1

National Library
Canada

Bibliothèque nationale
du Canada

CANADIAN THESES
ON MICROFICHE

THÈSES CANADIENNES
SUR MICROFICHE
0-315-04898-0

52792

NAME OF AUTHOR/NOM DE L'AUTEUR Claude Laferriere

TITLE OF THESIS/TITRE DE LA THÈSE Reliable Protocols for Synchronization and
Updating in Distributed Databases

UNIVERSITY/UNIVERSITÉ Carleton University

DEGREE FOR WHICH THESIS WAS PRESENTED/GRADÉ POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE Doctor of Philosophy (Engineering)

YEAR THIS DEGREE CONFERRED/ANNÉE D'OBTENTION DE CE DEGRÉ 1981

NAME OF SUPERVISOR/NOM DU DIRECTEUR DE THÈSE Professor S.A. Mahmoud

Permission is hereby granted to the NATIONAL LIBRARY OF
CANADA to microfilm this thesis and to lend or sell copies
of the film.

The author reserves other publication rights, and neither the
thesis nor extensive extracts from it may be printed or other-
wise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE
NATIONALE DU CANADA de microfilmer cette thèse et
de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la
thèse ni de longs extraits de celle-ci ne doivent être imprimés
ou autrement reproduits sans l'autorisation écrite de l'auteur.

DATE/DATE Friday 11th SEPT. 1981

SIGNED/SIGNÉ

PERMANENT ADDRESS/RÉSIDENCE FIXE

474 Wilbrod St., Apt. 704

Ottawa, Ontario,

Canada, K1N 6M9

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

**THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED**

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

**LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE**

RELIABLE PROTOCOLS FOR
SYNCHRONIZATION AND UPDATING
IN DISTRIBUTED DATABASES

by

C

Claude Laferriere

A thesis submitted to
the Faculty of Graduate Studies and Research,
Carleton University, in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy

Department of Systems and Computer Engineering
Carleton University

Ottawa, Canada

August, 1981

The undersigned recommend to the
Faculty of Graduate Studies and Research
acceptance of the thesis:

RELIABLE PROTOCOLS FOR SYNCHRONIZATION
AND UPDATING IN DISTRIBUTED DATABASES

submitted by Claude Laferriere in
partial fulfilment of the requirements
for the degree of Doctor of Philosophy.

Thesis Supervisor,

Chairman,

Department of Systems
and Computer Engineering

Carleton University,

September, 1981.

Abstract

The problems associated with the execution of distributed transactions in a distributed database are examined. Those problems are: definition of DDB software structure, correctness of operations, avoidance of deadlocks and robustness of algorithms. This thesis addresses each of those topics and produces a complete set of protocols together with a site architecture suited for distributed transactions. Correctness of operations and avoidance of deadlocks are taken care of by a synchronization protocol layer which uses a timestamp-based algorithm to order transactions consistently and a pre-declaration scheme to prevent deadlock. Distributed control is among the main features of this layer as well as of all the other protocol layers in this work. Robustness of execution is guaranteed by the execute protocol layer in the form of a distinct monitoring facility for both sites and transactions. Robust mechanisms for updating distributed relations are also presented. In a distributed database environment it is important to keep all distributed relations mutually consistent when updating activities are taking place. The techniques that are used to achieve this goal include: recursion, distributed failure detection, the use of sequences and up-lists coupled with a modified two-phase commit protocol enhanced by an extra wait strategy. The

updating of distributed relations is logically entrusted to a global structure called the commit protocol layer. As a rule, updating is carried out at all sites successfully. If it becomes impossible to update due to failures in the system, the failing nodes are removed from the up-lists until the updating can take place. Mechanisms to merge nodes that crashed but have now recovered are also described. The merging operation is also done concurrently with other system functions in order to minimize interference with normal system operation.

Acknowledgements

The author wishes to acknowledge gratefully the guidance and encouragement of his supervisor, Professor S.A. Mahmoud of the Department of Systems and Computer Engineering.

Thanks are due to all members of the department of Systems Engineering and to all the graduate students for providing a suitable atmosphere for research. More specifically, the author would like to mention: Dr. N. Janakiraman, Mr. S.S. Kamal, Mr. A.F. Guibord, Mr. D.A. Stanford and Miss D.L. Dodds for their help, co-operation and friendship during those years at Carleton.

The author would like to thank his wife, Dianne, for her support and her understanding which proved invaluable to him.

This research was made possible by graduate scholarships from the National Science and Engineering Research Council and the Ontario Graduate Scholarships programme.

Table of Contents

Chapter 1: Introduction

1.1 Characteristics of the Environment.....	1-4
1.2 DDB Problems.....	1-4
1.2.1 In a reliable system.....	1-4
1.2.2 In an unreliable system.....	1-8
1.3 Basic Approach.....	1-10
1.4 Thesis Overview.....	1-13

Chapter 2: Synchronization Protocol Layer

2.1 Introduction.....	2-1
2.1.1 State of the system at the entry point to SPL.....	2-1
2.1.2 Problem definition.....	2-2
2.1.3 Literature survey.....	2-6
2.1.4 Characteristics of SPL.....	2-15
2.2 Description of SPL.....	2-18
2.2.1 SPL structure.....	2-18
2.2.2 SPL strategy.....	2-19
2.2.2.1 Flow of control.....	2-19
2.2.2.2 Synchronization strategy.....	2-21
2.3 Specification of SPL.....	2-23
2.3.1 SPL data structures.....	2-23
2.3.1.1 Transaction notation.....	2-23
2.3.1.2 Resource access table.....	2-25
2.3.1.3 Synchronization messages.....	2-25
2.3.2 Synchronization protocol.....	2-26
2.3.2.1 Transaction acceptance test.....	2-27
2.3.2.2 Compute interference function.....	2-27
2.3.3 Performance considerations.....	2-28
2.3.3.1 Messages and rejections.....	2-28
2.3.3.2 One rejection phase.....	2-29
2.3.3.3 Message improvement.....	2-30
2.3.3.4 Blocking and re-starts.....	2-32
2.4 Validation of SPL.....	2-33
Assertion 2.4.1.....	2-34
Assertion 2.4.2.....	2-35
Assertion 2.4.3.....	2-36
Assertion 2.4.4.....	2-37
Assertion 2.4.5.....	2-39
2.5 Robustness of SPL.....	2-40
2.6 Specification of Robust SPL.....	2-42
2.7 Validation of robust SPL.....	2-43
Assertion 2.7.1.....	2-43
Assertion 2.7.2.....	2-44
2.8 Interaction of SPL/EPL/CPL.....	2-45

Chapter 3: Execute Protocol Layer

3.1 Introduction.....	3-1
3.1.1 State of the system at the entry point to EPL.....	3-1
3.1.2 Problem definition.....	3-4
3.1.3 Literature survey.....	3-5
3.1.4 Characteristics of EPL.....	3-6
3.2 Description of EPL.....	3-7
3.2.1 EPL structure.....	3-7
3.2.2 EPL strategy.....	3-8
3.2.3 Observations on EPL strategy.....	3-12
3.3 Specification of EPL.....	3-14
3.3.1 Introduction to specification.....	3-14
3.3.2 EPL data structures.....	3-14
3.3.3 EPL processes.....	3-14
3.4 Validation of EPL.....	3-15
Assertion 3.4.1.....	3-15
Assertion 3.4.2.....	3-17
Assertion 3.4.3.....	3-17

Chapter 4: Commit Protocol Layer

4.1 Introduction.....	4-1
4.1.1 State of the system at the entry point to CPL.....	4-1
4.1.2 Problem definition.....	4-2
4.1.3 Literature survey.....	4-5
4.1.4 Characteristics of CPL.....	4-8
4.2 Description of CPL.....	4-10
4.2.1 CPL strategy.....	4-10
4.2.2 CPL structure.....	4-13
4.2.2.1 Release and update table.....	4-15
4.2.2.2 Updating relations.....	4-15
4.2.2.3 Network access table.....	4-17
4.3 Specification of CPL.....	4-19
4.3.1 S-node and D-node Commit.....	4-19
4.3.2 NATupdate procedure.....	4-20
4.3.3 Resynch procedure.....	4-21
4.4 Validation of CPL.....	4-21
4.4.1 Assumptions.....	4-21
4.4.2 Validation assertions (up-list mechanisms).....	4-23
Assertion 4.4.2.1.....	4-23
Assertion 4.4.2.2.....	4-27
Assertion 4.4.2.3.....	4-28
Assertion 4.4.2.4.....	4-32
4.4.3 Validation assertions (distributed protocol).....	4-33
Assertion 4.4.3.1.....	4-34
Assertion 4.4.3.2.....	4-35
Assertion 4.4.3.3.....	4-43

4.5 Observations.....	4-49
4.5.1 Remotely initiated NATupdate.....	4-49
Rule I.....	4-49
Rule II.....	4-50
4.5.2 Special change in status.....	4-51
4.5.3 Cost considerations in CPL.....	4-52

Chapter 5: The Merge Procedure

5.1 Introduction.....	5-1
5.1.1 Purpose of merge procedure.....	5-1
5.1.2 Single node vs subnetwork merge.....	5-2
5.2 Literature Survey.....	5-3
5.3 Recovery Detection.....	5-4
5.4 Specification of Single Node Merge.....	5-6
5.4.1 Introduction to SNM.....	5-6
5.4.2 Single node merge.....	5-8
5.4.2.1 CPL merge.....	5-9
5.4.2.2 SPL/EPL merge.....	5-12
5.5 Validation of Single Node Merge.....	5-13
Assertion 5.5.1.....	5-14
Assertion 5.5.2.....	5-15
Assertion 5.5.3.....	5-21
Assertion 5.5.4.....	5-28
5.6 Subnetwork Merge.....	5-29
5.6.1 Options for SM.....	5-29
Options I.....	5-29
Options II.....	5-30
Options III.....	5-31
5.6.2 Specification of SM.....	5-32
5.6.2.1 CPL subnetwork merge.....	5-32
5.6.2.2 SPL/EPL subnetwork merge.....	5-34
5.6.2.3 Exception procedures.....	5-35
Exception procedure I.....	5-35
Exception procedure II.....	5-36
5.6.3 Validation of SM.....	5-37

Chapter 6: Conclusions and Recommendations for Further Study

6.1 Thesis Contributions.....	6-1
6.2 Appraisal of the System.....	6-2
6.2.1 Summary of basic assumptions.....	6-2
6.2.2 System strategy.....	6-3
6.2.3 Timer considerations.....	6-5
6.3 Recommendations for Further Study.....	6-7

Appendix A: Pseudo-Pascal Specification of SPL

A.1 Structure of the Appendix.....	A-1
A.2 Data Structure Definition.....	A-4
A.3 Procedures.....	A-5
A.3.1 Procedure remove.....	A-5

A.3.2 Procedure insert.....	A-5
A.4 Functions.....	A-6
A.4.1 Function oldesttransaction.....	A-6
A.4.2 Function choosetimestamp.....	A-6
A.4.3 Function preparemessage.....	A-7
A.4.4 Function checktimestamp.....	A-7
A.4.5 Function computeinterference.....	A-7
A.5 Submission Manager.....	A-8
A.6 Resource Access Table Manager.....	A-10

Appendix B: Abbreviations and Notation

B.1 Abbreviations.....	B-1
B.2 Notation.....	B-2

References.....	R-1
-----------------	-----

List of Figures

Figure 1.1	A distributed database.....	1-16
Figure 1.2a	Transactions T1 and T2.....	1-17
Figure 1.2b	Example of data inconsistency.....	1-17
Figure 1.3a	Transactions T1 and T2 (revised).....	1-18
Figure 1.3b	Example of deadlock.....	1-18
Figure 1.4	Phases of a distributed transaction.....	1-19
Figure 1.5	System access graph.....	1-20
Figure 2.1	SPL access graph.....	2-49
Figure 2.2	S-nodeSynch procedure.....	2-50
Figure 2.3	D-nodeSynch procedure.....	2-51
Figure 2.4	FSM of S-nodeSynch procedure.....	2-53
Figure 2.5	FSM of D-nodeSynch procedure.....	2-54
Figure 2.6	Robust S-nodeSynch procedure.....	2-55
Figure 2.7	Robust D-nodeSynch procedure.....	2-57
Figure 2.8	Time-out procedure for S-node/D-nodeSynch.....	2-59
Figure 2.9	Procedure processmessage.....	2-60
Figure 2.10a	Newsynch and associated time-out procedure.....	2-62
Figure 2.10b	Function operationpossible.....	2-63
Figure 2.11	FSM of robust S-nodeSynch procedure.....	2-64
Figure 2.12	FSM of robust D-nodeSynch procedure.....	2-65
Figure 3.1	EPL access graph.....	3-22
Figure 3.2	EPL strategy.....	3-23
Figure 3.3	EPL data structures.....	3-24
Figure 3.4	Activity monitoring process.....	3-25
Figure 3.5	Transaction process.....	3-28
Figure 4.1	CPL strategy.....	4-55
Figure 4.2	CPL access graph.....	4-56
Figure 4.3	Release and update table.....	4-57
Figure 4.4	Modified two-phase commit protocol.....	4-58
Figure 4.5	Complete protocol.....	4-58
Figure 4.6	Network access table.....	4-59
Figure 4.7	S-nodeCommit procedure.....	4-60
Figure 4.8	D-nodeCommit procedure.....	4-61
Figure 4.9	Time-out procedure for S-node/D-nodeCommit.....	4-62
Figure 4.10	Exchange of messages in protocol.....	4-63
Figure 4.11	NATupdate and associated time-out procedure.....	4-64
Figure 4.12	S-node*NATupdate procedure.....	4-65
Figure 4.13	D-node*NATupdate procedure.....	4-66
Figure 4.14	Associated time-out procedure and re-evaluatestatus of relations procedure for S-node/D-node*NATupdate procedure...	4-67

Figure 4.15	Resynch procedure.....	4-68
Figure 4.16	Resynch timeout procedure and exception procedure.....	4-69
Figure 4.17	FSM of S-nodeCommit procedure.....	4-90
Figure 4.18	FSM of D-nodeCommit procedure.....	4-71
Figure 4.19	FSM of NATupdate and Resynch procedures.....	4-72
Figure 5.1	Merge process.....	5-40
Figure 5.2	Detection algorithm.....	5-41
Figure 5.3	Symbols of detection algorithm.....	5-42
Figure 5.4	Restore procedure.....	5-43
Figure 5.5	Example of inconsistency during merge...	5-45

Figure 1.1: A distributed database

Chapter 4

CPL	Commit protocol layer
DT	Distributed transaction
STP	Supervisory transaction process
TU	Target update
TUD	Target update directory
DURQ	Distributed update request queue
NAT	Network access table
AT	Abort table
CC	Concurrency control
RUM	Release and update manager
RUT	Release and update table
TF	Timefield
RTU	Ready to update
CSS	Communication subsystem
SN	Suspected nodes set

Chapter 5

NAT	Network access table
NSQ	Network access table sequence number
MP	Merge process
DR	Distributed relation
RUT	Release and update table
GC	Global co-ordinator
RAT	Resource access table
TF	Timefield
STP	Supervisory transaction process
SNM	Single node merge
SM	Subnetwork merge

2. By alphabetical order

AC:	Abort co-ordinator
ACK:	Acknowledgement messages exchanged by GC's or RUM's
AMP:	Activity monitoring process
APL:	Abort protocol layer
AT:	Abort table
CC:	Concurrency control
CPL:	Commit protocol layer
CSS:	Communication subsystem
DB:	Database
DBM:	Database manager
DDB:	Distributed database
DLRQ:	Distributed lock request queue
DR:	Distributed relation
DRQ:	Distributed request queue
DT:	Distributed transaction

DURQ: Distributed update request queue
 EPL: Execute protocol layer
 GC: Global co-ordinator
 INT: Interference set
 LC: Local co-ordinator
 LRLQ: Local release queue
 LRQ: Local request queue
 MP: Merge process
 NAT: Network access table
 NI: Network interface
 NTS: New timestamp message sent by originating GC in
 SPL in case of rejection
 O/S: Operating system
 PMRQ: Process monitoring request queue
 Psem: Private semaphore
 QPP: Query processing process
 RAT: Resource access table
 REJ: Rejection message sent to originating GC in SPL
 when TS of T is unacceptable
 REQ: Request message used by originating GC in SPL
 RS: Recovery set of relations to be brought up to date
 during a merge procedure
 RTBLQ: Ready to be locked queue
 RTBRQ: Ready to be released queue
 RTS: Required time to achieve synchronization in SPL
 RTU: Ready to update message
 RUM: Release and update manager
 RUT: Release and update table
 SM: Subnetwork merge
 SN: Suspected nodes set
 SNM: Single node merge
 SPL: Synchronization protocol layer
 STP: Supervisory transaction process
 T: Transaction
 TN: Target nodes
 TP: Transaction process
 TPM: Transaction process monitoring (table)
 TR: Target relations
 TS: Timestamp
 TU: Target update
 TUD: Target update directory
 U: Update operation

To Dianne

CHAPTER 1

Introduction

1.1 Characteristics of the Environment

With the advent of computer networks, such as ARPANET, centralized computer systems were interconnected, and new areas for investigation were thus opened. Computer systems, linked in such a fashion and properly controlled and co-ordinated, displayed enhanced performance in terms of speed and reliability. Those interconnected systems found acceptance under various forms in many applications and are called Distributed Systems (DS).

Designers of databases were quick to see the advantages of distribution. On one hand, database relations could be replicated at different sites so as to reduce access time and network communication traffic. On the other hand, control and co-ordination of operations became issues that had to be carefully studied if the reliability and robustness advantages, inherent to distributed systems, were

to be conserved. Investigation of co-ordination of activities, reliability and robustness, in the context of a Distributed DataBase (DBB), are the topics of this thesis. Before identifying the problems which will be of interest, a few definitions are in order:

1. Distributed Systems (DS)

A distributed system is one in which independent computers are connected together by a network. Such system usually features independent scheduling of activity (i.e. processes) at each site and co-ordination of processes through exchange of messages. Since co-ordination depends on the network, some of its characteristics have to be taken into account. The communication network introduces variable message transit delays which are not negligible when compared to the time between significant events in the distributed system; message ordering is only guaranteed between two parties. The communication network also poses reliability problems, in the form of lost messages (*), which further complicate the reliability problems of the distributed system itself.

2. Distributed DataBases (DDB)

A distributed database is one which is supported by a

(*) Which are symptomatic of either crash of the sender or crash of network links.

distributed system, with individual databases at each site of the distributed system. Those individual databases could be of similar contents to give a fully redundant DDB. The databases may also be different, while still having some common relations with each other. This is the more general case which is shown in figure 1.1 and which will be considered throughout this thesis. All software mechanisms for the general case apply to the fully redundant case.

3. Distributed Relations (DR)

The unit of information in a DB is a relation. A distributed relation is a relation which is present at more than one site. In other words, relation r_1 may have copies at sites n_1 , n_2 , n_3 and n_4 , thus making its access easier for users at those sites but making the co-ordination of its updating more difficult.

4. Transaction (T)

A transaction is a sequence of actions that a user wants to perform on the database. Those actions, (read, write, merge, etc.,...), are grouped together under a transaction for consistency reasons [ESWA76], with the result being that the transaction is the basic unit of work in the DB. There are two types of transactions: local and distributed. Local transactions, as the name implies, access the database at one site and do not affect other sites. Distributed

Transactions (DT) affect more than one site either because they work on relations with copies at different sites (i.e. DR's) or simply because their required relations reside at several sites. For the purpose of this work, a distributed transaction will be represented at each site by a Transaction Process (TP). The co-ordination of those TP's will be entrusted to a Supervisory Transaction Process (STP) which will be the TP at the site where the transaction originated.

1.2 DDB Problems

1.2.1 In a reliable system

The design of a distributed database facility has been entrusted to a database designer and his job is to provide the users with consistent retrieve and update operations. The DDB system is assumed to be entirely reliable and only the properties of the distributed system will complicate the designer's task.

From a distributed co-ordination point of view, the retrieve operation seems to be the easiest function to implement. A retrieve operation does not change the contents of a relation and for that reason any copy of the

relation can be used. More often than not, the closest copy will be used for efficiency reasons. Reading is not critical as long as no write operation is in progress. Co-ordination with other read operations is therefore not necessary and multiple read operations are allowed on a single relation if so required.

When updating operations are introduced, the task of co-ordinating reads and writes goes beyond the typical Reader-Writer problem described in [HOAR74]. Initially the designer was not aware of the extra dimension taken by the task of co-ordinating read/write activities. While experimenting with the system, the designer became aware of a certain condition called "data inconsistency" which sometimes happened when transactions of the type shown in figure 1.2a were run on the system. The time graph of figure 1.2b shows what course of events led to data inconsistency. In figure 1.2b, r_1 , which is the relation being accessed, is present at both site 1 and 2.

In order to prevent this type of inconsistency in the database, the designer came up with the idea of using locks [ESWA76] on the relations being used. He therefore re-wrote T_1 and T_2 of figure 1.2a to take locking into account. His reasoning was that for any transaction which is planning to use distributed relations, no access shall be granted until

the transaction has successfully locked all the copies of the affected relations. With that rule, the designer hoped to solve, once and for all, the inconsistency problem. The new version of T1 and T2 is in figure 1.3a.

Much to his delight his solution seemed to work for a while. Later on it became clear that, occasionally, transactions would meet in a deadly embrace, effectively preventing one another from acting any further. This condition is shown in figure 1.3b and is known as deadlock.

The example of the disgruntled designer, although not realistic, illustrates some of the difficulties in DDB. Those difficulties are brought about by the independent scheduling of transactions at sites in the DDB, by the variable message transit delays introduced by the network, and, by the absence of guarantees regarding the ordering of messages in multiparty conversation.

After some careful thinking, the designer came up with several observations which are listed below:

1. Reads in themselves cause no problem. It is the combination of concurrent read and write operations that is troublesome. Some form of synchronization will have to be used.

2. Intersecting write operations have to be synchronized with each other to prevent deadlock, and to prevent those writes from being applied in different order at different sites.

3. In a read/write transaction, the results of the read are used to produce a target update for the write. Therefore the results of the read operation have to be consistent. In other words, a read operation on a distributed relation should always, at any given time, yield the same result, irrespective of the site it is obtained from.

4. For a DDB supporting concurrent operations, the result of any transaction should not be affected by other transactions. In actual fact, the concurrent execution of transactions should be equivalent to their serial execution where no concurrency is allowed.

Wiser from his experiences, the designer concluded that the solution to his problems will take the form of a protocol. The functions of this protocol will be to synchronize transactions among themselves and to allocate relations so as to prevent deadlock.

1.2.2 In an unreliable system

Real life systems are not entirely reliable. Failures may happen which could force a protocol out of its safe states into, perhaps, some trapping or incorrect states. Failure situations have to be identified and protected against.

The failures which are common in distributed systems are: failures of individual processes, total failure or crash of a site, failure of communication links, failure of the local database storage facilities, etc... The causes of such failures are numerous. A transaction process, for example, may fail if it encounters during its execution over/underflow conditions, failures of certain system services, coding errors, etc., which would force the Operating System (O/S) to abort it.

It should be emphasized that this work is concerned with reliable interprocess co-ordination in an unreliable system. Consequently, throughout the thesis, some basic assumptions are made about the file storage system and the local database. Those two system entities are assumed to work well and to protect themselves against failures by such techniques as described in [LAMPSxx], [MENA80b], [STUR80] and [VERH79].

In section 1.2.1, it was indicated that the solution to the synchronization and resource allocation problems took the form of a protocol. The synchronization protocol may be affected adversely if failures, such as loss of protocol messages and/or failures of processes managing the protocol, happen in the system. One of the purposes of the synchronization protocol is the creation of an ordering of events. A major consequence of a failure in the execution of the protocol would be the hold up of all other transactions which have to be ordered with respect to the transaction currently stalled.

Similarly, when a distributed transaction has components (i.e. TP's) executing at various sites, failure(s) of any of those TP's could prevent the transaction from completing. The resources allocated to the transaction could not be returned either to the database manager or to the O/S. The result of this will be, again, the hold up of all transactions which have to access blocked resources.

Clearly, the synchronization protocol has to have mechanisms to prevent transaction jams. Furthermore another protocol is needed to ensure that failures of sites or TP's will not cause a crippled transaction to hold onto resources indefinitely during its execution.

Hereto, the effects of failures have been to degrade system performance. Other situations exist where failures can cause inconsistencies in the database, such as during the updating of distributed relations. If the updating of a distributed relation is not successful at all sites, the copies of the relation will no longer be mutually consistent. This problem, or in other words, how to ensure complete success or complete abort of any updating activity, is very difficult. It is solved using protocols that guarantee reliable updating of distributed relations.

1.3 Basic Approach

The reliable process co-ordination protocols of subsequent chapters, are meant to form the Distributed Processing (DP) layer of the Architecture for Distributed Database (ADD) system [TOTH80]. The basic approach to the design of these protocols rests on a particular conception of a distributed transaction, shown in figure 1.4. The meaning of this figure can be summarized as follows:

A user's request for performing some actions, (read relation, merge, process relation, write relation; etc...), is processed by the upper layers of ADD. At the DP level, a set of requirements emerges, indicating, in the case of a distributed transaction, the sites and relations that will

be accessed and also the necessary instructions pertaining to the execution of the transaction.

The transaction then passes through a synchronization phase, corresponding to the synchronization protocol layer of chapter 2. This guarantees correct access to needed relations at all sites of interest. Then, the transaction will go through the execute phase; reliable execution of the transaction is guaranteed by the execute protocol layer of chapter 3. At the end of the execute phase, the transaction would have produced target updates which will be "committed" (i.e. entered) in the database. The commit phase corresponds to the commit protocol layer of chapter 4.

If a transaction failure takes place at any time during those three phases, the transaction will enter an abort phase. The abort phase is represented by a timer protocol in the case of the first two phases and by a commit operation on an abort table in the case of the commit phase. The abort phase is thus an integral part of other phases but is included for the sake of clarity.

The system supporting this conception of a transaction is made out of architecturally identical sites. Such a site architecture is shown in the access graph of figure 1.5. In that figure, circles denote processes and rectangles denote

data structures. Each of the protocol layers corresponding to transaction phases is represented by a group of processes and data structures to be specified and validated in later chapters.

In the access graph of figure 1.5, the DataBase Manager (DBM), the local DataBase (DB), the lock table and process/DBM interface constitute a local DB facility. The synchronization protocol layer is represented by the Global Co-ordinator (GC), the Local Co-ordinator (LC), the Resource Access Table (RAT) and other various queues. The execute protocol layer is made out of the Activity Monitoring Process (AMP), the Transaction Process Monitoring (TPM) table and the Target Update Directory (TUD). The commit protocol layer is composed of the Release and Update Manager (RUM), the Release and Update Table (RUT), the Network Access Table (NAT), the Abort Co-ordinator (AC) and the Abort Table (AT).

