

**CONCURRENCY CONTROL AND RECOVERY IN
REPLICATED DISTRIBUTED DATABASES**

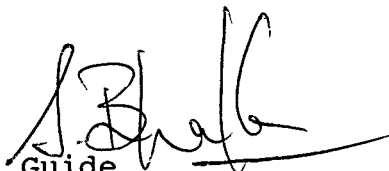
*Dissertation submitted to Jawaharlal Nehru University
in partial fulfilment of the requirements for the
award of the Degree of*
MASTER OF TECHNOLOGY
IN
COMPUTER SCIENCE

POLEPALLI KRISHNA REDDY


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

CERTIFICATE

The research work entitled **CONCURRENCY CONTROL AND RECOVERY IN REPLICATED DISTRIBUTED DATABASES**, embodied in this Dissertation has been carried out in the School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi. This work is original and has not been submitted in part or in full for any other degree or diploma of any university.



Guide
SCSS, JNU



Dean
SCSS, JNU

ACKNOWLEDGEMENTS

I owe my sincere thanks to my supervisor Dr. Subhash Bhalla for his uncompromising guidance, constant supervision and constructive criticism without which this work would not have been completed successfully.

I thank the dean and faculty of the School of Computer and System Sciences at Jawaharlal Nehru University (JNU) for teaching me Computer Science in a friendly and lucid environment.

I thank library staff of JNU and I.I.T, Delhi, for helping in getting references about this subject, and for providing xeroxing facilities.

I thank my parents for their support, which has encouraged me to do my M.Tech.

I wish to thank my friends who are directly and indirectly helped me to complete this project.

POLEPALLI KRISHNA REDDY
January 1991

ABSTRACT

Many distributed systems replicate data for fault tolerance, availability and improved response time. In a one copy distributed database each data item is stored at exactly one site. In a replicated database, some data items are stored at multiple sites. In such systems, a logical update on data item results in physical update on a number of copies. By storing important data at multiple sites the distributed database system can operate even though some sites have failed.

Recently several strategies have been proposed for transaction processing in replicated distributed database systems. In this report some of the strategies have been surveyed. At first the motivation to replication is discussed. Then, the problems and correctness criterion are presented. Next, different strategies are presented, after, these are compared in the light of cost, achieving high availability and fault tolerance.

A new algorithm is discussed based on **executing the transaction after visiting majority of sites**. The algorithm is discussed with proof and examples.

CONTENTS

ABSTRACT

CHAPTER 1 - CONCURRENCY CONTROL AND RECOVERY IN DISTRIBUTED REPLICATED DATABASE ENVIRONMENT	1-27
1.1 INTRODUCTION	1
1.2 DISTRIBUTED DATABASES	1
1.3 DATA REPLICATION	2
1.4 ADVANTAGES OF REPLICATION	4
1.5 PROBLEMS AND ISSUES OF REPLICATION	6
1.6 FAILURES AND RECOVERY	12
1.7 CORRECTNESS CRITERIA	17
1.8 ORGANIZATION OF THE STUDY	27
CHAPTER 2 - REPLICA CONTROL ALGORITHMS	28-53
2.1 INTRODUCTION	28
2.2 COMMON ASSUMPTIONS	28
2.3 CONCURRENCY CONTROL AND CONSISTENCY OF MULTIPLE COPIES OF DATA IN DISTRIBUTED 'INGRES'	29
2.4 A MAJORITY CONSENSUS APPROACH TO CONCURRENCY CONTROL FOR MULTICOPY DATABASES	31
2.5 THE QUORUM CONSENSUS ALGORITHM	35
2.6 AN ALGORITHM FOR CONCURRENCY CONTROL AND RECOVERY IN REPLICATED DISTRIBUTED DATABASES	37
2.7 MISSING WRITES ALGORITHM	40
2.8 RESILIENT EXTENDED TRUE COPY TOKEN SCHEME FOR A DISTRIBUTED DATABASE SYSTEM	42

2.9	SEMANTICS BASED TRANSACTION MANAGEMENT TECHNIQUE FOR REPLICATED DATA	48
2.10	CONCLUSION	53
CHAPTER 3 -	COMPARISON OF ALGORITHMS	54-74
3.1	INTRODUCTION	54
3.2	COST COMPARISON	56
3.3	SITE FAILURES	67
3.4	PARTITIONING BEHAVIOUR	69
3.5	READ-WRITE RATIO	71
3.6	DEADLOCK DETECTION AND OTHERS	73
3.7	CONCLUSION	74
CHAPTER 4 -	AN ALGORITHM BASED ON EXECUTING REQUEST AFTER VISITING MAJORITY OF SITES	75-93
4.1	INTRODUCTION	75
4.2	ENVIRONMENT	76
4.3	ASSUMPTIONS AND TERMINOLOGY	79
4.4	INTER SITE COMMUNICATION	81
4.5	TIME STAMPS	83
4.6	FAILURE ASSUMPTIONS	84
4.7	ALGORITHM	84
4.8	EXAMPLE	86
4.9	SITE CRASHES AND PARTITIONING BEHAVIOUR	89
4.10	WHY THE ALGORITHM WORKS	89
4.11	CONCLUSION	93
CHAPTER 5 -	CONCLUSION	94
BIBLIOGRAPHY		

CHAPTER 1

CONCURRENCY CONTROL AND RECOVERY IN A DISTRIBUTED REPLICATED DATA BASE ENVIRONMENT

1.1 INTRODUCTION

Developments in technology have made it possible to inter-connect a number of computer systems to form a computer network. The problem of distributing a database among the different computer systems or sites to form a distributed database system is an active research area.

1.2 DISTRIBUTED DATABASES

The main difference between centralized and distributed data base systems is that, in the former the data resides in one location while in the later the data resides in the several locations. There are number of differences between supporting a centralized database and distributed database such as transaction execution, reliability of processing, and problems in supporting distributed database are listed below.

Consider a distributed database, each data item 't' having only one copy at any site. If any transaction [ESWA76] (which issues reads and writes) needs that data item then, the message must be communicated to that site and response is also delayed. A lot of communication overhead

on the communication (telephone or microwave) system. In this, both read and write requests need equal communication overhead, even though read request does not modify the database (which is not in case of replicated one). That is, the request is communicated to that site where a data item resides, and response is also transmitted to the site, where the request is originated.

The second aspect is that of reliability problem. In a single copy database system, (DBS) if site crashes, then the data at that site is lost. The transactions which depend on the data cannot be executed, and the system must be shutdown until recovery. This is undesirable in the case of many applications such as in airline reservation system, railway reservation system, telephone operating systems. These and similar applications, must be supported with uninterrupted service.

The other problem is that of improving response time. For example, if the number of transactions needs single item (by maintaining two phase locking or time stamping), the queue would be very long, and response is delayed considerably.

