

(888)
(678)

**RECOVERY FROM MULTIPLE PARTITIONS
USING DYNAMIC VOTING
IN
DISTRIBUTED SYSTEMS**

72p.

Dissertation submitted
in partial fulfilment of the requirements
for the award of the Degree of
MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE & TECHNOLOGY

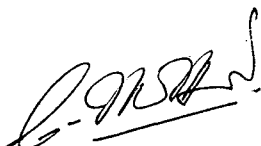
M. V SREENIVAS


SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110067
JANUARY 1991

C E R T I F I C A T E

The work embodied in this dissertation has been carried out at the School of Computer and Systems Sciences, Jawaharlal Nehru University, by me. This work is original and has not been submitted so far, in part or full, for any degree or diploma of any university.


M.V. SREENIVAS


Col. C.P.C. Nath,
Asst. Prof.,
SC&SS, JNU.
4 Jan 91


Prof. N.P. Mukherjee,
Dean, SC&SS,
JNU.

To my Esteemed Teachers

A C K N O W L E D G E M E N T

I owe my sincere thanks to Col. C.P.C. Nath, who introduced me to, and created a sense of confidence in me for this fascinating field of Distributed Systems. This effort would not have succeeded without the valuable discussions, encouragement to pursue my thoughts in my own way, apt criticism and excellent guidance.

I extend my thanks to Prof. N.P. Mukherjee, Dean, SC&SS, for giving me an opportunity to take up this project. I would like to acknowledge all the faculty members of the school, especially Dr. Subhash Bhalla for patiently listening to my talks, his excellent suggestions and for the reference material provided by him. I also thank the staff of School, who rendered their full cooperation in completing this project work.

I am grateful to all my friends, who helped me a lot without any hesitation, all through my project work.

I also take this opportunity to thank library staff of JNU and IIT, New Delhi, for their cooperation.

M.V. SREENIVAS

A B S T R A C T

In distributed systems the database is replicated over a number of sites. The transactions are concurrently executed in number of sites, where a transaction (a task) is divided into number of subtasks and may be carried out concurrently at the sites where these are assigned. In such environment, the failure of nodes or links may lead to inconsistent updates to the database.

The dissertation contributes in suggesting measures for retaining consistency in the event of the above mentioned failures. The node or link failures might lead to partitioning of the network in which a set of nodes cannot communicate with rest of the nodes. Recovery from failures poses a large number of problems and a great amount of research work has been done in order to solve these. Recovery from partitions is the area in which extensive research is going on. The recovery techniques studied in this dissertation have been designed to solve the problem of consistency. This dissertation also gives a solution to recovery from multiple partitions using dynamic voting technique and thus increasing the availability of database.

C O N T E N T S

1. INTRODUCTION	1
2. FAULT RESILIENT DESIGN TECHNIQUES	9
2.1 INTRODUCTION	10
2.2 RECOVERY BLOCKS	10
2.2.1 Recovery from Internal Errors	11
2.2.2 Recovery from External Errors	12
2.3 CONVERSATION AND EXCHANGE	15
2.4 N-VERSION APPROACH	18
2.5 CHECKPOINTING AND ROLLBACK RECOVERY	18
3. OPTIMISTIC RECOVERY IN DISTRIBUTED SYSTEMS	23
3.1 INTRODUCTION	24
3.2 RECOVERY FROM SITE FAILURES	26
3.2.1 Determining System State	26
3.2.2 Checkpointing and Message Logging	30
3.2.3 Recoverable System State	31
3.2.4 Finding Unique Maximum System State	31
3.3 RECOVERY FROM PARTITIONS	35
3.3.1 Version Vector Model	36
3.3.2 Precedence Graph Model	39
3.4 CONCLUSION	41
4. PESSIMISTIC RECOVERY IN DISTRIBUTED SYSTEMS	42
4.1 INTRODUCTION	43
4.2 LOCK SERVER (LS)	43
4.2.1 Lock Server Selection and Services	45
4.2.2 Lock and Release Grants	46
4.3 CONSISTENCY FROM CRASHES	47
4.3.1 Logical Partition Recovery	48

4.4 RECOVERY FROM SINGLE NODE FAILURES	52
4.5 MERGING OF PARTITIONS	53
4.6 RECOVERY FROM MULTIPLE PARTITIONS AND DYNAMIC VOTING	55
4.6.1 To Determine Last Node(s) Failed	58
4.7 CONCLUSION	59
5. CURRENT AND FURTHER RESEARCH	60
REFERENCES	62

CHAPTER 1

INTRODUCTION

A distributed system is one in which the processors are distributed over large geographical area with each of the processors working autonomously without sharing any memory but communicating with the other processors in the system by messages.

There is disagreement in defining what a distributed system is? This varies from multiprocessors, multicomputers, workstation LANs to interconnected WANs. Not only the memory distribution but also the type of communication between the processors determine the characteristics of a distributed system. This can vary by any degree from tightly coupled systems to loosely coupled systems. The degree of communication is measured in **grain size**. The systems with large grain size communicate infrequently and spend much of their time in computation. The system with fine grain size communicate more frequently. The goal of the distributed system is to achieve speedup through parallelism in running an application.

The difference between an ordinary computer system and distributed system is that the distributed system has got **partial failure property**. Even though part of the system fails (it can be single site to multiple site) the rest of the system must be in a position to continue with its work. This makes the distributed system attractive as the failure frequency or duration of failure is unpredictable. To

achieve partial failure property the data items should be replicated over distributed sites running autonomously.

The distributed system differs from other sequential systems in three issues:

- i) it should be in a position to assign different parts of the program to different processors to use the processors optimally.
- ii) these set of processors working on some program segment should co-operate to exchange results and for synchronization.
- iii) implementing partial failure property.

The properties (ii) and (iii) together pose the problem of globally consistent state of the system. The failure can be in node or communication. This can lead to network partitioning isolating set of nodes from rest of the nodes in the system. Increasing demand for higher throughput and higher availability makes the distributed system attractive. But increasing the availability by replication results in inconsistencies in the event of failure of one or more nodes. This is because of complicated failure modes of multiprocessor configurations. To support partial failure property failure of one node or communication failures should not stop the system. But a transaction involving set of nodes leads to inconsistent state of the system. This demands the execution of

operations in the transactions to be complete or the transaction should have no effect on the system at all. Such a transaction is called **atomic transaction**. We call an operation atomic if it satisfies both the properties of indivisibility and recoverability. A transaction is indivisible if its intermediate states are transparent to the external world. An operation is recoverable if the failure of the transaction will force the system to get back its state before the operation is started and thus the operation has got no effect at all. The transaction outcome can be one of the three instances given below:

BEGIN	BEGIN	BEGIN
action	action	action
action	action	action
.	.	.
.	.	.
.	.	.
action	action	action
COMMIT	ABORT	ABORT ==>
Successful Transaction	Aborted by Client	Aborted by Server

The important properties of atomic actions as put forth by Randell[RAND 78] are as follows:

- i) "An action is atomic if the process (processes) performing it is (are) not aware of the existence of any other active processes, and no other process is aware of the

activity of the process (processes) during the time the process is (processes are) performing the action".

ii) "An action is atomic if the process (processes) performing it does(do) not communicate with other processes while the action is being performed".

iii) "An action is atomic if the process (processes) performing it can detect no state changes except those performed by itself(themselves) and if it does (they do) not reveal its(their) state changes until the action is complete".

(iv) "Actions are atomic if they can be considered so far as other processes are concerned, to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent".

An atomic action can be achieved as follows: If a transaction contains an operation that tries to change an object, the changes are not applied to original object, but to a new copy of the object called **version**. If the entire transaction fails(or aborts), the new versions are simply discarded. If the transaction succeeds, it commits so that the new version becomes permanent. All objects modified by the transaction will retain this new version. It is not feasible to abort the transaction because of node failure or some other internal error. For this reason latest value of each object is placed in stable storage which has high chances of surviving processor crashes.

In distributed systems data items are replicated at various sites to increase the availability. This increases the reliability of database by making the database to be resilient to the site failures. Also it decreases the communication in the network when the data is present at the requester's site. But here the user should see that the updates to the database is made at all sites where the replica exists. But this has got a serious problem of inconsistency when the communication fails. This may lead to conflicting updates to the database. This problem will become more severe when the communication failure divides the system into two partitions (**partition failure**). This will update the replica of some data items at different partitions compromising the correctness of data.

In addition to the above complexities the problem still unsolved is the detection of the failure modes in the system. This includes the three anomalies which the system can not differentiate

- i) the node is down or
- ii) the communication channel to the node is down or
- iii) node and communication channels are fine but the delay is high.

When partitioning occurs, the transactions may or may not be allowed to continue and new transactions can be allowed to enter the partition. If the transactions are

allowed then conflicting updates can be taken care of by undoing the transaction and its effect by running compensating transaction which undo the effect of transaction which is the cause of inconsistency of database. But this may not be suitable in certain cases like banking system where money is withdrawn. If the transaction is suspended on partition then this decreases the availability. But the database will be consistent. This may not be acceptable in certain applications like:

- i) airline reservation where partition means losing customer or
- ii) in military Command and Control systems where availability is more important than correctness or consistency.

Two strategies associated with partitioned recovery are optimistic and pessimistic. In optimistic strategy there is a threat to the global consistency of the system as the system allows each partition to continue with its operation even when the partition occurs. The inconsistencies are resolved at the recovery time when the two partitions merge. In the pessimistic strategy, inconsistencies never occur as care is taken to see that same data item can not be updated in both the partitions.

The optimistic strategies include version vectors [chapter 4], optimistic protocol [chapter 3]. Some of the

pessimistic strategies include primary copy [ALSB 76, STON 79], tokens [MINO 82], voting [GIFF 79], missing writes [EAGE 83] etc.

My contribution in this dissertation is to find, how to recover from multiple partitions using dynamic voting. This is a pessimistic recovery mechanism. Chapter 2 is a survey of fault resilient design techniques. Chapter 3 is a survey on optimistic recovery. It gives solutions to both transient and persistent failure recovery. Chapter 4 discusses pessimistic recovery. It tries to increase availability in distributed systems using dynamic voting while recovering from multiple partitions. Chapter 5 focusses on current and further research in this area.

FAULT RESILIENT DESIGN TECHNIQUES

2.1 INTRODUCTION

2.2 RECOVERY BLOCKS

2.3 CONVERSATION AND EXCHANGE

2.4 N-VERSION APPROACH

2.5 CHECKPOINTING AND ROLLBACK RECOVERY

2.1 INTRODUCTION

This chapter is the survey of various strategies used in providing fault resiliency in case of process failures. Checkpointing and rollback recovery techniques used in the event of failure in distributed systems is also considered. The name distributed system itself conveys that the recovery from a failure is not only local to the failed node but also includes the whole system or a subset of the whole system which is affected by failure.

Recovery in centralised system is rather clear as the failure is local to the failed node. Failure must have occurred either before or after the completion of the operation. The recovery in distributed system is two fold. In the first case data items are distributed over sites. In the second case the data items are also replicated over various nodes.

The various recovery techniques outlined in this chapter are recovery blocks, conversation, exchange, N-version programming and checkpointing and rollback recovery.

2.2 RECOVERY BLOCKS

The failure of a node or site can be due to internal or external errors. **Internal errors** are the errors in which the system can recover when the same set of

operations are rerun or repeated. In case of **external errors** the failure is limited to the local node and it cannot be handled by the local node. It could be because of hardware or software. Here multiplicity of hardware or software is of no avail. So diversified hardware and software is used. In the following section we discuss how to recover from internal errors developed by Randell [RAND 75] and external errors by Kim [KIM 84].

The recovery block mechanism needs the program to be structured into set of blocks like subroutines, modules, procedures, functions etc. Each of these program units can be run on the primary as well as backup processes to recover from failures. The backup block becomes the recovery block in case of failure of the primary block.

2.2.1 Recovery from internal errors

The recovery block consists of tryblocks. One of the tryblocks is a primary block and the rest are alternate blocks. Each of the try blocks consists of acceptance test which needs to be satisfied after the module is run in a block to determine the correctness of the execution. In case of failure of a block the system state is restored and the next alternate block is tried. If the acceptance test is passed then further try blocks are ignored. If all the alternate blocks fail then the whole recovery block is

regarded as failed. The recovery block structure is as follows:

BEGIN

{ PRIMARY BLOCK }

.