Finally, the transaction is executed by the Transaction Process (TP) which receives its instructions either from the Query Processing Process (QPP) or from another TP called the supervisory TP. All exchanges of messages with other sites are carried out by the Communication SubSystem (CSS).

The layered approach to the execution of a transaction,

combined with the separation of functions in the architecture, should yield a flexible system that can be tailored to any application.

1.4 Thesis Overview

The main objectives of this thesis are the specification and validation of the protocol layers mentioned previously. These objectives will be achieved, at times, in the context of actual processes, and at other times, in the context of abstract algorithms.

In order to make the presentation easier, the four remaining chapters will be self contained chapters, each with its own introduction, problem definition and literature survey.

In chapter 2, a synchronization protocol is presented which guarantees correct access to database relations. This protocol features distributed control, the use of synchronized counters and a pre-declaration scheme to ensure correct database operation in a reliable system. A modified version of the protocol is also presented; this modified protocol is to operate in a system which can be affected by failures of individual processes and/or failures of the communication network.

In chapter 3, the execution of a transaction is closely monitored by an execute protocol. The protocol will provide system services to monitor transactions and will also inform the supervisory transaction process of the completion of the execute phase. Chapter 3 presents this execute protocol together with a description of a transaction process.

The problem of preserving the mutual consistency property of distributed relations is treated in chapter 4. In that chapter a commit protocol layer is introduced with the objectives of providing correct and robust updating facilities. Several mechanisms are used by this layer, namely: the use of up-lists (i.e. list of up nodes), status indicators for database relations and a modified two-phase commit protocol.

Chapter 5 addresses the problems associated with the re-insertion of nodes which failed but are now recovering. The local copy of the database as well as all the other control data structures have to be brought up to date with those of the rest of the network. The up-lists maintained by other nodes also have to be updated with the fact that a node has recovered. Algorithms to merge correctly such nodes are presented in that chapter.

Conclusions and recommendations for further study are

to be found in chapter 6. Appendix A contains a pseudo-Pascal description of a possible implementation of the synchronization algorithm of chapter 2. Appendix B includes a list of abbreviations used throughout the thesis and a definition of the symbols used in the algorithms.

Figure 1.1: A distributed database

T1:begin

T2:begin

for i:=1 to 2 do

for i:=2 to 1 do

read r1 @ i

read r1 @ i

if r1=A

if r1=A

then r1:=A'

then r1:=A'

else r1:=A*

else r1:=A*

fi

fi

od

od

end

end

{r1 = relation r1 and residing at sites 1 and 2. T1 executes at site 1 and T2 executes at site 2}

Figure 1.2a: Transactions T1 and T2

Figure 1.2b: Example of data inconsistency

T1:begin

```

lock r1 @ site 1
lock r1 @ site 2
for i:=1 to 2 do
  read r1 @ i
  if r1=A
  then r1:=A'
  else r1:=A*
  fi
od
release r1 @ 2
release r1 @ 1

```

end

T2:begin

```

lock r1 @ site 2
lock r1 @ site 1
for i:=2 to 1 do
  read r1 @ i
  if r1=A
  then r1:=A'
  else r1:=A$
  fi
od
release r1 @ 1
release r1 @ 2

```

end

Figure 1.3a: Transactions T1 and T2 (revised)

Figure 1.3b: Example of deadlock

Figure 1.4: Phases of a Distributed Transaction

Figure 1.5: System access graph.

CHAPTER 2

Synchronization Protocol Layer

2.1 Introduction

2.1.1 State of the system at the entry point to SPL.

The execution of a synchronization session begins when a user issues a transaction. The transaction is processed through various layers of the ADD system [TOTH80] and finally emerges as a set of instructions and requirements (i.e. relations to be accessed, sites to be contacted, etc.,) at the originating site. A transaction process, in time, dequeues directives from higher levels (see TP specification in chapter 3) and starts initiation of the distributed operation. Other TP's at other sites are not yet aware of this new transaction.

2.1.2 Problem definition

In a DDB environment, the scheduling of transactions is done independently at all sites in the network. Several difficulties arise from this situation and they are exacerbated by the behaviour of the network itself.

In order to visualize those difficulties, logs [BERN79] will be introduced. A log, L ,

$$L = \{T_1, T_2, T_3, T_4, \dots\},$$

is a chronological list of events that took place in the system. In this case, the events in the log are transactions, T_i ,

$$T_i = \{R_i, U_i\},$$

which are made up of a read operation (R_i) and of a write operation (U_i). It is assumed that, at the beginning of its execution, the transaction can read all the relations it requires into a private workspace. Consequently, for the purpose of this simple representation, it is not necessary to take into account the actual processing of information done by the transaction between its read and write operations. A transaction is also partially specified by the following sets:

$RS(T_i)$ the read set of T_i which includes all the relations used by T_i for read purposes

$WS(T_i)$ the write set of T_i which includes all the

relations used by T_i for write purposes
 $R(T_i)$ which is equal to $RS(T_i) \cup WS(T_i)$.

The case that will now be considered is that of a set of transactions $\mathcal{T}$,

$$\mathcal{T} = \{T_1, T_2, T_3, T_4, T_5\},$$

with the elements of $\mathcal{T}$ being characterized as follows:

$$\forall T_i, T_j \in \mathcal{T}, \exists T_j, T_j \in \mathcal{T}, \text{ such that } R(T_i) \cap R(T_j) \neq \emptyset.$$

In other words, all the T 's in $\mathcal{T}$ are somehow linked by their read set and write set. The case where they are not linked is not of interest since the sequence of execution would be of no consequence.

The elements of $\mathcal{T}$ will now execute on the system with each T_i being independently scheduled. In such a case, the criterion for correctness of execution will be the serial log L ,

$$L = \{T_1, T_2, T_3, T_4, T_5\}.$$

This log can also be expanded to yield

$$L = \{R_1, U_1, R_2, U_2, R_3, U_3, R_4, U_4, R_5, U_5\}.$$

For the sake of simplicity, it will be assumed at this point that only two sites of the DDB are affected (i.e. n_1 and n_2). Each site, n_i , has its own log, L_i , and in general $L_i \subseteq L$ (L being the system log) since the database is assumed not to be fully redundant. Consequently, some T 's may be in L but not in L_i . Two situations have to be

considered with respect to the execution of transactions at n_1 and n_2 .

Situation I

In situation I, T_1 , T_2 and T_4 are in L_1 and L_2 in the following way:

$L_1 = \{R_1, U_1, R_2, U_2, R_4, U_4\}$

$L_2 = \{R_1, R_2, U_1, U_2, R_4, U_4\}$.

In that case the result of R_2 is not consistent since part of it was done before U_1 and another part after U_1 . There are two solutions to rectify this problem:

1. Careful pre-analysis (*) could be used to determine whether or not this situation is of concern. If it does matter, however, the second solution should be adopted.
2. The read and write operations are joined under a transaction and cannot be separated. This is an attractive solution since, in most cases, pre-analysis is an expensive procedure and the savings that it affords in terms of enhanced concurrency may not always materialize.

 (*) This is basically SDD-1, a system which will be discussed in the literature survey.

Situation II

Even after having joined the read and write operations of transactions, there still remains a potential source of inconsistency during the execution of T . This situation is shown in the following two logs:

$L1 = \{R1, U1, R2, U2, R4, U4\}$

$L2 = \{R2, U2, R1, U1, R4, U4\}$.

The logs $L1$ and $L2$ are clearly not serializable into L and such a situation has to be prevented from occurring. Therefore, transactions have to be ordered by some suitable mechanisms which will hereafter be referred to as Concurrency Control (CC) mechanisms. Orderings created by the CC mechanisms can be of two types:

Total ordering in which transactions are ordered with respect to the absolute time of their coming into the system, and

Partial ordering which may be different from total ordering but is nevertheless the same at all sites in the DDB. In other words, a partial ordering is reducible to a serial ordering.

Besides the synchronization problem just discussed,

there also exist deadlock possibilities during the concurrent execution of some transactions. This was shown in section 1.2.1 of chapter 1. In summary, the problems encountered in a DDB with respect to concurrency of operations are those of synchronization of transactions and of deadlock prevention..

2.1.3 Literature survey

The task of synchronizing a distributed transaction can be accomplished in various ways. However, two major strategies stand out based on two distinct philosophies:

1. Centralized control

Centralized control has been used in many systems [MAHM75], [MAHM77], [MENA80a]. In this strategy, all control functions are assigned to one node, (i.e. the network controller). The central network controller being unique (*) a consistent ordering of actions and an effective deadlock control policy can be implemented. Centralized control is relatively simple, straightforward and easy to implement. However, the strategy suffers from many problems, namely: a failure of the central network controller incapacitates the system. As well, bottlenecks

(*) In some systems, there may exist more than one controller with non-overlapping responsibilities.

at the central controller are detrimental to system performance. Furthermore, the overall amount of concurrency is small when compared to distributed control schemes.

2. Distributed control

Distributed control schemes in which participating nodes arrive at a consensus on how to control system activities were created in order to alleviate some of the undesirable properties of centralized systems. These distributed control schemes fall into four broad categories [LELA79] with respect to the creation of a correct ordering of actions in the DDB. The problem of deadlock (which is closely linked to ordering) in DDB using distributed control is covered later. The four categories are summarized below:

a. Utilization of physical clocks

Physical clocks are used as a reference to create an ordering of all events in a system

i. Single clock system

This particular approach goes against the general philosophy of distributed systems and it is also highly vulnerable to failure of the central clock. Furthermore a reliable communication system is necessary as well as an accurate estimation of interprocess message

delays.

ii. Multiple clock systems

As in the previous case, an accurate estimation of interprocess message delay has to be available. This is necessary in order to correlate individual local times with each other. There is also some bounds on how synchronized those clocks should be, as explained in [LAMP78a]. Another consideration is that timing messages (e.g. during a re-synch phase) can sometimes be lost thus making synchronization probabilistic. The probability of not achieving synchronization can be reduced but at the expense of performance. [ELLI73] covers other applications using the above technique.

b. Explicit utilization of control privileges

Ordering of events is achieved through circulation of a control token. Only those with the control token can issue commands. The two subclasses are distinct by the way the token circulates.

i. Circulating token [LELA77a] [LELA77b]

The underlying assumption, which is also common

to the next subclass, is that all processes are distinct. All those distinct processes are located on a virtual ring which is not bound to any physical set-up or configuration. It is a good vehicle for interprocess communication since it provides readily for mutual exclusion but admittedly the parallelism is poor. There might also exist some problems in case of failures although a method to uniquely re-generate the control token is described in [LELA77b] and refined in [CHANG79]. Additions to and deletions from the virtual ring structure are not straightforward.

ii. Shared variables [DIJK74]

This subclass also embodies the virtual ring concept. Although there is no explicit token circulation like in the previous case, there is, nevertheless, an implicit one. Possession of the control privileges is inferred from the observation of variables shared with adjacent processes on the virtual ring. This scheme is not too resistant to failures and recovers only with great difficulty. Additions to and deletions from the ring structure are difficult.

c. Utilization of counters

In these solutions, the events are ordered with respect to a counter (or counters) instead of physical clocks. The differences between the two subclasses are mainly the number of counters used and the ordering imposed.

i. Synchronized counters

Synchronized counters are also known as logical clocks and are, by far, the most popular solutions. They have been used extensively (*) [BERN78], [BERN80], [LAMP78a], [MAHM79], [ROTH80], [ROSE78], [STON79], [THOM78], in either an implicit or explicit fashion. Usually each process is provided with its own local counter and each process also executes an algorithm to maintain the value of that counter. Events are given timestamps which are related to the value of some counter (logical clock) and a partial ordering is built on that provided that the timestamps are unique and that communication is done only through the network. The logical clock solution can be

(*) [ROSE78], [STON79] and especially [BERN80] are hybrid systems not fitting neatly in one class. They will be elaborated on later.

made to handle failures well, as far as logical time keeping is concerned.

ii. Independent counters

This approach ties ordering to operations rather than processes or resources. Independent counters, also known as event counts [REED79], reflect the number of operations that occurred in the system. Dynamic extension of independent counters is not easy and they are very sensitive to failures.

d. Utilization of sequences

It is possible to sequence operations and thereby create an ordering which is consistent and unique. The sequencing is done by issuing 'tickets' to processes. The two subclasses differ in the way tickets are given.

1. Static sequencer [REED79], [STON79]

The sequencer is a static entity that issues tickets and ensures a consistent ordering. This solution is sensitive to failures.

ii. Circulating sequencer [LELA78b]

The sequencer circulates on a virtual ring like the control token of a previous solution (b-1). The sequencer issues tickets for particular operations just as before. This approach is supposed to combine the advantages of the virtual ring concept while at the same time to increase the parallelism. It is also sensitive to failures.

The previous discussion of distributed ordering schemes would not be complete without mentioning SDD-1 [BERN80], Ingres [STON79], and [ROSE78]. In the distributed version of Ingres, each distributed relation has a primary copy against which all updates are committed. Ingres, therefore, uses a blend of concurrency and centralism: concurrency, because there are many relations in the system thus yielding many potential controllers; centralism, because the primary copy is a central controller with respect to update operations. Conflicts in multi-relation transactions are resolved through the use of timestamps.

In SDD-1, transactions are broken down into classes. Within each class, a strict serial execution is enforced by a unique controller for that class. Furthermore, conflicts among classes are assumed to be known because the sorting of transactions into classes is done by "pre-analysis". It

should also be mentioned that, in SDD-1, the read and write operations of a transaction need not always follow one another immediately. Pre-analysis is used to determine if this freedom may be allowed and if so, to what degree. Consequently, SDD-1 features four timestamp-based protocols, each of increasing power, with the fourth one equivalent to mutual exclusion among classes. Ordering of events and deadlock prevention are achieved by a mixture of pre-analysis, central control (within a class) and synchronized counters for class conflicts. Unfortunately for SDD-1 however, the methods by which this pre-analysis could be done have yet to be specified for even an experimental system. In fact, this pre-analysis may be a problem for which only heuristic solutions will be found; any addition or modification to the system would, of course, require another pre-analysis.

Finally, a system level concurrency scheme was proposed in [ROSE78] and it differs from other well known systems in that transactions do not have to pre-declare their needs (i.e. the relations they want to access). This means that transactions can evaluate their read-sets and their write-sets dynamically. Consequently [ROSE78] features a deadlock detection scheme as well as an ordering of events based on unique timestamps. Another feature of this system is that it uses transaction roll-back as a mean of achieving

correctness and breaking deadlocks (see deadlock section). Transaction roll-back is in fact mandatory because of the dynamic nature of the claims of a transaction (i.e. a transaction is rolled back if it cannot access relations of a suitable logical age in the system).

Solutions to the second problem, that of resource allocation, are concerned with preventing deadlock situations from occurring. Three methods for doing so are presented:

1. Priority Scheme [ROSE78]

- A given process can pre-empt others if it is older than the other processes. Suitable mechanisms for pre-emption and for rolling back the victimized processes are mandatory, thus complicating this approach.

2. Bulk request (pre-declaration)

- With this scheme, a transaction makes its needs known at the beginning of its execution. The request for resources covers everything that the transaction needs and, if granted, the transaction can go ahead safely without fear of being deadlocked. The release need not be 'bulk' and may be gradual as long as released resources will not

be needed again. Bulk request is obviously not very efficient but due to the nature of distributed systems, it has been, in various forms, the most popular solution.

3. On-line detection schemes

On-line detection schemes have been proposed in the literature [LOME79], [MENA79] etc.. Most of these schemes use complex algorithms which make use of control tables and introduce expensive overhead. Furthermore, great care has to be exercised with the maintenance of the tables which may be replicated throughout the system, depending whether the deadlock detection is a centralized or distributed scheme.

2.1.4 Characteristics of SPL

The purpose of the Synchronization Protocol Layer (SPL) is to provide a Distributed Transaction (DT) with means:

1. To create a set of Transaction Processes (TP) at all sites of interest with a Supervisory Transaction Process (STP) to interface with the query processing layers of the system.

2. To guarantee to that set of TP's access to the required DDB relations so that serial or concurrent execution of a DT gives the same results.
3. To prevent deadlock among TP's of various DT's.

The first objective is met by the site architecture of SPL, displayed in figure 2.1. The architecture of SPL is, of course, like the architecture of the layers that will follow (EPL, CPL), identical among all sites. This points to a very important characteristic of SPL, namely the fact that SPL is a totally distributed entity using distributed control.

The second and third objectives are met by a synchronization algorithm which is based on a synchronization protocol for the ADD system, found in [MAHM79]. This synchronization algorithm has the following characteristics:

1. The transaction is the basic unit of work (i.e. read and write operations are joined). This characteristic sacrifices some possible concurrency for simpler CC mechanisms.
2. Logical clocks are used to give unique timestamps

(TS) to transactions in order to produce a partial ordering.

3. Each relation has a timefield indicating when it was last used. Those timefields, which are updated by the transactions at the end of their synchronization, together with the logical clocks of the previous items, are used to maintain a time window during which transactions can be accepted for synchronization. A reservation table, the Resource Access Table (RAT), implements this logical time window.
4. In order to prevent deadlock, transactions pre-declare their needs and perform bulk request and bulk release.
5. Completely distributed control is used throughout. This property makes the algorithm more robust to failures. In fact, the synchronization algorithm of SPL is used as a basis for a study in reliability and robustness within the context of the ADD system.

2.2 Description of SPL

2.2.1 SPL structure

The structure of SPL is shown in the access graph of figure 2.1. In that figure, the Global Co-ordinator (GC) assumes the responsibility of carrying out the SPL algorithms in concert with other GC's at other sites.

The local co-ordinator (LC) merges the serially correct requests from the global co-ordinator with requests of local nature and local needs and sends them to the database manager. In the ADD system the database manager does not have extra functions to perform. In other words, the database manager and the local co-ordinator for that matter are unaware of any distributed operations.

The query processing process is not really of concern here but is included for the sake of completeness. It implements the system query processing strategy, that is, it translates requests from higher levels into a set of actions and directives which will be carried out by the transaction process. The latter process is not unique and there can be a great many similar processes created as the O/S needs them. The functions of the transaction process are to carry out the directives of the query processing strategy.

2.2.2 SPL strategy

2.2.2.1 Flow of control

1. At the originating site

When a user, at a given site, generates a transaction, it is processed by several layers of the ADD system [TOTH80] before getting to the query processing level. Assuming that the transaction satisfies the system consistency criteria, the query processing process creates an entry in the job queue which will eventually be dequeued by a transaction process. The transaction process (TP) will then find out by looking at the submission whether or not the transaction has to be synchronized. If so, the TP creates an entry in the distributed request queue (DRQ) and then waits on its private semaphore.

The entry in the DRQ will eventually reach the global co-ordinator which will initiate a synchronization operation for that entry. The synchronization operation is, in this case, the SPL algorithms to be specified in section 2.3. The synchronization, being a global operation, only involves global co-ordinators.

When the transaction is successfully synchronized, it is removed from the resource access table (RAT) and sent by

the global co-ordinator to the local co-ordinator through the distributed lock request queue (DLRQ). The local co-ordinator merges the serially correct entries of the DLRQ with the entries of the local request queue (LRQ) and sends them to the database manager through the ready to be locked queue (RTBLQ).

The transaction will wait in the RTBLQ (*) up until it is at the head of the queue for all the relations it needs and that those relations are available. When those two conditions are satisfied the DBM assigns the relations to the transaction and activates its transaction process (TP) through a signal operation on that process' private semaphore. The transaction process now being active is then in a position, together with other TP's in case of a distributed transaction, to carry out the transaction.

2. At a remote site

At a site other than the originating site, but still nevertheless involved in the operation, the procedure is very similar. The global co-ordinator at a remote site takes part in the synchronization of the transaction. Since the transaction was not locally originated, the global

(*) It is important to realize that successful completion of the synchronization algorithm for a transaction does not guarantee the immediate availability of the requested resources.

co-ordinator would put an entry for that transaction in the job queue. A transaction process will eventually dequeue that entry and since it is of a remote origin, the TP will wait for the go-ahead signal from the DBM by simply doing a wait operation on its private semaphore. The rest is identical with the previous case.

2.2.2.2 Synchronization strategy

The synchronization algorithm, which will be informally described in this section, creates a partial ordering of events and prevents deadlock situations from occurring. The methods to do so are as follows:

1. Logical counters (i.e. counting logical time) are assigned to all relations in the system. More precisely, each copy of a given relation has a logical counter associated with it, which will hereafter be referred to as a timefield.
2. When a transaction is submitted for synchronization to the global co-ordinator at the originating site, it is given a unique timestamp. The value of that timestamp is chosen so that it is greater than the timefield of any relation requested by the transaction at the originating site.

3. With that condition satisfied, the global co-ordinator enters the transaction into the Resource Access Table (RAT). It then sends request messages to all other GC's whose site will be involved in the operation. The message contains the transaction's identification, its unique timestamp and other information which will be defined later.
4. Upon receiving the request message, other GC's will test the timestamp of the transaction to see if it is greater than the timefield of any relation requested by the transaction. If it is, the transaction is entered in the RAT of the node where that condition is satisfied and acknowledgements are sent to all other GC's involved in the operation. If it is not, a rejection message is sent to the GC at the originating site which will, in time, send new timestamp information, thus re-starting the timestamp checking process. Acknowledgement messages contain the transaction's identification and the transaction's unique timestamp. In case of new timestamp information being sent (because of rejection), only acknowledgements with the transaction's current timestamp will be considered.

5. Within the RAT of a given node, the transaction is ordered by logical age with respect to transactions with conflicting needs (i.e. this will be called interference set). When the transaction has been fully acknowledged (i.e. acknowledgements with the proper timestamp have been received from all other nodes involved in the synchronization operation) and when it has become the one with the oldest timestamp among all the transactions with conflicting needs at that node, it is removed from the RAT of that node and its timestamp is used to update the timefield of all the relations it requested.

6. The synchronization operation terminates when the transaction has been removed from the RAT at all nodes which were included in the operation.

2.3 Specification of SPL

2.3.1 SPL data structures

2.3.1.1 Transaction notation

When dealing with a transaction T, the following notations will be used:

1. TS: timestamp of T which is unique throughout the system. This property is obtained by appending to the logical clock value given to T, the identification code (i.d.) of the originating site. [THOM78].
2. R(T): set of required relations which will be used by T either for read or write (or for both) purposes.
3. N(T): set of nodes which will be part of the operation. This set will be formed in the following way:
 - a. $N'(T)$ = all nodes with copies of relations in WS(T) (write set of T).
 - b. $RS'(T) = RS(T) - [RS(T) \cap WS(T)]$, with RS(T) being the read set of T.
 - c. $N(T) = N'(T) + \{\text{for each relation } ri \text{ in } RS'(T), \text{ one node with a copy of } ri \text{ if among all nodes with a copy of } ri \text{ none is in } N'(T)\}$.

2.3.1.2 Resource access table

The Resource Access Table (RAT) is a data structure used by the global co-ordinators when carrying out the synchronization algorithms. As specified formally in section A.2 of appendix A, the RAT is in fact a queueing table as well as a reservation table. It contains entries for all the relations in the system together with their timefields. In addition, it includes all the active transactions that require synchronization and that have requested relations residing at that particular site. The timestamp of those transactions is also stored in the RAT together with a list of the relations required by any individual transaction. In its role as a reservation table which will become evident in the description of the algorithms, the RAT also helps in reducing the number of rejections in the system.

For each transaction, a set of interfering transactions whose $R(T)$'s are intersecting is included in the RAT. This set, $INT(T)$, is necessary in order to create an ordering of events in the system.

2.3.1.3 Synchronization messages

Various messages are used by the SPL algorithms; for a

given transaction they are as follows:

1. Request message from originating GC to other GC's in $N(T)$: $RQS(RQS, T, TS, N(T), R(T))$.
2. Update message from originating GC to other GC's in $N(T)$ in response to a rejection of TS in original request: $NTS(NTS, T, TS)$.
3. Acknowledgement message, sent by GC's in $N(T)$ to all other GC's in $N(T)$: $ACK(ACK, T, TS)$.
4. Rejection message from GC('s) in $N(T)$ to originating GC when TS of T is unacceptable: $REJ(REJ, T, TS, ...)$.

2.3.2 Synchronization protocol

The algorithms will be introduced for the case of a transaction T with $R(T)$ and $N(T)$. The GC at the originating site will execute:

$S\text{-nodeSynch}(T, N(T), R(T))$

which is to be found in figure 2.2 and the GC's of other nodes in $N(T)$ will execute:

$D\text{-nodeSynch}\{\text{arguments supplied in message}\}$

which is to be found in figure 2.3. The algorithms of figures 2.2 and 2.3 have also been put in the context of an operational implementation. Such a description of the algorithms, in terms of RAT manager and submission manager (i.e. the functions of QC) is presented in appendix A.

2.3.2.1 Transaction acceptance test

In the SPL algorithms, a test on a transaction's timestamp is carried out when a request is initially received and when a new timestamp arrives in case of rejection. To perform the test, the algorithm compares TS with the timefield of each relation in $R(T)$. If TS is greater (*) than any of these, then TS is acceptable. If, on the contrary, TS is smaller than the timefield of, at least, one relation, TS is unacceptable since it is an indication that a transaction has already been successfully synchronized with a younger timestamp (in logical age) than that of the current transaction.

2.3.2.2 Compute interference function

The function "computeinterference" which is used to

(*) TS cannot be equal because the way the timestamps are constructed.

obtain the interference set $INT(T)$ performs two functions:

1. It generates $INT(T)$ which is a list of all other transactions currently being synchronized and with conflicting needs (i.e. overlapping $R(T)$'s).
2. It updates $INT(T_i)$, $\forall T_i | T_i \in INT(T)$, so that those other transactions T_i 's become aware of the existence of T .

2.3.3 Performance considerations

2.3.3.1 Messages and rejections

The number of messages required for such a synchronization is $(m^2 - m)$ in the no rejection case and $(2m^2 - (2+R)m + 2R)$ in the R node rejection case, with only one rejection phase (m is the number of nodes involved in T). In this regard, it is interesting to note that there are schemes whereby only one rejection phase is guaranteed. In other words, when the global co-ordinator at the site, where the request originated sends new timestamp information, full acknowledgements are guaranteed.

2.3.3.2 One rejection phase

One way to accomplish this goal is as follows: Given a transaction T , with $TS(T)$, originating at site i and affecting sites $j_1, j_2, \dots, j_m$, and that a global co-ordinator GC_j has found $TS(T)$ unacceptable and has thus rejected T .

The new method calls for GC_j to give T a new timestamp $TS(T)'$ such that T can be entered in the RAT of GC_j . GC_j would still send a rejection message but this time incorporating $TS(T)'$ as extra information.

When GC_i , the global co-ordinator at the originating site, has received messages from all GC_j 's, it then picks the highest $TS(T)'$ from the rejection messages it got (if any) and uses this new value in all the timestamp update messages it sends out. GC_i also updates its own RAT with $TS(T)'$.