1.3 DATA REPLICATION

One of the approaches to the above problem is that data is stored reduntantly at various sites. But there is a lot

of over head on storage. Can we tolerate this extra burden? Consider the recent advances in technology [HEVM88].

- * Hardware advances include high speed VLSI processors, large capacity memories, rapid magnetic and optical disk drives and sophisticated input/output devices. These components enabled the developments of micro computer work stations with impressive capabilities to serve as a distributed system sites.
- * Communication and networking advances include rapid data transmission media, such as optical fibre, microwave and satellite.
- * Software advances allowed most application systems to function efficiently and correctly on distributed networks with multiple users.

In addition, standards for integrating all the above technologies into a complete distributed system are being developed and accepted by most of the international community.

With these advances, we can go for replicated data base systems (RDBS).

Basically, replication means that, data items are stored redundantly at geographically dispersed locations. In replicated environment, read can access at local site

which is nearest. But for write operation, each copy must be updated. That is, read operations executed efficiently. But, for write operations there is a lot of communication overhead.

1.4 ADVANTAGES OF REPLICATION

The main advantages for going replication are reliability, increase in parallelism and performance.

* **Reliability:-** There are many types of failures. For example, node failures, communication link failures, malevolent failures etc. Some of the failures are detectable, while others are not. We should know which type of failures the system is protected against and also how many. In the RDBS if one site fails the remaining sites are able to continue operating. Transactions can be run at any site. After detecting failure the transactions which belong to that failed site can be forwarded to other sites. Thus the failure of the site does not necessarily imply the shutdown of the system.

The failure of any site must be detected by the system, then appropriate action may be needed to recover from failure. The system no longer uses the services of failed site. Finally, when the failed site recovers (or repaired) mechanisms must be available to

integrate it smoothly back into the system. In RDBS the ability of the most of the system to continue to operate, despite the failure of the one site results in increase in availability. Availability is crucial for real time applications. Loss of access to data, for example in airline may result in the loss of potential ticket buyers to competitors.

- * **Increased parallelism:-** In case, where the majority of access to the data items results in any reading of the data item, the several sites can process queries involving that data item, in parallel. The more replicas of data item there are the greater the chance that needed data is found in the site, where the transaction is being executed. Hence, data replication minimizes data accessing overhead. Data copies are placed to provide acceptable availability to all system sites that are closely located.

Till now it is clear as to how the replication technique is superior to single copy distributed systems. However, there remains a fair amount of overhead and problems involved in designing and implementation phase. The benefits and cost of data replication are very difficult to measure.

1.5 PROBLEMS AND ISSUES OF REPLICATION

Some of the problems of replication technique are presented below.

Redundant update problem:- The system must ensure that all replicas of data item 'x' are consistent [DAVI86]. But, the inherent communication between sites maintain copies of database makes it impossible to ensure that all copies remain identical at all times when update requests are being processed. The principal goal of update mechanism is, to guarantee that updates get applied to the database copies in a way that, preserves the mutual consistency [STON83] of the collection of database copies as well as the internal consistency of each database copy. Otherwise, erroneous computations may result. The notion of database consistency has two aspects.

Mutual consistency of redundant copies

The update transactions incur greater overhead to ensure mutual consistency. Mutual consistency requires all copies of database must be identical. This means whenever a data item is updated, then update must be propagated to all sites containing, replicas, resulting in an increased data maintenance overhead. For example, in a banking system where account information is replicated in various sites, it is necessary that transactions must ensure, the balance in a particular account agrees at all sites, to a common value.

Consider a banking database that contains a checking account and saving account for a certain customer, with a copy of each account stored at both sites A and B. Figure 1.1 shows the value of accounts at site A and site B.

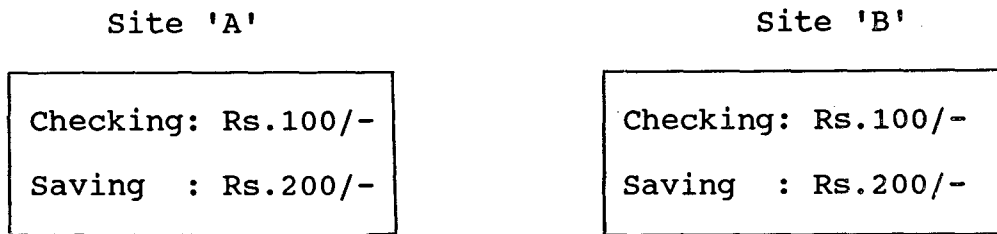


Figure 1.1

Suppose two update requests come at the same time at site A and site B.

R_1 : Checking = checking -50; R_2 : Checking = checking +100.
Saving = saving + 100.

Each update is based on the database state in Figure 1.1. If both updates are processed at their home sites there is no ordering then the new state of the database is shown in the Figure 1.2



Figure 1.2

In the figure 1.2, the value of each account is not same in both sites. So, the mutual consistency of database has been disturbed.

Internal consistency

The internal consistency requires that each copy of the database remain consistent within itself just as a non-redundant database must. It concerns the preservation of invariant relations that exist among items within a database. Maintaining internal consistency is overhead as compared to single copy database. Most of the responsibility for the internal consistency of database must rest with the application process which updates it. The update mechanism should not destroy the internal data relationships of the database.

Consider the same banking database as shown in the figure (1.1)

Further assume that the relation

$$(\text{Checking} + \text{saving}) \geq 0$$

must be preserved for the database.

Consider the update requests 'R3' and 'R4'

$$R_3: \text{Checking} := \text{Checking} - 200;$$

$$R_4: \text{Checking} := \text{Checking} - 150;$$

Each of which is based on the initial database state. If both R_3 and R_4 are applied regardless of the order of

application, the internal consistency of the database (the relation (checking + saving) $> = 0$) will be destroyed. Hence, one of the requests must be rejected, in order to preserve the internal consistency of the database.

Concurrency control

Concurrency control [BERN81] is the activity of coordinating concurrent accesses to a database in a distributed database system. Concurrency control permits users to access a database in a multi-programmed fashion while preserving the illusion that each user is executing alone on a dedicated system. The main technical difficulty in attaining this goal is, to prevent database updates performed by one user from interfering with database retrievals and updates performed by another. Read and write actions issued by users, concurrently can corrupt data correctness.

The goal of concurrency control is to prevent interference among users who are simultaneously accessing a database. Let us illustrate the problem with two examples.

Example #1: Suppose two customers simultaneously try to deposit money into the same account. In the absence of concurrency control, the two activities could interfere. Initially the balance in the account is Rs.500/-.