. (operations)

.

while (acceptance-test-not-satisfied and
alternates-exist) do

begin

restore the system state

use alternate block for operations

end

if acceptance-test-satisfied then

return(success)

else return(failure)

END

2.2.2 Recovery from external error

External error recovery is done by distributed execution of recovery blocks. It is suitable for tolerating failures in both hardware and software components. The recovery block here consists of tryblocks, one in local node and the rest in standby nodes. Acceptance test is used to check the correctness of result. In case of error the standby node containing a tryblock is invoked. To gain time concurrently all the tryblocks can be initiated. Two schemes are considered. We assume that the tryblocks will not update global variables and for ease of explaining we consider only two tryblocks, primary and backup.

Scheme I

The primary block and the standby block work on the same program module and at the end of the run the acceptance test is taken. If the primary block passes the acceptance test then its result will be directed to successor computing station else the alternate block will be notified of the failure of primary block. The alternate block after the acceptance test, on passing the test, passes the result to the successor computing station. In addition to logic acceptance test, time acceptance test is also used to put bounds on the time of execution of module in all of the tryblocks (see fig.2.1).

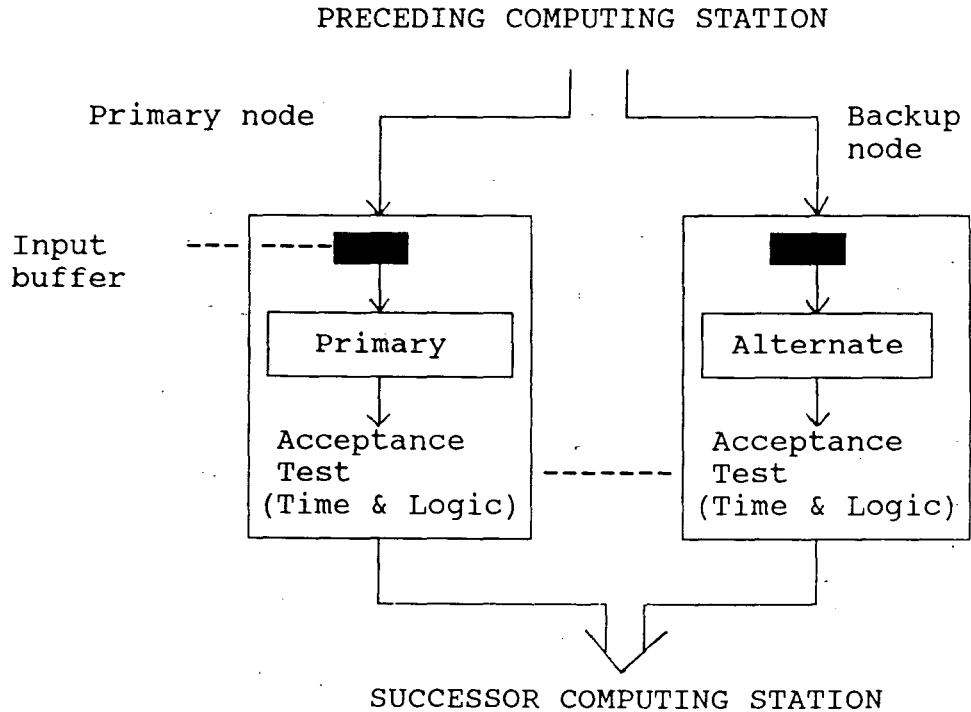


fig. 2.1 Scheme I

Scheme II

In this scheme the primary and backup nodes will have two tryblocks each. One is designated as primary and the other as backup block. In case of failure of primary node the roles of the primary and backup nodes are exchanged. Also in case of failure of any of the nodes the roles of its primary and backup blocks will be changed.

For example in fig. 2.2, the primary block in the primary and backup nodes are A and B respectively. The failure of primary node acceptance test causes the backup node to be new primary node and primary node to be new backup node. The roles of A and B in primary are changed.

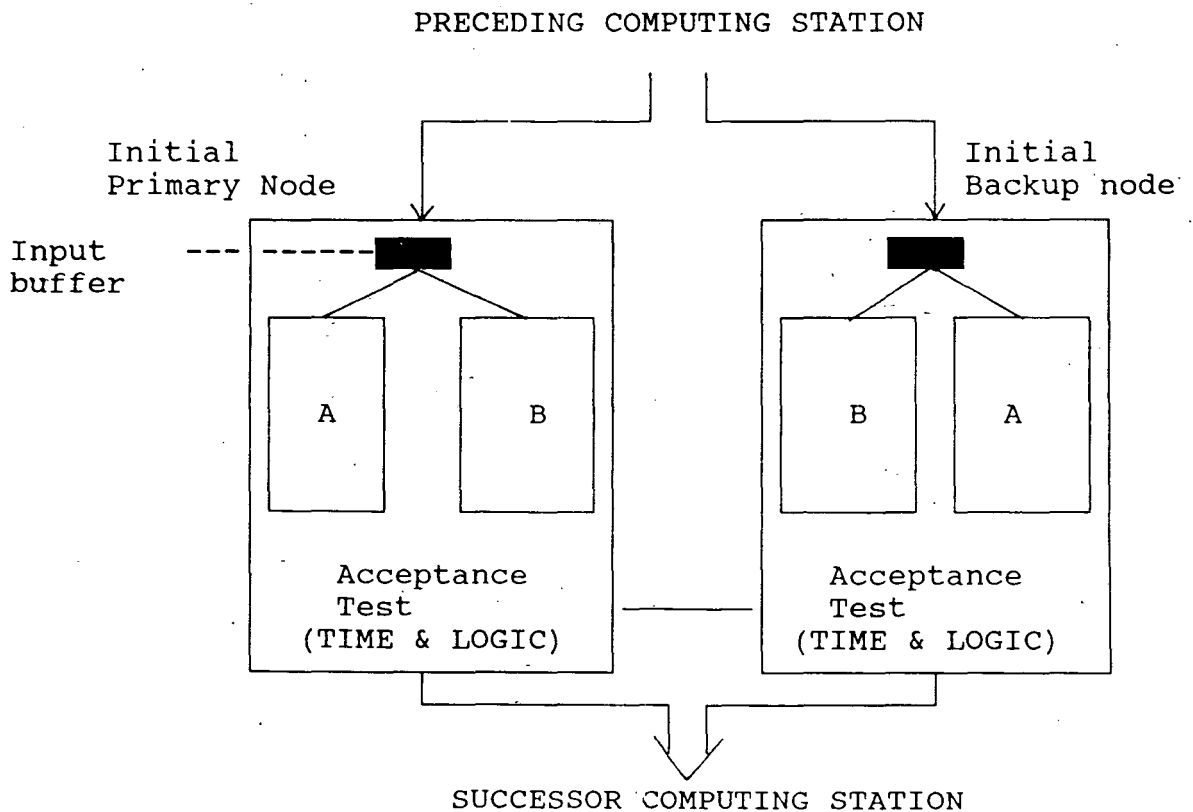


fig. 2.2 Scheme II

The failure of backup node causes the backup node to change the roles of its primary and backup blocks and it starts with its new primary block. The failure of backup node will have no effect on the primary node functions.

2.3 CONVERSATION AND EXCHANGE

In the previous section of recovery blocks notion of interaction between the processes is not considered. This makes the system slow in case of failures either persistent or transient. The slowness is owing to starting of the system from the initial state in the event of failure of any of the process. In the strategy to be presented communicating processes start from some intermediate state rather than from the initial state. This works well in case the failure is transient. This strategy conversation and exchange is due to Randell [RAND 75] and Andrew [ANDR 86].

The process creates recovery points as it goes on. Whenever the process fails, it starts from a recovery point. Since the processes are interacting, the rolling back of the failed process may force other process or set of processes to rollback to maintain the consistency of the system. This may lead to uncontrolled rolling back of processes and the system starts from initial state compromising the idea to restore from some intermediate state. This is called the **domino effect**. The reason is that the processes are dependent on each others progress as

they are involved in message passing. For example in figure 2.3, if the process 1 fails then it will be backed up to recovery point 4. If process 2 fails it will be backed up to recovery point 3 and this forces process 1 to rollback to 3 and this in turn forces process 3 to rollback to recovery point 4. In case process 3 fails all the processes rollback to initial state due to domino effect.

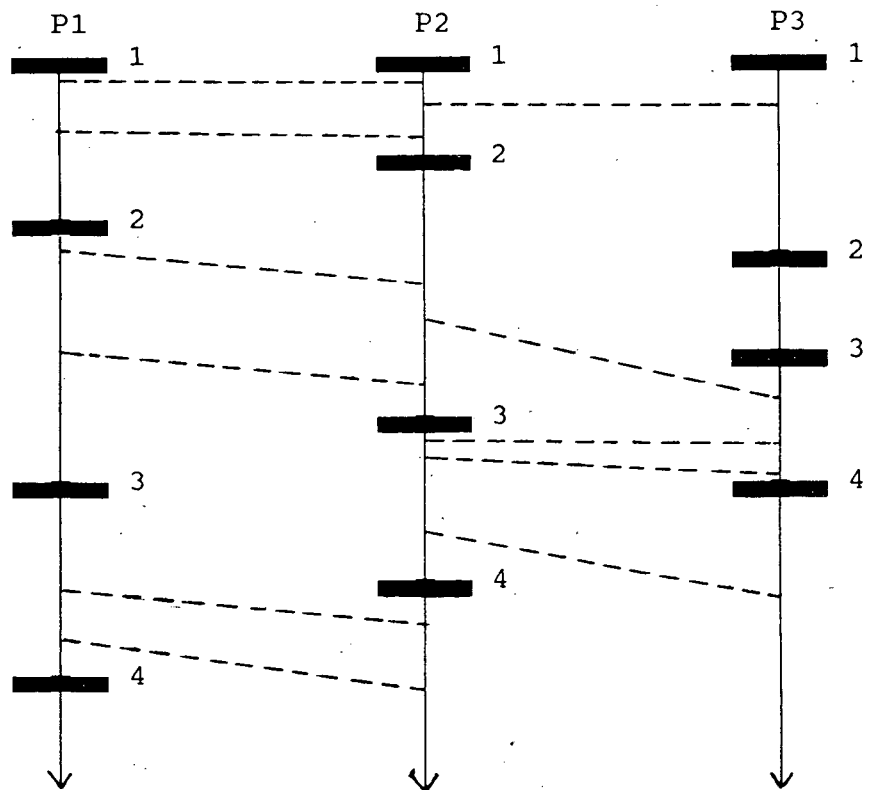


fig. 2.3 Domino Effect

The set of processes are said to be in conversation if they are communicating among themselves and not with any other process outside this set. Any process needs to take recovery point on entering the conversation

and all the processes involved in conversation should terminate at the same time(fig. 2.4). Failure of any of the process in the set will cause all of them to rollback to their respective recovery points taken on entering the conversation and start again.

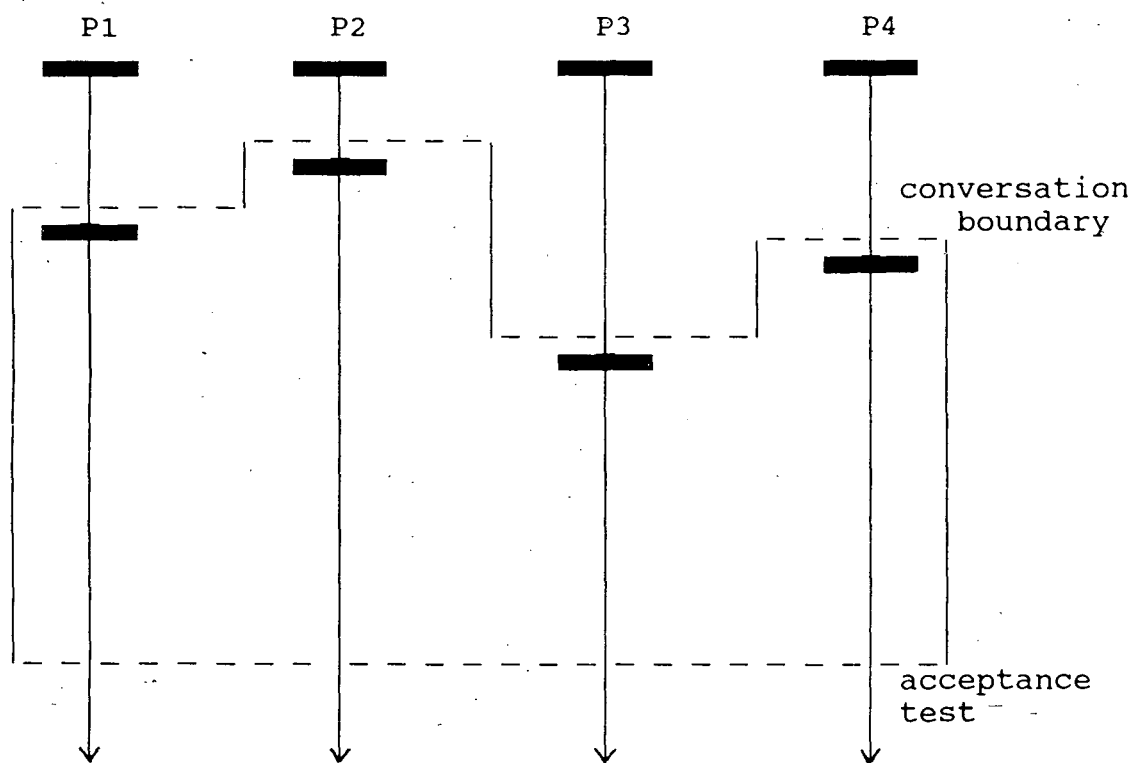


fig. 2.4 Conversation

The exchange is a restricted form of conversation. In exchange each of the process will be taking a recovery point on initiation. The set of processes are said to be in exchange now and they are allowed to terminate only when all the other processes in the excahange terminate. The

processes involved in the exchange are now allowed to discard their initial recovery points. The failure of any of the process forces all the processes in the exchange to rollback to their initial recovery points:

2.4 N-VERSION APPROACH

N-version approach is based on multiple computation technique used in hardware fault tolerance. It is designed by Algirdas [ALGI 85]. Here diversified software(N-versions) running on N different hardware units generates results. The results are checked with acceptance tests as used in recovery blocks. The difference between the recovery block and N-version approach is that in recovery block technique the acceptance test is for identical result from the tryblocks but in N-version programming since diversified software is used the results of a version also depends upon the local software design. So check will be for similar results, not for identical results. But the problem is of choosing incorrect result when the similar errors outnumber the set of good results at a decision point.