Upon receiving the timestamp update message, the GC_j 's update their own RAT and accept T . The rest of the algorithm is then executed. As mentioned, the cost in messages of the algorithm is $(2m^2 - (2+R)m + 2R)$ for an R node rejection.

2.3.3.3 Message improvement

The algorithm however can be modified to operate with fewer messages but to the detriment of synchronization time. The algorithm in its present form requires two phases:

1. Broadcast by the originator GC_i of synchronization requests to all other GC_j 's in $N(T)$.
2. Broadcast by receiving GC_j 's of acceptance (or rejection) to all other GC_j 's in $N(T)$.

Phase 2 can be modified to give:

- 2a. Reply by receiving GC_j 's of acceptance (or rejection) to originating GC_i .
- 2b. Broadcast of go-ahead signals by the originating GC_i to all other GC_j 's in $N(T)$.

The cost of this modified algorithm is $3(m-1)$ messages in the no-rejection case. The modified algorithm can also be made to handle rejection in a very efficient manner which is similar to the improvement to the standard algorithm.

1. A receiving GC_j unable to enter T_1 in its RAT because of too old a $TS(T_1)$ would give it a new

TS(T1)' and enter it in its RAT. Obviously TS(T1)' is chosen so that the insertion of T1 in the RAT of the GCj is possible.

2. It would then send a rejection message for T1 with TS(T1)' to the originating GCi.
3. The originating GCi would choose among all the rejection messages the highest TS(T1)' and include this new timestamp TS(T1)' in its go-ahead messages to all other GCj's in N(T1). It would also modify the old TS(T1) with TS(T1)' in its own RAT.
4. Upon receiving the go-ahead message, a receiving GCj would take the new TS(T1)' and update its own RAT accordingly.

The modified algorithm can therefore be made to have only one rejection phase. Furthermore the cost in messages is the same as in the no-rejection case, that is $3(m-1)$. Briefly, to compare the case of one-node rejection, i.e. $R=1$, the standard algorithm requires $(2m^2-3m+2)$ messages as opposed to $3(m-1)$ messages for the modified version. The synchronization time is also better with the modified algorithm only if the standard algorithm suffers rejection. If the standard algorithm runs without rejection, it is

faster than the modified one. The no-rejection situation can be made to hold by judicious choice of timestamps and/or by keeping the logical clocks throughout the network as closely synchronized as possible. The choice of which algorithm to use then becomes dependent upon the type of distributed database dealt with.

2.3.3.4 Blocking and re-starts

Besides the number of messages required to achieve synchronization, there is another performance criterion to consider: the required time to achieve synchronization (abbreviated RTS). RTS depends on two factors:

1. Re-starts,

that is, the number of times a transaction will be rejected because its timestamp is too old.

Re-starts can be minimized by careful choice of TS, keeping TF's of relations in the RAT in as close a synchrony as possible with those of other RAT's, and adopting a one-rejection-only scheme.

2. Blocking,

that is, the amount of time a transaction is forced to wait in the RAT. This waiting time in the RAT can be made longer by transactions literally

squeezing in, at the last instant, ahead of a nearly fully acknowledged transaction. Since the RAT is also a reservation table, it does, on one hand, help reduce the amount of rejection but, on the other hand, may make possible long blocking waits, hence long RTS.

2.4 Validation of SPL

The next step in the presentation of SPL is the proof of correctness of the synchronization algorithms. The main criterion for correctness is that given a set of transactions $J = \{T_1, T_2, T_3, \dots, T_p\}$ executing in that order in a serial execution, a concurrent execution of J shall be taken as correct if, when all activity ceases in the database, the end result it produced is the same as in the serial execution case. In order to satisfy this criterion, several mechanisms, implemented in SPL, are necessary; they are presented as assertions. Besides serial correctness, the algorithms should also guarantee a deadlock free execution in, at least, the reliable case. Furthermore the protocol should be live, that is its execution should not have the possibility of getting to a trapping state.

Assertion 2.4.1

The synchronization algorithms run without deadlock.

Proof of Assertion 2.4.1

Assertion 2.4.1 involves two aspects of execution:

1. A transaction being synchronized by the algorithms of SPL will not run into systems deadlocks since all its needs have been pre-declared and requested in bulk. The proof of this statement can be found in operating systems textbooks such as [HANS73].

2. With respect to itself, the protocol (i.e. the algorithms and their interaction) does not end up in trapping states or in loops of infinite duration. This is readily seen from the Finite State Machine (FSM) representation of the S-node/D-nodeSynch algorithms which are shown in figures 2.4 and 2.5 respectively. There is, however, a loop in those two FSM's which represents the rejection mechanism. It could be possible that the S-node and its D-nodes engage in a continual exchange of update and rejection messages. This is not going to happen because: the GC's that have already accepted T with its then current TS are not going to reject it anymore, and the originating

GC can always, when faced with persistent rejections, choose an extremely high value of timestamp, thereby ensuring acceptance albeit to the expense of blocking for T and further re-starts for other T's.

Furthermore, a one-rejection-only scheme can be adopted, bypassing the problem entirely. In this connection, it should also be said that the frequency of rejection observed by a given GC is dependent on the level of distributed activity involving, on one hand, the DR's at a given node and, on the other hand, those same DR's at other nodes (for read purposes only since a write would force them all to have the same TF). In a reasonably busy system, but still far away from saturation, rejections would not occur repeatedly.

Assertion 2.4.2

The timestamp $TS(T)$ of a transaction T originating at node i is unique throughout the system.

Proof of Assertion 2.4.2

The uniqueness of $TS(T)$ has to be proved along two lines:

1. Within a node itself, the uniqueness of $TS(T)$ is guaranteed by advancing the logical clock after each significant event.
2. Within the network, the uniqueness of $TS(T)$ is guaranteed by appending the unique processor id to $TS(T)$ in the following fashion:

$$TS(T) := \{\text{Logical clock @ } i : i\}$$

Assertion 2.4.3

The database changes dynamically. In other words, when, at site ns , the TS of a transaction is tested, it is so tested against a logical clock made out of

$$\text{MAX } \{TF(r_i), \forall r_i \mid r_i \in R(T) \text{ and } r_i \in R(ns)\}, (*)$$

and therefore, for the system to function properly, this logical clock changes dynamically.

Proof of Assertion 2.4.3

The database changes dynamically because, when a transaction T is removed from the RAT of site ns ,

$$TF(r_i) := TS(T), \forall r_i \mid r_i \in R(T) \cap R(ns).$$

Furthermore, if the same mechanism is applied at all n_j 's,

(*) $R(ns)$ = relations found at ns

$\forall n_j \in N(T)$, the DDB with distributed relations will indeed change dynamically.

Assertion 2.4.4

An ordering of events exists within each node and is mutually consistent with that of other nodes.

Proof of Assertion 2.4.4

Within each node there exist two streams of activity:

1. Local actions

Local actions within a node are totally ordered by the simple fact that, no distribution being present, the local clock is an absolute reference.

2. Distributed actions

Distributed activity is the most important part of this assertion. Given a set $\mathcal{T} = \{T_1, T_2, T_3, \dots, T_P\}$ of transactions with a function

$$\tau: \mathcal{T} \rightarrow TS$$

where $TS = \{t_1, t_2, t_3, \dots\}$

with t_i being a logical instant,

an ordering is possible since

$$TS(T_i) \neq TS(T_j),$$

$$\forall i \text{ and } \forall j \mid i \neq j \text{ and } T_i \in \mathcal{T}, T_j \in \mathcal{T}.$$

An ordering function O can therefore exist such that

$O: \mathcal{T} \rightarrow \mathcal{T}'$ such that for all T_i and T_j in $\mathcal{T}'$,

$T_i \rightarrow T_j$ if $TS(T_i) < TS(T_j)$

where $\rightarrow$ means "precede".

$\mathcal{T}'$ is always unique since individual TS 's are unique.

In the fully redundant case $\mathcal{T}$ is the same at all nodes in the network. However, if the database is not fully redundant there is still a global set $\mathcal{T} = \{T_1, T_2, T_3, \dots, T_P\}$ of transactions but this time $\mathcal{T}$ is partitioned into $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \dots$ corresponding to nodes $n_1, n_2, n_3, \dots$. The global set $\mathcal{T}$ is now equal to:

$$\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_3 \cup \dots$$

Clearly, $\mathcal{T}_i$ is not necessarily equal to $\mathcal{T}_j$ and the sequence of transaction processing is not exactly the same at n_i and n_j . However, because $\mathcal{T}$ was subjected to O , the sets $\mathcal{T}_1, \mathcal{T}_2, \dots$ will also be producing the ordered sets $\mathcal{T}_1', \mathcal{T}_2', \dots$. The ordering of transactions in those sets is strictly by increasing timestamp order, hence consistent ordering among all sites.

Finally the ordering produced is guaranteed to be the same throughout the network by the application of one of the rules of exit from the RAT which says that for a transaction T , with $TS(T)$, T has to be fully acknowledged (*) before

(*) There are two rules to satisfy in order to exit from the RAT. See proof of assertion 2.4.5 for the other.

being removed from the RAT, hence before updating the TF's of the relations in $R(T)$.

Assertion 2.4.5

The database accessed by a given transaction T , with $TS(T)$, is unique to T .

Proof of Assertion 2.4.5

The actions of a transaction in a DB system can be viewed as:

$$T: RS(T) \rightarrow WS(T),$$

where $RS(T)$ and $WS(T)$ are respectively the read-set and write-set of T . The action of going from $RS(T)$ to $WS(T)$ is done in the same logical instant. That is the reason why locks are used on the relations so that physical time can equal logical time.

For T to be entered in the RAT,

$$TS(T) < \max \{TF(r_i), \forall r_i | r_i \in R(T)\},$$

where $R(T) = RS(T) \cup WS(T)$. In the RAT that condition is preserved even though insertion of other T_i 's, with $TS(T_i) < TS(T)$, can take place. $TS(T_i)$ would of course have to satisfy the entrance conditions into the RAT because, if $TS(T_i)$ failed the test, it would mean that T_i cannot be

guaranteed access to an $R(T_i)$ of suitable logical age, everywhere in $N(T_i)$.

Any insertion in the RAT would preserve the ordering of assertion 2.4.4 because, for T with $TS(T)$, the following rule of exit from the RAT applies:

$$TS(T) = \min \{TS(T_i), \forall T_i | T_i \in \text{INT}(T)\},$$

where $T_i \in \text{INT}(T)$ if $R(T_i) \cap R(T) \neq \emptyset$.

Once a transaction leaves the RAT it updates the timefields of the relations it used, so as to ensure dynamic changes of the database (assertion 2.4.3). The action of leaving the RAT moves the entrance condition along an imaginary sequence of action. The database seen by T is therefore unique since when T leaves the RAT, the relations in $R(T)$ are now of logical age $TS(T)$.

4.

2.5 Robustness in SPL

In any system there always exists, however small, a finite non-zero probability of failure. The concurrency control should be able, to a certain degree, to cope with events such as:

1. loss of synchronization messages,

2. crash of individual processes,
3. crash of concurrency control mechanisms at one or more sites in the network.

The result of such failures translates in a transaction jam in the RAT. This jam is produced because transactions are linked through their interference sets and, clearly, such situations have to be protected against. The approach to robustness in SPL will borrow some of its ideas from the Commit Protocol Layer (CPL) of chapter 4 (*), With the main features of a robust SPL being:

1. Use of diagnostics from the Communication Subsystem (CSS). When a GC gives a message to CSS to deliver, CSS, being the interface to the network, can tell the GC whether or not the message was delivered successfully (see chapter 4, section 4.3.1).
2. Use of stop messages which would be sent to stop the synchronization operation, upon the detection of a failure (see chapter 4, section 4.2.2.2).
3. Use of a timer protocol to abort the

(*) The reader is referenced to chapter 4 for a more complete description of robust mechanisms.

synchronization operation. This timer protocol is based on mutual suspicion and assumes that if no positive acknowledgement of a partner's presence is received, that partner is deemed to have failed and the current transaction is aborted.

At this point, it is interesting to compare the SPL robustness approach to that of CPL. In SPL, a failure may create a jam in the RAT but does not introduce inconsistencies in the database as a failure might do during an update operation in CPL. Consequently, the strategy used in SPL calls for aborting faulty or crippled transactions. The case of some sites in $N(T)$ not being aware of the abort is not critical since such a situation will be detected in the Execute Protocol Layer (EPL) of chapter 3.

In CPL, on the other hand, an update operation or its abort have to be completely successful, thus entailing use of costly mechanisms which are not necessary in SPL.

2.6 Specification of Robust SPL

The previous algorithms will now be presented with extra features to make them failure resistant. The S-nodeSynch and the D-nodeSynch can be found in figures 2.6

and 2.7 respectively with the procedures "Time-out", "Processmessage", and "Newsynch" in figures 2.8, 2.9 and 2.10(a and b). The comments of section 2.4 still apply to this section.

In connection with the use of stop messages to stop the synchronization operation, it should be mentioned that, in order to be able to receive stop messages, the protocol calls for an extra wait after the conditions for the removal of a transaction have been satisfied. This corresponds to "wait for extra time delay" in the algorithms.

2.7 Validation of Robust SPL

The algorithms specified in section 2.6 are basically those of section 2.3 enhanced with special features. Therefore, this validation will only be concerned with the new features, namely: the liveness of the augmented protocol and the consistency of its abort.

Assertion 2.7.1

The robust SPL algorithms are free from deadlock (i.e. live execution).

Proof of Assertion 2.7.1

The FSM representation of S-nodeSynch and D-nodeSynch is shown in figures 2.11 and 2.12. By comparing them with those of figures 2.4 and 2.5, it is readily seen that states that could be trapping states in the event of failures, are protected by time-out mechanisms in the new figures to ensure that the protocol will always be live.

Assertion 2.7.2

The abort mechanism provided by SPL is consistent, that is to say will abort the transaction at all sites of $N(T)$. In the event of a decision to continue, the re-grouping of the GC's in $N(T)$ will also be consistent.

Proof of Assertion 2.7.2

Assuming that the timer mechanisms work properly, it is easy to see that, in the FSM's of figures 2.11 and 2.12, any failure will cause a time-out at sites where it is detected. The first action of any GC when it times out, is to stop the algorithms through the use of stop messages. At this point, $N(T)$ will be partitioned into the nodes that can be reached and those that cannot (either they failed or they are part of their own subnetwork(s)).

1. For the nodes that can be reached, the procedure "Newsynch" is executed to determine whether the S-node GC is still in the subnetwork. If it still is, it will send new request messages if, according to it, the transaction can still go ahead. If the S-node GC is not in the subnetwork, a second time-out will occur (i.e. in the procedure "Newsynch") upon which the transaction will be aborted in the subnetwork.
2. For the nodes that cannot be reached, they may or may not have received any stop messages. If they did, the previous case applies. If they did not then, when the transaction processes at those sites try to form the virtual ring structure of EPL of chapter 3, the TP's will detect the anomalous situation and abort themselves.

2.8 Interaction of SPL/EPL/CPL

In later chapters (3,4,5), some robust mechanisms will be developed in order to ensure reliable transaction execution in the case of EPL and reliable committing of updates in the case of CPL. The fact that those mechanisms have not been introduced creates an awkward situation

regarding the interaction of various layers. To the mechanisms of SPL, those of QPL appear only as an up-list (Network Access Table (NAT)) mechanism together with status indicators (i.e. MAJ= majority, MIN= minority, INC=inconsistent) associated with each relation at a given node. The up-list mechanism is also a form of status indication for nodes (i.e. up, down, merging) in consequence of which, the following restrictions are imposed on the original request submitted by STP to GC via the DRQ.

1. When QPP, in conjunction with STP, formats the request, all the nodes in $N(T)$ should be up in the NAT of chapter 4. In fact, should some of the nodes be down, the operation would eventually be aborted without any chance of success.

2. At the STP node, (say n_k), the following relation has to hold:

$$\forall r_i \mid r_i \in R(T) \cap R(n_k), r_i, \text{status} = \text{MAJ.}$$

The status indicators for those relations are in the Release and Update Table (RUT) described in chapter 4.

3. STP can check the status of the relations at its own node, but cannot do likewise for other nodes in $N(T)$. Therefore, $\forall TP \mid TP \text{ resides at } n_j \text{ and}$

$n_j \in N(T)$; the following has to hold:

$$\forall r_i \mid r_i \in R(T) \cap R(n_j), r_i.\text{status} = \text{MAJ}.$$

Clearly, if the above test proves negative, some mechanisms have to be in place to notify the originating GC that the request is not acceptable as formatted.

Readers already familiar with the strategy of CPL will see that STP can check the status a relation should have, with the help of the NAT and system catalogues. This status is only an approximation in the sense that a relation could be MAJ or a mix of MIN and INC. The main concern of STP is that all relations in $R(T)$ should have this potential MAJ status. If, at some other sites in $N(T)$, the test of item 3 fails (i.e. some $r.\text{status}$ are MIN or INC), two courses of action can be envisaged:

1. The site where the test failed could simply ignore the request, thus causing its eventual abort. Furthermore, the QPP in charge of that particular request may want to alter its strategy with respect to $N(T)$ and submit a new request.
2. Rejection messages could be expanded and the algorithms modified so that the transaction could be put on hold while the QPP tries to re-configure

the request (if at all possible) and then sends updated timestamps in case of success.

Various other possibilities could no doubt be considered but it is important to emphasize that the two already mentioned do not use a NATupdate procedure (chapter 4) to remove the nodes which could not participate fully. It was felt that such an expensive alternative should only be used in CPL where either complete success or complete abort are mandatory.

Figure 2.1: SPL access graph

```

S-nodeSynch (T: transaction; TN: target nodes;
              TR: target relations)

begin
  TS := gettimestamp (T)
  {TS is chosen so that it is greater than the
   timefield of any relation in TN (i.e. R(T))}
  insert (T, RAT)
  INT(T) := computeinterference (T, RAT)
  {INT(T) should contain all the transactions with
   conflicting requirements}
  RQS := (RQS, T, TS, TN, TR)
  send (RQS, TN)
  repeat
    msg := receivemessage
    case msg of
      ACK: begin
        if ACK.TS = TS
          {does this ACK correspond to
           the current transaction}
        then
          ACKcount := ACKcount + 1
          if ACKcount = |TN| - 1
            then
              fullyACKed := true
            fi
          fi {ACK.TS <> TS
             does not mean anything}
        end
      REJ: begin
        TS := newtimestamp (T)
        update (TS, RAT) {new TS}
        NTS := (NTS, T, TS)
        send (NTS, TN)
        ACKcount := 0
      end
    endcase
  until fullyACKed = true
  wait until T is oldest in INT(T)
  update (TR, TS)
  {that is, update the timefields of
   the relations in TR}
  remove (T, RAT)
  remove (T, other INT sets)
end

```

Figure 2.2: S-nodeSynch procedure

D-nodeSynch {arguments supplied in RQS message}

```

begin
  submission := receivemessage
  with submission do
    if TS is acceptable
      {for TS to be acceptable, TS should be
       greater than the timefield of any
       relation in TN (i.e. R(T))}
    then
      insert (T, RAT)
      INT(T) := computeinterference(T, RAT)
      {INT(T) should contain all transactions
       in local RAT with conflicting needs;
       .INT(T) is obviously dependent on the
       site where it is evaluated}
      ACK := (ACK, T, TS)
      send (ACK, TN)
    else
      REJ := (REJ, T, TS, ...)
      send (REJ, S-node)
    fi
  repeat
    mssg := receivemessage
    case mssg of

      NTS: begin
        if NTS.TS is acceptable
          then
            if NTS.T is not in RAT
              then
                insert (T, RAT)
                INT(T) :=
                  computeinterference(T, RAT)
              fi
            ACK := (ACK, T, NTS.TS)
            send (ACK, TN)
            ACKcount := 0
            update (NTS.TS, RAT)
          else
            REJ := (REJ, T, NTS.TS, ...)
            send (REJ, S-node)
          fi
        end
      NTS:
    fi
  end

```

```

ACK: begin
      if ACK.TS = TS
      then
        ACKcount := ACKcount + 1
        if ACKcount = |TN| - 2
          {i.e. minus S-node
            and oneself}
          then
            fullyACKed := true
          fi
        fi
      end

      endcase
      until fullyACKed = true
      wait until T is the oldest in INT(T)
      update (TR, TS)
        {that is, update the timefield
          of the relations in TR}
      remove (T, RAT)
      remove (T, from other INT sets)
    endwith
end

```

Figure 2.3: D-nodeSynch procedure

Figure 2.4: FSM of S-nodeSynch procedure

Figure 2.5: FSM of D-nodeSynch procedure

S-nodeSynch (T:transaction; TN:target node;
TR:target relations)

begin

TS:= gettimestamp (T)
insert (T, RAT)
INT(T):= computeinterference (T, RAT)
RQS:= (RQS, T, TS, TN, TR)
send (RQS, TN)
set timer
if all RQS's properly delivered
then

repeat

mssg:= receivemessage
case mssg of

ACK: begin

if ACK.TS= TS
then

ACKcount:= ACKcount + 1

if ACKcount= |TN| - 1

then

fullyACKed:= true

fi

fi

end

REJ: begin

disable time-outs

{to optimize the performance of
the algorithm, S-node could wait for
all possible rejections}

TS:= newtimestamp (T)

update (TS, RAT) {new TS}

NTS:= (NTS, T, TS)

send (NTS, TN)

if all NTS's properly
delivered

then

set timer

ACKcount:= 0

else

abort:= true

SN:= nodes to which
message could not
be delivered

stop:= (stop, T, TS, SN)

send (stop, TN)

newsynch (T, TN, TR)

fi

end

```

      stop:begin
        disable time-outs
        abort:= true
        newsynch (T, TN, TR)
      end
    endcase
  until fullyACKed= true
    or abort= true
  else {some RQS's not properly delivered}
    disable time-outs
    SN:= nodes to which message
      could not be delivered
    stop:= (stop, T, TS, SN)
    send (stop, TN)
    newsynch (T, TN, TR)
  fi
  if fullyACKed= true
  then
    wait for extra time delay
      {in case some stops arrive}
    if any stop(s) received
    then
      newsynch (T, TN, TR)
    else
      wait until T is oldest in INT(T)
      update (TR, TS)
      remove (T; RAT)
      remove (T, other INT sets)
    fi
  fi
end

```

Figure 2.6: Robust S-nodeSynch procedure

D-nodeSynch {arguments supplied in RQS message}

begin

submission := receivemessage

with message do

if TS is acceptable

then

 insert (T, RAT)

 INT(T) := computeinterference(T, RAT)

 ACK := (ACK, T, TS)

 send (ACK, TN)

if all ACK's properly delivered

then

 set timer

 processmessage

else

 {some ACK's not properly

 delivered}

 disable time-outs

 SN := nodes to which messages

 could not be delivered

 stop := (stop, T, TS, SN)

 send (stop, TN)

 newsynch (T, TN, TR)

fi

else {TS is not acceptable}

 REJ := (REJ, T, TS, ...)

 send (REJ, S-node)

if REJ properly delivered

then

 set timer

 processmessage

else

 disable time-outs

 SN := nodes to which message

 could not be delivered

 stop := (stop, T, TS, SN)

 send (stop, TN)

 newsynch (T, TN, TR)

fi

fi

```

if fullyACKed=true
  {fullyACKed will be set by the
   procedure processmessage}
then
  wait for extra time delay
  {in case stop(s) arrive}
  if any stop(s) received
    then
      newsynch (T, TN, TR)
    else
      wait until T is oldest in INT(T)
      update (TR, TS)
      remove (T, RAT)
      remove (T, from other INT.sets)
    fi
  fi
endwith
end

```

Figure 2.7: Robust D-nodeSynch procedure


```
Procedure Time-out
{during either S-nodeSynch or D-nodeSynch
 and corresponding to "set timer" instruction}

begin
  SN:= nodes from which ACK
      did not arrive
  stop:= (stop, T, TS, SN)
  send (stop, TN)
  newsynch (T, TN, TR)
end
```

Figure 2.8: Time-out procedure
for S-node/D-nodeSynch

procedure processmessage

begin

repeat

mssg:= receivemessage

case mssg of

begin

disable time-outs

if NTS.TS is acceptable

then

if NTS.T is not in RAT

then

insert (NTS.T, RAT)

INT(T):=

computeinterference(T,RAT)

fi

update (NTS.TS, RAT)

{i.e. new TS}

ACK:= (ACK, T, NTS.TS)

send (ACK, TN)

if all ACK's properly
delivered

then

set timer

ACKcount:= 0

else

SN:= nodes to which message
could not be delivered

stop:= (stop, T, TS, SN)

send (stop, TN)

newsynch (T, TN, TR)

fi

else

{NTS.TS is not acceptable}

REJ:= (REJ, T, NTS.TS,...)

send (REJ, S-node)

if REJ properly delivered

then

set timer

ACKcount:= 0

else

abort:= true

remove (T, from system)

{i.e. abort T}

fi

fi

end

2

```

ACK: begin
    if ACK.TS = TS
    then
        ACKcount := ACKcount + 1
        if ACKcount = TN - 2
        then
            fullyACKed := true
        fi
    fi
    end

    stop: begin
        disable time-outs
        abort := true
        newsynch(T, TN, TR)
    end

    endcase
until fullyACKed = true
    or abort = true
end {of procedure processmessage}

```

Figure 2.9: Procedure processmessage

Procedure newsynch (T:transaction;
TN:target node; TR:target relation)

```

begin
  put T on hold
  wait for all possible stops
  if S-node
  then
    SN:= union of all SN's in stops
    NAT*:= NAT - SN
    test:= operationpossible(T,TN,TR,NAT*,...)
    {function operationpossible is in
     figure 2.10b}
    if TEST.check= true
    then
      S-nodeSynch (T,TEST.TN,TR)
    else
      remove (T, from system)
    fi
  else
    set timer
    wait for beginning of interaction
    D-nodeSynch
  fi
end {end of newsynch}

```

Procedure time-out {for newsynch}

```

begin
  remove (T, from system)
end

```

Figure 2.10a: Newsynch and time-out procedure

```

Function operationpossible (T: transaction;
    TN: target node; TR: target relation;
    NAT*: NAT; ...): TEST
    {TEST= record
        check: boolean;
        TN: target node
        end;}

begin
    TEST.check:= true
    repeat
        r:= pickarelacion (TR)
        if r.status <> MAJ in NAT*
        then
            TEST.check:= false
        fi
    until all r's in TR have been processed
        or TEST.check= false
    if TEST.check= true
    then
        TEST.TN:= rebuildTN (TR, NAT*)
        {all available copies of r's in write set
        and one copy of r's in read set but not
        write set}
    fi
end

```

Figure 2,10b: Function operationpossible

Figure 2.11: FSM of robust 3-nodeSynch procedure

Figure 2.12: Full of robust D-nodeSynch procedure

CHAPTER 3

Execute Protocol Layer

3.1 Introduction

3.1.1 State of the system at the entry point to EPL

Special protocol layers, such as SPL and later CPL, are introduced to ensure that transaction execution is correct and robust. So far, chapter 2 has described a synchronization protocol which guarantees correctness of operations for distributed transactions (*).