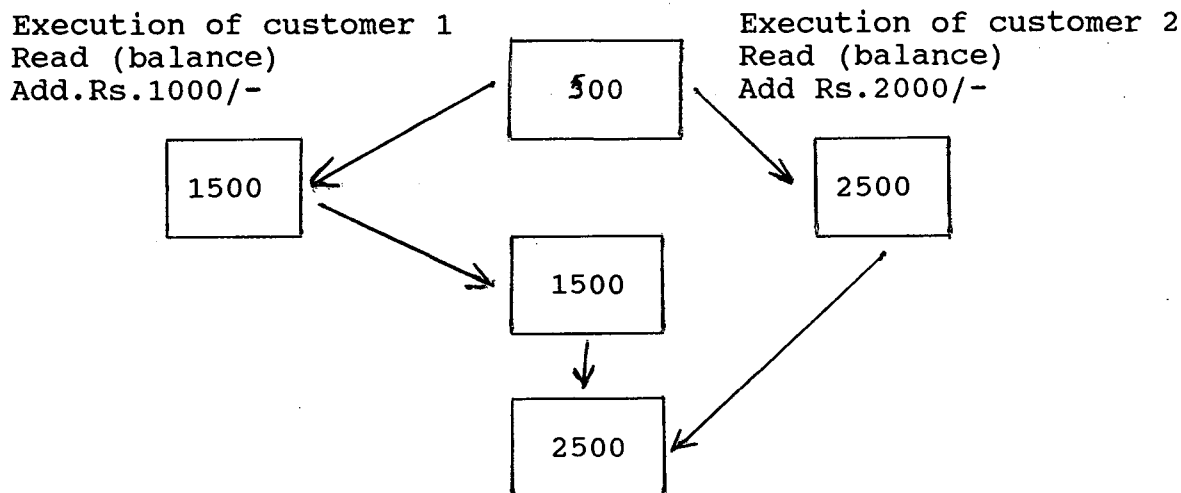


Figure 1.3

The two machines handling the two customers could read the account balance at approximately same time, compute new balance in parallel and then store new balance back into the database. The net effect is incorrect (Figure 1.3). Although two customers deposited the money, the database only reflects one activity; the other deposit is lost by the system.

Example #2: Suppose two customers simultaneously execute the following transactions with the initial database state given in figure 1.4.

Customer 3: Move Rs.1000/- from saving account to checking account.

Customer 4: Read the total balance in saving and checking.

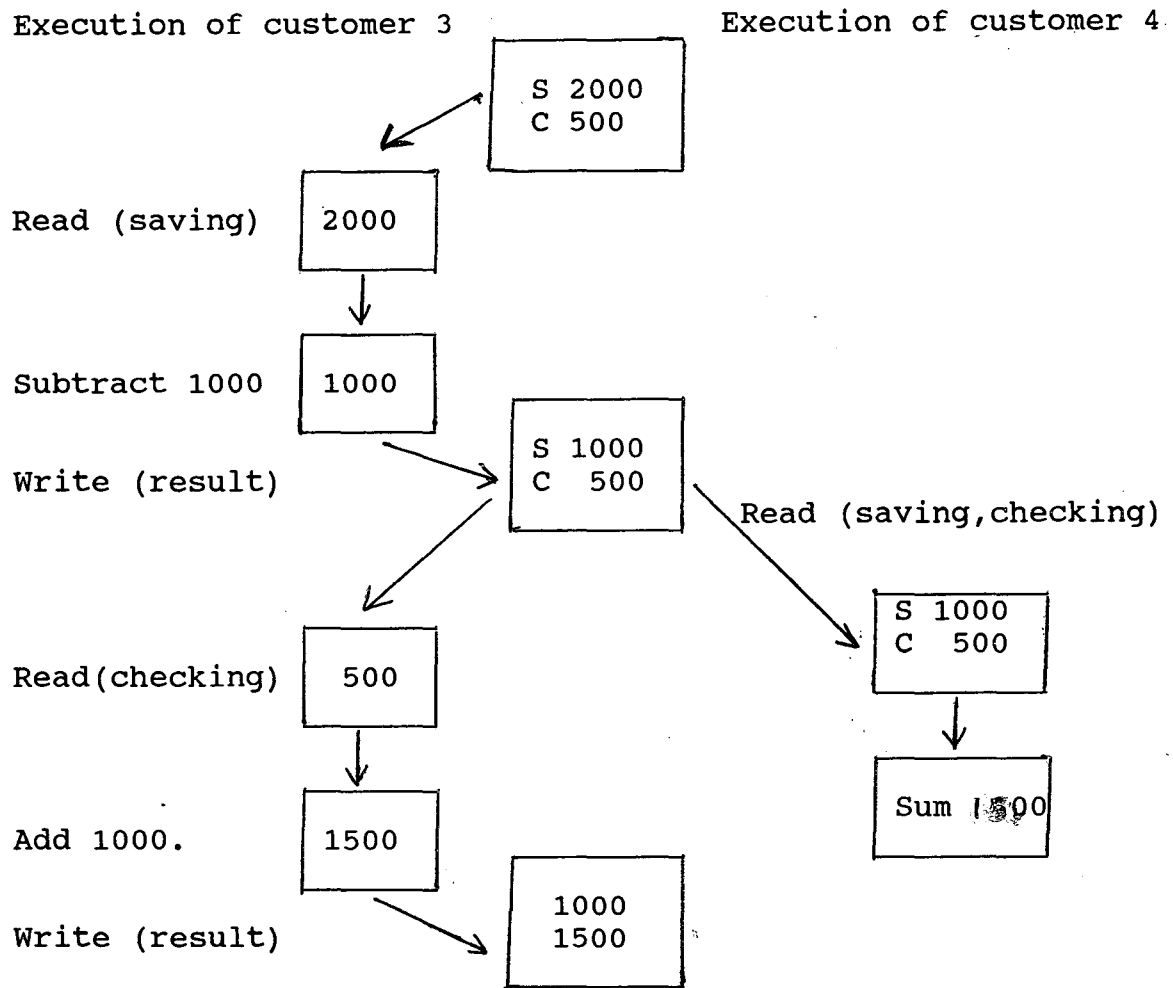


Figure 1.4

In the absence of concurrency control these two transactions could interfere. The first transaction might read the saving account balance. Subtract Rs.1000/- and store result back into the database. The second transaction might read the savings and checking account balance and print the total. Then, the first transaction might finish the funds transfer by reading the checking account balance, adding Rs.1000/-, and finally storing the result in the

database (Figure 1.4). In this, the final value placed into the database by this execution is correct. But, still the execution is incorrect because the balance received by the customer 4 is Rs.1000/- short.

The above examples don't exhaust all possible ways in which concurrent users can interfere. However, these examples are typical concurrency control problems that arise in distributed DBS. Concurrency control methods such as two-phase locking [KORT86] are expensive and may lead to system dead lock.

However, the problem of controlling concurrent updates by several transactions to replicated data is more complex than the centralized approach to concurrency control.