2.5 CHECKPOINTING AND ROLLBACK RECOVERY

The conversation technique discussed places a restraint on the set of processes taking part in conversation to end the conversation at the same time. This causes extended delay in a process which finishes the task first. So this brings down the throughput of the system.

Let a process send a message to some other process and the other process received the message. After some time the received process fails and forgets the information regarding the receipt of message. Then the process which has sent the message should undo its actions and send the message again. This is called **roll back**.

When a process fails, its state is lost and the whole system (set of process involved in conversation) should start from beginning. But this is not acceptable as the transaction might have already taken considerable amount of time. So after some interval of time the status of the process is stored in non-volatile memory called **stable storage**. In the event of failure the process copies its state from its stable storage and starts from this point. This stored status of the system is called **checkpoint**. This checkpointing and rollback recovery technique is discussed by Richard [RICH 87].

In the checkpointing and rollback recovery technique presented here, when a process takes checkpoint it forces a minimal set of processes to take recovery point or checkpoint. Also rollback of a process after failure forces a minimal set of processes to rollback. This strategy is tolerant to failures at the time of recovery. Since the state of a process is dependent upon the state of other processes, any process is allowed to take checkpoint only

when the system remains consistent in case of failure of any of the process. For example (fig. 2.5) when process 'p' sends a message to process 'q'. After receiving the message say process q takes a checkpoint. Now if process p fails, on repair it rolls back to its latest checkpoint and thus forgets about sending a message to q. This leads to inconsistency in the system.

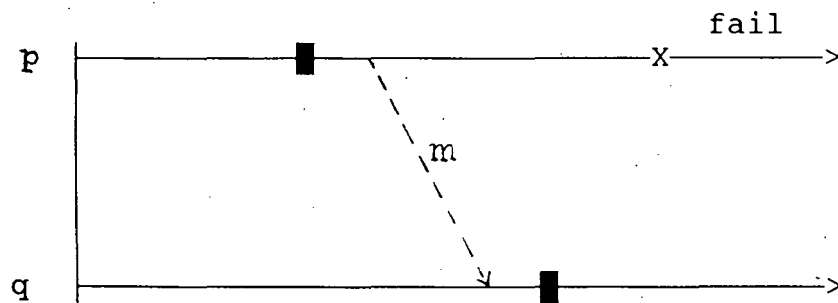


fig. 2.5 Inconsistency

In the second example (fig. 2.6), say process p takes a checkpoint after sending the message and process q receives the message and fails after some time. The process q rolls back to its latest checkpoint which is taken before the message from p is received. So p has the knowledge of sending the message whereas q does not have (lost messages).

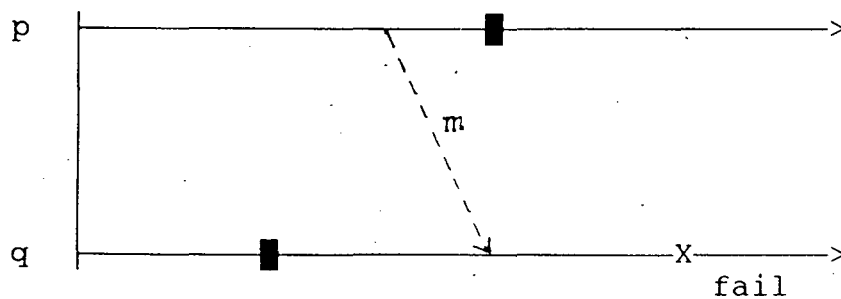


fig. 2.6 Lost Messages

In this model two checkpoints are taken, one tentative and the other permanent. The tentative checkpoint can be revoked or changed into permanent checkpoint. To overcome inconsistencies a process is allowed to take checkpoint only when all the processes which have sent messages to process p since its last checkpoint (called ckpt-cohorts of p) are checkpointed. Process p initially takes a temporary checkpoint and sends the request to all the cohorts of p. They, in turn, take temporary checkpoints and inform its own cohorts and so on. Once cohorts agree on checkpointing, permanent checkpoint is taken by two way commit protocol. When a checkpointing process is going on the processes which take tentative checkpoints do not receive any messages. Also they cannot take another checkpoint or rollback to avoid cycles leading to deadlock.

TH-3653

When a process p takes a rollback it sends request for rollback of all the processes to which it has sent messages after its latest checkpoint (called rback-cohorts of p) to rollback. The mechanism is similar to one used while taking checkpointing. When it receives positive response from all the cohorts then it will be ready to take rollback action. During recovery a process taking part in recovery is not allowed to send messages excepting the request for rollback to its cohorts.

Dissertation

681.3:342.82

Sr 17

21

de.



In the event of negative response from a cohort or failure of a cohort the rollback is not allowed. This is needed to maintain the consistency of the system.

OPTIMISTIC RECOVERY IN DISTRIBUTED SYSTEMS

3.1 INTRODUCTION

3.2 RECOVERY FROM SITE FAILURES

3.3 RECOVERY FROM PARTITIONS

3.4 CONCLUSION

3.1 INTRODUCTION

The optimistic recovery technique is used when probability of failure is very low and failure latency is small. The overhead incurred using this technique is small and its complexity increases linearly with increase in the number of failures. The computation will continue even when the sites fail. It can withstand single site or multisite failures. There will be no synchronization among communication, computation and checkpointing. But synchronization is needed when a failed process recovers and starts from recent stored state. Here the set of processes whose states depend upon the failed process are also forced to backout and replay messages to start from unique maximum system state. A method which does not need synchronization but uses incarnation numbers to detect duplicated messages is discussed by [YEMI 85].

Checkpointing and rollback recovery technique is used to achieve resiliency(see section 2.5). Let a process p send a message m to process q. After receiving m the process q fails and then loses the knowledge of receiving message m. But process p still thinks that the message is received by process q. Here process p is called **orphan process** and the computations if any performed by p are called orphans. Here we need to undo the actions of process p by forcing it to some earlier state that does not depend on the lost state. To undo the actions of p we

will restore the status of p to some earlier checkpoint from stable storage and then replay the logged messages. The messages are said to be logged when the messages that the process received is stored in stable storage. Here, care is taken to see that the extent of roll back is restricted so that it will not lead to **domino effect**.

We have seen that when process p sends a message to process q, process p becomes orphan when q fails and thus forgetting the receipt of message. The sender may keep on sending the same message until the receiver logs the message. But the number of messages are usually written to stable storage depending upon the size of buffer and in a single transfer. Writing each message on its receipt reduces the throughput. The intention is to allow the sender to continue with its actions after sending the message, no matter the receiver has logged the message or not and still be in a position to recover from failures [JOHN 90]. Also to see that, based on current checkpoint and logged messages, recover to unique maximum recoverable system state. This relinquishes the restriction to log all the received messages.

This optimistic strategy will never limit the availability of the system. In case of single or multiple site failures, even a single copy of the data item will make the system to continue with its task. But this creates

inconsistencies when the network is partitioned. In each of the partitions data will be consistent. But as each partition is allowed access to data the global state [CHAN 85] of the system will be inconsistent. When the partitions are to be merged these inconsistencies should be detected and restored. Two techniques, an optimistic protocol [DAVI 84] and version vector approach [PARK 83] are discussed.

3.2 RECOVERY FROM SITE FAILURES

3.2.1 Determining System State

The state of the process is defined in terms of the state of its dependencies i.e. the set of processes which can make this process an orphan. The state of the system is the collection of states of all the processes in the system.

Here the state of the process is defined in terms of the number of messages it received. Each time the process receives a message the state interval of the process is incremented by one. So a process can send any number of messages during a state interval and it changes each time when it receives a message (see fig. 3.1). The current state of a process will be its current state interval (or the count of number of messages it received). The process tags with each of the message sent by it, the current state interval.

State Interval = 0

send mesg₁

send mesg₂

:

:

Rec mesg₁

State Interval = 1

send mesg_p

send mesg_q

:

:

Rec mesg₂

State Interval = 2

:

:

Rec mesg_i

State Interval = i

:

:

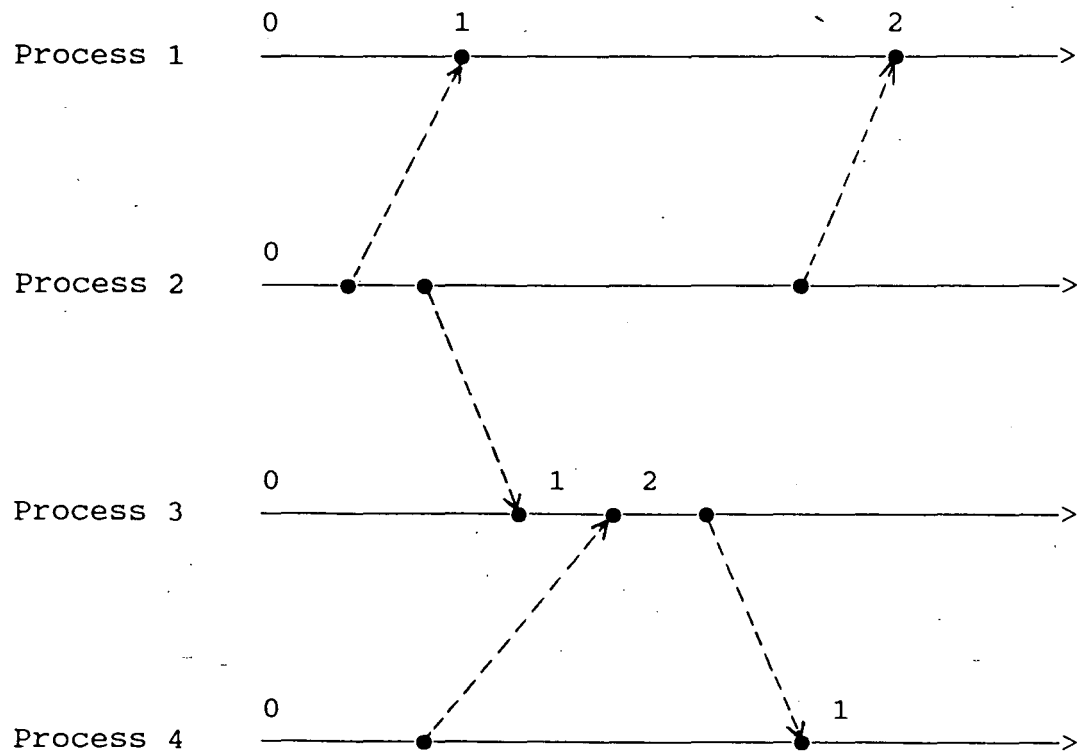
:

fig. 3.1 Numbering State Intervals

The dependency vector of process i (DV_i) is the set of state intervals of all the processes on which the process i depends. We call process i depends on some

process j when process i receives a message from process j . When a process doesn't receive any message from process j then the value corresponding to process j is kept at some minimum value (\perp) at process i (see fig. 3.2).