When a transaction, viewed as a distributed entity, comes out of SPL, it consists of a set of Transaction Processes (TP), waiting to execute at each site in $N(T)$. One of those TP's is the Supervisory Transaction Process (STP) whose functions are to carry out the wishes of the

(*) Throughout this thesis, transaction and distributed transaction (DT) are synonymous. When a transaction is purely local, it will be specified as such.

user (or originator of the transaction) as instructed by the Query Processing Process (QPP). The various aspects of query processing strategy are not of concern here and a treatment of the topic, in the context of the ADD architecture, can be found in [TOTH80].

Assuming the existence of a suitable strategy for co-ordinating processes, the STP will instruct the TP's as to the proper course of action. The main goal of a transaction is to produce a target update (TU) either in the form of a read-only transaction producing a TU which will never be installed in the DB or in the form of a read-write or write-only transaction producing a new updated version of some data items.

In order to achieve this goal, some important steps have to be followed. They are summarized below:

1. Upon being given the locks on the local relations that the transaction requested, the TP (or STP) reads the relations in the read-set of T and stores them in a separate workspace. Then, relations which were requested for read-only purposes can be released. This early release is not entirely necessary since those read-only relations can be released at the end of T, with all the other

relations in R(T). An early release does improve the performance of the system by possibly allowing other transactions to run.

2. All the TP's under the STP's supervision perform processing of the information contained in their private workspaces. Some TP's may be instructed to send the contents of their workspace to some other TP's. In such a case, those TP's (i.e. the ones who sent their workspaces) may end their execution or, if some target relations reside at their node, wait for either some other interaction or the target updates.
3. Eventually target updates will be produced and the STP will make sure that the TU's (or relevant portions of the TU's) are sent to TP's at nodes with target relations.
4. Finally, the STP triggers entry into the CPL of chapter 4 so that the TU's will be entered in the database.

In this chapter, the transaction and its TP's will be viewed as:

$$T = \{STP, TP1, TP2, TP3, \dots\} (*)$$

with the aim of T as being the production of TU's and with the TP's performing three important actions: send, receive and process information in workspace.

3.1.2 Problem definition

Failures can disrupt the distributed activities of a transaction in several ways. The results end up being the failure of either STP or any of the TP's, thus preventing the completion of the transaction's execution. Without proper protection, the components of a damaged transaction would simply wait indefinitely while holding the locks on the relations of their write-set and thereby preventing other transactions from executing. Furthermore, even if the transaction has been aborted, the failed components still have to be cleaned up, that is, their relations have to be released and their allocated resources given back to their local executives. Decisions regarding distributed activities cannot be made on a purely local basis. Distributed mechanisms to oversee the distributed execution of transactions have to be in place if the system is not to

(*) Such an arrangement is also known as a cohort [GRAY78].

be unduly affected by failures.

Failures of local transactions also have harmful effects on both local and distributed transactions. A failed local transaction will not release its write-set relations, thus blocking the execution of others. A clean abortion of such transactions does not present any problem, but for the sake of simplicity, the monitoring of both local and distributed transactions will be entrusted to a single entity called the Activity Monitoring Process (AMP) to be described in section 3.2.

3.1.3 Literature survey

The topic of protecting independently scheduled co-operating processes has not received, so far, wide attention in the context of both general multi-processor systems and distributed databases.

The reliability mechanisms of SDD-1 [HAMM80] provide for site monitoring. This monitoring is a lower level service and individual components of a transaction can put a "watch" on a given site (i.e. either as it fails or recovers). However, in SDD-1, sites that are recovering are assumed to know of their previous failure.

Besides site monitoring, SDD-1 also offers a complex arrangement of back-up processes so that if the process in charge of committing the updates (through the use of a two-phase commit protocol [LAMPSxx], [GRAY78], [STUR80]) fails, then one of the back-ups would take over. The number of back-up processes determines how many failures the system can tolerate.

Back-up processes are not provided in EPL. It was felt that a failure of STP would be severe enough to justify aborting the transaction. The CPL of chapter 4, however, does explicitly provide for back-ups.

Other works, such as [GRAY78], concentrated on the co-ordination aspects of co-operating processes. The possibility of failure was mentioned but the mechanisms to detect such a failure and to take care of it were not discussed.

3.1.4 Characteristics of EPL

The Execute Protocol Layer (EPL) described in this chapter will provide failure protection to the DDB. This is achieved by providing for individual monitoring of TP's on a per transaction basis. This monitoring will cover:

1. The time period during which the TP's are waiting for the go-ahead signal from the DataBase Manager (DBM) in the form of a signal operation on the TP's private semaphore.
2. The start of each TP of a given transaction is asynchronous with the starts of other TP's of the same T and that period of relative inactivity for a TP will also be monitored.
3. The execute portion of T is very much application dependent. Nevertheless, the monitoring should continue up until the successful transition into CPL. The consistency of this transition should, of course, be guaranteed by EPL.

3.2 Description of EPL

3.2.1 EPL structure

The access graph of figure 3.1 shows the components necessary for the proper functioning of EPL. Besides the transaction process (TP or STP) which is familiar by now, figure 3.1 contains the Activity Monitoring Process (AMP), the Transaction Process Monitoring (TPM) table and the

Target Update Directory (TUD).

The AMP is the entity which, at a particular site, is responsible for providing monitoring to all TP's which have requested it. The monitoring and failure protection of AMP is also given on a per transaction basis.

To perform its tasks, the AMP uses three main data structures, namely: the TPM which is a table containing i.d.'s of TP's under monitoring, the TUD which is a directory containing pointers to TU's and finally the Process Monitoring Request Queue (PMRQ) which is the means by which TP's and AMP can communicate.

3.2.2 EPL strategy

The basic approach of EPL is better expressed on a global level; a virtual ring (VR) is set up for each Distributed Transaction (DT) in the system and this virtual ring is initially made up of sites where the transaction will be active. A monitoring message is initially issued by the Activity Monitoring Process (AMP) local to the site where the transaction originated and this message is kept circulating on the virtual ring as a sign that all the transaction processes of a given transaction are functioning properly. This approach is illustrated in figure 3.2.

Failure detection is carried out locally by each AMP on

the component of DT local to it and on the other TP's as well. This failure detection has three aspects:

1. The AMP interacts with the site's local Operating System (O/S) with the former being notified by the latter in case of failure of TP's. Failures that can be so detected include: TP failures (i.e. overflow, underflow, coding errors, etc.,) caught by the O/S since it had to abort the TP, corrupted memory words, privilege violations, etc., again forcing the O/S to abort the TP.
2. A TP performing certain tasks such as send, receive, etc., can itself perform error detection on the operations. An example of this could be a TP sending a message and being told by the communication subsystem (CSS) that the message cannot be transmitted successfully. Upon detecting such failure, the TP will interact with AMP in order to have AMP perform an abort against DT.
3. Other TP's and by extension the sites they are running at, are also monitored by an AMP since, failure(s) of any remote components (i.e. remote TP, sites, or other services) will cause a break in VR monitoring. Such a break triggers a time-out of

AMP(s) which in turn causes DT to be aborted.

A consistent distributed abort is provided by AMP's and is accomplished by breaking the VR structure surrounding the DT in question. Other AMP's on that particular VR will time-out on the missing monitoring message and will abort the local component (i.e. TP) of the DT. The mutual consistency of the distributed abort is thus based on mutual monitoring among AMP's (*).

Besides providing for failure detection, the VR structure lends itself well to an orderly transition into CPL. This is achieved by allowing the VR to shrink as TP's at nodes in N(T) finish their work. When a TP which will not participate in updating has finished its work as per instructed by STP, it simply removes itself from the system and also instructs its local AMP that it should disappear from the VR. The AMP accomplishes this VR shrinkage by deleting the node from the routing list in the monitoring message of a given transaction. Similar action will take place for TP's at sites where updating will take place with the difference that such a removal from VR will be requested by a TP only after it has successfully received the TU's and

(*) Remote TP and remote site monitoring and distributed abortion seem to use the same mechanisms. They are, however, conceptually different.

inserted them in TUD. This method of VR shrinkage provides convenient means by which STP can ascertain if the DT has completed its execution. When the size of the routing list of a VR is 1 (i.e. the STP is the only one left) it basically tells the STP that the TU's are in place at all target sites and that STP can now effect entry into CPL by queueing a request in the DURQ. After having done that, the STP can then remove itself from the transaction process monitoring table of the activity monitoring process.

The STP is protected until it triggers entry into CPL although that is not the case for any other TP's. In fact, when TP's have stored the TU's and have entered them into TUD, they will remove themselves from the VR monitoring with the transaction still being in the execute phase. The transition from EPL to CPL is not instantaneous for TP's and they have no guarantee that the STP will not fail before it can request entry into CPL. For a given transaction, the locks on the relations in its write set will only be released at the end of CPL. Consequently, a failure of STP in the above circumstances would prevent other users from accessing those locked relations. To protect against this possibility, a timer is attached to each entry in TUD (one entry per transaction) so that failure to enter CPL within a reasonable delay would cause a time-out procedure to execute. The purposes of this time-out procedure would be to delete the TU's and to release the locked relations. In

other words, it would be the protracted abortion of a distributed transaction which locally completed its execution a long time ago.

3.2.3 Observations on EPL strategy

Some characteristics of the proposed EPL strategy should be mentioned:

1. AMP(s) and VR structure remove the burden of failure detection and recovery from the TP's (or in fact the DT structure) and instead provide reliability as a system service.
2. As a result of this system service, overhead can be minimized in two ways. On one hand, the number of messages per site (i.e. for all the TP's of a site) can be minimized by grouping them according to destination. On the other hand, monitoring messages, because of their small sizes, can be appended to existing ones without adverse effects.
3. Reliability during the execution of a transaction can be parametrized as a function of the speed of revolution of the monitoring message through the VR. A faster revolution rate would give smaller

response time when failures occur.

4. The actual number of messages is difficult to evaluate. It depends on factors such as the duration of the transaction, the average message transit delay and the number of nodes in $N(T)$. The number of messages will therefore vary widely from one transaction to another.

5. Finally, the virtual ring concept in which all processes are equal affords greater ease of use than a master/slave arrangement. In the latter scheme, the master monitors the slaves but the slaves have, in turn, to monitor the master. The hierarchy of the master/slave arrangement restricts the contacts of slaves among themselves for monitoring purposes. In fact, the guarantee that the group of processes is functioning properly will come from the master. Consequently, if the master fails, a slave has to be elected to become master so as to provide this guarantee and to carry out the master's control functions. The virtual ring approach bypasses that difficulty and, as a by product, reduces the number of monitoring messages.

3.3 Specification of EPL

3.3.1 Introduction to specification

The algorithms of chapter were introduced as a set of S-node (for initiating node) and D-node (for participants) algorithms. This chapter requires a different approach because of the very loose nature of the application and also, partly because EPL is mostly co-operation among TP's and AMP's. For that purpose, the specification of EPL will look at processes and the co-ordination of their activities through queueing of messages in appropriate queues.

3.3.2 EPL data structures

In their operations, the components of EPL make use of various control data structures: TPM, TUD, monitoring message and submission into PMRQ, which are shown in pseudo-Pascal in figure 3.3.

3.3.3 EPL processes

The two processes of importance, namely the AMP and the TP (execute phase) are shown in pseudo-code in figures 3.4 and 3.5 respectively.

3.4 Validation of EPL

Validation of EPL cannot be as thorough as that of SPL or CPL since EPL is not entirely a system layer but, rather, a mixture of system services and application dependent tasks. However, the following assertions will prove that EPL guarantees protection against process/processor failure(s), ensures consistent TP abort and provides for consistent transition into CPL. The only assumption made in this section is that the AMP, due to its relatively simple nature, is not going to crash unless the site where it is running goes down.

Assertion 3.4.1

EPL will protect a distributed transaction (DT) against failure of one or more of its components, whether it be failure of a TP or failure of the site where the TP executed.

Proof of Assertion 3.4.1

The validation of EPL protection mechanisms goes along two lines:

1. Protection against TP failure(s).

The DT is, as a whole, protected against failures of individual components by the AMP which gets its diagnostics from the O/S (*) in case of TP failure and also from the TP's themselves, should any action initiated by a TP fail. Under such conditions, the AMP would, in effect, break the VR monitoring of a given DT which will cause the DT to be aborted (see case 2).

2. Protection against processor failures.

A processor failure entails the failure of all processes running at that site, including the AMP. With the AMP down, a break in the VR monitoring of a DT will occur. This break will, in time, cause a time-out procedure to be executed by the other AMP's involved in that particular VR (**). The fact that other AMP's in the VR structure will execute an exception procedure ensures detection of the failure(s).

(*) There obviously has to be some means of communication between AMP and the O/S regarding the monitoring of TP's.

(**) A given AMP will, of course, be involved in more than one VR.

Assertion 3.4.2

In case of TP or site failure, a DT will be aborted successfully at all other sites where it had a TP component.

Proof of Assertion 3.4.2

In the event of a failure causing a break in the virtual ring monitoring, the AMP's, at the sites that are still up, will time out and abort the component of the DT which is local to them.

As far as the site(s) that is (are) down is (are) concerned, the mechanisms of CPL (chapter 4) will ensure eventual detection and merge into the rest of the system. Even without detection the recovery would take place at a later time so that the recovering AMP would then notice the VR break and abort TP (or what is left of it).

Assertion 3.4.3

For a given DT, the mechanisms of EPL guarantee that when the AMP local to STP gives STP the go-ahead signal to enter CPL, all the TU's are in TUD at all target sites. Furthermore, if STP fails before entering CPL, the system will abort the corresponding DT.

Proof of Assertion 3.4.3

As far as STP is concerned, a given TPI has two ways of properly terminating its execution:

1. TPI performs read, processes information and then, under STP's directives, sends the results somewhere else. After performing those actions, TPI would enter a request for removal from the VR monitoring.
2. TPI could perform all of the actions mentioned previously and also, still under STP's directives, store the target updates (TU) in the target update directory (TUD). If all the relations affected by TU are all of the proper status, then TPI would enter a request to be removed from VR monitoring.

In either case, the local AMP will wait to get a last monitoring message. Upon receiving it, the local AMP will remove itself from the routing list in the VR message. If the last message is not coming, thus indicating a failure somewhere else, the local AMP will abort TPI and, in case 2, delete the corresponding TU's in TUD.

If no failure occurs, then the size of the VR will

shrink to 1 as all the TP's finish their work. When the size of the VR is 1, it is a definite sign that all the necessary TU's are in TUD of target sites; the AMP local to STP advises the latter accordingly.

If failures occur, the following two cases have to be considered:

1. STP's own failure which could occur:

a. before STP gets a signal from AMP that EPL phase has ended. In this case, the AMP local to STP will abort STP locally and will also break the VR monitoring. The corresponding TP's will then be aborted as well. It may be, however, that some TP's had already put TU's in their local TUD and were removed from the VR monitoring. In such a case, the timer in the TU entry in those TUD's would eventually run out, thereby causing the removal of those TU's and the release of the corresponding relations.

b. after STP gets the go-ahead signal but before it actually enters the CPL. The AMP local to STP would detect the failure and abort STP properly. However, since the size of the VR is 1, no break.

will be noticed. The timer in corresponding TU entries in the TUD's of target sites will run out, causing the removal and deletion of those TU's, and the release of the affected relations.

2. TPI's failure

a. A failure of TPI could occur before TPI puts TU in TUD locally. Such a failure would be detected by AMP and the corresponding DT aborted.

b. Once TU's are in TUD, TPI will vanish but there remains one concern: the timer for TU's in TUD could run out prematurely and when the CPL begins, the site would not have its TU's anymore.

1. This is not likely to happen because of the high value given to the timer. It is also further assumed that if either STP or its site fails, it will be so for a long time and when STP tries to interact again, then:

1. if STP alone failed, the AMP would have aborted it,

2. otherwise the whole site failed, in

which case it will be detected as such by mechanisms of CPL.

ii. If it does happen, then the RUM (*) with missing TU's will set the status of corresponding relations to inconsistent. No further operations will take place because each TP checks the status of target relations and if the test is not successful, aborts DT. Furthermore, a situation such as the above will force the RUM to update its copies to remove the status of inconsistent since MAJ and inconsistent cannot co-exist.

(*) RUM is Release and Update Manager, an entity of CPL. Part of the proof of this assertion uses notions developed later in CPL. The reader is therefore referred to chapter 4.

Figure 3.1: EPL access graph

Figure 3.2: EPL strategy

type

TPM: array [1..X] of entry

entry=

record

transaction i.d.: integer

flag: (distributed, local)

TPtype: (S-node, D-node)

D-nodeRemoveflag: boolean

ERT: integer

{ERT= expected round-trip time}

TPlist: array [1..S] of TP

NodeList: array [1..S] of nodes

end

submission=

record

submissiontype: (S-nodeEnter,

D-nodeEnter, S-nodeRemove,

D-nodeRemove, localEnter, cancel)

TPlist: array [1..S] of TP

nodeList: array [1..S] of nodes

{any other relevant information}

end

monitoringmessage=

record

transaction i.d.: integer

TPlist: array [1..S] of TP

nodeList: array [1..S] of nodes

routinglist: array [1..S] of node

sender: node

destination: node

end

TUD: array [1..X] of TUDentry

TUDentry=

record

transaction i.d.: integer

CLflag: boolean

ETC: integer {expected time to commit}

TUlist: array [1..T] of TU

pointers: array [1..T] of ↑TU

end

Figure 3.3: EPL Data Structures

Activity Monitoring Process

```

begin
  repeat
    submission:=.dequeue (PMRQ)
    case submission of
      S-nodeEnter:
        begin
          insert (TP,TPM)
          flag:= distributed
          TPtype:= S-node
          routinglist:=
            createroutinglist(TP)
          sender:= STP node
          destination:=
            firstnode(routinglist)
          shift (routinglist)
          message:= (monitoring, TP's,,
            routinglist, sender,
            destination)
          send(message,destination)
          ERT:= .evaluate(expected
            round-trip time)
        end
      D-nodeEnter:
        begin
          insert (TP,TPM)
          flag:= distributed
          TPtype:= D-node
          ERT:= evaluate(expected
            round-trip,time)
        end
      S-nodeRemove: {distributed or local}
        begin
          remove (TP,TPM)
        end
      D-nodeRemove:
        begin
          for TPentry in TPM do
            D-nodeRemoveflag:= true
          od
        end
    end
  end

```

Cancel:

```

begin
  insert (TP,AT) {AT= abort table}
  remove (TP,TPM)
end

```

LocalEnter:

```

begin
  insert (TP,TPM)
  flag:= local
end

```

endcase

```

while mssgbuffer <> empty do
  {not empty of messages for AMP}
  mssg:= receivemessage
  with entry[mssg.transaction i.d.] do
    if entry <> nil
      {i.e. entry has not been deleted}
    then
      ERT:= re-evaluate (expected
                        round-trip time)
      mssg.sender:= node
      mssg.destination:=
        firstnode (routinglist)
      shift (mssg.routinglist)
      if [mssg.routinglist]=1
      then
        if TPtype= S-node
        then
          { [RL]=1--> end of EPL}
          signal (psem (TP))
        fi
      fi
      if D-nodeRemoveflag= true
      then
        delete(node,mssg.routinglist)
        remove(TP,TPM)
      fi
      send(mssg,destination)
    fi {if entry is nil,
        message dies here}
  endwith
endwhile

```

```

for each entry in TPM do
  if entry.flag= distributed
    and entry.ERT is expired
  then
    insert(TP,AT)
    remove(TP,TPM)
  fi
od

for each entry in TUD do
  if CLflag= false
  then
    {transition into CPL has not
    happened yet}
    if ETC is expired
    then
      delete target updates
      release(locks, on TU's relations)
      delete (entry, TUD)
    fi
  fi
od

with interface from O/S do
  for each entry in TPM do
    if entry's TP has failed
    then
      insert (TP,AT)
      remove (TP,TPM)
    fi
  od
endwith

until machine halts
end {of AMP}

```

Figure 3.4: Activity Monitoring Process

Transaction Process

```

begin
  job:= dequeue(jobqueue)
  if job= supervisory
  then
    {TP is to be STP}
    if jobtype= distributed
    then
      request:= (synchronization, ...)
      queue (request, DRQ)
      {DRQ=distributed request queue}
      wait (psem(TP))
      {signal will come from GC when TP
       has been successfully synchronized}
      request:= (S-nodeEnter, TP, ...)
      queue (request, PMRQ)
      {PMRQ=process monitoring
       request queue}
      wait (psem(TP))
      {signal will come from DBM when TP
       has been granted the locks on its
       requested relations}
      read (read-set relations, TPworkspace)
      if read <> successful
      then
        request:= (Cancel, TP, ...)
        queue (request, PMRQ)
        Exit
      fi
    fi
  release (read-set, relations)

  {what follows is the EXECUTE part of
   of the transaction. STP gets its
   instructions from QPP through QPP/STP
   interface. The execute is then a
   series of operations (i.e. read, send,
   receive, process) with a test for
   success at the end of each of them.
   A failure of the test entails
   aborting the transaction.}

```

```

message:= (put TU in TUD)
send (message, nodes with TU)
put (TU, TUD)
wait (psem(TP))
    {signal will come from AMP when the
      size of the VR protection has gone
      down to 1, indicating that the TU's
      are in TUD at all relevant sites}
request:= (S-nodeCommit, TR, TN, TU)
queue (request, DURQ)
    {DURQ=distributed update
      release queue}
request:= (S-nodeRemove, TP, ...)
queue (request, PMRQ)
Exit

```

else

```

    {jobtype=local}
    request:= (local DB access)
    queue (request, LRQ)
    {LRQ=local request queue}
    wait (psem(TP))
    {signal will come from LC when the
      transaction has been put in the
      ready to be locked queue}
    request:= (LocalEnter, TP, ...)
    queue (request, PMRQ)
    wait (psem(TP))
    {signal will come from DBM when TP
      has been granted the locks on the
      relations it requested}
    read (read-set relations, workspace)
    if read <> successful
    then
        request:= (Cancel, TP, ...)
        queue (request, PMRQ)
        Exit

```

fi

```

    release (read-set relations)

```

```

    {EXECUTE; see previous comments on
    -execute phase of a transaction}

```

```

    update (local DB, TU)
    request:= (release, write relations)
    queue (request, local release queue)
    request:= (S-nodeRemove, TP, ...)
    queue (request, PMRQ)
    Exit

```

fi

```

else
  {request from GC for TP to be participant
   in an operation remotely originated.
   Transaction is being currently
   synchronized.}
  request:= (D-nodeEnter, TP,...)
  queue (request,PMRQ)
  wait (psem(TP))
  ; {signal will come from DBM when TP has
   been granted its requested locks}
  read (read-set relations, workspace)
  if read <> successful
  then
    request:= (Cancel, TP,...)
    queue (request,PMRQ)
    Exit.
  fi
  release (read-set relations)

  {EXECUTE; see previous comments on
   the execute phase of a transaction}

  {At this point, either TU is in TUD or
   read has been completed and contents of
   workspace successfully sent to other
   TP's under STP directives.}

  request (D-nodeRemove, TP,...)
  queue (request,PMRQ)
  Exit
fi
end {of transaction process}

```

Figure 3.5: Transaction Process

CHAPTER 4

Commit Protocol Layer

4.1 Introduction

4.1.1 State of the system at the entry point to CPL

The situation when a Distributed Transaction (DT), through the actions of its Supervisory Transaction Process (STP), decides to enter the CPL is as follows:

1. The Target Updates (TU) produced by DT during its execution are now residing in stable storage and have been entered into the Target Update Directory (TUD) which also resides in stable storage (*). These actions have taken place at all target sites.
2. The structure of the Execute Protocol Layer (EPL)

(*). Stable storage indicates that the information is not in volatile memory. For more on stable storage see [STUR80] and [MENA80b].

is such that the STP can determine with certainty that the above fact (i.e. that the TU's are in TUD at all target nodes) really took place. Following that, the STP, which at this point is the only surviving Transaction Process (TP) of the DT, requests entry into CRL by queueing a properly formatted submission into the Distributed Update and Release Queue (DURQ).

4.1.2 Problem definition

The problems addressed in this chapter are different from those of a single site database or data storage facility. Reliable and correct operations in such an environment have been studied previously and several solutions have been proposed. These solutions fall into two broad categories:

1. Updating procedures for relations

Under this heading are grouped: files and records handling techniques [VERH77], intention lists [LAMPSxx], stable storage [STUR80] and recovery points and commitment intervals [COLL79].

2. Process control techniques

Techniques to ensure proper execution of processes as well as proper re-starts in case of failures are covered in [RAND78].

Throughout the rest of this chapter, it will be assumed that the data storage subsystem at each site in the DDB network features stable storage, correct file updating and consistent process re-starts from the last checkpoint.

Grouping individual DB's into a DDB network introduces problems in co-ordination of updating activities (*). One of the main characteristics of a DDB is the existence of relations which are present at more than one site. Such relations are called Distributed Relations (DR). The occurrence of failures during a critical phase of a transaction updating some DR's may result in inconsistencies. To illustrate this point, consider for example a DR, r_i , with three identical copies located at n_i , n_j and n_k . If the updating of r_i by some transaction is successful at n_i and n_j but not at n_k , r_i is not mutually consistent anymore, since, for a user at site n_s who wants to read r_i , the result of the read will be dependent upon the site it is obtained from. This condition should not be confused with the fact that as far as absolute physical time is concerned, a DR may for a certain period of time be

(*) just to mention this one since it is the one of interest in this chapter.

inconsistent during update. The Concurrency Control (CC) mechanisms of the SPL are used to make the action of updating atomic in logical time and to guarantee database update consistency for all types of transactions. However, when dealing with an unreliable medium [YEME79] and the possibility of partial or total site failure, it may happen that the period of inconsistency never terminates. In this case, additional updating mechanisms will be needed to cope with site failure and recovery.

The primary concern of this chapter is the design of reliable updating mechanisms in the form of a Commit Protocol Layer (CPL) which would take over after the execute phase of a distributed transaction. The main objectives of the CPL will be to provide reliable updating of DR's. The CPL will guarantee that, following each DR update, all the available copies of a DR will be mutually consistent and that currently unavailable components will be brought up to date as soon as they are recovered.

The network environment in which the CPL operates has specific characteristics. DDB systems are supported by a collection of individual processors linked by a communication network. Each processor is running independently but is co-ordinating all distributed activities through exchanges of messages. The network only

guarantees the ordering of messages between two parties and introduces variable message transit delays that are not negligible when compared to the time between synchronization events at individual sites. Nodes may fail and upon recovery may be unaware of the failure. Similarly, the network may be partitioned into subnetworks in which distributed activities may take place independently.

4.1.3 Literature survey

The first attempts to design reliable CC and updating mechanisms used the two-phase commit protocol [STUR80], [LAMPSxx], [GRAY78] as a means of reliably bringing the status of several nodes to a point at which they will all update or none will. The two-phase commit protocol has also been used in centralized CC mechanisms such as the one described in Menasce [MENA80a].