1.6 FAILURES AND RECOVERY

System failures and transaction abortions due to concurrency control requires recovery methods that ensure consistent recovery [KOHL81] to all data copies. The failure and recovery problem for a distributed DBS is more complex than that for centralized DBS. More kinds of failures must be considered. Communication links cause new kinds of failures such as partition failures.

A distributed system consists of two kinds of components: sites which process information, and communication links, which transmit information from site to

site. A distributed system is commonly depicted as a graph where nodes are sites and undirected edges are bi-directional communication links as shown in the Figure 1.5.

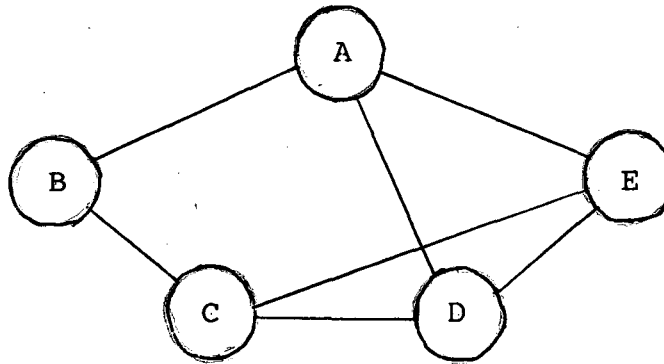


Figure 1.5

We assume that the graph is connected, means that there is a path from every site to every other. Thus, every two sites can communicate either directly via link joining them, or indirectly via a chain of links. The combination of hardware and software that is responsible for moving messages between sites is called a computer network.

A distributed system may suffer from the same types of failures that a centralized system does (for example: memory failure, disk crash). There are however additional failures that need to be dealt with in a distributed environment, including:

- a) The failure of a site;
- b) The failure of link;
- c) Loss of message;
- d) Network partition.

In order to be robust, the system must therefore detect any of these failures, reconfigure the system so that computation may continue, and recover, when a processor or link is repaired.

Site failures

When a site experiences a system failure, processing stops abruptly and contents of volatile storage are destroyed. In this case, we will say the site has failed. When the site recovers from failure it first executes a recovery procedure, which brings the site to a consistent state so that, it can resume normal processing.

In this model of failure, a site is always either working correctly or not working at all; it never performs incorrect actions. This type of behaviour is called fail-stop [SCHL88], because sites fail only by stopping.

Even though each site is functioning properly or has failed, different sites may be in different states. A partial failure is a situation where some sites are operational while others are down. Total failures occur when all sites are down.

Partial failures are tricky to deal with. Mainly this is because, operational sites may be uncertain about the state of failed ones. Atomic commitment protocols are designed to minimize the effect of one sites failure on

other site's ability to continue processing.

Communication failures

Communication links are also subject to failures. Such failures may prevent processes at different sites from communicating. A message may be corrupted due to noise in a link; a link may malfunction temporarily, causing a message to be completely lost; or link may be broken for a while, causing all messages sent through it to be lost.

Message corruption can be effectively handled by using error detecting codes, and by retransmitting a message in which the receiver detects an error. Loss of messages due to transient link failures can be handled by retransmitting lost messages. If a message is sent from site A to site B, but, the network is unable to deliver the message due to broken link, it may attempt to find another path from A to B whose intermediate links and sites are functioning properly. Error correcting codes, message retransmission, and rerouting are usually provided by computer network protocols.

Network partitioning

Unfortunately, even with automatic routing a combination of site and line failures [BERN87] can disable the communication between sites. This will happen if all the paths between two sites A and B contain a failed site or

a broken link. This phenomenon is called a network partition. In general a network partition divides the operational sites into two or more components, where every two sites within a component can communicate with each other, but sites in different components cannot. Figure 1.6 shows the partitioning of the system of Figure 1.5. The partition consist of two components, {B,C} and {D,E}, and is caused by the failure of site A and links (C,D) and (C,E).

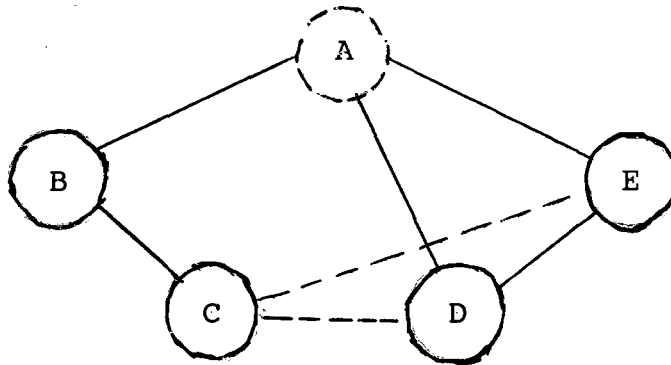


Figure 1.6

As sites recover and communication links are repaired, communication is re-established between sites that could not previously exchange messages, thereby merging the components. For example, in figure 1.6, if site A recovers or if either link (C,D) or (C,E) is repaired, the two components merge and every pair of operational sites can communicate.

It is generally not possible to differentiate between link failure and network partition. We can usually detect that a failure has occurred.

Security

Replication of data increases the security risk of exposing sensitive information and of providing an opportunity for the corruption of data.

Although there is a considerable overhead with replication, on the whole, it enhances the performance of read operations and increases the availability, parallelism and reliability. Because of advances in technology people are working for better replicated techniques to minimize the overhead.

1.7 CORRECTNESS CRITERIA

There are two correctness criteria for replicated databases.

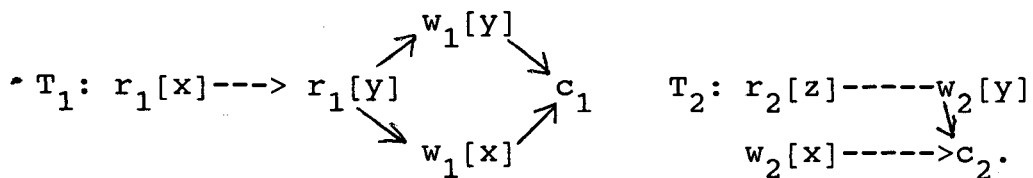
(1) Concurrency control

When a set of transactions execute concurrently, the operations may be interleaved. We model such an execution by a structure called a history. A history indicates the order in which the operations of the transactions executed relative to each other. Since some of these operations may be executed in parallel, a history is defined as partial order [BERN87]. If transaction ' T_i ' specifies the order of its operations, there two operations must appear in that order in any history that includes ' T_i '. In addition, we

require that a history specify the order of all conflicting operations that appear in it.