$DV_i = \langle d_1, d_2, d_3, \dots, d_n \rangle$ where, n denotes the number of processes in the system.



$$DV_1 = \langle 2 \quad 0 \quad \perp \quad \perp \rangle$$

$$DV_2 = \langle \perp \quad 0 \quad \perp \quad \perp \rangle$$

$$DV_3 = \langle \perp \quad 0 \quad 2 \quad 0 \rangle$$

$$DV_4 = \langle \perp \quad \perp \quad 2 \quad 1 \rangle$$

$$DV_{xx} = \begin{bmatrix} 2 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 2 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix}$$

fig. 3.2 The System State

The matrix DV_{xx} determines the system state at some point in time. The dependency vector values determine only the receipt of messages not the logging of messages. The diagonal values in DV_{xx} determine the current state intervals of the corresponding processes.

The dependency vectors are sent to the centralised controller. The centralised controller determines the whole state of the system (D_{xx}) from the information provided by the processes. The state of the system at an instant of time will be consistent state of the system. When a process fails and then recovers the system is forced back to this consistent system state and starts all over again so that it is equivalent to some failure free execution. The system state is said to be inconsistent if a message not yet sent by the system is received by a process. It can be due to failure of the sender and it rolls back on failure and forgets about sending the message. This is same as saying that the system state is inconsistent if for some process p the status of some of the processes depends beyond the

process p 's current state interval. This can be ensured from the dependency matrix. In each of the column the diagonal element (the current state interval of the process) should not be less than any other value in that column so that the system is consistent.

3.2.2 Checkpointing and Message Logging

When a process receives a message it may place the message in stable storage so that in the event of failure of process it will retain some stable state and replay the messages from stable storage. This storing the messages in stable storage is called message logging. When ever a process receives a message it changes (or increments) its current state interval index. We can say that the message received started the new state interval index. The message received and the state interval index it started are both logged (logged (i, α)). Here i denotes process number and α the new state interval index, the message which started new state i .

A state interval α of a process is said to be stable if the process can be started from its checkpoint and the messages logged can be replayed so that it will get back to state interval α . We say stable (i, β) if for process i the state interval β is stable. Let us say process i 's state interval p is checkpointed and r will be its stable state interval if

$$\forall q \quad p < q \leq r \quad [\text{logged}(i, q)]$$

Here the process will roll back to its state interval p in the checkpoint and starts replaying the messages from $p+1$ to r and hence its state interval will be r again.

3.2.3 Recoverable System State

The system state α is recoverable if the component processes can be backed to their latest checkpoint and then by replaying the logged messages the process can get back to state α as if some equivalent failure free execution is going on. So the system state $D = [D_{xx}]$ is recoverable if in D_{xx} ,

$$\forall i [\text{Stable}(i, D_{ii})]$$

The current recoverable state (CRS) is the state to which the whole system can be restored in the event of failures in the system.

3.2.4 Finding Unique maximum System State

The old recovery vector ($R = \alpha_{xx}$) and the dependency vector are needed in determining the new recovery vector. At first when the system starts, the recovery vector of the system is initialised to zero vector. The recovery vector (RV_i) of a process i can change when

- i) a new process state interval became stable ($p > \alpha_{ii}$)
- or
- ii) a checkpoint has been taken or
- iii) messages received after effective checkpoint are logged.

Now the system recovery vector (R) is changed to reflect this new recovery vector value for process i by replacing its ith row by new recovery vector value. Now the new system recovery vector and dependency vectors are checked to find new current recovery state (see fig. 3.3a and 3.3b). In the beginning the recovery vectors of each process and the current recovery state are initialised to zero vectors.

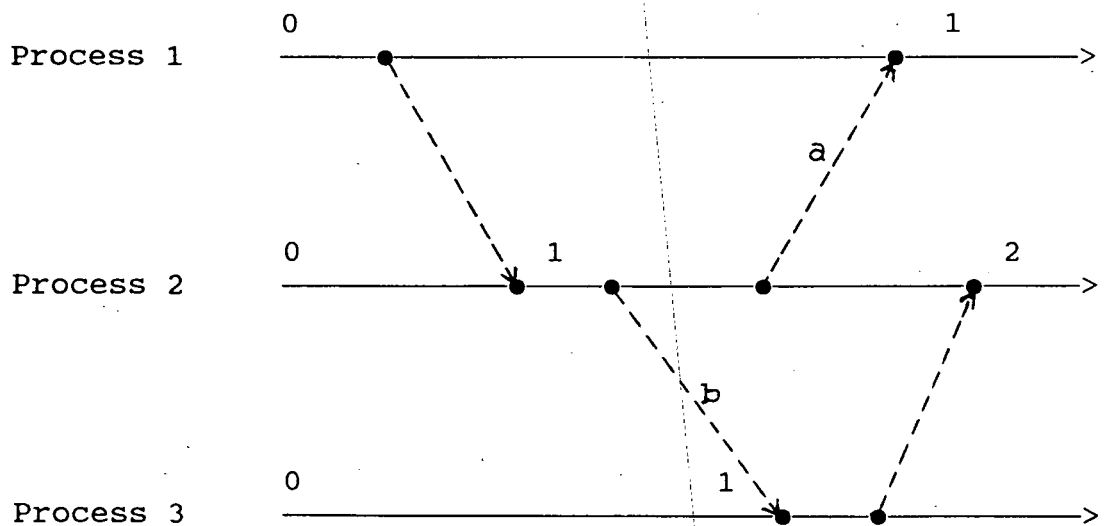


fig. 3.3a The system state

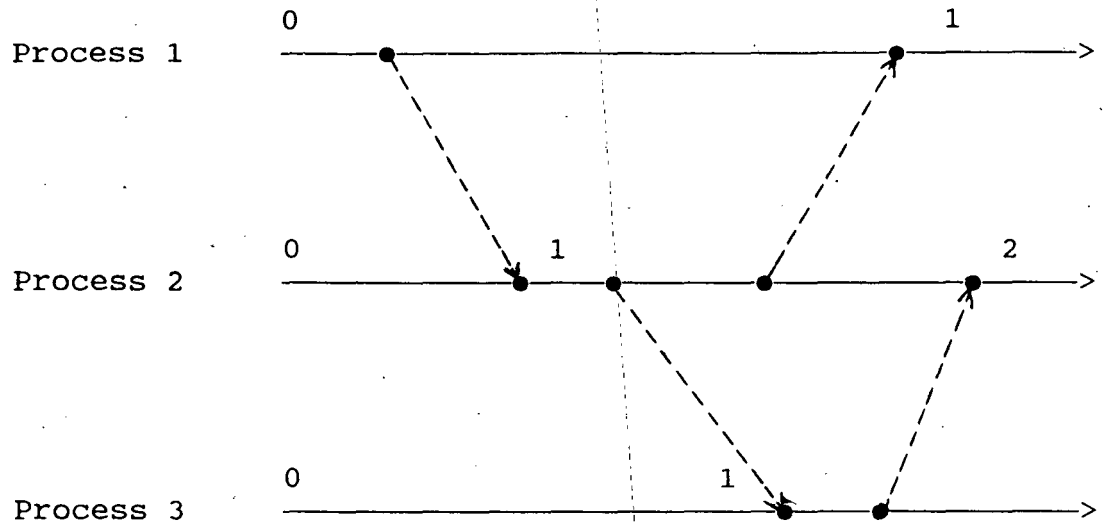
$$\text{CRS} = \langle 0 \quad 0 \quad 0 \rangle$$

$$\text{DV} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{R} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

R - VECTOR	ACTION	MODIFIED R - VECTOR	
	mesg. p logged		
$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	replace row 1 by row 1 of DV.	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	INCONSIS TENT
	Process 2 takes Checkpoint		
$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	replace row 2 by row 2 of DV	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 0 \end{bmatrix}$	INCONSIS TENT
	mesg. q is logged		
$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	replace row 3 by row 3 of DV	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 1 & 1 & 1 \end{bmatrix}$	CONSIS TENT
	CRS = < 0 0 1 >		
$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	replace row 2 by row 2 of DV	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	CONSIS TENT
	CRS = < 0 2 1 >		
$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	replace row 1 by row 1 of DV	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	CONSIS TENT
	CRS = < 1 2 1 >		

fig. 3.3b Determining maximum recoverable states

So the maximum recoverable system state will be $\langle 1 \ 2 \ 1 \rangle$. For this same example I will explain what happens when the same sequence of actions takes place.



Initially the current recovery state will be zero vector ($CRS = \langle 0 \ 0 \ 0 \rangle$). When the message a is logged, it can't be included in the CRS because process 2 has not yet logged its first message and hence the failure of process-2 makes it impossible for process 1 to send the message again since its recovery state is one. It defers this till process 2 makes its state interval 1 stable.

Now say process 2 takes checkpoint in state interval 2. Here also its stable state 2 is not included in the current recovery state. It defers this till message b becomes stable in process 3.

If process 3 logs its received message b, it can be included in the current recovery state. So the CRS becomes $\langle 0 \ 0 \ 1 \rangle$. This makes the waiting process 2 to put its

recovery state into CRS making it $\langle 0 \ 2 \ 1 \rangle$. Process 1 eventually finds that the state interval 1 of process 2 is stable and hence the current recovery state is changed to $\langle 1 \ 2 \ 1 \rangle$. Here the messages 1 and 2 of process 2 are never logged. So this method relinquishes from the constraint that all the messages received by a process should be logged. For algorithm and other definitions see [DAVI 90].

From the above method we can say that the current recovery state never decreases and hence domino effect will be taken care. Also the calculation of CRS goes concurrently with the other processes and the unique maximum system state is calculated at the earliest. By broadcasting CRS to all the sites, the failure of the central controller can be made robust. For the selection of central controller election algorithms are discussed in [RAYN 88].

3.3 RECOVERY FROM PARTITIONS

The data items are replicated to increase the availability of data. This reduces the network load caused by remote data accesses, improves the access time and increases the reliability against site failures. But this creates new problem due to communication failures, partitioning of network and lengthy communication delays. This can lead to conflicting updates in each of partition (communication delay can be considered as single node partition) compromising the consistency of the database.

The optimistic recovery strategy assumes that the failures rarely occur. So the data items are updated and are read in each of the partitions. Though the database will be consistent in each of the partitions but the global system state will be inconsistent. Two problems are associated with the partitions, one detection of partitions and two recovery from inconsistencies if any. The partition can only be detected during recovery phase as the each of the partitions will assume that the set of processes to which it could not communicate failed. Once the partitions are detected the inconsistencies if any are checked. To recover from inconsistencies the transactions need to be rolled back in both the partitions. Two strategies to recover from the partitions leading to inconsistencies are discussed. They are

- i) Version vector model and
- ii) Precedence graph model.

3.3.1 Version Vectors

This model is used in the design of LOCUS operating system [POPE 83]. The version vectors only detect write-write conflicts between the copies of same data item. Each copy of the logical data item maintains array of two fields. The size of the array is equal to the number of copies of the logical data item. The fields are a) the name of the site and b) number of times the file is updated in this

site. The conflicts due to partition can be found by comparing the version vector values; once the partition is detected. Two partitions are said to be not in conflict if the vector in one of the partitions dominates the vector in other partition (see examples below).