Voting algorithms [THOM78] designed to handle the CC problems have, as a by product, some interesting reliability properties. These algorithms are not too efficient and their performance degrade rapidly with increased load. They were originally designed for fully redundant systems and do not, in general, handle network partitioning efficiently. A reliable multiprocess system [LAMP78b] proposed by Lamport also makes use of voting. In this case it is to preserve

the mutual consistency of fully redundant relations. It is an interesting system but it suffers from the same shortcomings as [THOM78].

The concepts of primary copy together with recovery algorithms have been used [STON79] to provide resiliency. This scheme imposes a centralized approach to the control of updating and requires having an - up - to date back-up for primary copies.

Menasce [MENA80a] introduces a reliable CC and update system. Centralized control and distributed recovery mechanisms are used to maintain the mutual consistency of DR's. A parallel can be drawn between [MENA80a] and [LELA77b] although the latter did not address the DDB issue explicitly. The system of [LELA77b] can be thought of as a central controller circulating on a virtual ring. Breaks in the ring are handled by regenerating the control token, hence the central controller. Independent operational subnetworks can also be handled by special techniques. The system of [MENA80a] is conceptually the same except that the controller is resident at one node. The case of subnetwork is handled in a very similar way. Both those systems use a form of centralized control which makes them attractive for particular network configuration such as a star topology. In other topologies, centralized control is prone to

failure, RUM failure, network interface failure, and network partitioning (i.e., isolation of some NI's). These failures will result in the following symptoms: message loss, incomplete interaction, and creation of subnetworks. A typical failure may combine any of the above.

2. The duration of failures is long as compared to the execution of an update operation.

3. Detection of failed nodes is also assumed because of assumption 2 above and also because of the fact that a reasonable amount of distributed activity will take place.

4. The sending of multiple messages is considered atomic with respect to a possible crash of the sender.

5. If a node i can communicate with node j but not with node k , then node j cannot communicate with node k .

6. Periods of correct functioning during which operations can complete successfully will occur frequently.

4.1.4 Characteristics of the CPL

The reliability mechanisms of the CPL presented in this chapter make use of up-lists and a two-phase commit protocol but differ substantially from what has been done before by featuring: 1-distributed initiation of all operations, 2-the use of diagnostics from the communication subsystem and 3-correct operation in presence of independent subnetworks. Furthermore the CPL does not assume that recovering nodes know of their previous failures.

It is also interesting to contrast the Guardian Ward Site Monitoring (GWSM) facility of the RelNet [HAMM80] and the up-list strategy used in this thesis and others [MENA80a], [ELLI77].

It should be first pointed out that some memory mechanisms have to be provided to a system in order to make it robust. The reason for this is that, when a node fails, it is assumed to be in that state for a period of time which is long when compared to normal DBB activities. During that period of time, the node may miss several update operations so that its DB is not consistent anymore with that of the operational part of the network. When the node is repaired and tries to join the other nodes, it has to be recognized as a recovering node and then brought up to date.

In the RelNet of SDD-1, this is accomplished by having certain nodes (guardian) keep a watch on a given node (ward). When a recovering node has been detected, the system, through its guaranteed message delivery service, sends it all the update messages it missed.

In systems using up-lists, any operational node is able to detect a recovering node by simply looking at its up-list. Methods to re-insert the recovering node are system dependent and those of SPL/EPL/CPL are described in chapter 5.

Up-lists are a special case of GWSM in which, for a n node network, the number of guardians for each ward is $(n-1)$. Therefore up-lists offer a greater degree of reliability than GWSM and also, because up-lists are fully redundant relations, their updating extend a logical time benchmark to all nodes in the network. This obviates the need for careful timekeeping as in the GWSM. On the negative side, however, updating up-lists is costly of time and messages, with the number of messages growing quickly with the size of the network.

4.2 Description of CPL

4.2.1 CPL strategy

The general strategy of the CPL is illustrated in figure 4.1. Although single updates are considered, the CPL can support, in general, concurrent activity with updates affecting more than one relation.

The basic building block of the flowchart of figure 4.1 is the two entry point-two exit point block. It allows a distributed operation to be started and to either succeed or fail. After a failure some remedial action can be taken and the original operation can then be re-started.

The flow of control in the flowchart of figure 4.1 can be described as follows: A request for distributed update is first processed by a synchronization operation in which all the participants have to interact properly. If this synchronization succeeds then each individual participant goes on to update the target relations and release the locks on them. In case of failure the participants that did not respond properly are removed (*) from an up-list called the

(*) Remove in the sense that the status of a node is set to down.

Network Access Table (NAT). The NAT carries an indication of the status of the network as perceived by the node maintaining it. In other words if a site is marked "down" in the NAT it is excluded from any network operation until it has fully recovered and has been re-inserted in the NAT. If the removal operation did not succeed, it will loop back and re-do the NAT update until the network (as represented by the NAT) is composed uniquely of sites that are up.

At that point a decision can be made whether to continue the operation or not. The update operation will go ahead, if and only if for each target relation, a majority of copies is available in the network. This will be known as "Majority Rule". It should also be mentioned that there exists another rule, namely the "Minority Rule" which says that if only a minority of copies of a DR exists in the network then the DR can not be used at all, even for read purposes. This information is stored in the form of a status indicator for a given relation, with the status indicator being MAJ for majority and MIN for minority.

If a decision is made to continue then the control is passed to the synchronization block through the Re-synch entry point. The outcome of the synchronization operation is still either success or failure, hence the recursive nature of the strategy.

On the other hand, the decision may be to stop in which case the operation has to be aborted at all participating sites (which are marked as up in the NAT). The abort operation is very similar to a NAT update and consists of entering the identification code of an update operation in the Abort Table (AT) of the involved sites. The AT is a data structure which is present at all sites in the network but its updating does not require network wide operation.

Successfully completing the abort block means the end of a particular update operation. Failures at that point entail removing the failed participants from the NAT and looping back to complete the abort block.

To complete this general description a few points should be emphasized:

1. The information available to the CPL is not restricted to its own use but should be shared with the CC mechanisms. For this reason it is assumed that when an update is initially submitted to the CPL, a majority of copies of all its target relations is available.
2. This approach implies that if too many sites fail, the DDB will end up fragmented to a point where

barely any distributed activity can take place.

3. All the blocks described previously assumed some means of distributed failure detection conducted by a modified two-phase commit protocol.
4. Special cases may arise from the execution of the protocol in which failure situations would go undetected. Such cases are taken care of by expanding the status indicators of relations to include the status of inconsistent and by setting the status of involved relations to inconsistent when such cases are thought to have occurred. The status of a relation already covers "MAJ" when a majority of copies of the relation is available and "MIN" when it is not. A status of inconsistent, as will be obvious from the algorithms, is used to remove logically certain copies of a relation from the database.

4.2.2 CPL structure

In order to implement the general strategy of figure 4.1, a set of reliable algorithms using distributed control and a conceptual architecture to support them are needed.

The reliable algorithms will be introduced in section 4.3 while the conceptual architecture is the topic of this section.

Every site in the network features the same CPL architecture which is described in figure 4.2. The access graph of the CPL does not show the structures of the CC mechanisms. Instead it assumes that all lock requests submitted to the DataBase Manager (DBM) through the "ready to be locked" queue (RTBLQ) are consistently ordered with respect to all the RTBLQ's of all other sites. This allows the CPL to function with any concurrency control mechanisms using two-phase locking policy.

Within a site the Release and Update Manager (RUM) is responsible for executing the reliable algorithms to be presented shortly. As seen before, a request to update has been submitted by STP at the end of DT's execution and is now in the Distributed Update and Request Queue (DURQ) where it will be dequeued by the RUM which will initiate the execution of the reliable algorithms. The RUM that initiated an operation is referred to as the S-node while the other RUM's which will also be part of the interaction will be called the D-nodes.

It should also be mentioned that the abort operation,

which is referred to in the algorithms, involves entering the identification code of a particular operation in the abort table (AT). From figure 4.2, it can be seen that the role of the abort co-ordinator is to delete from a site all traces of a transaction whose identification code is in the abort table. The abort table is updated by using the reliable algorithms, thus making the Abort Protocol Layer (APL) an integral part of the CPL.

4.2.2.1 Release and Update Table (RUT)

The Release and Update Table (RUT) is used in synchronizing updates among target sites. Its structure in pseudo-Pascal is shown in figure 4.3. The RUT contains individual entries for each relation present at the site. It keeps an up-to-date timefield (TF) for each relation together with status information. At first sight, it would seem that the timefield information is not really necessary since all the update requests have been serialized correctly by the CC mechanisms. However, they are essential later to the merge operation.

4.2.2.2 Updating Relations

The basic protocol to co-ordinate updating operations

is shown in the time graph of figure 4.4. The example uses only three nodes but it is easy to extend it to any number by adding extra D-nodes. The basic updating procedure is very similar to the synchronization procedure of the SPL.

Upon receiving an update request U , the RUM, which will become the S-node, forms a set of target nodes $N(U)$ and target relations $R(U)$, generates a unique timestamp $TS(U)$ and enters U in the RUT. It then broadcasts Ready to Update messages (RTU) to all other nodes in $N(U)$. The RTU messages carry $TS(U)$ which is unique to the operation (*) and which is used to update the timefield of all relations in $R(U)$ when U is fully acknowledged.

When a RUM receives an RTU message, it enters the update request U into its own RUT. It then sends acknowledgements (ACK) for that particular U to all other nodes in $N(U)$. When a RUM in $N(U)$ has received ACK's from all other nodes in $N(U)$, then as far as that node is concerned, U has completed successfully. It then updates with $TS(U)$ the timefield (TF(ri)) of all the relations in $R(U)$ which exist at that site. After that, U is removed from the RUT and sent to the local DBM. Once U is fully ACKed, assumptions about the system components, namely

 (*) TS 's are made unique by appending site number. TS 's are also always increasing.

stable storage and process re-starts, guarantee that the actual updating will eventually take place.

In presence of failures, the message exchange part of this protocol has to be modified. First of all, all the operations will be protected by timers and time-out periods. If any RUM times out, it sends "Stop" messages to all other nodes in $N(U)$. Furthermore the update operation is not carried out as soon as all the ACK's are received. A RUM which received all the ACK's for a given update operation would wait till the end of the time-out period. It would then also wait some extra period of time in order to be able to receive "Stop" messages should any be sent. If the RUM does not receive any stop messages then the operation is carried out but if some stops are indeed received, the operation will be halted. This procedure is shown in figure 4.5 for the case where an ACK message is assumed either lost or not received. (This description is basically that of the algorithms of section 4.3.1).

4.2.2.3 Network Access Table

The NAT is the central element of the general strategy outlined in figure 4.1. Its structure is shown in figure 4.6 in a pseudo-Pascal description. Besides a list of nodes and their status, the NAT also has a sequence number (NSQ)

which is an indication of the last update performed on it. Also included is a series of the most recent NSQ's, one for each node, reflecting the last NAT update triggered by a given node (LSQ_i for node i).

Updating the NAT is very similar to the updating of ordinary relations and consists of changing the status of a node (or nodes) to up, down, merging or subnetwork merge. As a general rule, only up nodes can initiate NAT update operations. The only exception is that for a given node n_k , the transition from $n_k.\text{status} = \text{merging}$ to $n_k.\text{status} = \text{up}$ is initiated by n_k (and similarly for subnetwork merge).

One very important feature of the NAT is the NSQ. Its purpose is to identify a particular version of the NAT and is used by the communication subsystem to detect sites that crashed previously but are now recovering. The communication subsystem (CSS) would then perform three extra tasks:

1. Append the current NSQ to all outgoing messages.
2. Trigger a "merge" operation (see chapter 5) when a message is received with an incorrect NSQ.
3. Periodically send "Inquire" messages to nodes which

are "down" in the NAT.

4.3 Specification of CPL

4.3.1 S-node and D-node commit

The S-node commit algorithm is shown in pseudo-code form in figure 4.7. It is initiated by a RUM executing the procedure `S-nodeCommit(N(U), R(U), target update)`. This in turn forces the RUM's located at other nodes in `N(U)` to execute the procedure `D-nodeCommit`. The D-node commit algorithm is shown in figure 4.8 and the time-out procedure for both the S-node and D-node commit is in figure 4.9.

In the algorithms, the indication as to whether a message is successfully delivered or not comes from the CSS local to the RUM. For a given message, the CSS will give the RUM either an ack, indicating that the message has been successfully received by the destination CSS, or a nack, indicating that the message cannot be delivered successfully.

The flow of messages from RUM to RUM is shown in figure 4.10. Distinctions are made between ACK's, which are part of the reliable algorithms (i.e. RUM to RUM message),

ack/nack, which are diagnostics given to a RUM by the local CSS, and message acknowledgements, which are part of the PAR protocol (i.e. CSS to CSS). The PAR protocol [SUN75] (i.e. Positive Acknowledgement + Retransmission) is used by the CSS's to communicate among themselves. The PAR protocol takes care of intermittent failures by retransmitting (up to a limit) messages that were not acknowledged properly.

4.3.2 "NAT update" procedure

The NAT update procedure and its time-out procedure are both shown in figure 4.11 in pseudo-code form. The corresponding S-node*NATupdate and D-node*NATupdate procedures are shown in figure 4.12 and 4.13, and their time-out procedure is in figure 4.14. The NATupdate procedure calls for a wait for all possible stop messages. The magnitude of the wait can be determined by using an upper bound on the message transit delays and using worst case situations in the exchange of messages.

Following the receiving of the stop messages, a sequence of initiation using some pre-determined ordering policy is obtained. Furthermore, the sequence number of a given node is used to determine the value of a timer protection, should preceding nodes in the initiation sequence fail.

4.3.3 Resynch procedure

The Resynch procedure is shown in figure 4.15 and the Resynch time-out procedure and the Resynch exception procedure are both shown in figure 4.16.

The Resynch procedure first removes from $N(U)$ the nodes that failed. Then, in a fashion similar to the NAT update procedure, a sequence of initiation is decided upon, with the first node initiating either an S-node commit or an S-node Abort (*). The other nodes will set the value of their timer protection to be proportional to their sequence number, and will wait for the interaction to begin.

4.4 Validation of CPL

4.4.1 Assumptions

The basic assumptions used in this section are as follows:

1. The failures to be taken care of are: link failure (from a node CSS to a network interface (NI)), CSS

(*) S-node/D-node Abort is basically an S-node/D-node Commit ($N(U)$, AT, transaction id.).

failure, RUM failure, network interface failure, and network partitioning (i.e., isolation of some NI's). These failures will result in the following symptoms: message loss, incomplete interaction, and creation of subnetworks. A typical failure may combine any of the above.

2. The duration of failures is long as compared to the execution of an update operation.

3. Detection of failed nodes is also assumed because of assumption 2 above and also because of the fact that a reasonable amount of distributed activity will take place.

4. The sending of multiple messages is considered atomic with respect to a possible crash of the sender.

5. If a node i can communicate with node j but not with node k , then node j cannot communicate with node k .

6. Periods of correct functioning during which operations can complete successfully will occur frequently.

The proper functioning of the reliable algorithms is guaranteed by three basic mechanisms: up-lists, status indicators for relations and nodes and a modified two-phase commit distributed protocol. These mechanisms preserve the mutual consistency of DR's in the network and provide consistent updating. These goals are achieved by the close co-operation of all mechanisms, supplemented by the merge procedure (chapter 5). What follows is an informal proof of the correctness of the Commit Protocol Layer algorithms described previously.

4.4.2 Validation Assertions (Up-list mechanism)

Assertion 4.4.2.1

Up-list mechanism (NAT) together with up-list version number checking (NSQ) provide for positive identification of: failed node(s) that is/are recovering, and of subnetworks that are merging.

Proof of Assertion 4.4.2.1

In order to obtain a complete proof, it is necessary to

consider all the likely cases, namely: single failures, repeated single failures, concurrent multiple failures, and subnetworks. In all cases, detection is achieved by comparing NSQ's of communicating sites and is normally followed by the merge procedure. When detection is not possible, it is shown that no harm is done to the mutual consistency of distributed relations.

Case 1: Single Failure

In the case of a single failure, the network N is partitioned into an operational subnetwork N_u and a set of crashed nodes, N_c , with $|N_c| = 1$ and $nc \in N_c$ (with nc as a crashed node). The node in N_c may either have failed (in which case there will be no activity there) or be isolated. If isolated, the node will provide for activity of local nature only, thus not endangering the mutual consistency of DR's. When a node in N_c recovers and tries to interact with other nodes (or receive an Inquire message(s)) there are two possibilities:

1. The failure was never noticed by the rest of the network. This means that there never was any distributed activity involving the crashed or isolated node while the effects of the failure lasted. Therefore when nc is involved in distributed activity with N_u , it is as though

nothing had happened.

2. The failure was noticed by Nu. Any message either from Nu to Nc or Nc to Nu will have conflicting NSQ hence detection.

Case 2: Repeated Single Failures

The case of repeated failures is very similar to the previous case with the exception, however, that subnetworks can be formed among some recovering nodes. (e.g. $n1, n2 \in Nu2$ with $N = \{Nu1, Nu2, Nc\}$ and $Nc = Nc1 \cap Nc2$). Limited distributed activity can take place within subnetworks and does not endanger the mutual consistency of the relations. Subnetworks may try to join other subnetworks and eventually the network will be whole again.

Case 3: Concurrent Multiple Failures

Two or more nodes may fail at the same time in the sense that the updating of their status from up to down takes place in the same NATupdate. Two possibilities exist:

1. Such nodes may recover at different time and previous cases will then apply. (i.e. NSQ's will be different).
2. Such nodes recover at the same time and will then,

de facto, form a subnetwork since they have no way of detecting their respective failures until they actually interact with other nodes. Distributed activity may take place among such nodes and does not violate any consistency criteria since:

- a. as that activity does not trigger a detection, it is contained in that subnetwork,
- b. its effects are the same as though it happened before the failure(s) (*).

When that subnetwork is detected, its nodes will be detected as single failures since they themselves never noticed any disturbance.

Case 4: Subnetworks

When a split occurs in the network, subnetworks are created. Subnetworks, being composed of operational sites, some distributed activity may take place within their confines.

All the cases involving single nodes can be extended to subnetworks with similar results.

(*) This activity is representative of an older logical age.

Assertion 4.4.2.2

Up-list mechanism and the Majority Rule (MAR) protect the mutual consistency of distributed relations with respect to "write" operations (i.e. updates).

Proof of Assertion 4.4.2.2

Given a series of update operations, it is possible to form a log L [BERN79]

$$L = \{U_1, U_2, U_3, \dots\}$$

so that L is representative of the past distributed activity in the network. When faced with failures, the network may become fragmented into subnetworks; N_i 's, each with its own log, L_i , covering the distributed activity that took place while the subnetwork existed.

At some point in time, the network will be re-connected and will function as a unit. In forming the log for the network, the individual logs of previous subnetworks have to be considered so that, for a time interval in which subnetworks existed, the total log, L_n , is given by

$$L_n = L_1 \cup L_2 \cup L_3 \cup \dots$$

Now consider $U_1 \in L_1$ and $U_2 \in L_2$ with $R(U_1) \cap R(U_2) \neq \emptyset$.

Clearly L_1 and L_2 cannot be incorporated into L_n because the

effects of U_1 and U_2 are not serializable. If U_1 and U_2 were allowed to run in their respective subnetworks the distributed relations in $R(U_1) \cap R(U_2)$ would not be mutually consistent.

In order to prevent unserializable operations, updates will be allowed only if, for all the DR's in $R(U)$, a majority of copies is available within the confines of the network as represented by the NAT.

The MAR therefore ensures that if a given update satisfies it, then at least one component of each DR in $R(U)$ was present in the last update performed on it. The MAR together with the merge procedure will guarantee the consistency of all relations with respect to "write" operations.

Assertion 4.4.2.3

Up-list mechanism and the Minority Rule (MIR) protect the mutual consistency of DR's with respect to "read" operations.

Proof of Assertion 4.4.2.3

The main purpose of the CPL presented in this paper is to ensure consistent updating. Consequently the CPL impacts "read" operations since a "read" accesses relations that have been updated in the past. There are two criteria for a "read" to be successful:

1. The "read" should be independent of the site it is performed at.
2. If a relation can be read, then the contents of that "read" are such that, after some processing, they can be used to update some relations.

The first criterion is satisfied by the fact that:

1. For an operational site in an operational subnetwork, the "read" will be performed on sites that are up and whose relations are mutually consistent, hence independence of location. Furthermore, "read" operations will never be performed at sites which are marked as down and/or have not merged yet.
2. For a recovering site that is not aware of having failed, a "read" operation involving foreign sites

would entail detection, followed by a merge.

The second criterion is satisfied by the Minority Rule (MIR) which states that if a subnetwork is in a position where, for a given DR, only a minority of copies is available, then not only can it not be updated (MAR) but it cannot be read either.

To explain this, the logs of the previous proof have to be enlarged with "read" operations (R_i). A transaction is therefore a "read" followed by a "write" (i.e. $T_i = \{R_i, U_i\}$). Consider two transactions, T_1 and T_2 , each with its read-set, $RS(T_1)$ and $RS(T_2)$, and its write-set, $WS(T_1)$ and $WS(T_2)$, related in the following way:

$$RS(T_1) \cap WS(T_2) = x \quad RS(T_2) \cap WS(T_1) = y.$$

T_1 will run in Nu_1 and T_2 will run in Nu_2 (i.e. $T_1 \in L_1$ and $T_2 \in L_2$). T_1 and T_2 are assumed to run and since the MAR is satisfied by the relations in $WS(T_1)$ and $WS(T_2)$, it follows that the relations in x are majority in Nu_2 and minority in Nu_1 and vice-versa for those of y .

Furthermore $RS(T_1) = x \cup z$ and $RS(T_2) = y \cup s$ so that the relations in z and s are majority in their respective subnetworks and are changed dynamically by other transactions. This means that the following logs

$$L_1 = \{ \dots T_1 \dots T_1 \dots T_1 \dots T_1 \dots \}$$

$L2 = \{ \dots T2 \dots T2 \dots T2 \dots T2 \dots \}$

are not serializable into an L_n because the output of T_1 and T_2 which is a function of the "read" is now dependent upon a particular partitioning of the network (i.e. dependent upon those transactions in Nu_1 and Nu_2 which are changing relations in z and s). Therefore, enforcing the MIR guarantees that the contents of the "read" are valid and can be used later for updating purposes.

Before concluding on the proof, there remains an abnormal situation to be covered. It may happen that subnetworks are formed but that the NAT of their respective nodes does not reflect the fragmentation. In such case, transactions could read relations which have a status of "MAJ" but which should in fact have a status of "MIN". Reading those relations violates the MIR, but no harm has been done since if a transaction unknowingly reads relations with a "MIN" status then:

1. if the write-set of the transaction is contained in that subnetwork then no NATupdate will be triggered and, conceptually, the operation can be thought of as having happened before the failure.
2. if the write-set is not contained within that subnetwork, then the update operation will trigger

a NATupdate followed by a re-evaluation of the status of the relations at the up-nodes. After that, the above relations would be seen as being "MIN" hence abort the transaction.

Assertion 4.4.2.4

The up-list mechanism as described in this paper does not lead to deadlock in the sense that recovering node(s) would remain isolated.

Proof of Assertion 4.4.2.4

A recovering node which may not know that it previously failed will assume that the NAT it has in stable storage is valid. When some distributed activity takes place at that node, its outbound messages will carry an outdated NSQ which will ensure detection and eventual recovery.

If, on the other hand, some of the nodes with which the recovering node wants to interact are down, then a NATupdate will be triggered by the recovering node and this would ensure detection and recovery. At this point, if all the nodes which would participate in the NATupdate are down

(although in the network some other nodes may be up but marked as down in the outdated NAT that the recovering node has) then the result of the NATupdate will be that the recovering node will mark itself as the only up-node in the network. The recovering node is then a one node subnetwork.

The case of subnetwork is the only source of potential deadlock in the up-list mechanism. Since, because of the operating rules, a node will send messages only to other up-nodes, subnetworks will never detect each other's presence, hence deadlock. However, since nodes periodically send "Inquire" messages to down nodes, this potential deadlock situation is averted.

4.4.3 Validation Assertions (Distributed Protocol)

This subsection is concerned with the validation of the distributed protocol which implements update synchronization through message passing, and distributed failure detection through sending stop messages and extra wait strategy.

Assertion 4.4.3.1

The execution of the distributed protocol (i.e. modified two-phase commit) does not lead to deadlock.

Proof of Assertion 4.4.3.1

A finite state machine (FSM) representation of the distributed protocol is to be found in figure 4.17, 4.18 and 4.19 dealing respectively with S-nodeCommit, D-nodeCommit and NATupdate/Resynch. From those FSM's, it can be concluded that deadlock situations do not arise in the protocol since:

1. there are no loops in the FSM's (although the protocol uses recursion there is no risk of indefinite postponement; the result of the protocol is either success or abort in subnetworks of varying sizes).
2. there are no terminal states
3. all "wait" states which could become trapping states are protected with timer mechanisms

Assertion 4.4.3.2

The update operation, whose co-ordination is the responsibility of the distributed protocol, will be carried out by all surviving sites (even in separate subnetworks) or will be stopped by all. Cases where the assertion does not hold are shown to be recoverable.

Proof of Assertion 4.4.3.2

The responsibility for the triggering of a given update operation does not lie with the commit layer but rather with the execute layer (see figure 1). It is therefore a transaction control issue and it is reasonably easy to envisage mechanisms to ensure proper start up of an update operation. Furthermore, it is also assumed that at the onset of an operation, the MAR is satisfied and that the set $N(U)$ is composed of up-nodes which are connected (i.e. in the same subnetwork).

At this point, it should be stressed again that a node participating in an update operation only has two ways of detecting that something may be wrong with the execution of the algorithms.

1. Its own CSS tells it (through nack's) that some messages sent to other nodes cannot be delivered. This case covers both the S-node and its RTU messages and the D-nodes and their ACK messages.
2. The node times out because ACK's from some nodes have not reached it within a specified amount of time.

Upon detecting a failure, a node can stop the updating by sending "stop" messages.

There exist three basic scenarios to the execution of the distributed protocol:

Situation 1: (a or b)

a-(RTU messages sent successfully and all ACK's received).

b-(all ACK's sent successfully and all ACK's received).

This situation is a no failure situation and does ensure proper updating at all sites in $N(U)$ since the RTU messages were received by all and were followed by full acknowledgements (ACK's). Interestingly enough, this situation also takes care of the case where, at a remote node, the RUM failed but the CSS was still operational. In such case, the fact that the remote CSS positively received the messages means that those messages are in stable storage.

at that node and will be available to that remote RUM once it recovers.