Two operations are said to conflict if they both operate on the same data item and at least one of them is a write. Thus $\text{read}(x)$ conflicts with $\text{write}(x)$, while $\text{write}(x)$ conflicts with both operations $\text{read}(x)$ and $\text{write}(x)$. If two operations conflict, their order of execution matters. The value of 'x' returned by $\text{read}(x)$ depends up on whether or not that operation proceeds or follows a particular $\text{write}(x)$. Also, the final value of 'x' depends on which of the $\text{write}(x)$ operations is processed last.

To illustrate, consider the two transactions. (r = read, w = write, c = commit)



The possible history (H_1) is shown in Figure 1.7.

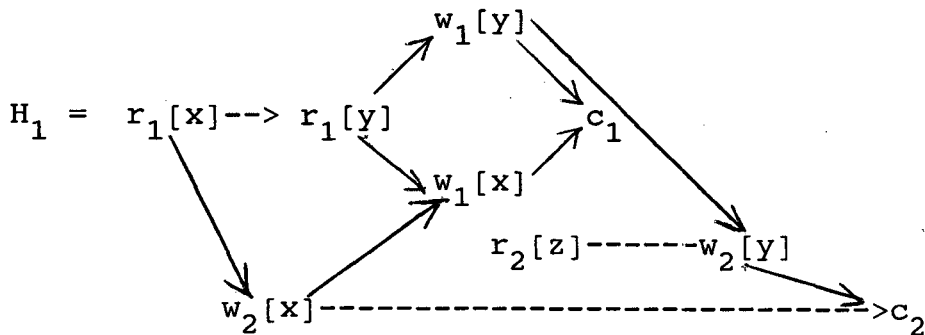


Figure 1.7

Consider figure 1.3 suppose two transactions are executed one at a time in the order 'customer 1' followed by customer 2. The execution sequence is as shown in ^{Fig} 1.8(a).

Customer 2 after Customer 1	Customer 1 after Customer 2
Road (balance)	Road (balance)
Add Rs.1000/-	Add Rs.2000/-
Read (balance)	Read (balance)
Add Rs.2000/-	Add Rs.1000/-
(a)	(b)

Figure 1.8

The final value of balance is correct. Similarly if the transactions are executed one at a time customer 2 followed by customer 1, then corresponding execution sequence is shown in ^{Fig} 1.8(b).

The execution sequences described above are called serial histories. Thus, a serial history represents an execution in which there is no interleaving of the operations of different transactions. Each transaction executes from beginning to end before the next one can start.

The histories described above are called serial histories. When serial transactions are executed in parallel, the corresponding history need no longer be serial. But not all parallel executions result in an incorrect state.

Each transaction when executed alone, transfers the system from one consistent state to another consistent state. A natural way to define a correctness in a concurrent system, is to require that the outcome of processing a set of transactions concurrently, be the same as one produced by running these transactions serially in the same order. A system that ensures this property is said to ensure serializability [BERN79].

In order to formalize the concept of serializability, we need to define a notion of equivalent histories [BERN87]. We can say two histories are equivalent(=) if

- 1) They are defined over the same set of transactions and have the same operation; and
- 2) They order conflicting operations of non-aborted transactions in the same way that is, for any conflicting operations 'p_i' and 'q_j' belonging to transactions T_i and T_j (respectively if p_i <_H q_j then p_i <_H q_j).

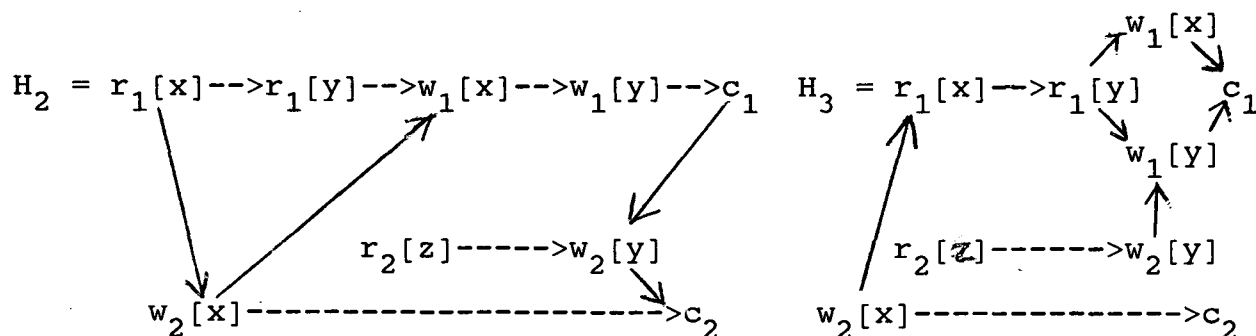


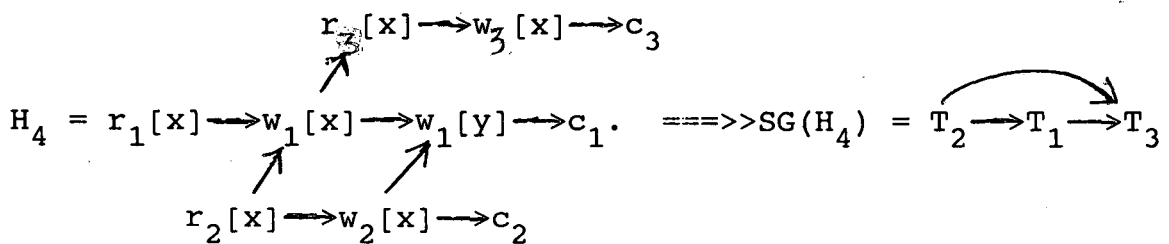
Figure 1.9

Underlying this definition, the outcome of a concurrent execution of transactions depends only on the relative ordering of conflicting operations. To see this observe that executing nonconflicting operations in either order has the same computational effect. Conversely, the computational effect of executing two conflicting operations depends on their relative order.

For example, given the three histories shown in Fig.1.9, $H_1 = H_2$ but, H_3 is not equivalent to either.

111

We can determine whether a history is serializable by analyzing a graph derived from the history called a serialization graph. Let H be a history over $T = \{T_1, \dots, T_n\}$. The serialization graph (SG) for H , denoted $SG(H)$, is a directed graph whose nodes are the transactions in T that are committed in H and whose edges are all $T_i \rightarrow T_j$ ($i \neq j$) such that one of the T_i 's operations precedes and conflicts with one of T_j 's operations in H . For example:



The edge $T_1 \rightarrow T_3$ is in $SG(H_4)$, because $w_1[x] < r_3[x]$, and the edge $T_2 \rightarrow T_3$ is in $SG[H_4]$ because $r_2[x] < w_3[x]$. Notice that a single edge in $SG(H_4)$ can be present because



of more than one pair of conflicting operations. For instance,

$T_2 \rightarrow T_1$ is caused both by $r_2[x] < w_1[x]$ and $w_2[y] < w_1[y]$.