Partition 1. <A:1, B:2, C:4, D:3>	NO CONFLICT
Partition 2. <A:1, B:2, C:2, D:3>	Part.1 dominates part. 2
Partition 1. <A:1, B:2, C:2, D:3>	CONFLICT
Partition 2. <A:1, B:2, C:3, D:4>	Both part.1(C) and part. 2(D) trying to dominate.
Partition 1. <A:1, B:2, C:4, D:3>	NO CONFLICT
Partition 2. <A:1, B:2, C:3, D:4>	Part. 3 dominates other two
Partition 3. <A:1, B:2, C:4, D:4>	

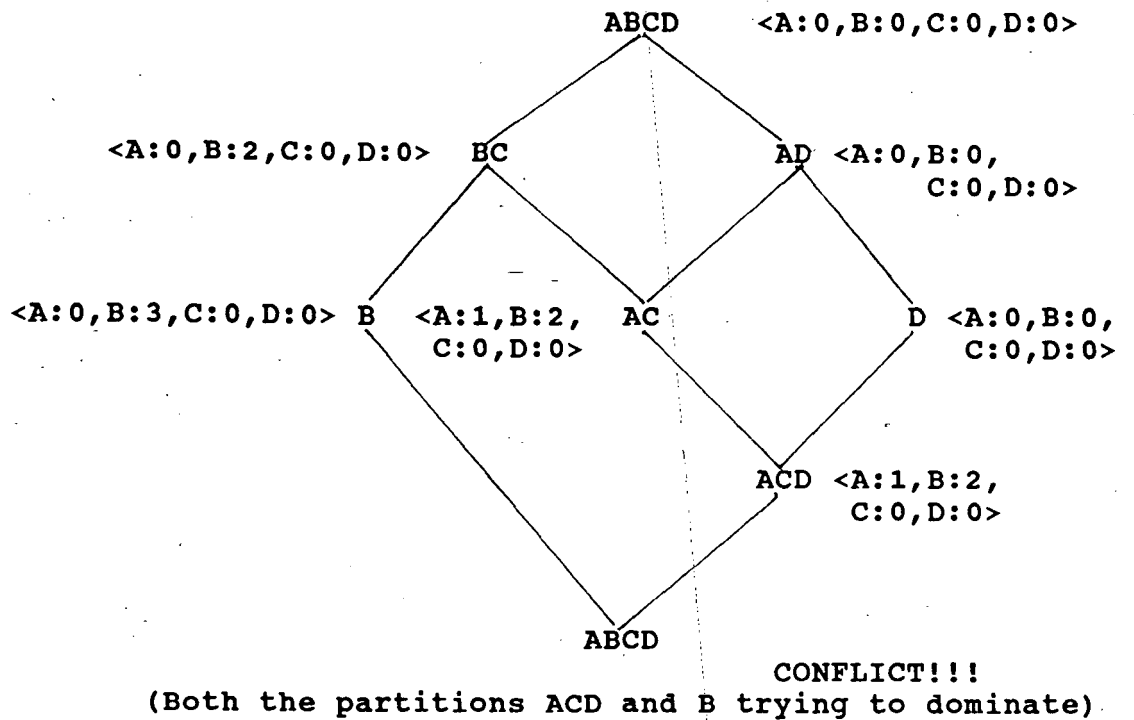


fig. 3.4 Recovery from partitions

The version vector method is not automatic i.e. once the conflicts between the partitions is detected the resolution is left to the database administrator (see fig.3.4).

This version vector model will work only in case of single file transactions. The conflicts are not detected in case of multifile transactions. For example consider a bank in which a person can withdraw money based on sum of his two account balances. The information regarding his balances are replicated at two sites A and B. The transaction and the version vector applicability are shown in fig. 3.5a and 3.5b. Though only Rs. 300 is in both accounts put together he could collect Rs. 450 and there is no conflict in the database detected by version vectors.

SITE-A	SITE-B
acctx.bal = Rs. 100	acctx.bal = Rs. 100
accty.bal = Rs. 200	accty.bal = Rs. 200
If acctx.bal+accty.bal>150	If acctx.bal+accty.bal>200
then	then
acctx.bal:=acctx.bal-150	accty.bal:=accty.bal-200
acctx.bal = Rs. -50	acctx.bal = Rs. 100
accty.bal = Rs. 200	accty.bal = Rs. 0

fig. 3.5a

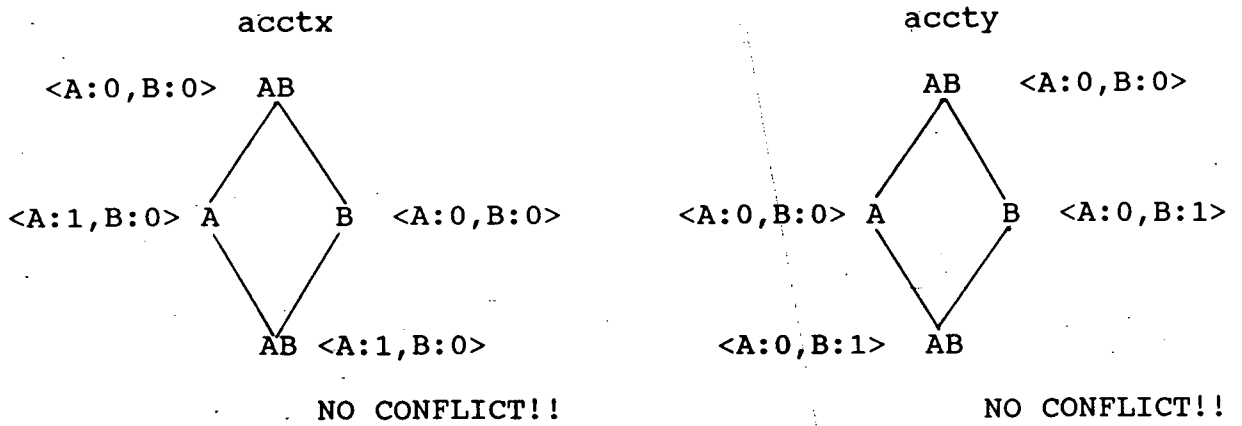


fig. 3.5b Undetected Inconsistencies

3.3.2 Precedence Graph Model

Precedence graphs are used to determine the serialization between the sites [DAVI 82,84]. Precedence graph model to check serialization in a single site is given in [PAPA 79]. To construct the precedence graphs the read and write requests are logged in stable storage. Once a partition is detected the logged requests in both the partitions are used to determine serializability. The conflicts between the partitions is detected by cycles in the precedence graph. The set of transactions $T_{i1}, T_{i2}, T_{i3}, \dots, T_{in}$ determine set of n transactions in serial order in partition i . The interaction between the transactions are represented by edges in the precedence graph. The interactions are of three types:

- i) Data dependency edges : $(T_{ij} \text{-----} \rightarrow T_{ik})$ Here the transaction k reads the value of data item updated by

ii) Precedence edges : $(T_{ij} \longrightarrow T_{ik})$ Here the transaction i reads a data item before transaction j writes it.

iii) Interference edges : $(T_{ij} \longrightarrow T_{lk})$ Here the transaction i reads the data item before the transaction k writes on to it in the other partition.

Here we assume that the readset of transactions include the writeset. Also the write-write conflict can be represented by a pair of read-write conflicts. The conflicts between the transactions and the cycle in the precedence graph is shown in fig. 3.6 .

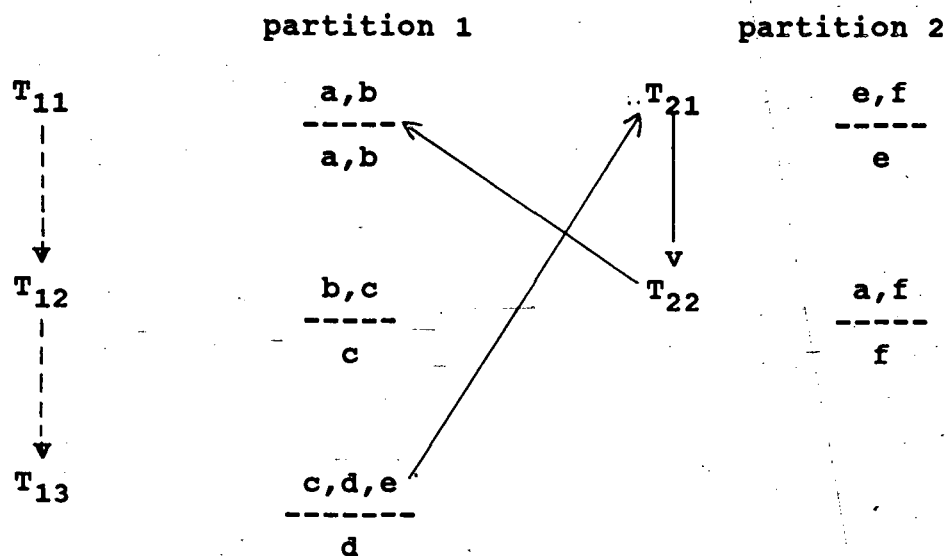


fig. 3.6 Precedence graph showing conflict

The precedence graph for a set of partitions is acyclic iff resulting database is consistent [DAVI 84]. If the precedence graph contain no cycles the last updated

copy of each data item is the correct value. Resolving the inconsistencies include undoing the transactions until the resulting graph is acyclic. It might result in uncontrolled rolling back of transactions. The transactions connected by precedence edges are not rolled back since they did not read the results of the rolled-back transactions. Also the transactions are rolled back opposite in direction to the execution of transactions.

Rolling back of the transactions may not be acceptable in some cases like banking. Once the money has been lent to the customer the rolling back of transactions is not just updates to the database. The precedence graph method is well suited

- i) when only small percentage of items were updated during partitioning and
- ii) when writesets are very small for most of the transactions.

3.4 CONCLUSION

This paper surveys the existing optimistic recovery techniques. The message logging and checkpointing strategy which also determines maximum recoverable system state is suitable when the failures are transient. When the failures are persistent either version vectors or precedence graph model can be used. But version vector model can be used to detect only write-write conflicts and can be used only with single file transactions.

PESSIMISTIC RECOVERY IN DISTRIBUTED SYSTEMS

4.1 INTRODUCTION

4.2 LOCK SERVER

4.3 CONSISTENCY FROM CRASHES

4.4 RECOVERY FROM SINGLE NODE FAILURES

4.5 MERGING OF PARTITIONS

4.6 RECOVERY FROM MULTIPLE PARTITIONS AND DYNAMIC VOTING

4.7 CONCLUSION

4.1 INTRODUCTION

The availability of the database can be increased by replicating the data at number of sites. But this poses problems of consistency in the events of nodal or communication failures which makes a node inaccessible or partition of the network. The pessimistic model tries to maximise availability of data items by using dynamic voting scheme but at the same time completely avoiding conflicting updates in the partitions. Also to ensure the consistency while updating the replicated data, the update needs to be done concurrently avoiding conflicting reads and writes. This needs locking of the logical portion of the database in all the sites where it is replicated. For this a centralized locking protocol with a decentralized recovery technique is discussed[POPEK 80]. The locking protocol itself is robust to the failures and the system recovers gracefully in the case of partition. The method to be discussed tolerates single node failures, partition and the failure of central controller. Maximum forward progress is achieved in case of any of the above mentioned failures.

4.2 LOCK SERVER (LS)

The central lock server maintains a lock table for the entire logical partition. The logical partition

consists of nodes which can communicate and have unique lock server. Each entry in the lock table is a tuple consisting of the host which initiated the request for the lock, the transaction or request sequence number and the part of the logical database locked by the request. Each host gives a unique number to the local request so that the host identifier and transaction number form the transaction or request identifier (see fig. 4.1).

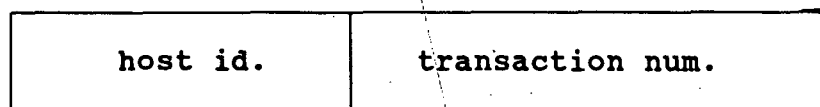


fig.4.1. Transaction Identifier

In addition to Central Lock Server (CLS) each of the nodes will have its own local lock servers (LLS). These local lock servers will contain information regarding data items locked at the site. The need for the LLS is to make the system resilient to the failures of CLS. In the event of the failure of CLS the lock table can be reconstructed from the information at the local lock servers. The selection of the new lock server is on the basis of some static priority assigned to nodes. This recovery mechanism is called **logical partition recovery**.

The lock controller contains the information pertaining to the sites where the data is being replicated and the list of nodes which are in its logical partition (the

uplist). The lock controller broadcasts these lists to all the sites (by piggybacking with outgoing messages). Also each site maintains the failure list. This consists of list of nodes which were in this partition and failed. These three lists, locations where each data item is replicated, uplist and failure list are maintained at each site in nonvolatile memory to make the partition resilient to the failures of the central lock server.

4.2.1 Lock Server - Selection and Services

The request to lock any data item is conveyed to the lock server. The lock server checks the lock table to determine whether the data item is already locked. If so the request is rejected. If not, it sends lock request to local lock servers of relevant sites containing replica of data item. The lock server also maintains for each lock x , $Loc(x)$, the set of sites which are relevant to lock x or the sites where replica of data items are locked by lock x [CHU 76].

The failure of lock server or partition of the system causes the new logical partition to undergo logical partition recovery. This includes the selection of new lock server and release of the locks for data items which failed majority voting. The transaction in this case is aborted in partition.

Once the communication is reestablished between the two partitions, the two lock controllers will communicate to establish single logical partition called **merging of partitions**. In addition to the selection of combined lock server, the number of majority votes for data items common to merged partitions is also changed accordingly.

4.2.2 Lock and Release Grants - Safe Talk Protocol

The requests from the external sources(application programs) are delivered to the lock server by the host which receives the request. These requests are for locking or releasing the locks at relevent sites where data item is replicated. Locking and releasing of locks arbitrarily leads to inconsistency. For this Safe Talk Protocol is used by lock server for locking and releasing at sites in the partition(see fig. 4.2).