Situation 2:

A node gets all the ACK's but some ACK's are not delivered successfully. In this case the result is that the node will send stop messages. This situation also supports a variation in which the S-node cannot deliver successfully all its RTU messages. In that case, the S-node will send stop messages. (The algorithm could not have been completed in any case).

Situation 3:

A node delivered all the ACK's successfully but timed out on some missing ACK's from other nodes. The node will send stop messages.

Once a decision to send stop messages has been made by a node, several possibilities exist because of the occurrence of other failures. Given that situation 2 or 3 happened, three cases have to be considered.

Case 1: Stops are sent; no further failures.

Stops are received in subnetworks where detection took place; the NAT is updated and the operation is re-started without problems since no extra failures will take place. The Resynch procedure will then decide whether to go ahead with the update or to simply abort.

Case 2: Stops are not sent; failure of the node(s) that originally detected the failure(s).

In the event of case 2 occurring, even if the subnetwork in which the failed node was is further subdivided, no stop message will be received and except for failures of other nodes (or detection of failures by other nodes), the update will be committed in that subnetwork.

However, the main claim of assertion 4.4.3.2 has been violated because of the fact that the failed node(s) had detected a situation which warranted the stopping of the algorithm but, (those) node(s) having failed, the algorithm was not stopped and some other nodes went ahead and updated.

Two possibilities should be considered in this connection:

1. All the other nodes in $N(U)$ will also commit, since

the update satisfies the MAR (according to their NAT which is no longer accurate). Assuming that those nodes do have, among themselves, a majority of copies of all the relations in $R(U)$ then the fact that the updating took place introduced no inconsistency. The other node(s) which may have failed, which may be isolated or may be in a subnetwork are not important since it is assumed (assumption 2 and 3) that no surreptitious re-connection will take place. In fact assumption 3 guarantees the detection of their failure which is reasonable because:

- a. If a single node crashed during the operation it would upon recovery time out and try to interact with an operation that has long since been done.
- b. If a single node was isolated (or nodes were in a subnetwork) then a NAT update is the logical conclusion. The following Resynch will also fail and the relation in $R(U)$ will have their status set to MIN (but see item 2 following).

2. The nodes that committed did not have all the copies of relations in $R(U)$ necessary to satisfy

the MAR. This situation is potentially damaging since the mutual consistency of the DR's in R(U) is not preserved anymore. The solution to this problem is for nodes that failed or were isolated while the operation was in progress to set to "inconsistent" the status of relations for which the status went from "MAJ" to "MIN" because of failures.

The meaning of a status of inconsistent is to logically remove from the database copies of DR's with that status. The implications of such a removal are twofold for a given copy of a DR:

1. The node where it is located did not fail but instead found itself in a subnetwork where the MAR was no longer satisfied. The setting of the status of that relation to inconsistent was done during the Resynch which followed the NAT update. As long as the node is up, then this relation cannot be accessed (just as relations with "MIN" status) but furthermore, if the node participates in the merging of a recovering node, the status of inconsistent will be an indication that this particular copy of a relation cannot be used to rebuild a valid copy which would have a MAJ

status (*)

2. The node where it is located crashed while the operation was in progress. The setting of the status of that relation to inconsistent was done when the node recovered and either tried to complete the operation (which had been completed elsewhere long ago by assumption 2) or executed the merge procedure.

Case 3: Stops partially received; further failures.

The fact that, within the subnetworks involved, the stops were partially received points to three subcases:

1. some nodes which belonged to that subnetwork (i.e. the one in which stops were sent) are now forming their own subnetwork and are therefore not reachable. These nodes will commit.
2. some nodes may be down within the subnetwork which accounts for their inability to receive the stop messages. These nodes will not commit and their

(*) An exception to this is when all the copies of a given DR are present in the system and that they all have a status of inconsistent.

having failed will be detected once they recover. The status of the relations in $R(U)$ will later be set to inconsistent.

3. only a subset of nodes in a given subnetwork got the stop messages sent by the nodes which once belonged to the same subnetwork but do not anymore. The subset of nodes that got the message(s) will eventually be involved in a NAT update in which the other nodes may or may not be involved depending on the ordering of the re-start sequence. In any case, those other nodes (i.e. the ones that did not get the stop messages) would have committed the update because of timing considerations. This would, in effect, be consistent with some other subnetworks in $N(U)$ which may have updated while the others did not.

In all those cases, it is assumed that whether a node commits rightly or wrongly, the mutual consistency is preserved by the setting of the status of certain relations to inconsistent by other nodes. Discussions relating to setting status to inconsistent are similar to those of case 2. Assumptions 2 and 3 also guarantee that failed nodes will not participate further and can thus be ignored for the rest of the operation. Assumption 4 is useful in reducing

the cases that have to be considered by eliminating partial broadcast of messages.

Assertion 4.4.3.3

Given that the occurrence of failures forced an update operation to stop, the following claims will be made:

Claim 1: The distributed protocol will ensure safe re-start and proper updating of the NAT.

Claim 2: It will also carry out the operation in the new N(U) or if not possible to do so will abort it.

Claim 3: The protocol will eventually terminate.

Proof of Claim 1

The stopping of the distributed protocol is done through the exchange of stop messages. The proof of assertion 4.4.3.2 covered the various cases arising from such an exchange in the presence of further failures. It should be evident that the set SN (suspected nodes) which is also used to generate the NAT update initiation sequence is potentially different for nodes belonging to the same subnetwork. The execution of the NAT update will be examined while keeping in mind that a stable subnetwork is.

at one time, assumed to exist for the NAT update to complete successfully.

1-No further failures (in a given subnetwork)

In this case, the NAT update will take place with the set SN of the S-node as the first tentative list of nodes whose status should be set to down. The set of target nodes will be {NAT-SN} with the nodes in SN being excluded to prevent the operation from looping endlessly. Simultaneous initiations (i.e. more than one S-node) will result in a new set SN which will be the union of the SN's of all other S-nodes, thus reducing further the set of target nodes.

The NAT update operation and the commit algorithms are essentially the same and will have the same properties except for the absence of status indication for the NAT. Since no further failures take place in this case, the NAT update will terminate properly and will be followed by Resynch.

2-Further failures

Further failures will either incapacitate potential initiators (i.e. S-nodes) or isolate them. The time-out protection of the operation ensures that other node(s) will

take over and increase their set SN so as to reflect the new situation.

Failure(s) during the execution of the NAT update operation result in a stopping of the algorithm and in an increase of the set SN, followed by another attempt at updating the NAT. These actions take place independently in each subnetwork.

During the execution of the protocol, situations arise (see proof of assertion 4.4.3.2) in which nodes update mistakenly. Such a malfunction of the protocol is, in the commit algorithms, taken care of by setting the status of involved relations to inconsistent. However, in the case of the NAT update, malfunctions of that kind are not critical because:

1. When these nodes, after having mistakenly updated their NAT, try to resynch, they will realize, either during commit or abort, that other failures occurred. They will therefore update their NAT accordingly.
2. The other nodes which took part in the original operation would have stopped and increased their set SN, thus excluding the nodes whose failures

were responsible for the stopping of the protocol. Furthermore, their next attempt at updating the NAT will fail because of nodes which have mistakenly updated their NAT and are, by assumption 5, unreachable. This implies that the SN of these other nodes will be increased and that the NAT update operation will be re-started.

3-Complete failure

A final possibility to consider, especially if the number of target nodes is small, is the failure or isolation of all the target nodes while an operation was in progress. In such a situation, no re-start would take place and the NAT maintained by these nodes would not be updated. Detection of the failed nodes is guaranteed by assumptions 2 and 3 and the mutual consistency of the DR's is protected by the merge procedure which will be invoked upon detecting the recovery of those failed nodes.

Proof of Claim 2

Claim 2 is concerned with the Resynch procedure which takes over once the NAT update terminates in a stable subnetwork. At this point, further failures may happen

thereby generating extra cases to consider:

1-No further failure

In this case the Resynch procedure generates a unique sequence of initiation from the new set of target nodes which is unique within that subnetwork. Uniqueness is guaranteed by the fact that, since the NAT update operation terminated successfully, the subnetwork is stable (for the time being) and the NAT is the same for every node in it. The first node in the sequence makes a decision to continue or abort the operation.

2-Further failures

A sequence of initiation is generated from the new set of target nodes and this sequence is unique within a subnetwork at the end of the NAT update operation. Further failures will therefore only affect a node's capabilities to either initiate the Resynch or to take part in it.

Failure of the initiator (either real or perceived, to be so by an isolated node) will result in the time-out of node(s) further down the sequence. These nodes will then be initiators (one at a time) and will first remove from the NAT previous initiators that failed. A NAT update results

which will again be followed by a Resynch.

3-Complete failure

As in the proof of claim 1, this situation results in no re-start at all. The mutual consistency of the DR's is guaranteed by the fact that eventual detection will take place, followed by a merge procedure.

Proof of Claim 3

The eventual completion of the distributed protocol is guaranteed by assumption 6 which says that, at one time in the network, stable subnetworks will exist so that the commit algorithms (either for update or abort) will execute successfully.

If that assumption does not hold, then the distributed protocol may simply loop endlessly, until all the nodes are down or isolated. If all the nodes are down, then clearly no updating will take place and the mutual consistency of the DR's is not endangered. Assumption 2 and 3 guarantee detection of those failures and, when a failed node recovers, the merge procedure will bring its relations up to date and will also set the status of any relation.

Figure 4.1: CPL strategy

Rule II

A node should not look at its message buffer between the time a NAT update is fully ACKed and the time it is actually committed.

Rule I guarantees that when receiving an ACK from node j, for example, all messages from j with old NSQ would have been received and that consequently new messages from j will have the new NSQ.

It is also possible that, for example, node i commits the NAT update earlier than j and right away sends a message to j. The message from i to j may reach j before j itself committed but, because of timing consideration, after all ACK's were received at j. This has to be the case because, if the NAT update was not fully ACKed, j would have sent stop messages which i would have received since i can communicate with j. However since the message has a new NSQ and j's NAT does not, a merge procedure may accidentally and erroneously be triggered. Rule II prevents that situation from occurring.

4.5.2 Special change of status

Remotely initiated NATupdate (discussed in 4.5.1) may cause a change in status, from MAJ to MIN, for some relations in the R(U) of a given update operation. This needs not be of concern however because:

1. if it happens after the CPL then, even though the TU's may not have yet been committed, assumptions about the behaviour of nodes guarantee that they will (either if the node is isolated or while the node is recovering).
2. if it happens during CPL, then the affected relations will have their status set to inconsistent just as though some failures had been detected by the algorithm itself. In actual fact, the algorithm would have eventually detected those failures.

4.5.3 Cost considerations in CPL

In CPL, the cost of the layer has several components:

1. All the TP's of a given T are forced to hold onto their allocated resources until such time as the updating can take place. Doing it any other way, for example, during the execution of a transaction, presents serious problems. In fact, if failures occur during the execution of a transaction it may happen that the transaction has to be aborted. If some TU's had already been committed, their updating in the DDB has to be rolled back. Furthermore, additional failures complicate the situation quite substantially. It would appear that the re-grouping of TP's before entering CPL makes the task of safeguarding mutual consistency easier.
2. The cost in messages is dependent upon the frequency and severity of the failures. The number of messages required in CPL (abbreviated CPL_{mssg}) is given by

$$\text{CPLmsg} = (|N(U)|^2 - |N(U)|) + \sum_{f=1}^F \{ a_f |N(U)| + b_f (N^2 - N) + c_f (|N(U)|^2 - |N(U)|) \}$$

In a no-failure situation, the number of CPL messages would be only $(|N(U)|^2 - |N(U)|)$, where $|N(U)|$ is the number of nodes involved in a given update operation. In the presence of failures, the number of CPL messages will obviously increase since it becomes necessary to send stop messages, to update the NAT and to try to synchronize again, albeit with a reduced $N(U)$. The summation part of CPLmsg evaluates the number of messages so required; the variable f represents each individual failure, with F as being the total number of failures that afflicted a particular operation. The other terms in the summation are defined as follows:

$a_f |N(U)|$ represents how many stop messages were sent in response to the occurrence of a particular failure f ($a_f < 1$).

$b_f (N^2 - N)$ $N^2 - N$ represent the cost in messages of a NATupdate with N being the number of nodes in the network. The coefficient b_f , $b_f < 1$, is used to take into account the fact that not N nodes will participate in the NATupdate (i.e. because of failures, subnetworks, etc...).

$c_f (|N(U)|^2 - |N(U)|)$ represents the number of messages ~~required~~ in the new commit or abort operation. $(|N(U)|^2 - |N(U)|)$ is the original cost reduced by c_f , $c_f < 1$.

3. Another component of the cost is the wasted processing time spent on the execution of those NATupdate and Resynch procedures. Associated with the processing time is the cost of holding resources (i.e. relations) while CPL tries to commit the TU's. Those costs are difficult to ascertain because they are dependent upon the time required to complete CPL.

In summary, the cost of CPL is high but, on the other hand, CPL provides mutual consistency in a failure prone environment. The basic question is whether to protect against failures or not. Once that decision has been made, the level of protection desired determines the cost incurred.

Figure 4.1: CPL strategy

Figure 4.2: CPL access graph

{Release and Update Table}

type

requirement: boolean;
timefield, timestamp: integer;

update= record

identity: integer;
timestamp: integer;
targetrelation: array[1..M] of requirement;
targetnode: array[1..N] of requirement;
end

relation= record

identity: integer;
timefield: integer;
status: (MAJ, MIN, merging, inconsistent);
end

releaseupdatetable=record

resource: array[1..M] of relation;
entry: array[1..P] of update;
ackcount: array[1..P] of integer;
fullyACKed: array[1..P] of boolean
end

Figure 4.3: Release and Update Table.

RTU= Ready to Update

Figure 4.4: Modified 2-phase commit protocol

RTU: Ready to Update

TO: Time-out

EW: Extra wait

Figure 4.5: Complete protocol

{Network Access Table}

type

state= (up, down, merging, subnetworkmerging);

sequencenumber=

record

clockvalue: integer;

pid: integer; {pid= processor id}

end

networkaccesstable=

record

NSQ: sequencenumber;

node: array[1..N] of integer;

status: array[1..N] of state;

LSQ: array[1..N] of sequencenumber

end

Figure 4.6: Network Access Table

S-nodeCommit (TN: target nodes; TR: target relations;
TU: target updates)

{TN contains all the relevant up nodes as
well as merging or subnetwork merge nodes}

begin

{RTU= Ready To Update message}

CLflag:=true

{disable timer on TU in TUD; see chapter 3}

TS:= timestamp(update)

RTU:= (update, TN, TR, TU, TS)

send (RTU, TN)

set-timer

if all RTU's properly delivered

then

wait for ACK's from all other nodes in TN

~~wait~~ till the end of the time-out period

wait extra time delay

if any stop(s) received

then

NATupdate(TN)

TN:= TN $\cap$ NAT

Resynch(TN, TR, TU)

else

commit (TR, TU)

updatetable (RUT, TR, TS)

fi

else

SN:= nodes to which messages could not be
delivered

stop:= (stop, TN, TR, U, TS, SN)

send (stop, TN)

NATupdate(TN)

TN:= TN $\cap$ NAT

Resynch(TN, TR, TU)

fi

end

Figure 4.7: S-nodeCommit procedure

D-nodeCommit

{Parameters are supplied by the RTU message}

```

begin
  RTU:= receivemessage
        {RTU= (update, TN, TR, TU, TS)}
  CLflag:=true
        {disable timer on TU in TUD; see chapter 3}
  set-timer
  ACK:= (ACK, U, TS)
  send (ACK, TN)
  if all ACK's properly delivered
  then
    wait for ACK's from all other nodes in TN
      {except from S-node}
    wait till the end of the time-out period
    wait for extra time delay
    if any stop(s) received
    then
      NATupdate(TN)
      TN:= TN  $\cap$  NAT
      Resynch(TN, TR, TU)
    else
      commit (TR, TU)
      updatetable (RUT, TR, TS)

      {Should it occur that some TU's have
       been deleted by the misfiring of
       timer protection in TUD, the node
       should set the status of the affected
       relations to INC, carry out the update
       and then restore those relations
       through possible use of the merge}

    fi
  else
    SN:= nodes to which messages could not be
          delivered
    stop:= (stop, TN, TR, U, TS, SN)
    send (stop, TN)
    NATupdate(TN)
    TN:= TN  $\cap$  NAT
    Resynch(TN, TR, TU)
  fi
end

```

Figure 4.8: D-nodeCommit procedure

Time-out (S-node/D-nodeCommit)

```

begin {time-out on wait for ACK
      for both S and D-nodeCommit}
  SN:= nodes from which ACK was not received
  stop:= (stop, TN, TR, U, TS, SN)
  send (stop, TN)
  NATupdate(TN)
  TN:= TN  $\cap$  NAT
  Resynch(TN, TR, TU)
end

```

Figure 4.9: time-out procedure for S-D-nodeCommit

3 3

OF / DE

Figure 4.10: Exchange of messages in protocol

NATupdate (PTN: target nodes of previous operation)

```

begin
    Wait for all possible stop messages
    SN:= union of SN's of all stop messages
    X:= [PTN-SN]
    pick a sequencenumber (X)
    if first {i.e. sequence number= 1}
    then
        S-node*NATupdate((NAT-SN), SN)
    else
        set timer according to sequence number
        wait for beginning of interaction
        D-node*NATupdate
        {D-node*NATupdate has no arguments since
         they are supplied in RTU*NATupdate message}
    fi
end

```

Time-out procedure (wait in else part of if)

```

begin
    SN:=SN° + (all previous nodes in X)
    S-node*NATupdate((NAT-SN), SN)
end

```

Figure 4.11: NATupdate and time-out procedure

```

S-node*NATupdate (TN: target nodes; SN: target update)
{SN is the set of suspected nodes and i is the
 S-node}

begin
    tentativeNSQ.clockvalue := NAT[i].NSQ.clockvalue + 1
    tentativeNSQ.pid := i
    RTU*NATupdate := (i, SN, TN, tentativeNSQ, ...)
    send (RTU*NATupdate, TN)
    wait for a pre-determined amount of time
        {in order to allow other RTU*NATupdate messages
         (if any) to arrive}
    if other RTU*NATupdate(s) were received
    then
        combine operations
        SN := union of SN and SN's of other RTU*NATupdates
        TN := TN - SN
    fi
    set timer
    if all RTU*NATupdate(s) for which destination site
        is still in TN have been delivered successfully
    then
        wait for ACK's from all other nodes in TN
        wait till the end of the time-out period
        wait for extra time delay
        if stop(s) were received
        then
            NATupdate(TN)
        else
            with RTU*NATupdate do
                NAT[i].NSQ := tentativeNSQ
                NAT[i].LSQ[tentativeNSQ.pid] := tentativeNSQ
                {in case of more than one S-node, perform
                 similar step on their respective LSQ's}
                updatestatus (NAT, SN)
            endwith
            re-evaluate status of relations in RUT
        fi
    else
        SN := SN + unreachable nodes
        stop := (stop, TN, NAT, U, tentativeNSQ, SN)
        send (stop, TN)
        NATupdate(TN)
    fi
end

```

Figure 4.12: S-node*NATupdate procedure

D-node*NATupdate

{Arguments for this procedure are supplied in the RTU*NATupdate message.
 SN and TN are therefore those of the initiating S-node(s) and may not be the same as those generated by the node in the first phase of the NATupdate;
 j is the D-node}

begin

RTU*NATupdate := receivemessage
 wait for a pre-determined amount of time
 {in order to allow other RTU*NATupdate(s) to arrive (if any)}
 if other RTU*NATupdate(s) were received
 then
 combine operations
 SN := union of SN and SN's of other RTU*NATupdates
 TN := TN $\cup$ SN

fi

set timer
 ACK := (ACK, NATupdate, tentativeNSQ, (others if any))
 send(ACK, TN)
 if all ACK's successfully delivered
 then

 wait for ACK's from all other nodes in TN
 {except S-node}
 wait till the end of the time-out period
 wait for extra time delay
 if stop(s) were received
 then

 NATupdate(TN)

else

 with RTU*NATupdate ~~do~~

 NAT[j].NSQ := tentativeNSQ
 NAT[j].LSQ[tentativeNSQ.pid] := tentativeNSQ
 {in case of more than one S-node perform
 similar steps on their respective LSQ's}
 updatestatus(NAT, SN)

 endwith

 re-evaluate status of relations in RUT

fi

else

 SN := SN + unreachable nodes
 stop := (stop, TN, NAT, U, tentativeNSQ, SN)
 send(stop, TN)
 NATupdate(TN)

fi

end

Figure 4.13: D-node*NATupdate

Time-out procedure(wait for ACK's in S/D-node*NATupdate)

```

begin
  SN:= SN + nodes that did not respond
  stop:= (stop, TN, NAT, U, tentativeNSQ, SN)
  send (stop, TN)
  NATupdate(TN)
end

```

Procedure re-evaluate status of relations

{used in S-node/D-node*NATupdate}

```

begin
  if NATupdate downgraded node status
  then
    for all r's with r.status= MAJ do
      if r.status<> MAJ in new NAT
      then
        r.status:= MIN
      fi
    od
  fi
end

```

Figure 4.14: Time-out procedure for NATupdate and
re-evaluate status of relations
procedure for S-node/D-node*NATupdate

Resynch (TN: target nodes; TR: target relations;
TU: target update)

```

begin
  pick a sequence number from TN
  if first
  then
    if update possible
      {all the relations in TR and all the
       relations in the read-set of the
       transaction that originated the update
       should have a status of MAJ for the
       update to be possible}
    then
      S-nodeCommit(TN, TR, TU)
    else
      S-nodeCommit(TN, AT, U)
      {AT is the abort table. Entering U in the AT
       does in fact abort the operation}
      if for any r's in R(U), r.status=MIN,
        because of NATupdate
      then
        r.status:= inconsistent
      fi
    fi
  else
    set timer according to sequence number
    wait for beginning of interaction
    {interaction may be update of R(U)
     or Abort of U}
    if interaction= update of R(U)
    then
      D-nodeCommit
    else
      {interaction= Abort U}
      D-nodeCommit {on the Abort table}
      if for any r's in R(U), r.status=MIN
        because of NATupdate
      then
        r.status:= inconsistent
      fi
    fi
  fi
end

```

Figure 4.15: Resynch procedure

```

begin {time-out procedure; wait
      in else part of 1st if}
  SN:= previous nodes in the sequence
  resynchmssg:= (SN, ...)
  send (resynchmssg, TN)
  S-node*NATupdate((TN-SN), SN)
  TN:= TN  $\cap$  NAT
  Resynch(TN, TR, TU)
end

```

```

begin {exception procedure when a
      resynch message is received}
  resynchmssg:= receivemessage
  D-node*NATupdate
  TN:= TN  $\cap$  NAT
  Resynch(TN, TR, TU)
end

```

Figure 4.16: Resynch time-out procedure
and exception procedure

Figure 4.17: FSM of S-nodeCommit procedure

Figure 4.18: FSM of D-nodeCommit procedure

Figure 4.19: FSII of NATupdate and Resynch procedures

CHAPTER 5

The Merge Procedure

5.1 Introduction

5.1.1 Purpose of merge procedure

In the description of the system so far, nodes which had failed were removed from the NAT. There comes a time however when such failing nodes are repaired or when isolated subnetworks are re-connected. For any given recovering site trying to join the network, control data structures as well as local database relations have to be brought up to date with those of the rest of the network. This is in essence the purpose of the merge procedure and is independent of the failure duration of a particular node.

5.1.2 Single node vs subnetwork merge

The merge procedure should be able to handle the case of a node that crashed but is now recovering and subnetworks being re-connected. The single node merge procedure (SNM), presented in section 5.4, deals with nodes that failed or were isolated. For these nodes, no distributed activity took place while the failure conditions prevailed.

On the other hand, the case of subnetworks differs substantially. It should be noted however that, strictly speaking, a node n_i in N_{ui} can be re-connected into N_{uj} even though the fact that n_i belonged to N_{ui} was not detected. Such a forced re-connection through the use of the merge procedure of section 5.4 entails:

1. the failure of any in progress distributed operations involving n_i in N_{ui} .
2. the eventual and disorderly re-connection of nodes in N_{ui} into N_{uj} when present or future operations in N_{ui} involving n_i fail. The NAT update operation triggered in N_{ui} would involve n_i which, being in N_{uj} , would absorb all these nodes into N_{uj} .

The forced re-connection of n_i into N_{uj} would perform

poorly with respect to a subnetwork merge procedure. Under such a procedure:

1. the total merge of Nui and Nuj would have been expedited through the use of the respective NAT's.
2. DR's common to both Nui and Nuj could have been restored quicker because of the extra information available.
3. the failure of distributed operations in either Nui or Nuj would have been averted.
4. the total processing efforts would have been minimized.

These facts emphasize the need for the merge procedure to be able to distinguish between failure/isolation and subnetwork.

5.2 Literature Survey

The task of re-inserting a node into a system or re-connecting subnetworks is very much dependent upon the control algorithms used by the DDB. Even then, not too much attention has been paid to the re-insertion and the

re-connection procedures.

Menasce [MENA80a] introduces a centralized concurrency control mechanism in which the tasks of rebuilding relations and synchronizing control data structures are relatively easy since a single reference is used as the central controller.

In SDD-1 [HAMM80], distributed control is used and relations are brought up to date by having the rest of the network send to the recovering node the messages it missed while it was down. The case of subnetworks is not covered and furthermore, if some messages are missing (*), an SDD-1 catastrophe is said to have happened. The reliability mechanisms of SDD-1 do not cater to catastrophes.

5.3 Recovery Detection

The detection of a recovering node is done primarily by inspecting the NSQ appended to every message. When the NSQ of the NAT maintained by a node does not coincide with the NSQ appended to an incoming message, an abnormal situation has been detected. In such a case, CSS which is responsible for detection will trigger the Merge Process (MP) (see

(*) A situation brought about by too many failures.

figure 1.5)). The pseudo-code description of MP is to be found in figure 5.1. The merge process serves two purposes:

1. to interact with a recovering node which requests information to bring its relations up to date,
2. to take over control of a recovering site and merge it properly.

In the latter case, MP first runs a detection algorithm to ascertain if a single node merge or subnetwork merge will be executed. An algorithm to determine the status of a pair of nodes which has detected an abnormal situation is presented in pseudo-code in figure 5.2. The meaning of all the symbols used in the algorithm is shown in figure 5.3.

The detection algorithm does have its limitations as in certain situations it will not detect the presence of subnetworks. The limitation is due to the fact that the NAT is updated only if a distributed operation fails.

Messages are relied upon as the medium through which detection is carried out. However, detection is dependent upon the amount of traffic among nodes and moreover, the rules of the protocol prevent up nodes from sending messages to nodes believed to be down. In order to speed up detection and to circumvent deadlock occurring

possibilities, nodes will send "Inquire" messages at regular intervals to all other nodes marked "down" in the NAT (carried out by CSS).