Each edge $T_i \rightarrow T_j$ in $SG(H)$ means that at least one of T_i 's operations precedes and conflicts with one of T_j 's. This suggests that T_i should precede T_j in any serial history that is equivalent to H . If we can find a serial history, H_s , consistent with all edges in $SG(H)$, then $H_s = H$ and so H is SR. We can do this as long as $SG(H)$ is acyclic.

The correctness criteria for concurrency control is, as long as $SG(H)$ of the history of corresponding parallel execution is acyclic the database is consistent.

(2) Replication control

In replicated database, each data item 'x' has one or more copies, denoted by x_A, x_B, \dots , at different sites. Users interact with the system by running transactions that issues reads and writes on data items.

But, as far as users are concerned they should not feel that complexity of replication. So, for the DBS managing a replicated database should behave like a DBS managing a one copy (non replicated) database. In a one copy database, users expect the interleaved execution of their transactions

to be equivalent to a serial execution of those transactions. Since replicated data should be transparent to them, they would like the interleaved execution of their transactions on a replicated database to be equivalent to serial executions of those transactions on a one copy database. Such executions are called one copy serializable (1-SR). This is a goal of concurrency control for replicated data.

The correctness criteria for replication can be explained by considering two types of histories: replicated data (RD) histories and one copy histories.

Replicated data histories [BERN85]

Let $T = \{T_0, \dots, T_n\}$ be a set of transactions. To process operations from T , a DBS translates T 's operations on data items into operations on the replicated copies of those data items. We formalize this translation by a function 'h' that maps each $r_i[x]$ into $r_i[x_A]$, where x_A is a copy of x ; each $w_i[x]$ into $w_i[x_{A1}], \dots, w_i[x_{Am}]$ for some copies x_{A1}, \dots, x_{Am} of x ($m > 0$); each c_i to c_i and each a_i (abort) into a_i .

A complete replicated data(RD) history H over $T = (T_0, \dots, T_n)$ is a partial order with ordering relation $<$ where:

1. $H = h(\bigcup_{i=0}^n T_i)$ for some translation function h ;

2. For each T_i and all operations p_i, q_i in T_i , if $p_i < q_i$, then every operation in $h(p_i)$ is related by $<$ to every operation in $h(q_i)$;
3. For every $r_j[x_A]$, there is at least one $w_i[x_A] < r_j[x_A]$;
4. All pairs of conflicting operations are related by $<$, where two operations conflict if they operate on the same copy and at least one of them is write; and
5. If $w_i[x] <_i r_i[x]$ and $h(r_i[x]) = r_i[x_A]$ belongs to $h(w_i[x])$. For example consider the following transactions (T_0, T_1, T_2, T_3) (figure 1.10): $w_0[x]$.

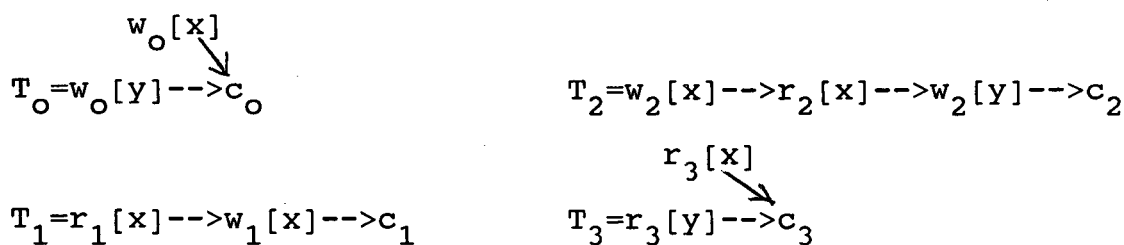


Figure 1.10

The database consists of data items x, y with copies x_A, x_B, Y_C, Y_D . The following history, H_5 (Figure 1.11) is an RD history over $\{T_0..T_3\}$:

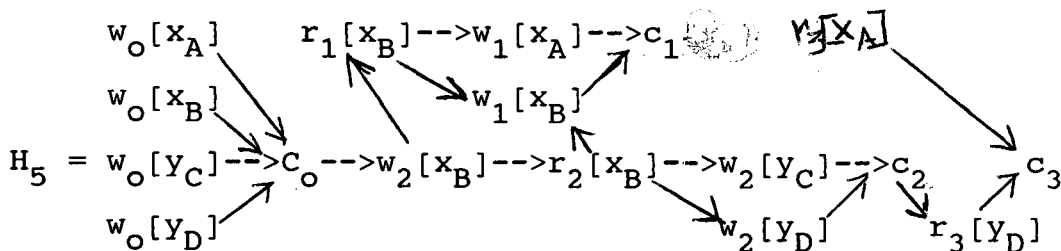


Figure 1.11

Two histories are equivalent (=) if they have same reads-from's relationship. One copy histories *are described below.*

A RD log is one-copy serializable (1-SR) if it is equivalent to one-copy history.

One copy serializability is our correctness criterion for managing replicated data.

An serializable RD history need not be 1-SR. The following example illustrates this fact. The transactions are:

$$T_0 = w_0[x], T_1 = r_1[x]w_1[y], T_2 = r_2[y]w_2[x]$$

The history of above is:

$$H_5 = w_0[x] w_0[x_B] w_0[y_C] w_0[y_d] r_1[x_A] w_1[y_C] r_2[y_D] w_2[x_B]$$

In any serial one copy-log over $\{T_0, T_1, T_2\}$, either T_1 or T_2 must read from the other. But in the H_5 , both T_1 and T_2 read from T_0 . Thus H_5 is not 1-SR.

To ensure that an RD log is 1-SR, the DBS must ensure that each transaction reads from correct transaction i.e. the transaction it would have read from there had been only one copy. This notion is captured by a graph called a logical serialization graph (LSG) [BERN85], defined below.

Given RD history H, Let 'G' be a directed graph whose nodes represent the transactions in H .

G induces write order for \mathcal{H} if all data items x , and transactions T_i and T_k ($i \neq k$) that write x , either $T_i \ll T_k$ or $T_k \ll T_i$. This definition just says that if two transactions write x , one transaction must precede the other.

G induced a read order for \mathcal{H} is for all x : (1) if T_j reads x from T_i , then $T_i \ll T_j$; and (2) if T_j reads x from T_i , T_k writes x (i, j, k distinct), and $T_i \ll T_k$, then $T_j \ll T_k$. This definition says that T_j follows the transaction, T_i , from which it reads x , and precedes all transactions, T_k that subsequently write x .

One possible LSG for H_5 is shown in the figure 1.12. If the LSG is acyclic then it is in 1-SR. H_5 is acyclic so, it is not 1-SR.