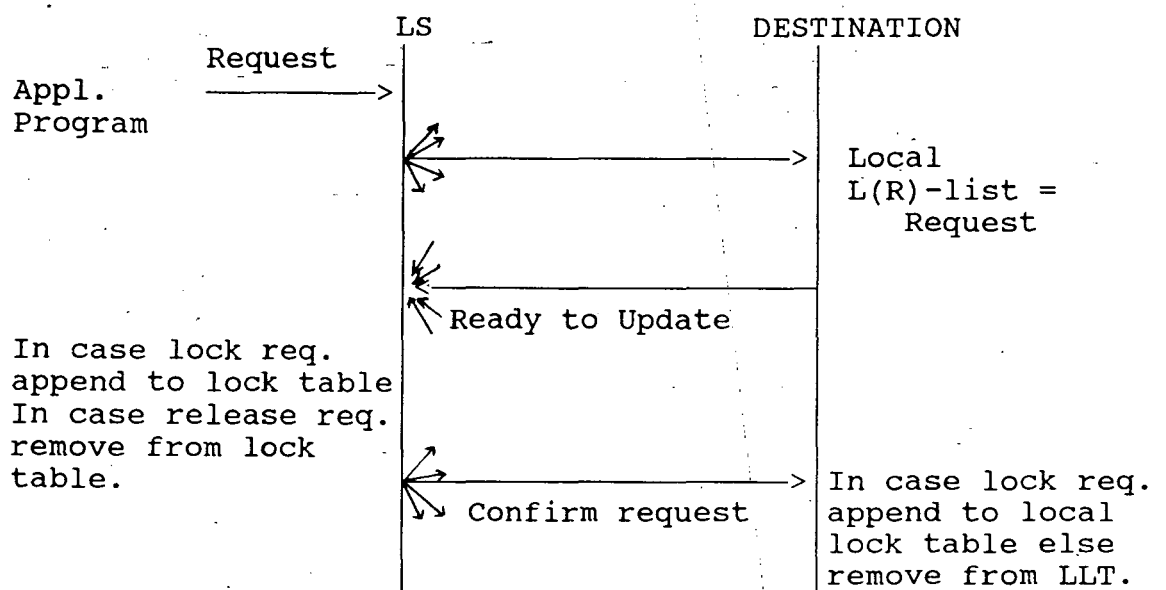


fig. 4.2 Safe Talk Protocol

The lock and release requests are placed in temporary buffers, L-lists for locks and R-lists for release. Then the lock server sends the requests to the relevant local lock servers and the remote sites also use temporary buffers. When the acknowledgement is received from relevant nodes, the lock server places the lock requests (removes in case of release) from the temporary buffers to the lock table and confirms the requests to local lock servers. On receiving the confirm request the local lock server commits the request by placing (removing) it in local lock tables.

The failure of node is determined by some timeouts and retransmissions at any stage of the Safe Talk Protocol. The communication failures leading to inaccessibility of nodes is treated the same way as failure of nodes.

4.3 CONSISTENCY FROM CRASHES

In central server $Loc(x)$ denotes the set of sites where the lock x is effective. From the Safe talk Protocol we can be sure that if x is a lock table entry in LS, then it must be present in the local lock servers at $Loc(x)$. If x is not present in the lock table of LS then it is not present in the local lock tables of any of $Loc(x)$ or it is present in the release tables in $Loc(x)$. This determines the consistency of the lock table.

The logical partition is internally consistent if the lock table is consistent and there is only one lock

server to the partition present at any time. The logical partitions are mutually consistent if the lock tables of partitions will not conflict at any time.

The recovery from failure will be consistent if the failure occurs before the recovery is complete or after the completion of the recovery. We denote recovery point as completion point.

Completion Point

TERMINAL FAILURE
(Before selection of new LS)

TRANSPARENT FAILURE
(After selection of new LS)

4.3.1 Logical Partition Recovery

When the lock server of a partition fails or the logical partition is partitioned resulting in one of the partitions being unable to communicate with the lock server, a new lock server is selected. This is called **Logical Partition Recovery**. The recovery from lock server crashes has two phases i) Nomination phase ii) Lock table update phase.

i) Nomination phase:

During the nomination phase the lock server will be selected and the lock and release tables are constructed from the local lock(L) and release(R) tables, at the local lock servers at all sites of the partition. The process which detects the failure of LS is responsible for nominating a new lock server. Some priority order called nomination order is used in selecting a site as LS. If a

node is selected as lock server it is the responsibility of the lock server to intimate other sites in the partition of its selection and creation of lock and release lists. The lock and release lists are circulated through entire partition to determine global L and R lists (see fig. 4.3).

We assume that lock request is delivered by LS and it reaches all the nodes in the partition. When LS delivers the release request, say due to communication failure, it may not reach some of the nodes. At this point of time, say LS fails and before selection of new LS the communication is restored. So for the same data item some of the nodes will contain lock-request and some will contain release requests leading to conflict. But sequence number of requests(see fig. 4.1) are used here to determine which request is the latest.

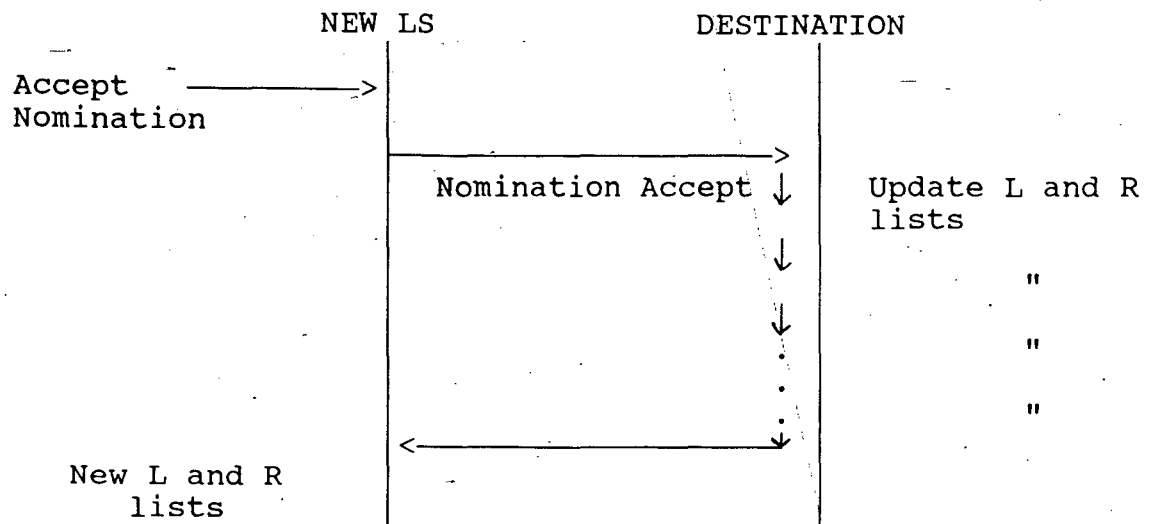


fig. 4.3 Nomination Phase

ii) Lock table update phase:

During the nomination phase the L and R lists of the central lock server, which are initially empty are constructed from the L and R lists of the sites present in the partition by passing the request and tables to all the sites in the partition in some order. The lock server determines majority vote for all locked data items. This can be obtained by determining identity of nodes as L and R lists pass through the sites.

During lock table update phase, the lock server adds the release requests for data items which lost majority due to partition to the R-list resulted from nomination phase and thus aborting the transaction in that partition. The Safe Talk Protocol is used in updating the L and R lists of all the sites(see fig. 4.4). The lock server determines uplist and new majority needed for updates after getting **Ready to Update** response from the sites.

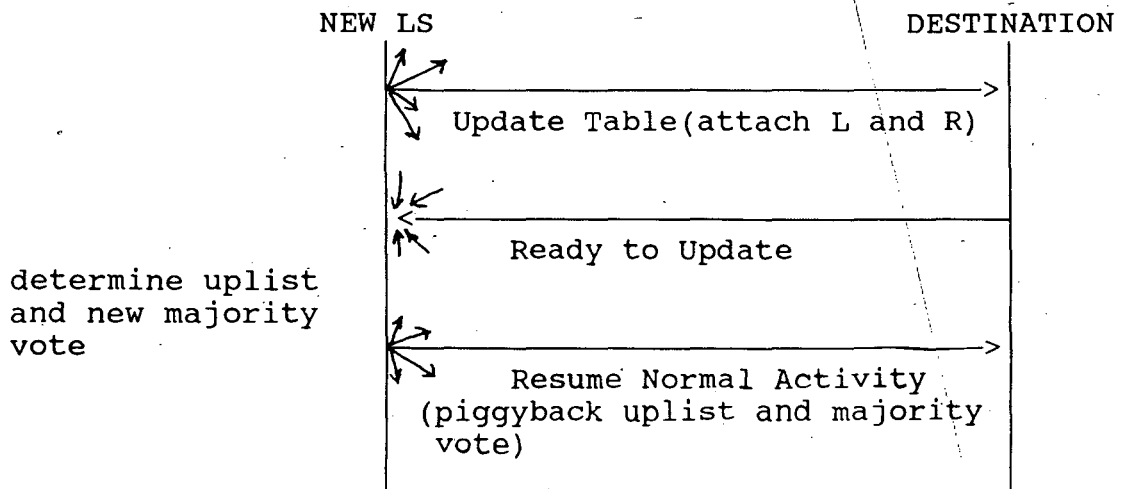


fig. 4.4 Lock Table Update Phase

The failure of the lock server during recovery causes restarting of the logical partition recovery mechanism. This logical partition recovery mechanism is in the event of failure of the new lock server. We will see that the new lock server is robust to the failures. The crash of new lock server can occur

- i) before sending request to local lock servers to update lock tables.
- ii) After sending update tables to local lock servers but before getting ready to update response from all the sites.
- iii) After all of the lock tables at the local lock servers are updated i.e. after issuing "Resume Normal Activity".

In case (i) logical partition recovery is restarted as there will be no change to L and R lists. In case (iii) the lists are updated completely and LPR is restarted. In case (ii) as we are using the Safe Talk Protocol the updated tables are kept in temporary buffers and unless Resume Normal Activity is issued, the local tables can not anyway use the information. This needs just logical partition recovery mechanism to be started again without any loss of consistency.

Let us see how the failures of nodes can be taken care of. The transparent failure of any node can be during nomination phase or lock table update phase. During nomination phase the failure causes the node to be removed

from the uplist and added to the failure list. During lock table update phase as we are using Safe Talk Protocol the failure will cause changes in the up and failure lists and also the majority is reduced by one for the replicated data items in the failed partition.

4.4 RECOVERY FROM SINGLE NODE FAILURES

The failure of a node forces the lock server to remove its identity from the partition. When the failed node is recovered, it sends a message to lock server to include it in the partition through any of the nodes that it could communicate. The lock controller sends part of the database for which this node maintains a replicated copy so as to preserve consistency of the database. In addition to this it sends L and R lists and any outstanding lock or release requests.

In case the logical partition is itself undergoing logical partition recovery then the recovering node becomes a partition on its own nominating itself as lock server to the single node partition. It initialises its L and R lists to empty. When the LPR is completed it will undergo Logical Partition Merge with the partition (see sec. 4.5).

The failure of a recovering node restores the state of the partition to the previous state. During recovery, if the LS of the partition fails then single node recovery mechanism is started all over again.

4.5 MERGING OF PARTITIONS

When partition occurs new lock server will be chosen at the partition which does not have access to lock server. When communication is reestablished between the partitions, a combined lock server is chosen and the tables are updated from the information present at the two lock controllers. This is called **Merging of Partitions**. Partitions are always merged pairwise. The merging of partitions has got two phases:

- i) Detection of communication between the partitions and
- ii) Merge of partitions.

During the first phase central lock server at any of the partitions is responsible for detection of communication between the partitions. It will try to send messages, after each timeout, to the nodes which are in the failure list to determine whether they are up. One of the partition is denoted as primary and the other secondary. The selection of primary is on the basis of some priority information or on the basis of which of the two partitions detects the communication between the partitions first. Then the transaction common to both the partitions are aborted at both partitions to enable consistency of information replicated in both partitions. For this uplists of both partitions are exchanged.

During the merge phase the data items are exchanged to enable both the partitions to contain latest values of data as maximum voting would have allowed only one partition to update replicated data items. The uplist of the merged partition will be the union of the uplists of the two partitions. The lock server which is primary will be the lock server for the combined partition. The updates to tables, data items and new majority vote are done by Safe Talk Protocol with piggybacked updates to enable both sites to finish the recovery at the earliest.

Either the primary or secondary lock servers can fail during detection phase or merge phase. If it is during detection phase the merge process is aborted and the partition chooses new server. If it is during the merge phase then also the partition which lost the lock server will undergo the selection of new lock server after aborting the merge. If updates to data items are made it will not cause any conflicts as Safe Talk Protocol is used. Because the partition again will lose the majority to update and this makes the data item inaccessible. So updates will be reflected at all nodes in the partition unless a failure of node within partition, which we already proved is robust in single node recovery. Once the selection of new lock server is completed the merge is restarted.