5.4 Specification of Single Node Merge

5.4.1 Introduction to SNM

Once a node has been determined to have crashed, two options are open for its merging with the rest of the network. The node could be brought up to date using a system wide freeze. This approach entails a very high cost since it forces the operational part of the network to stop and help the node join in. On the other hand, the merging could be done concurrently with normal system functions and in such a way as to minimize the interference to the operational network. The merge algorithm which will be presented, will adopt that latter strategy and will ensure that the burden of the merge is mainly on the recovering node(s).

The situation for which the merge algorithm will apply is one in which a node n_r has crashed previously but is now recovering. Through interaction with node n_k , (n_k belonging to the rest of the network) n_r discovered its previous

crash. There were no subnetworks involved.

The first step of the procedure involves updating the status of nr from "down" to "merging" which is done by nk on nr 's behalf. With a status of merging, nr is then in a position to participate in (although not initiate) update operations on local copies of DR 's. In other words, nr would have to participate fully in any commit operation where it is involved. A failure of nr would force the other nodes to downgrade its status from merging to down. The purpose of the status of merging is to enable nr to be aware of ongoing updating activities while it tries to bring its own DB up to date. From that point onwards, there exist two parallel streams of activities within nr .

1. One task is concerned with storing the updates for the latter part of the procedure. If, for a given update operation, the status of a relation in $R(U)$ is merging, then the update will be stored together with its timestamp. The stored updates will be used when the status of these relations goes to "MAJ". If the status of the relations in $R(U)$ is "MAJ" then the update is committed in the usual fashion.

2. The purpose of the other task is to bring local

relations to a point where stored updates can be committed. This is achieved by grouping relations according to four status indications relevant to the new NAT (i.e. 1-inconsistent, 2-MIN and staying MIN, 3-MIN but going MAJ, 4-MAJ) and observing that no distributed updating operation will affect (*) status 1, 2 and 3. The merge procedure will try to bring all relations up to date to a "MAJ" status.

Potential problems due to several nodes executing the merge procedure concurrently are avoided by allowing only one node at a time to execute the procedure.

5.4.2 Single node merge

The single node merge procedure is divided into two parts: namely CPL merge and SPL/EPL merge. The procedure "restore" used in the merge is to be found in figure 5.4.

(*) Except in the case of status 1 when other copies are "MAJ" in the network.

5.4.2.1 CPL merge

1. If possible, complete outstanding operations (possibly by retrieving messages from message buffer in stable storage). If unable to complete pending operations, set the status of involved relations to inconsistent. Ignore any other Inquire messages.
2. Node nr starts interacting with node nk so as to get the current NAT maintained by nk. Node nk will then start a NATupdate procedure to bring the status of nr from "down" to "merging". This will be done when no other node has a status of merging and following the NATupdate, nk will report back to nr on the success of the operation. Any updating activities on the NAT which may have taken place in the meantime will also be included in the reply from nk to nr. Node nr therefore has a valid NAT which will be kept up to date because nr is now merging. This interaction between nr and nk should be protected by a timer. In case nk fails, nr could wait for another Inquire message or try to contact other nodes in order to start merging again.

3. The various data structures are re-initialized but the timefields of all the relations are kept. A status of inconsistent is never erased. Using the new NAT obtained in step 2, nr obtains RSr,
- $$RSr := (r \in RSr \text{ if, } \forall r, r \in R(nr), \\ r \in R(nr) \cap R(\text{network})).$$

4. With RSr four recovery sets are formed:

RS1 = {relations with a status of inconsistent}

RS2 = {relations which, according to the new NAT plus recovering node counted as "up", will be "MIN"}

RS3 = {relations which, according to the new NAT plus recovering node counted as "up", will go from "MIN" to "MAJ"}

RS4 = {relations which should be "MAJ" in the new NAT}

RSr = RS1 U RS2 U RS3 U RS4. Set the status of relations in RS2, RS3 and RS4 to merging.

5. With RS1

repeat

r := pickarelation (RS1)

if majority of copies of r with MAJ status exists

then

restore (r, RS0)

else

if all copies of r are available

then

restore (r, RS1)

else

r.status := inconsistent

fi

until fi all r's in RS1 have been processed

6. With RS2

repeat

r := pickarelation (RS2)

r.status := MIN

until all relations in RS2 have been processed

7. With RS3

repeat

r := pickarelation (RS3)

if r.status = inconsistent elsewhere
{or in fact if one or more sites is (are)
unresponsive}

then

r.status := MIN

else

restore (r, RS3)

fi

until all relations in RS3 have been processed

8. With RS4

repeat

r := pickarelation (RS4)

if majority of copies of r
with MAJ status exists

then

restore (r, RS4)

else

if (majority - 1) of copies of r
with MIN status exists

then

restore (r, RS3)

else

if all copies of r available

then

restore (r, RS1)

else

r.status := MIN

fi

fi

until fi all relations in RS4 have been processed

9.

for each relation with $r.status = merging$ do
 commit stored update(s) from the timefield
 of r to timestamp of latest update
 $r.status := MAJ$

od

10. When $status \neq merging$ for all relations in RUT then
 nr initiates a NATupdate operation to take its
 status from merging to up.

5.1.2.2 SPL/EPL merge

1. The global co-ordinator at nr , GCr, responds to any
 attempts to synchronization with a rejection.
 (i.e. updating the timestamp will follow). Also
 for all relations in both the RUT and RAT, the time
 fields (TF) of relations in RUT are copied over to
 those of the RAT.

2. Concurrently with step 1, GCr prepares a
 transaction Tr whose purpose will be to
 re-synchronize the time fields in the RAT with
 those of the RUT. Tr is characterized as follows:

a. $TS(Tr)$, timestamp

b. $R(Tr) = R(nr)$ with a further qualification

$ri \in R(Tr)$ if $ri \in R(nr)$ and $ri.status = MAJ$

c. $N(Tr) = \text{all } ni's \mid R(Tr) \cap R(ni) \neq \emptyset$.

3. Tr is a null operation whose purpose is to set the timefields of all relations in $R(Tr)$ to $TS(Tr)$. The execute phase of Tr is only needed to allow the transaction processes (of Tr) to regroup. Then the STP at nr initiates:

S-nodeCommit ($N(Tr)$, $R(Tr)$, $\emptyset$).

4. When Tr is fullyACKed, GCr updates the timefields of all the relations in the RAT and then starts functioning normally, that is not rejecting transactions.

5.5 Validation of Single Node Merge

The correctness of the single node merge procedure will be proved by four assertions which cover the operations of both parts of the SNM.

Assertion 5.5.1

The SNM is free from deadlock.

Proof of Assertion 5.5.1

The only deadlock causing steps in the procedures are the ones in which the recovering node, nr, interacts with other up-nodes. Only in step 2 is nr entirely dependent upon nk for updating nr's status to merging. This step is therefore protected by a timer. Upon timing out nr would wait for contact from another node in the form of an Inquire message. Eventual completion of the single node merge is treated in assertion 5.5.4..

During other steps; namely step 5, 6, 7, 8, nr would respond to failure of other nodes by switching to nodes with copies of the relation currently being recovered. The status of the relation may be affected by the switch. There is therefore no deadlock in those steps.

Finally when nr participates in NATupdate operations, deadlock cannot occur by definition of those operations.

Assertion 5.5.2

The single node merge procedure functions correctly.

Proof of Assertion 5.5.2

The first part of the SNM alters the status of the recovering node, nr, and the status of the copies of the relations which reside at nr. The status of nr goes from down to merging at the beginning of part 1 and then goes from merging to up at the end of part 1. The correctness of those two changes is guaranteed by the NATupdate triggered by nk and nr respectively.

The significance of a status of merging is that nr is included as a full participant (D-node) in all operations of the CPL in which $R(U) \cap R(nr)$. However, nr does not initiate any operation, be it in the CPL or in the SPL. Updates resulting from CPL operations are stored if the status of relations in the target relation set is "merging". The status of merging for a copy of a relation is therefore an indication that the copy is still in the process of being recovered.

Besides upgrading the status of nr, SNM alters the

status of relations in steps 5, 6, 7 and 8. Apart from relations with a status of inconsistent which are treated differently (step 5), all other relations see their status set to merging and see themselves being classified into recovery sets RS2, RS3 and RS4.

1. Relations with a prior status of inconsistent retain that status and are classified into RS1. For such relations, the upgrading of their status from inconsistent to MAJ (or merging) (*) will only take place if either all the other copies of the DR are available or if there exists a majority of copies each with a status of MAJ. Any other combination does not allow the status of inconsistent to be removed.
2. Relations in RS2 are those for which, according to system directory and the new NAT (obtained in step 2), a majority of copies is not available (with any kind of status). Since those relations do not have a status of inconsistent, they should be tagged with a status of MIN and should be restored to MAJ as other nodes join in.

(*) Which is the only possibility if one remembers that the purpose of a status of inconsistent is to remove logically a relation from the system.

3. If, however, according to system directory and the new NAT, a copy of a relation would tip the balance in favour of a MAJ status, then the relation is incorporated into RS3. During the execution of step 7, if there is indeed a number of copies (equal to majority-1) with status of MIN, then the SNM will restore them all to a status of MAJ directly. This is possible since there was no activity on these before. Otherwise if some remote copies have a status of inconsistent, the status of the local copy is set to MIN.

4. Finally, relations in RS4 are those for which it is believed there exists a majority of copies in the system. Three cases are of interest:

a. There exists indeed a majority of copies, each with a status of MAJ. In that case the local copy is brought up to date (as much as possible, keeping in mind that CPL activity is going on) and its status set to merging.

b. Some remote copies have a status of inconsistent but there exist (majority-1) copies of the relation, each with a status of MIN. This case is similar to that of relations

in RS3 and is dealt with accordingly.

- c. Some remote copies have a status of inconsistent and not enough copies with a status of MIN exist so that the DR can be recovered. In such a case the status of the local copy is set to MIN.

At the end of this step, (step 8), the relations that still have a status of merging are those for which a majority of copies with a status of MAJ existed elsewhere in the network. Since CPL activities may be taking place, such relations are kept waiting up until step 9, during which the stored updates are committed and the status of those relations brought to MAJ. It is important to realize that, after having committed the updates and despite the fact that the status of the node is still merging, the value of those relations is mutually consistent with the other remote copies.

The last point in the proof of part 1 of SNM concerns the inconsistent status and the way in which other nodes can update the status of their relations. As can be seen from the S-node/D-node*NATupdate procedures (chapter 4, figure 4.12 and 4.13), the status of relations can only be downgraded during a NATupdate. On the other hand, upgrading

the status of these relations only takes place during the merge procedure. Consequently, a status of inconsistent does not have to be propagated in order to remove effectively a given copy of a relation from the system.

The second part of the SNM is concerned with synchronizing the timefields of the relations in the RAT with those of the RUT. The situation at the beginning of this second part is that the RUT at nr is synchronized with the RUT of the other nodes in the same subnetwork. By synchronized, it is meant that, for a given logical instant, the relations have similar status and timefields, and, should any changes take place in the network, the relations will change in a mutually consistent fashion.

However, even though nr has been successfully re-inserted into the CPL, there still exist potential sources of inconsistencies in the system. These are caused mainly by transactions which originated remotely and were synchronized (SPL) before nr recovered and that are still in progress at other sites (i.e. executing). This difficulty is illustrated in figure 5.5 which shows two nodes, nk and nr , with nr having just been successfully re-inserted in the CPL. Relation $r5$ is singled out for the example and thus appear in the RAT of both nodes. There may exist transactions, TW's, with $r5$ in their write-set either in the

RAT of nk (and possibly others) still unacknowledged and having originated before nr went up or in the DLRQ of nk (and others).

If a transaction Tx originating from ns , with $r5 \in R(ns)$, arrives at nr just after the status of nr has been set to up, then it would be allowed to run through and could possibly be allowed to read $r5$ before the writes of transactions TW 's. However, if Tx had been directed at nk , $r5$ would have been read properly, hence a read with location dependency. This situation has to be protected against; two solutions are possible:

1. The first approach which could be taken is to merge separately the EPL (a difficult task because of the loosely distributed nature of EPL) and then to merge the SPL. That last operation (i.e. SPL merge) would in fact only involve ensuring mutual consistency among the RAT's.
2. The second approach, which is more practical and easier to implement, is to reject any transactions up until a specific recovery transaction, Tr , is fully acknowledged and then to proceed normally. The purpose of Tr is to establish a time bench mark which, at nr , would force any remotely originated

transaction to be synchronized after Tr. Since Tr is a write-only (with null, write) transaction, it will be forced to regroup all its components (i.e. TP's) before entering CPL. This will delay any other transactions until such time as all the other transactions ahead of Tr in the DLRQ of all other sites with $R(n1) \wedge R(nr)$ have executed and committed. Subsequent transactions will therefore be synchronized properly with respect to that synchronization bench mark.

This completes the proof of assertion 5.5.2.

Assertion 5.5.3

The single node merge is robust, that is to say resistant to failures to a certain degree, and, if it has to be abnormally terminated, will not introduce inconsistencies in the system.

Proof of Assertion 5.5.3

The robustness property of the SNM will be proved separately for part 1 and part 2 since those two parts have different functions.

In part 1, the SNM is robust because of the four following observations.

Observation I

Should nr fail at any time during the execution of the SNM, its status will be brought back to down during a NATupdate triggered by remote nodes which detected the failure while interacting with nr. Upon recovering, nr would presumably be detected and forced to re-start the single node merge. The consistency of DR's is therefore maintained.

Observation II

During the first NATupdate procedure which is initiated by nk on nr's behalf, three cases have to be considered:

1. Case 1

A failure of nk during the NATupdate would result in the status of nk being downgraded to down and the operation re-started to finally upgrade the status of nr. If all the nodes in the subnetwork fail, then the status of nr goes to merging in a

one node subnetwork (but there is little point in merging!).

2. Case 2

A failure of nr would result in the status of nr staying down. The merge procedure will be re-started at nr when nr recovers.

3. Case 3

Failures of nodes in the subnetwork other than nr and nk simply results in their status being set to down. There are no adverse effects from this situation except, perhaps, that the number of nodes with which nr can interact in order to restore its relations is diminished.

Observation III

This third point is concerned with failures of nr, nk and possibly other nodes when nr is performing recovery on its relations (i.e. steps 5, 6, 7 and 8 of SNM). In this respect, four separate cases have to be considered:

1. Case 1

For relation r at nr being added to an already MAJ

situation, two types of failures have to be considered. Firstly, nr could fail during the restoring of its copy of r which is, in itself, of no consequence to the rest of the network. Secondly, failures of other nodes, whether the MAJ status is changed or not do not endanger consistency since:

- a. if the status is not changed then the former situation prevails and,
- b. if the status changed, then either a NATupdate will be triggered or any activity not triggering a NATupdate can be taken as having happened before the failure.

2. Case 2

For relation r at nr being rebuilt during SNM, a failed broadcast of up-to-date copies to other nodes may be the source of inconsistencies. However, it need not be so because of the assumption made about the atomicity of broadcast which says that all messages in a given broadcast are sent at the same instant. Looking at the situation with this assumption in mind, one sees that:

a. if nr fails, it is either before or after, a broadcast in which case the new copy is obtained by no one or by every one,

b. if a node (not nr) fails and does not receive its copy of the relation (i.e. in strict terms the broadcast failed) then its failure will be detected later and trigger a NATupdate followed by a re-evaluation of relation status.

3. Case 3

After a relation r , at nr , is restored to MAJ status, activity in the CPL on r is possible even though nr is still merging. During that period (i.e. up until $nr.status=up$), nr , nk or other nodes may fail, thus making the MAJ status of r invalid. When those failures are detected, a NATupdate is triggered which takes care of re-evaluating the status of all relations. On the other hand, there exists a period of uncertainty which covers the time from the occurrence of the failure(s) to the NATupdate. During that period of time, transactions with r in their read-set could execute, leading to two cases:

a. The first case is that the transactions are

read-only, or if they are read-write, the write is only among operational sites. Therefore any operations performed by those transactions can be taken as having happened before the failure since they did not trigger a NATupdate.

- b. The second case is that of transactions with write-sets covering down sites. Their execution will therefore trigger a NATupdate and subsequent re-evaluation of the status of all relations.

4. Case 4

The last case to be considered is that of failure of a node which happens to affect the recovery of relations not yet processed by SNM. An unnoticed failure is picked up by the algorithm when it queries the relevant nodes. A failure that is detected forces *nr* to re-evaluate some of its recovery sets and also modifies the status a relation will eventually get.

Observation IV

The last point in the proof of part 1 concerns possible failure in the NATupdate initiated by nr to take its status from merging to up. If nr fails then it obviously has to re-do the SNM; failures of other nodes will result in a downgrading of their status. This completes the four observations.

The second part of SNM is basically a transaction, Tr, which originates from nr. Failures will affect the complete merging of nr in two ways:

1. If nr itself fails then Tr will either abort or execute elsewhere. In either case, no damage is done to the system.
2. If other nodes in $N(Tr)$ fail then Tr will go through in the nodes that remain while transactions directed at nr are delayed. The status of the failed nodes will eventually be set to down by a NATupdate and in the meantime those nodes are assumed not to recover (*).

Furthermore it is worthwhile mentioning that the

(*) Another alternative would be to trigger a NATupdate for those nodes.

synchronization protocol layer will provide basic failure protection to nr when nr is submitting Tr . This completes the proof of assertion 5.5.3.

Assertion 5.5.4

The SNM will eventually terminate.

Proof of Assertion 5.5.4

The proof of this assertion bears strong resemblance to a similar assertion concerning the eventual termination of the protocol of the CPL. In both cases, it rests on an assumption made earlier that, at certain time in the network, there will be periods of time during which no failures will occur. Given that this assumption is true, the SNM will terminate successfully.

5.6 Subnetwork Merge (SM)

5.6.1 Options for SM

The task of re-connecting operational subnetworks is more complicated than SNM. Before specifying the SM, three strategies for subnetwork merge will be presented, each with its own level of complexity and its cost.

Option I

Given that two subnetworks, Nu1 and Nu2, became aware of each other's existence because of re-connection, the following actions are executed:

1. Update NAT on both sides
2. Determine direction of merge,
(source-->destination)
3. Elect controller in source Nu and merge source Nu nodes one at a time using SNM algorithm.
4. Controller is last to join in destination Nu.

This is a useful strategy when the source subnetwork is

small and the destination subnetwork is large since it leaves the latter undisturbed. Its complexity is also reasonable.

Option II

Two subnetworks, Nu1 and Nu2, detected each other's presence and updated their NAT reciprocally.

1. Nu1 and Nu2 finish up the outstanding transactions but accept no more work. They will eventually both grind to a halt (freeze).
2. A unique recovery controller in Nu1 U Nu2 is chosen and its job is to co-ordinate the recovery of relations.
3. When the relations of Nu1 and Nu2 have been brought up to date, the controller updates the NAT so that it reflects the new situation (i.e. Nu1 U Nu2).

Option II is basically a centrally controlled recovery. As such, it is vulnerable to failure of the controller but on the other hand is simple to implement and relatively economical of messages.

Option III

Two subnetworks, Nu1 and Nu2, detected each other's presence and updated their NAT so that, for the nodes in Nu1, the status of the nodes in their NAT will be:

$$\forall ni \mid ni \in Nu1, ni.status = up$$

$$\forall nj \mid nj \in Nu2, nj.status = subnetworkmerging$$

(and vice-versa for NAT of nodes in Nu2). Relations in both Nu's also have an extra data type in the form of a flag which can either be tentative or firm. In each Nu, nodes continue operations as before on relations with a status of MAJ. Any updates on those are also transmitted to nodes with special subnetworkmerging indicator (just as in SNM for nodes with a status of merging).

1. Relations are partitioned depending on their chances of upgrading their status and depending upon which node shall carry out their recovery.
2. Merge is done by all nodes having relations to restore.
3. After all merging relations have been restored, nodes upgrade their status in the NAT of the other subnetwork.

Option III is the most complicated but has definite advantages in term of efficiency and in not disrupting the

activities of the respective subnetworks. It also tries to minimize the work to be done by guessing at the end status of relations to be merged. Therefore, option III was chosen and is specified in section 5.6.2.

5.6.2 Specification of SM

{Procedure restricted to subnetworks Nu1 and Nu2; procedure executed by the merge process of node ni in Nu1.

The situation:

nk in Nu1 and nl in Nu2 have detected the re-connection of Nu1 and Nu2 and triggered a special NATupdate in their respective subnetworks; therefore the NAT of nodes in Nu1 shows all the nodes of Nu1 with up status and all the nodes of Nu2 with a special subnetworkmerging indicator (the situation is reversed in Nu2))

5.6.2.1 CPL subnetwork merge

1. $RSi := (r \in RSi \text{ if, } \forall r | r \in R(ni), r \in R(ni) \cap (R(Nu1) \cap R(Nu2)) \text{ and } r.status = INC \text{ or } MIN)$
2. $RSi' := (r \in RSi' \text{ if, } \forall r | r \in RSi, \text{ no. of copies of } r \text{ in } Nu1 \cup Nu2 \geq \lceil \text{no. of copies of } r / 2 \rceil + 1)$
3. $RSi'' := (r \in RSi'' \text{ if, } \forall r | r \in RSi' \text{ and } ni \in Nu1, \text{ } ni \text{ is the node with the lowest i.d. of all nodes with a copy of } r \text{ in } Nu1 \text{ and } Nu1 \text{ is the subnetwork with fewer copies of } r)$

4. $RS1i := (r \in RS1i \text{ if, } \forall r | r \in RS1i, r.\text{status} = \text{INC})$
 $RS2i := (r \in RS2i \text{ if, } \forall r | r \in RS1i, r.\text{status} = \text{MIN})$

```

5. for all r in RSi' do
    r.flag := tentative
od

```

```
6. for all r in RSi' do
    case r.status of
```

a) r.status could be MAJ in Nu2
ascertain if r.status = MAJ in Nu2
{i.e. if only one copy has a status
of MAJ then they all do}

```
if r.status= MAJ in Nu2
then
```

```

getcopy (r)
commit stored updates
    {in timestamp order}
r.status:= MAJ
broadcast(r, relevant sites in Nul)
r.flag:= firm

```

```

else
    execute case b)
fi

```

```

b) r.status could be MAJ in Nu1 U Nu2
   ascertain whether number of copies of r
   with r.status= MIN in Nu1 + number of
   copies of r with r.status= MIN in Nu2 >=
   [no. of copies of r / 2] +1
   if condition is true
   then

```

```
getlatestcopy (r)
    {with r.status= MIN only}
broadcast (r, relevant sites
           in Nul U Nu2)
r.status:= MAJ
r.flag:= firm
```

```

else
  if  $r \in RS11$  and all copies of  $r$  are in  $Nu1 \cup Nu2$ 
  then
    {all copies of  $r$  are present}.
    getlatestcopy( $r$ ).
    {with a status of MIN if any,
     otherwise with status of INC}
    broadcast( $r$ , relevant sites
              in  $Nu1 \cup Nu2$ )

```

in Nu1 U- (Nu2)


```

        r.status:= MAJ
        r.flag:= firm
    else
        mssg:= (r, done)
        send (mssg, to relevant sites
              in Nu1 U Nu2)
        r.flag:= firm
    fi
endcase
od

```

7. {concurrently with the rest of the algorithm}

```

mssg:= receivemessage
if mssg= update
    {update from a node recovering relation r}
    then
        commit update
        r.status:= MAJ
        r.flag:= firm
    else
        r.flag:= firm
    fi
fi

```

8. if $\forall r | r \in RS1$, $r.flag = firm$
then
 trigger NATupdate in Nu2 to take ni.status
 from SM to up
fi

5.6.2.2 SPL/EPL subnetwork merge

1. Reject any transaction T for which
 $R(T) \cap RS1' \neq \emptyset$.

2. Prepare a synchronizing transaction, STi , with

$R(ST) = RS1'$

$N(ST) = \text{all } ni's | ni \in Nu1 \cup Nu2$

and $R(ni) \cap RS1' \neq \emptyset$.

3. Synchronize ST

4. Accept transaction T, for which $R(T) \cap RS_i' \neq \emptyset$, if, $\forall r | r \in R(T) \cap RS_i'$, r has been synchronized by ST either from ni if $r \in RS_i''$ or from other nodes in $Nu1 \cup Nu2$. When all the relations in RS_i' have been synchronized by an ST then the subnetwork merge has been completed.

5.6.2.3 Exception Procedures (EP)

Exception Procedure I

A time-out at ni is possible if the subnetwork merge is protected by a timer. The cause of the time-out could be that, after ni has completed its work on the relations of RS_i'' , some relations in RS_i' (but not RS_i'') still have a flag of tentative. This situation may be indicative of a failure of a node in $Nu1 \cup Nu2$. The required actions in such a case would be:

1. Wait longer and/or query node so as to determine its status (the node could be

sluggish or extremely busy).

2. In case of failure, perform NATupdate to remove suspected node and choose either a replacement node or re-distribute the RS' of the failed node.

Exception Procedure II

A time-out at ni could also be caused by a synchronizing transaction which has not arrived yet.

Two options are possible:

1. Do nothing and do not accept transactions for the affected relations.
2. Query node or wait longer and then query node. If node has failed then trigger a NATupdate to remove it and choose a replacement node to carry out ST on those affected relations.

5.6.3 Validation of SM

The mechanisms used by the subnetwork merge (SM), namely: broadcast of relations, storing updates and committing them in timestamp order, treatment of failing nodes during merge and, use of synchronizing transactions are similar to those of the single node merge (SNM). For this reason, a complete validation of SM is not required; however, some points are worth emphasizing if not clarifying.

1. Only relations for which a possibility of recovery exists are of concern. For the other relations, they are either of a majority status and should not be interfered with or of a status such that not enough copies of them are available. Furthermore, work on the relations to be recovered is distributed among the recovering nodes of a given subnetwork. This avoids replication of tasks.

2. For a given node n_i , the completion of SM occurs when all the relations in RS_i have either been recovered by n_i and also broadcast to other nodes or a recovered version (*) has been received from the node in charge of the recovery of that relation. This feature is thus akin to full acknowledgement for the nodes sharing relations.
3. The completion of SM does not occur simultaneously throughout a subnetwork. However, for node n_i , when the completion condition is satisfied, the status of relations in $R(Nu1) \cap R(Nu2)$ (but not in $R(n_i)$) is irrelevant. Therefore n_i upgrades its status in the NAT of the other subnetwork.
4. The series of NATupdate operations which will take place does so on the combined NAT of the two subnetworks. Those two NAT's, however, are not the same (i.e. nodes of $Nu1$ have a status of up and those of $Nu2$ have a status of subnetwork merge in the NAT of $Nu1$ and

(*) or a done message on the relation indicating that several copies are inconsistent thus precluding its recovery.

vice-versa) and the purpose of the series of NATupdate is to make them similar.

5. Robustness is provided at the node level by timer protection on completion of merge and also on synchronization of synchronizing transactions.