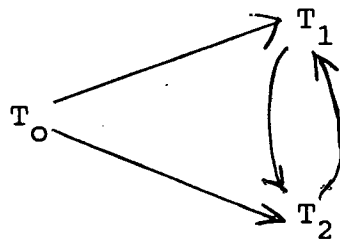


Figure 1.12

Over all the main idea for correctness is one-copy serializability. An execution of transactions in a replicated database is one copy serializable (1-SR), if it is equivalent to a serial execution of the same transactions

in the nonreplicated database. A replicated data algorithm is correct if all of its executions are 1-SR. An execution is 1-SR if and only if it has an acyclic LSG.

1.8 ORGANIZATION OF THE STUDY

Chapter 2 presents the various replica control algorithms and brief discussions on them. Chapter 3 provides the details comparison of various algorithms which are discussed in chapter 2. This comparison is based on the cost in term of number of messages, recovery mechanisms in the case of site failures and partitioning failures, ratio of read requests and write requests and so on. Chapter 4 represents new replica control algorithm. In this transaction visits majority of sites before execution. Chapter 5 concludes the report. At the end, up-to-date bibliography is given.

CHAPTER 2

REPLICA CONTROL ALGORITHMS

2.1 INTRODUCTION

Concurrency control in replicated environment is more complex than centralized one. The basic requirement for any replica control algorithm is that databases should mutually and internally remain consistent. In addition we assume that the algorithm should be robust with site and partition failures. So, the design of replica management algorithm is a notoriously hard problem.

This chapter deals with the description of various replica control algorithms. Here, these are described in briefly, to highlight the central idea behind existing ones. Mathematical description is avoided in order to describe these in simplified form. We have further grouped these algorithms into various categories. The comparison based on various criteria of requirement is presented in chapter 3.

2.2 COMMON ASSUMPTIONS

In almost all algorithms we have certain common assumptions.

A Site failures are clean. When a site fails it simply stops running. Most times when a fault occurs, the site runs incorrectly some time until it detects the fault. By assuming failures are clean, we are assuming that

faults are detected before serious damage is done. We assume that site failures are detectable. While a site is down the other sites can detect this fact.

- B** Communication system is reliable.
- C** All sites are treated equal. i.e., any site can generate update requests.
- D** The database is assumed, it consists of a collection of named elements. It may be records.
- E** Routine communication errors, lost duplicates, and garbled messages are handled by the network.

2.3 CONCURRENCY CONTROL AND CONSISTENCY OF MULTIPLE COPIES OF DATA IN DISTRIBUTED 'INGRES' [STON 79]

Introduction

The algorithm presented here is based on primary site model. In this each object possesses a known primary site to which all updates in the network for that object are first directed. Different objects may have different number of primary sites.

The environment of DBS, in which this algorithm has been designed is for 'INGRES' system. For distributed environment it can be assumed that for each data item, one replica is considered as a primary copy. Different primary

copies may be stored at various sites. The sites which do not contain any primary copy can fail, i.e. in case of failing of such sites the system will not be affected. In this algorithm fragmentation is considered. In case of partitioning of network, primary copies are divided among two partitions. The system response is poor. Deadlock occurs. It needs special treatment.

In this, recovery entails two tasks: handling the failures of a single node in the network and dealing with the failures that partitions the network.

Assumptions

- A** Atleast 95 percent of the traffic to be processed by a distributed data base system is local.
- B** Closeness : Non local interaction are not expected to be scaled on closeness and notion of near by site is not assumed to be useful.
- C** The algorithm should work well for both broad cast and point to point network.

Algorithm and Environment

In this environment an object is a subset of the rows of a relation. A relation is partitioned in to fragments, each with a primary site and some number of redundant copies.

In this each transaction originates from user process at some site 'i' in the network. A 'MASTER' or coordinating collection of INGRES processes at other site ensures that each slave knows identity of all other slaves. SLAVE INGRES exists at all sites where processing takes place. A local concurrency controller (CC) runs at each site. Then CC sees a transaction by saying 'done' or no response. Deadlock detection and resolution can also be distributed. But, this task is allocated to one machine called the SNOOP.

Failure handling

In this Algorithm MASTER, SLAVE and COPY are executed. The other three algorithms are run in the context of failure. Algorithm LOCAL RECOVERY performs local clean up and is run when a site wishes to resume service. Algorithm RECONFIGURE is used to adjust the uplist after a failure or a service restoration. Algorithm SLAVE PROMOTE is run when a MASTER crashes.

2.4 A MAJORITY CONSENSUS APPROACH TO CONCURRENCY CONTROL FOR MULTY COPY DATABASES. [THOM79]

Introduction

In this algorithm database sites vote on the acceptability of update requests. The important property of majority consensus is that the intersection of any two majorities has at least one DBMP in common. For a request

to be accepted and applied to all database copies only a majority need approve it. For any two requests that are accepted at least one DBMP voted OK for both [This situation will not occur]. Query and update access to the database are initiated by application process (AP).

Assumption

- A** It employs time stamping mechanism, used both in the voting procedure and in the application of accepted updates to the database copies.
- B** It is deadlock free and preserves both internal consistency and mutual consistency of the database.
- C** It can recover from and function effectively in the presence of communication system and database site failures. It does not require special recovery mode operation.
- D** The data item copy at each site is accessible only through database managing process (DBMP).

Algorithm

- 1** At first AP locates the data items which are used in its update computation. These variables are called Base variables. The DBMP also supplies the time stamps of base variables to the AP.

- 2 The AP computes new values for the data elements to be updated. The set of elements to be updated are called the Update variables. This algorithm requires that the update variables be a subset of the base variables. Then AP constructs an update request composed of the update variables with their new values and base variables with their time stamps and submits it to a DBMP.
- 3 The update request is transmitted to all DBMPs by broadcasting or daisy chaining type of communication. Request are said to be in conflict if the intersection of the base variables of one request and update variables of other requests are not empty. The voting process is as stated below.
 - 3A Compare the time stamps for the request base variables with the corresponding time stamps in the local database copy.
 - 3B Vote 'REJ' (REJECTED) if any of the base variable is obsolete.
 - 3C Vote 'OK' and mark the request as pending if each base variables is current and request does not conflict with any pending request.
 - 3D Vote 'PASS' if each base variable is current but the request conflicts with a pending request does not

conflict with a pending request of higher priority.

3F Otherwise defer voting and remember the request for later reconsideration.

4 After voting on request 'R':

4A If vote was OK and majority consensus exists accept 'R' and notify all DBMPs that 'R' has accepted.

4B If vote was 'REJ' or 'PASS' then majority consensus is no longer possible, then reject R.

4C Otherwise forward R and votes accumulated so far to a DBMP, that has not voted on it.

5 If R has been accepted then it updates its local copy then notify all DBMPs that R has accepted and reject conflicting requests that were deferred because of R. If R is rejected then use the voting procedure again.