4.6 RECOVERY FROM MULTIPLE PARTITIONS & DYNAMIC VOTING

When partition occurs (either single node or multinode) the partition uses majority voting [THOM 76] to determine which partition is allowed to have access to data item. But this reduces the availability when multiple partitions occurs.

This dissertation contributes in increasing the availability of data items in the event of multiple partitions by using dynamic voting technique. This is a refinement of the techniques suggested by researchers like Barbara, D., and Garcia-Molina, H. [BARB 86], Jajodia, S., and Mutchler, D. [JAJO 87].

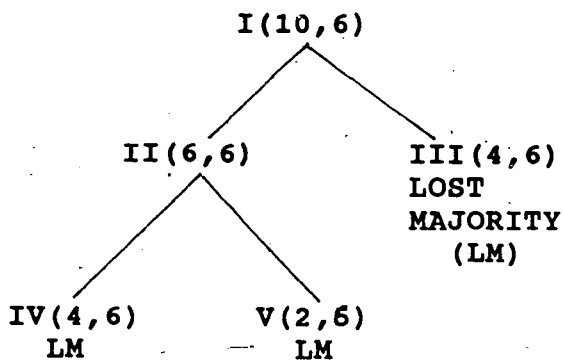


fig. 4.5a STATIC VOTING

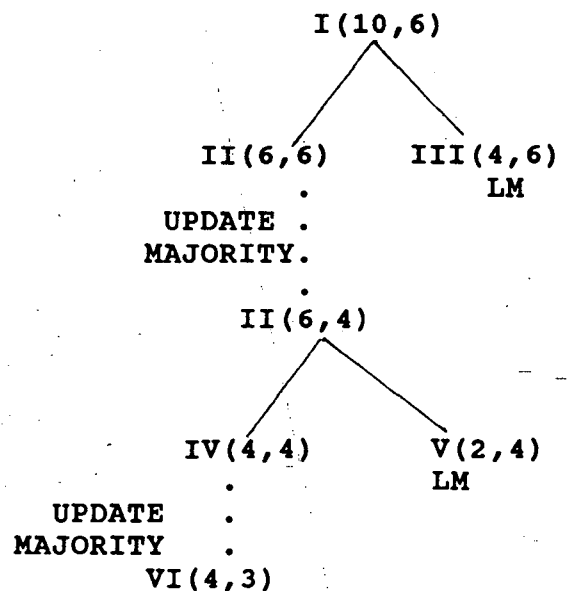


fig. 4.5b DYNAMIC VOTING

fig. 4.5 The tuple (x,y) , x denotes number of sites where the data item is replicated, y denotes number of votes needed to access the data item. New level denotes partition.

To increase availability, it is needed to dynamically reduce the number of votes required to access a data item when a node failure or partition failure occurs. For example when a partition occurs the partition which gets majority vote will be having right to access data item. Once it further partitions it is possible that no partition can access the data item (fig. 4.5a).

This method allows access to replicated data even when only one node in which data replicated is up but under certain conditions as follows:

- i) partition occurs only pairwise
- ii) partition of already partitioned network will occur only after the partition detects the partition and some table management is done and
- iii) In the event of partition divides the nodes into exactly two halves then some priority of the nodes shall be taken to decide upon which partition gets the right to access the data item.

The problem with this method is that when two partitions merge, even though they retain majority vote for some data item, they are not allowed to access the data item as some partitions (even though it is a single node) will be having majority to access the data item. For example in fig. 4.5b if partitions V and III merge, six replicas of data item will be available in the merged partition, but they

should not be allowed to access the data item. The partition gets right to access the data item only when it merges with a partition having right to the access data item.

The above solution gives rise to another problem. Assume that a node having right to access a data item fails. In the partition in which it is present majority vote and uplists are updated and it continues with updates. When the failed node recovers, it individually has got the right to access data item or in case it merges with some partition which is not having the right to access data item will get the right resulting in the conflicting updates to the data items. For this reason when the node recovers it should check its uplist to determine whether it is the only node in the partition before it failed or not. If not, it will lose majority otherwise it retains the right.

But the above implementation has got one more problem. In the partition which is having majority vote to access a data item, all the nodes fail at a time. Here each node during recovery will disqualify itself from accessing the data item and hence the data item cannot be accessed any longer. But one of the solutions is to wait till all the nodes in which the data item is replicated to be recovered and by looking into the uplists we can determine the last set of nodes that have failed concurrently. But this is not a feasible solution. Another solution which determines the

the last set of node(s) which failed(concurrently) is discussed in the next section.

4.6.1 To Determine Last Set of Node(s) Failed

We assume that some data item X is replicated in 'n' nodes and each node maintains up and failure lists. The uplist(failurelist) of a node 'i' gives the set of nodes that are up(down) as far as the node knows. Each time when some node 'i' recovers, we remove from set S (initially S = n) the nodes which failed before 'i'. This will be done until set S is a subset of set of recovered nodes. Now this S gives the set of nodes which have failed concurrently (see fig. 4.6) .

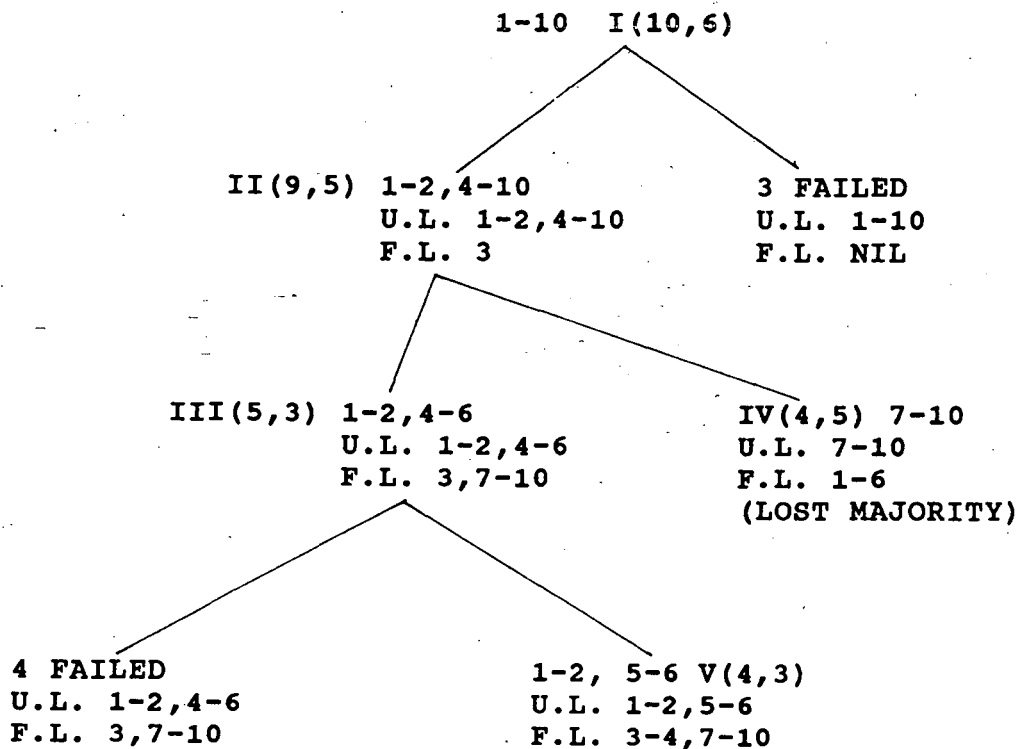


fig. 4.6 Up and Failure Lists on Partition(n=10)

Now say all the nodes of partition V failed concurrently with right to access the data item and up and failure lists. Also failed nodes 3 and 4 have right to have access to data item. Now we will see how to determine the last node(s) to fail. We will continue removing nodes from set S (initialized to n, the set of nodes where data item X is replicated) until S is a subset of recovered nodes (R). When this condition is satisfied, S gives the last set of nodes failed.

RECOVERED NODES	CANDIDATES FOR LAST NODES
R = 0	S = 1 to 10
R = 4	S = (S - (F.L. of 4)) = 1 to 2, 4 to 6.
R = 4, 2	S = 1 to 2, 5 to 6
R = 4, 2, 3	S = 1 to 2, 5 to 6
R = 4, 2, 3, 1	S = 1 to 2, 5 to 6
R = 4, 2, 3, 1, 5	S = 1 to 2, 5 to 6
R = 4, 2, 3, 1, 5, 6	S = 1 to 2, 5 to 6

Here S is a subset of R. So the last set of nodes to fail are 1, 2, 5 and 6.

4.7 CONCLUSION

This model is an extension to the existing dynamic voting models [BARB 86, JAJO 87]. These models either need at least four sites to determine the majority or use a hybrid model to switch back to static voting when the number of sites in the model becomes less than four.

CHAPTER 5

CURRENT AND FURTHER RESEARCH

Further research in recovery in distributed systems is expected in the following directions:

- 1) Performance tradeoffs between various strategies of recovery (pessimistic and optimistic).
- 2) Detecting partition in the network.
- 3) Dealing with nested transactions both in pessimistic and optimistic cases.
- 4) Cost effective way of implementing fail-stop processors.
- 5) Rather than having single centralized server(although it is robust to the crashes), with the number of sites increasing in the system the load on centralised server increases. The solution is to divide the sites into groups so that each group will have it own group server. There will be a server sitting on top of the group server. But the problem is how to divide the system intelligently so that for transactions generated in a group the most probable sites to be accessed should be present in that group.
- 6) considerations of heuristics to determine majority of a partition in assymmetric networks which are not totally connected and in which some links are more reliable than others.

Recently two papers came out with extentions to the existing models. One is Bhargava's [BHAR 90] paper on increasing availability using tokens so that each partition is authorised to have atleast one operation on a data item. The other is by Jajodia [JAJO 90], which uses dynamic voting algorithm in determining the majority.

R E F E R E N C E S

[ALGI 85]

ALGIRDAS AVIZIENIS, The N-version Approach to Fault Tolerant Software, IEEE Transactions on Software Engineering, Vol. SE-11, No. 12, Dec. 1985, pp. 1491- 1501.

Diversified hardware, software and repetitive computation to check the correctness of result and hence by achieving the fault tolerance in distributed systems is the motivation behind this paper.

[ALSB 76]

ALSBERG, P.A., AND DAY, J.D., A Principle for Resilient Sharing of Distributed Resources. In proc. of the 2nd international conference on software engineering, Oct. 1976, IEEE Computer Society, pp. 627-644.

[ANDE 83]

ANDERSON THOMAS AND JOHN C. KNIGHT, A Framework for Software Fault Tolerance in Realtime Systems, IEEE Transactions on Software Engineering, Vol. SE-9, No. 3, May 1983, pp. 355-364.

This paper gives a software fault tolerant scheme for realtime systems. It introduces new concept "The Exchange" mechanism which is a restricted form of conversation. This also classifies various errors like Internal error, external error, pervasive error, persistent error and transient error.

[BARB 86]

BARBARA, D., GARACIA-MOLINA, H., AND SPAUSTER, A., Protocols for Dynamic Voting Reassignment, In Proceedings of the IEEE Conference on Distributed Computing, ACM, New York, 1986, pp. 195-205.

[BHAR 90]

BHARGAVA, B., AND LIAN, S.R., Typed Token Approach for Database Processing During Network Partitioning, Conference on Management of Data (COMAD 90), Dec. 12-14, 1990, New Delhi, India, (ed.) Naveen Prakash, Tata McGraw-Hill Publication Company Limited, pp. 162-194.

This model is designed to see that each of the partitions authorized to have atleast one operation in the event of multiple partitions.

This is done by redistributing the tokens when partition is anticipated and thus increasing the availability of the database.

[CHAN 85]

CHANDI K. MANI AND LAMPORT LESLIE, Distributed Snapshots: Determining Global States of Distributed Systems, ACM Transactions on Computer Systems, Vol. 3, No. 1, Feb. 1985, pp. 63-75.

Determining the global state of the system has got many advantages like avoiding deadlocks, taking checkpoints which are free of domino effect, duplicate messages and inconsistencies in the system etc. Algorithm with informal proof for determining global system state is being discussed in this paper.

[CHU 76]

CHU, W.W., Performance of File Directory Systems for Databases in Star and Distributed Networks. Proc. National Computer Conference, 1976, pp. 577-587.

[COUL 88]

COLOURIS F GEORGE, AND JEAN DOLLIMORE, Distributed Systems (Concepts and Design), Addison-Wesley Publishing Company, 1988.