Merge Process-

```

begin
  if Merge Process triggered by CSS
  then
    {local CSS has determined that the site
      had failed}
    execute detection algorithm
    if single node failure
    then
      execute single node merge
    else
      execute subnetwork merge
    fi
  else
    {the merge process is called upon to
      help a merging node}
    respond to queries from merging node
  fi
end

```

Figure 5.1: Merge Process

```

begin {detection algorithm}
  if pidi = pidj
  then
    if LCVi = LCVj
    then
      everything is OK
    else
      if LCVi > LCVj
      then
        i was always up
        j is recovering
      else
        i is recovering
        j was always up
      fi
    fi
  else
    {pidi <> pidj}
    if LCVi > LSQj[pidi] and
      LCVj > LSQi[pidj]
    then
      i and j were both operating
      in subnetworks of their own
    else
      if LCVi > LSQj[pidi] and
        LCVj <= LSQi[pidj]
      then
        i was operational
        j was down
      else
        if LCVi <= LSQj[pidi] and
          LCVj > LSQi[pidj]
        then
          i was down
          j was operational
        else
          other combinations
          indicate serious
          network troubles
        fi
      fi
    fi
  fi
end

```

Figure 5.2: Detection algorithm

Figure 5.3: Symbol of detection algorithm

Procedure restore (r: relation; code: condition)

begin

case code of

RS0: begin

r := getmajcopy (r)

if operation successful

then

r.status := merging

else

r.status := inconsistent

fi

end

RS1: begin

r := getlatestcopy (r, INC)

if operation successful

then

broadcast (r, relevant sites)

r.status := MAJ

else

r.status := inconsistent

fi

end

RS3: begin

r := getlatestcopy (r, MIN)

if operation successful

then

broadcast (r, relevant sites)

r.status := MAJ

else

r.status := MIN

fi

end

```

RS4: begin
      r := getmajcopy (r)
      if operation successful
      then
        r.status := merging
      else
        if there exist (majority-1) of copies
          of r with status <> inconsistent
        then
          r := getmajcopy (r)
          if operation successful
          then
            r.status := MAJ
          else
            r.status := MIN
          fi
        else
          r.status := MIN
        fi
      fi
    end
  endcase
end

```

Figure 5.4: Restore Procedure

Figure 5.5: Example of inconsistency during merge

CHAPTER 6

Conclusions and Recommendations for Further Study

6.1 Thesis Contributions

The problems associated with the co-ordination of activities in distributed databases were examined in the context of both reliable and unreliable systems. This thesis focused on the correctness and the robustness of interprocess co-ordination, and those criteria were satisfied by algorithms which were designed, specified and validated during the course of this work.

The correctness of DDB operations was ensured by a synchronization protocol based on an algorithm in [MAHM79]. This protocol was formally specified and validated and its software architecture was also defined.

Following that, a study of robustness in DDB operations was undertaken. Considerations were given to the algorithmic aspect of robustness and also to software

engineering requirements in designing those robust mechanisms. This investigation produced complete design, specification and validation of the following components:

1. A robust synchronization protocol layer,
2. An execute protocol layer, which guarantees robust execution of a transaction,
3. A commit protocol, which guarantees mutual consistency among local databases in the system and
4. Merge procedures, which are used to insert recovering nodes into the operational system or to join subnetworks.

Furthermore, the architecture of the system was also specified, as well as the relationships among system components.

6.2 Appraisal of the System

6.2.1 Summary of basic assumptions

The basic assumptions which were used throughout this thesis are listed below:

1. Message transit delays are upperbounded and are much greater than processing time.
2. The duration of the effects caused by the occurrence of a failure in the system is long as compared to processing of transactions.
3. The system is assumed to be reasonably busy, in terms of both distributed and local transactions. This is necessary in order to ensure proper detection of site failures.
4. Failure free periods during which operations can successfully complete will occur frequently.
5. Finally, nodes are assumed to be well behaved with respect to local actions and not to malfunction.

Those basic assumptions, as well as others made during the course of this work, are deemed to be realistic and representative of typical distributed systems.

6.2.2 System strategy

The originality of the proposed co-ordination mechanisms rests on the following characteristics:

1. A layered approach was used throughout to allow greater flexibility when it comes to transaction processing. A read-only (i.e. retrieve) transaction, for example, has to be synchronized and monitored, but does not have to enter CPL at all. The saving on overhead is substantial.
2. Distributed control was a feature of all the layers. Such a control scheme exhibits higher parallelism (i.e. greater concurrency) than centralized control schemes and is also inherently more robust.
3. Thanks to the robust protocol layers, the DDB will survive node crashes and network partitions. The reduced DDB will also continue to operate correctly even at a somewhat lower level of activity.
4. This network partitioning survivability property also encompasses the execution of a transaction. Special monitoring facilities exist at each site in the DDB so that the burden and the responsibility of monitoring do not weigh too heavily on individual TP's.
5. The DDB will be correctly rebuilt when failed nodes

recover or when subnetworks are re-connected. This is achieved by merge procedures which will perform concurrently with other system functions so as not to interfere with whatever DDB activity is possible.

It is also worth mentioning that the system can be made to accomodate other requirements. For example, the architecture and protocols of CPL can be modified to cater to file maintenance. This could be done by allowing the status of each relation to be manipulated by this file maintenance facility so that the status of corrupted relations would be set to INC. The merge procedure could then be called to restore the damaged relations.

6.2.3 Timer considerations

In all the algorithms that were described, it was assumed that the processing time required to perform desired actions was small when compared to the time spent by messages in the network. Furthermore, it was also assumed that a process would perform normally when it is activated, otherwise it is in a dormant state. In other words, malfunctions were not covered.

The setting of timers is very important in any robust system since the only way to detect the occurrence of a failure is by timing out on the non-occurrence of the desired event. To obtain time values with which to set timers, the following guidelines were followed:

1. The message transit delays were assumed to have an upper bound. This upper bound was used in estimating the maximum time that a message would take to reach its destination.
2. Actual timer values were obtained by analyzing the interaction of nodes in each protocol and by considering worst case situations in the exchange of messages.

It should be realized that the value of a timer, although not difficult to ascertain, greatly influences the behaviour of the system: too high a timer value would cause the system to operate inefficiently in the presence of failures; too small a timer value would lead to false alarms, i.e. the detection of imaginary failures.

The network topology is also of concern in determining the upper bound on message transit delays. In a loosely connected network, the upper bound on message transit

delays, taken on a node by node basis, could vary widely. Two choices could be made for setting timers: either choose the highest value or give flexibility to the algorithms to determine their timer values based on $N(T)$. The latter choice would produce faster response to failures at the expense of added complexity.

6.3 Recommendations for Further Study

This work has uncovered numerous interesting areas in the design of reliable and robust protocols in DDB. Some of these areas warrant further investigation:

1. Integrity constraints should be fully considered in the design of reliable protocols. A DDB operating without any integrity constraints has limited practical use.
2. Security is also another concern of DDB designers. Not only should DDB protocols be robust, they should also prevent unauthorized access by third parties.
3. The assumption covering malfunctions should be relaxed so that the system corresponds closely to practical situations. Malfunctions should be

linked with a study of integrity to yield robust and resilient protocols. This area presents a challenging research topic.

4. The reciprocal influence of robust protocols and the data network should be investigated in terms of the sensitivity of some solutions to the architecture, topology and services of the data network. In other words, the questions of how to design robust DDB's given a communication network and the reverse should be carefully examined.

APPENDIX A

Pseudo-Pascal Specification of SPL

A.1 Structure of the Appendix

The purpose of this appendix is to illustrate the differences that exist between the formal algorithmic specification of SPL and an actual implementation of the ADD synchronization algorithm. This appendix contains a formal specification in Pseudo-Pascal of the ADD synchronization algorithm and of the data structures it uses. The running of the synchronization algorithm is the responsibility of the global co-ordinator. Consequently the algorithm is described as a set of two concurrent processes:

1. the submission manager (A.5),
2. and the resource access table (RAT) manager (A.6).

Those two processes run concurrently and they co-ordinate

their activities and communicate between themselves through the resource access table. When processes operate in this fashion there usually arise problems if the processes are to be kept properly synchronized, that is, avoiding race conditions, and if the integrity and consistency of the data structures they jointly access are to be preserved. This can be guaranteed by defining "critical sections" in the execution of those processes so that:

1. access to a given structure is mutually exclusive
2. and that a process can execute a given series of instructions (i.e. a critical section) in that mode.

The algorithm presented here does not explicitly provide for that although considerations of that type would be extremely important in actual system design.

Certain functions and/or procedures appearing in the program have not been formally defined. They were left out either because of their dependence upon a given implementation or because they were not critical. Similarly some queues that are not in the system access graph are referenced in the program. Those are mainly message queues

within the network interface and are:

1. the network request queue,
2. the network ack queue,
3. the network rejection queue
4. and the network update queue.

When messages are received by the network interface they are, according to their type, put in one of those queues.

Finally, in connection with data structures, one can notice that, in the data structure definition section (A.2), the number of nodes, relations and transactions seem to be limited to N , M and P respectively. This is obviously not true in a real system. Similarly the resource access table may take quite a different form in an operational system than what is presented here. However, the purpose of this appendix is not so much to produce a completely specified, ready to be implemented design, but, rather, to contrast algorithmic and implementation view of a given synchronization procedure.

A.2 Data Structure Definition

constant

M, N, P; {M, N and P are the maximum number of
relations, nodes and transactions}

type

requirement, conflict, test, continue: boolean;
timefield, timestamp, newtime: integer;

transaction= record
Tid: integer;
Tts: timestamp;
Tneed: array[1..M] of requirement;
Tnode: array[1..N] of requirement;
interference: array[1..P] of conflict
end;

resource= record
Rid: integer;
Rtf: timefield
end;

accesstable= record
relation: array[1..M] of resource;
entry: array[1..P] of transaction;
ackcount: array[1..P] of integer;
fullyacked: array[1..P] of boolean
end;

message= record
msgtype: request, ack, rejection,
 update;
Tid: integer;
Tts: timestamp;
Tneed: array[1..M] of requirement;
Tnode: array[1..N] of requirement
end;

submission= record
Tid: integer;
Tts: timestamp;
transactiontype: local, distributed,
 network;
Tneed: array[1..M] of requirement;
Tnode: array[1..N] of requirement
end;

A.3 Procedures

A.3.1 procedure remove (j: integer);

```

begin
  with accesstable do
    begin
      for k:= 1 to M do
        if entry[j].Tneed[k]= true
          then relation[k].Rtf:= entry[j].Tts;
        entry[j].Tid:= nil;
        fullyacked[j]:= false;
        ackcount[j]:= 0;
        for k:= 1 to P do
          if entry[j].interference[k]= true
            then begin
                  entry[k].interference[j]:= false;
                  entry[j].interference[k]:= false;
                end;
        submission.Tid:= j;
        submission.Tneed:= entry[j].Tneed;
        queue (submission, distributed lock request queue)
      end
    end; {end of the procedure remove}

```

A.3.2 procedure insert (sub:submission; table:accesstable);

```

begin
  with table do
    begin
      entry[submission.Tid].Tid:= submission.Tid;
      entry[submission.Tid].Tts:= submission.Tts;
      entry[submission.Tid].Tneed:= submission.Tneed;
      entry[submission.Tid].Tnode:= submission.Tnode;
      entry[submission.Tid].interference:=
        computeinterference (submission.Tid,
          submission.Tneed);
      ackcount[submission.Tid]:= 0;
      fullyacked[submission.Tid]:= false
    end
  end; {end of the procedure insert}

```

A.4 Functions

A.4.1 function oldesttransaction

```

      (int: array[1..P] of conflict): integer;
var n, t: integer;
begin
    n := 0; t := 0;
    repeat
      n := n + 1;
      if int[n] = true
      then t := accesstable.entry[n].Tts
    until int[n] = true and accesstable.entry[n].Tts <> 0;
    for i := 1 to P do
      if int[i] = true and accesstable.entry[i].Tts < t
      and accesstable.entry[i].Tts <> 0
      then
        begin
          t := accesstable.entry[i].Tts;
          n := i;
        end;
    oldesttransaction := n;
end; {end of function oldesttransaction}

```

A.4.2 function choosetimestamp

```

      (need: array[1..M] of requirement):
      integer;
var t, increment: integer;
begin
    t := 0;
    for i := 1 to M do
      if need[i] = true and accesstable.relation[i].Rtf > t
      then t := accesstable.relation[i].Rtf;
    choosetimestamp := t + increment;
    increment := increment + 1;
    {increment can, in fact, be considered
     the logical clock of a given site}
end {end of function choosetimestamp}

```

A.4.3 function preparemessage (mtype:msgtype;

id:ts: integer; need: array[1..M] of
requirement; node: array[1..N] of
requirement): message;

begin

with preparemessage do

begin

msgtype:= mtype;

Tid:= id;

Tts:= ts;

Tneed:= need;

Tnode:= node

end

end; {end of function preparemessage}

A.4.4 function checktimestamp (ts: integer;

need: array[1..M] of requirement):
boolean;

begin

checktimestamp:= true;

for i:= 1 to M do

if need[i]= true and accesstable.relation[i].Rtf > ts

then checktimestamp:=false;

end; {end of function checktimestamp}

A.4.5 function computeinterference

(k: integer; need: array[1..M] of
requirement): array[1..P] of conflict;

begin

for i:= 1 to P do computeinterference[i]:= false;

for i:= 1 to M do

if need[i]= true

then

for j:= 1 to P do

if accesstable.entry[j].Tneed[i]= true

then

begin

accesstable.entry[j].interference[k]:= true;

computeinterference[j]:= true

end;

end; {end of computeinterference}

A.5 The Submission Manager

```

begin
  while true do
    begin
      if distributed request queue <> empty
      then
        begin
          submission:=dequeue (distributed request queue);
          with submission do
            begin
              Tts:= choosetimestamp (Tneed);
              Tid:= chooseid;
              insert (submission, accesstable);
              mssg:=\preparemessage (request, Tid, Tts, Tneed,
                Tnode);
              send (mssg, Tnode)
            end
          end;
        end;
      if network request queue <> empty
      then
        begin
          message:= dequeue (network request queue);
          with message do
            begin
              submission.Tid:= Tid;
              submission.Tts:= 0;
              submission.transactiontype:= network;
              submission.Tneed:= Tneed;
              submission.Tnode:= Tnode;
              insert (submission, accesstable);
              queue (submission, job queue);
              test:= checktimestamp (Tts, Tneed);
              if test= true
              then
                begin
                  accesstable.entry[Tid].Tts:= Tts;
                  mssg:= preparemessage (ack, Tid, Tts);
                  send (mssg, Tnode)
                end
              else {test is false}
                begin
                  mssg:= preparemessage (rejection, Tid, Tts);
                  send (mssg, origin)
                end
              end
            end
          end;
        end;
      if network rejection queue <> empty
      then

```

```

begin
message:= dequeue (network rejection queue);
with message do
begin
newtime:= choosenewtime (Tid);
accesstable.entry[Tid].Tts:= newtime;
accesstable.ackcount[Tid]:= 0;
mssg:= preparemessage (update, Tid, newtime);
send (mssg, accesstable.entry[Tid].Tnode)
end
end;
if network update queue <> empty
then
begin
message:=dequeue (network update queue);
with message do
begin
if accesstable.entry[Tid].Tts= 0
then
begin
test:= checktimestamp (Tts,
accesstable.entry[Tid].Tneed);
if test= true
then
begin
accesstable.entry[Tid].Tts:= Tts;
accesstable.ackcount[Tid]:= 0;
mssg:= preparemessage (ack, Tid, Tts);
send (mssg, accesstable.entry[Tid].Tnode)
end
else {test is false}
begin
mssg:= preparemessage (rejection, Tid, Tts);
send (mssg, origin)
end
end
else {accesstable.entry[Tid].Tts<> 0}
if accesstable.entry[Tid].Tts< Tts
then
begin
accesstable.entry[Tid].Tts:= Tts;
accesstable.ackcount[Tid]:= 0
end;
end
end; {end of if network update queue}
end
end.

```

A.6 The Resource Access Table Manager

```

begin
  while true do
    begin
      while network ack queue <> empty do
        begin
          message := dequeue (network ack queue);
          with message do
            begin
              if accesstable.entry[Tid].Tts = Tts
              then accesstable.ackcount[Tid] :=
                accesstable.ackcount[Tid] + 1
              else {if older disregard, if newer reset Tts}
                if accesstable.entry[Tid].Tts < Tts
                then
                  begin
                    accesstable.entry[Tid].Tts := Tts;
                    accesstable.ackcount[Tid] := 1;
                  end;
                  if accesstable.ackcount[Tid] = all nodes in Tnode
                  then accesstable.fullyacked[Tid] := true;
                end
              end;
            end;
          {end of while ack queue}
          for i := 1 to P do
            begin
              if accesstable.entry[i].Tid <> nil and
                accesstable.fullyacked[i] = true
              then
                begin
                  continue := true;
                  while continue do
                    begin
                      j := oldesttransaction
                        (accesstable.entry[i].interference);
                      if accesstable.fullyacked[j] = true
                      then remove (j)
                      else continue := false;
                    end
                  end
                end
              end
            end
          end
        end
      end
    end
  end.

```

APPENDIX B

Abbreviations and Notation

B.1 Abbreviations

AC: Abort co-ordinator
ACK: Acknowledgement messages exchanged by GC's or RUM's
AMP: Activity monitoring process
APL: Abort protocol layer
AT: Abort table
CC: Concurrency control
CPL: Commit protocol layer
CSS: Communication subsystem
DB: Database
DBM: Database manager
DDB: Distributed database
DLRQ: Distributed lock request queue
DR: Distributed relation
DRQ: Distributed request queue
DT: Distributed transaction
DURQ: Distributed update request queue
EPL: Execute protocol layer
GC: Global co-ordinator
INT: Interference set
LC: Local co-ordinator
LRLQ: Local release queue
LRQ: Local request queue
MP: Merge process
NAT: Network access table
NI: Network interface
NTS: New timestamp message sent by originating GC in SPL in case of rejection
O/S: Operating system
PMRQ: Process monitoring request queue
Psem: Private semaphore

QPP: Query processing process
 RAT: Resource access table
 REJ: Rejection message sent to originating GC in SPL
 when TS of T is unacceptable
 REQ: Request message used by originating GC in SPL
 RS: Recovery set of relations to be brought up to date
 during a merge procedure
 RTBLQ: Ready to be locked queue
 RTBRQ: Ready to be released queue
 RTS: Required time to achieve synchronization in SPL
 RTU: Ready to update message
 RUM: Release and update manager
 RUT: Release and update table
 SM: Subnetwork merge
 SN: Suspected nodes set
 SNM: Single node merge
 SPL: Synchronization protocol layer
 STP: Supervisory transaction process
 T: Transaction
 TN: Target nodes
 TP: Transaction process
 TPM: Transaction process monitoring (table)
 TR: Target relations
 TS: Timestamp
 TU: Target update
 TUD: Target update directory
 U: Update operation

B.2 Notation

$N = \text{network} = \{n_1, n_2, n_3, \dots\}$, with $n_i = \text{node } i$

$N = N_{u1} \cup N_{u2} \cup N_{u3} \cup \dots$, with $N_{ui} = \text{subnetwork } i$

Corresponding to N_{ui} , there is N_{ci} which is the set of nodes which crashed as perceived by those in N_{ui} .

$T_i = \text{transaction } i$ / $U_i = \text{update operation } i$, with

$R(T_i)$ / $R(U_i)$ set of target relations

$N(T_i)$ / $N(U_i)$ set of target nodes

$TS(Ti) / TS(Ui)$ timestamp of Ti / Ui

ri = relation i

$TF(ri)$ = timefield of relation ri

At a given node there exist: ATi , $NATi$, $RATi$, $RUTi$,
etc., and also by extension:

$R(ni) = R(RUTi)$ = all relations at ni

$N(ri)$ = all nodes where ri can be found

REFERENCES

- [BERN78] Bernstein, P.A., et al., "The concurrency control mechanisms of SDD-1: A system for distributed databases (the fully redundant case)", IEEE transactions on software engineering, vol. se-4, no. 3, pp. 154-158, May 1978.
- [BERN79] Bernstein, P.A., Shipman, D.W. and Wong, W.S.: "Formal aspects of serializability in database concurrency control", IEEE Transactions on Software Engineering, vol. SE-5, no. 3, pp. 203-216, May 1979.
- [BERN80] Bernstein, P.A., Shipman, D.W., and Rothnie, J.B.: "Concurrency control in a system for distributed databases (SDD-1)", ACM Transactions on Database Systems, vol. 5, no. 1, pp. 18-51, March 1980.
- [CHANG79] Chang, E., and Roberts, R.: "An improved algorithm for decentralized extrema finding in circular configuration of processes", CACM, vol. 22, no. 5, pp. 281-283, May 1979.
- [COLL79] Colliat, G., and Bachman, C.: "Commitment in a distributed database", Working conference on distributed database architecture, Venice, Italy, pp. 100-115, June 1979.
- [DIJK74] Dijkstra, E.W.: "Self stabilizing systems in spite of distributed control", CACM, vol. 17, no. 11, pp. 643-644, Nov. 1974.
- [ELLI73] Ellingson, C.E., and Kulpinski, R.J.: "Dissemination of system time", IEEE transactions on communications, vol. com-21, no. 5, May 1973.

- [ELLI77] Ellis, C.A.: "A robust algorithm for updating duplicate databases", Proceedings of the 2nd Berkeley Workshop on distributed data management and computer networks, pp. 146-158, 1977.
- [ESWA76] Eswaran, K.P., et al.: "The notions of consistency and predicate locks in a database system", CACM, vol. 19, no. 11, pp. 624-633, Nov. 1976.
- [GRAY78] Gray, J.N.: "Notes on database operating systems", Operating Systems: An Advanced Course, (ed. R. Bayer, R.M. Graham and G. Seegmuller), Springer Verlag, pp. 393-481, 1978.
- [HAMM80] Hammer, M. and Shipman, D.: "Reliability mechanisms for SDD-1: a system for distributed databases", ACM Transactions on Database Systems, vol. 5, no. 4, pp. 431-466, December 1980.
- [HANS73] Hansen, Per Brinch: "OPERATING SYSTEM PRINCIPLES", Prentice Hall, Englewood Cliffs. N.J., 1973.
- [HOAR74] Hoare, C.A.R.: "Monitors: an operating system structuring concept", CACM, vol. 17, no. 10, Oct. 1974.
- [LAMP78a] Lamport, L.: "Time, clocks, and the ordering of events in a distributed system", CACM, vol. 21, no. 7, pp. 558-565, July 1978.
- [LAMP78b] Lamport, L.: "The implementation of reliable distributed multiprocess systems", Computer Networks, vol. 2, pp. 95-114, 1978.
- [LAMPSxx] Lampson, B. and Sturgis, H.: "Crash recovery in distributed data storage systems", to appear in Communications of the ACM.

- [LELA77a] LeLann, G.,: "Introduction a l'analyse des systemes multireferenciels", Ph.D. thesis, University of Rennes, France, May 1977 (french).
- [LELA77b] LeLann, G.,: "Distributed systems-towards a formal approach", IFIP, North-Holland publishing co., Toronto, pp. 155-165, August 1977.
- [LELA78] LeLann, G.,: "Algorithms for distributed data sharing systems which use tickets", Proceedings of the 3rd. Berkeley workshop on distributed data management and computer networks, pp. 259-272, August 1978.
- [LELA79] LeLann, G.,: "An analysis of different approaches to distributed computing systems", Proceedings of the 1st international conference on distributed computing systems, Huntsville, pp. 222-232, Oct. 1979.
- [LOME79] Lomet, D.B.,: "Coping with deadlock in distributed systems", Working conference on distributed database architecture, Venice, Italy, pp. 89-99, June 1979.
- [MAHM75] Mahmoud, S.A.,: "Resource allocation and file access control in distributed information networks", Ph. D. thesis, Systems engineering department, Carleton university, Jan. 1975.
- [MAHM77] Mahmoud, S.A., and Riordon, J.S.,: "Software controlled access to distributed databases", Infor, vol. 15, no. 1, Feb. 1977.
- [MAHM79] Mahmoud, S.A., Riordon, J.S. and Toth, K.C.,: "Distributed database partitioning and query processing", IFIP working conference on distributed database architecture, Venice, Italy, pp. 32-51, June 1979.

- [MENA79] Menasce, D.A., and Muntz, R.R.: "Locking and deadlock detection in distributed databases", IEEE Transactions on software engineering, vol SE-5, no. 3, pp. 195-202, May 1979.
- [MENA80a] Menasce, D.A., Popek, G.J. and Muntz, R.R.: "A locking protocol for resource co-ordination in distributed databases", ACM Transactions on Database Systems, vol. 5, no. 2, pp. 103-138, June 1980.
- [MENA80b] Menasce, D.A. and Landes, O.E.: "On the design of reliable storage components for distributed database management systems", 6th Very Large Database Conference, (VLDB-6), Montreal, Canada, pp. 365-376, Oct. 1-3, 1980.
- [RAND78] Randell, B.: "Reliable computing systems", Operating Systems: An Advanced Course, (ed. R. Bayer, R.M. Graham and G. Seegmüller), Springer Verlag, pp. 282-392, 1978.
- [REED79] Reed, D.P., and Kanodia, K.: "Synchronization with eventcounts and sequencers", CACM, vol. 22, no. 2, pp. 115-123, Feb. 1979.
- [ROSE78] Rosenkrantz, D.J., et al.: "System level concurrency control for distributed database systems", ACM transactions on database systems, vol. 3, no. 2, pp. 178-198, June 1978.
- [ROTH80] Rothnie, J.B., et al.: "Introduction to a system for distributed databases (SDD-1)", ACM Transactions on database systems, vol. 5, no. 1, pp. 1-17, March 1980.
- [STON79] Stonebraker, M.: "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.

- [STUR80] Sturgis, H., Mitchell, J. and Israel, J.: "Issues in the design and use of a distributed file system", ACM SIGOPS Operating Systems Review, vol. 14, no. 3, pp. 55-69, July 1980.
- [SUNS73] Sunshine, C.A.: "INTERPROCESS COMMUNICATION PROTOCOLS FOR COMPUTER NETWORKS", Technical report no. 105, Stanford university, Dec. 1975.
- [THOM78] Thomas, R.H. "A solution to the concurrency control problem for multiple copy databases", Compcon 78 spring, pp. 509-515, 1978.
- [TOTH80] Toth, K.C.: "Distributed database architecture and query processing strategies", Ph.D. thesis, Department of Systems Engineering, Carleton University, Ottawa, 1980.
- [VERH79] Verhofstad, J.S.M.: "Recovery techniques for database systems", ACM Computing Surveys, vol. 10, no. 2, pp. 167-195, June 1978.
- [YEME79] Yemeni, Y., and Cohen, D.: "Some issues in distributed processes communication", Proceedings of the 1st international conference on distributed computing systems, Huntsville, pp. 199-203, Oct. 1979.

END

0	2	H	0	9	H	8	2	1
---	---	---	---	---	---	---	---	---

FIN