This text focuses on design of distributed systems and gives the design of Grapavine system developed and designed at Xerox Palo Alto Research centre. This also gives an account on load balancing, performance evaluation and distributed system management.

[DAVI 84]

DAVIDSON B SUSAN, Optimism and Consistency in Partitioned Distributed Database System, ACM Transactions on Database Systems, Vol. 9, No. 3, Sept. 1984, pp. 456-481.

This is an optimistic protocol. It uses incidence and other edges to determine cycles in the graph on the merging of the partitions. The conflicting updates are represented by cycles in the graph. In addition to this it describes how to backout and the selection of transactions for backout so that optimal recovery takes place and at low cost. Finally it gives how this model suits the

applications like baking systems, airline reservation system and periodic updating systems like weather forecasts etc.

[DAVI 90]

DAVID B JOHNSON AND WILLY ZWAENEPOEL, Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing, Journal of Algorithms, 11, Sept. 1990, pp. 462-491.

The primary concern of this paper is how to overcome domino effect and to find the maximum recoverable system state concurrent to the process execution and to see that this system state never decreases i.e. in the event of failure the system state will never go beyond this recoverable system state. Taking into account existing checkpoint at the same time avoiding logging of messages in the event of checkpoint how to find the maximum recoverable system state is the main idea behind this paper. It uses a centralized protocol to determine maximum recoverable system state. Synchronization is needed during recovery.

[EAGE 83]

EAGER, D., AND SEVCIK, K.C., Achieving Robustness in Distributed Database System. ACM Transactions on Database Systems, Sept. 1983, pp. 354-387.

[ENSL 78]

ENSLAW H PHILIP, Jr., What is "Distributed" Data Processing System ? , IEEE Computer, Jan. 1978, pp. 13-21.

This paper introduces distributed systems and its properties which lies in varying degrees. These include multiplicity of resources, distribution of resources, highlevel operating systems, transparency of complex system and autonomy and cooperation.

[FRED 84]

FRED B. SCHNEIDER, Byzantine Generals in Action : Implementing Fail-Stop Processors, ACM Transactions on Computer Systems, Vol. 2, No. 2, May 1984, pp. 145-154.

This fail-stop mechanism uses $(2k + 1)$ replicas as against $(k+1)$ replicas needed to k resilient database to overcome from Byzantine faults by

masking. The idea is not to induce uncertainties in the system, detection of failure of other nodes and using stable storage to avoid conflicting updates leading to inconsistencies.

[GARC-82]

GARCIA-MOLINA HECTOR, Reliability Issues for Fully Replicated Distributed Databases, IEEE Computer, Sept. 1982, pp. 34-42.

This paper discusses reliability issues and problem associated with fully replicated databases and the solutions. The whole problem is solved using seven design steps. Also the cost involved with these design considerations.

[GARC 85]

GARCIA-MOLINA HECTOR, SUSAN B DAVIDSON AND DALE SKEEN, Consistency in Partitioned Networks, ACM Computing Surveys, Vol. 17, No. 3, Sept. 1985, pp. 341-369.

Tradeoff between correctness and availability is discussed with examples. It surveys the various strategies, pessimistic and optimistic recovery techniques in the event of partition. The optimistic strategies discussed are version vectors, optimistic protocol. The pessimistic strategies include primary site copy, tokens, voting, missing writes, accessible copies and class conflict analysis. A combined strategy (optimistic + pessimistic) also discussed.

[GIFF 79]

GIFFORD, D.K., Weighted Voting for Replicated Data. In proc. of 7th symposium on operating system principles, ACM, New York, Dec. 1979, pp. 150-162.

[GOOD 81]

GOODMAN, NATHAN AND PHILIP B BERNSTEIN, Concurrency Control in Distributed Database Systems, ACM Computing Surveys, Vol. 13, No. 2, June 1981.

This is a survey paper on concurrency control in distributed systems. It examines thoroughly the serializability problem, two-phase protocol, primary copy, voting, Centralized two-phase locking, timestamp ordering, multiversion timestamp ordering, conservative time stamp ordering etc.

[GOOD 84]

GOODMAN, NATHAN AND PHILIPA BERNSTEIN, An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases, ACM Transactions on Database Systems, Vol. 9, No. 4, Dec. 1984, pp. 596-615.

This paper gives concurrency control algorithm in replicated environment which is robust to the failures of the nodes. The idea is to increase the availability by making the object item available to the extent that atleast one copy is available. Care is taken to see that this last copy is not obsolete. It will not work in the event of partitions in the network.

[JAJO 87]

JAJODIA, S., AND MUTCHLER, D., Dynamic Voting. In Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM, New York, 1987, pp. 227-238.

[JAJO 90]

JAJODIA, SUSHIL AND DAVID MUTCHLER, Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database, ACM Transactions on Database Systems, Vol. 15, No. 2, June 1990, pp. 230-280.

This paper gives two models for improving availability in distributed systems. In the first model only site failures are considered and in the second model only link failures are considered. Two algorithms considered are dynamic voting and dynamic linear voting.

[KENN 85]

KENNETH P. BIRMAN, THOMAS A. JOSEPH, THOMAS RAEUCHLE AND AMR E.L. ABBADI, Implementing Fault Tolerant Distributed Objects, IEEE Transactions on Software Engineering, Vol. SE-11, No. 6, June 1985, pp. 502-508.

This paper gives how to build a k-resilient system (i.e. which withstands the failure of k nodes) to make the system tolerant to the failures of nodes. It uses checkpointing to retain the status of the system. This need to be synchronized while taking the checkpoint. It also discusses optimization while taking checkpoints and how to cope with total system failures.

[KIM 84]

KIM K.H., Distributed Execution of Recovery Blocks : An Approach to Uniform Treatment of Hardware and Software Faults, IEEE, Proceedings of 4th International Conference on Distributed Computing Systems, May 1984, pp. 526-532.

Recovery Block strategy for software and hardware fault tolerance is presented. Three schemes of increasing complexities and with more effective tolerances are given.

[KOHL 81]

KOHLER WALTER H., A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems, ACM Computing Surveys, Vol. 13, No. 2, June 1981.

This paper surveys the various solutions for recovery and attaining concurrency in decentralized systems. The concurrency control mechanism chosen are locking, time stamps, permits, tokens and conflict analysis. Recovery techniques and atomic actions and their usage are clearly explained.

[KOO 87]

KOO RICHARD AND SAM TOUEG, Checkpointing and Rollback Recovery in Distributed Systems, IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, Jan. 1987, pp. 23-30.

This paper gives an algorithm to store the status of the system (checkpointing) and restoring the status of the system when inconsistencies are detected. The algorithm proposed is robust to the failures of system either partial or complete. The checkpointing forces minimal number of processes to take checkpointing to make this globally consistent.

[LAMP 78]

LAMPORT L, Time, Clocks and the Ordering of Events in a Distributed System, Communications of the ACM, 21, 7, July 1978, pp. 558-565.

It is shown that the ordering of events (happens before relation) forms partial order and which can be implemented by counters. It also descusses on how to synchronize the physical clocks for event synchronization.

[MINO 82]

MINOURA, T., AND WIEDERHOLD, G., Resilient External True Copy Token Scheme for Distributed Database System, IEEE Transactions on Software Engineering, May 1982, pp. 173-189.

[PAPA 79]

PAPADIMITRIOU, C.H., Serializability of Concurrent Database Updates, Journal of ACM, 26, 4, Oct. 1979, pp. 631-653.

It is shown that serialization of the interleaved transaction is NP - complete. It is also shown that the concurrency control protocols existing currently generates special classes of schedulers which can determine the order of events.

[PARK 83]

PARKER D SCOTT, POPEK G.J., GERARD RUDISIN, Detection of Mutual Inconsistency in Distributed systems, IEEE Transactions on Software Engineering, Vol. SE-9, No. 3, May 1983, pp. 240-247.

The idea behind this paper is to increase the availability in Distributed systems. This makes database to be updated even when the partition occurs. It uses version vectors to detect inconsistency after the partitions merge. It is a simple mechanism and makes it unnecessary to store all the operations on the object when the network is partitioned. It works only for transactions accessing a single file and it may fail to detect inconsistency when the transaction accesses multiple files.

[POPE 80]

POPEK, G.J., DANIEL A. MENASCE AND RICHARD R. MUNTZ, A Locking Protocol for Resource Coordination in Distributed Databases, ACM Trans. on Database Systems, Vol. 5, No. 2, June 1980, pp. 103-138.

This paper gives solution for how to update replicated database using locks by the transaction coordinators at each site. A centralized controller which is robust to the node crash and recovery which is distributed is proposed. Also the recovery is to maximum unique state. It is a

pessimistic recovery mechanism and it works only when all the sites where replica of data is present are up and are in the same partition. An informal proof for this centralized locking and distributed recovery mechanism is given.

[RAND 75]

RANDELL, BRIAN, System Structure for Software Fault-Tolerance, IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 220-232.

This paper gives detailed report on recovery blocks and conversation schemes, the fault resilient design techniques.

[RAYN 88]

RAYNAL MICHEL, Distributed Algorithms and Protocols, Translated from French by Jack Howlett, John Wiley and Sons, 1988, ISBN 0 471 91754 0.

Various algorithms in diversified areas in distributed systems like deadlock detection and resolution algorithms, termination algorithm, election and mutual exclusion algorithms, avoiding Byzantine quarrel algorithms are discussed.

[SKEE 85]

SKEEN, DALE, Determining Last Process to Fail, ACM Transactions on Computer Systems, Vol. 3, No. 1, Feb. 1985, pp. 15-30.

This paper discusses how to determine the last process or last set of processes which failed so as to get the information of which process has got latest information of certain data item. This is needed when a process continues with its execution (updatation) when the other process fail. To determine this each process will maintain up and failure lists of other processes. It is assumed that when a process fails it will inform other processes of its failure.

[STON 76]

STONEBRAKER, M., AND NEOHOLD, E., A Distributed Data Version of INGRES, MEMO ERL, M612, Electronics Res. Lab, Univ. of California, Berkeley, Sept. 1976.

[SUNI 85]

SUNIL, K.S., BARBARA BLAUSTEIN AND CHARLES W. KAUFMAN, System Architecture for Partition-Tolerant Distributed Databases, IEEE Transactions on Computers, Vol. C-34, No. 12, Dec. 1985, pp. 1158-1163.

This protocol uses time stamps to determine the latest copy of the data item. This works well in cases where integrity of the data is not important. Threat to integrity of data occurs due to failures or partitions in the system. It assumes full replication in the system and will not work in the event of nodes joining or dropping from the system.

[SOVB 89]

SVOBODOVA LIBA, Attaining Resilience in Distributed System, (ed.) Anderson T., Dependability of Resilient Computers, BSP Professional Books, 1989, pp. 98-124.

This paper explores various failure modes in distributed system and the inconsistencies that these failures lead to. The concept of atomic action is explained in detail. Achieving atomicity using Timestamp ordering, Commit protocols (like two-phase commit protocol) are also explained. The concept of nested atomic actions, replicated objects and partitioning problems are discussed.

[THOM 76]

THOMAS, R.H., A Solution to Update Problem for Multiple Copy Databases Which Uses Distributed Control, Tech. Rep. 3340. Bolt Beranek and Newman, July 1976.

[THOM 79]

THOMAS, R.H., A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases, ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979, pp. 180-209.

This protocol assumes that the database is totally replicated at all the sites. Each of the replica of database is associated with a controller. Also the replicas are associated with timestamps to distinguish from fresh and obsolete informations in the event of failures. Each request is resolved by voting by the controllers

and checking the time stamp. So a centralized time stamp is needed.

[TYRR 86]

TYRRELL, ANDREW M., AND DAVID J. HOLDING, Design of Reliable software in Distributed Systems Using Conversation Scheme, IEEE Transactions on Software Engineering, Vol. SE-12, No. 9, Sept. 1986, pp. 921-928.

This paper gives how the status of the system can be represented using petrinet model. It gives how to detect the conversation, the beginning and ending of the conversation between a set of processors. The assumption here made is that the system can be expressed as a set of communicating sequential processes.

[YEMI 85]

YEMINI SHAULA AND ROBERT E. STROM, Optimistic Recovery in Distributed Systems, ACM Transactions on Computer Systems, Vol. 3, No. 3, Aug. 1985, pp. 204-226.

This paper discussess an optimistic recovery mechanism in which communication, checkpointing and computation goes asynchronously. It uses session sequence numbers to determine whether the sender or receiver failed. It also uses incarnation numbers to determine whether the message is duplicate or not. Absolutely no synchronization during recovery or at any time needed.