Discussions

The basic characteristic of this algorithm is every update request collects the majority in case it limits its flexibility [GIFF79]. It is resilient to number of failures. It is sufficient that one request message succeeds in acquiring a majority vote set. At a moment when two nodes communicate over a link all other nodes and links may be down. It supports partitioning. Deadlock will not occur. But with this algorithm the internal consistency of database may be disturbed. This can be removed by

improvements [DROS88]. In case of conflicts the number of rejections are more. It employs time stamp mechanism for updates and consistency purpose which needs more cost and storage. This algorithm is proved correct using system-wide invariants [DROS88]. This is a first voting based algorithm in replication.

2.5 THE QUORUM CONSENSUS ALGORITHM [GIFF79]

Introduction

The first voting approach was the majority consensus algorithm. Quorum consensus is the generalization of majority consensus algorithm.

In this approach every copy of replicated item is assigned some number of votes. Every transaction must collect a read quorum of 'r' votes to read an item and write quorum of 'w' votes to write an item. Quorum must satisfy two constraints.

- 1 For each read quorum R and write quorum W, R intersection W should not be null i.e. there is at least one copy common and $(R+W)$ exceeds the total number of votes assigned to that item.
- 2 For each pair of write quorums there is at least one copy common or in other words the total number of votes for each write quorum must exceed half of the votes.

The first constraint ensures that there is a non null intersection between every read quorum and every write quorum. Any read quorum is therefore guaranteed to have a current copy of that item. Each copy has a version number, initially zero. When DBS processes write (x) on quorum 'w', it calculates VN. The maximum version number over all 'Xa' belongs to 'W' and updates each version number to (1+VN). When DBS processes read (x) on quorum R, each access returns its copies version number, and the DBS reads the copy with largest version number. In QC, the TM is responsible for translating reads and writes on data items into reads and writes in to copies.

Discussions

In this algorithm recovery of copies requires no special treatment. A copy of x that was down and therefore missed some writes will not have the largest version number. Therefore, transactions will automatically ignore its value until it has been brought up-to-date.

This algorithm guarantees serial consistency for update requests [Giff79]. This doesn't insist the majority of copies to be updated. This improves the flexibility by weighted voting. It supports site and partition failure. The major drawback of this algorithm is that it pays the same cost to reads and writes. QC needs more number of copies to tolerate a given number of site failure.

2.6 AN ALGORITHM FOR CONCURRENCY CONTROL AND RECOVERY IN REPLICATED DISTRIBUTED DATABASES [BERN84]

Introduction

Available copies algorithm handles replicated data by using simple technique called 'write-all-approach'.

In ideal world, where sites never fail, there is a simple way to manage replicated data. When user wishes to read 'x' the system reads any copy of 'x' and when user updates 'x' the system applies the updates to all copies of 'x'. Concurrency control is done by distributed two phase locking. This algorithm is nothing but an extension of this simple algorithm to an environment where sites fail and recover.

Assumptions

- A The network never becomes partitioned. If two sites are up they can always communicate.
- B Every site runs centralized recovery algorithm.
- C Distributed DBS runs a distributed atomic commit algorithm, such as two phase commit.
- D Site must fail infrequently.

Environment

This algorithm uses directories to define the set of sites that are currently stores the copies of an item. For

each data item 'x', there is a directory $D(x)$ listing the set of x's copies. Like a data item a directory may be replicated, it may be implemented as a set of directory copies and stored at different sites. The directory for 'x' at site U, denoted $D_u(x)$, contains a list of copies for x that site U believes are available. After a copy has been initialized and before it has failed, it is ~~set~~ to be available, otherwise it is ~~set~~ to be unavailable. Usually a site will store both directory and data item copies. Concurrent access to directory copies is controlled by same scheduler that controls concurrent access to data item copies.

This algorithm runs special transactions called status transactions, which makes copies available and unavailable. These are :

INCLUDE(X_a) --> makes X_a available.

EXCLUDE(X_a) --> makes X_a unavailable.

DIRECTORY-INCLUDE(D_t) --> makes D_t available.

The DBS involves EXCLUDE transactions when a site fails, and INCLUDE and DIRECTORY-INCLUDE transactions when site recovers.

Algorithm

- 1 To read, it can consult directory (D_t) of that site, then reads by locking it.

- 2 To write, it set lock on 'D_t' and test D_t. data items whether that data item still available and so, then lock it and write it. If it becomes unavailable then ignore it.
- 3 Consider the case if sites fail and some are recovering during execution. In this every transaction as its locking point. It will not reach its locking point until it gets all locks.
- 3A When failure occurs during the execution then the transaction will not reach its locking point until it gets exclude lock. Then, it is aborted.
- 3B When some site recovers it will not reach its locking point until it gets include lock. Then transaction. commits.

In both cases transaction either commits or aborts. So, database is consistent.

Discussions

In this algorithm locking is used for replication control. It pays more cost for writes, but for reads it requires no message. Deadlock may occur which require special algorithms to be run. In case of site failures, running status transactions increases complexity. When site failures occur frequently, this algorithm is not preferred. Overall, this algorithm is not resilient to more number of

system crashes, partitioning, which is not considered as a robust and flexible.

2.7 MISSING WRITES ALGORITHM [EAGE83]

Introduction

In missing writes (MW) algorithm, during reliable period, the DBS processes read(x) by reading any copy of 'x' and write(x) by writing all copies of 'x'. When a failure occurs the DBS resorts to quorum consensus (QC). After the failure is repaired, returns to available copies algorithm. Thus it only pays the cost of QC during periods in which there is a site or communication failures.

Algorithm Description

Each transaction executes in one of the two modes: normal mode, in which it reads any copy and write all copies or failure mode, in which it uses QC. A transaction must use failure mode if it is aware of 'missing writes'. Otherwise it can use normal mode.

A transaction is aware of missing writes (MW) if it knows that a copy 'X_a' does not contain updates, that have been applied to other copies of 'X'. For example, if transaction sends a write to 'X_a' but receives no acknowledgement, then it becomes aware of MWs.

To implement this algorithm we need a mechanism where by a transaction is aware of MWs. If a transaction 'T_i' becomes out an acknowledgement to one of its writes then its immediately becomes aware of MW. If a transaction 'T_j' comes after T_i, it must be aware of MWs of 'T_i'. Otherwise 'T_j' will read (write) missing copies which will not ensure serializability. To do this 'T_i' should attach a list 'L' of the MWs, it is aware of which copy 'Y_b' it accesses. It tags 'L', to indicate whether it read or write Y_b. When T_j accesses Y_b then it conflicts L's tag then it becomes aware of those MWs.

Data Manager (DM) should acknowledge T_i's access to 'Y_b' by returning a copy of L. 'T_i can now propagate 'L' along with other such list received to all the copies that it accesses. The way a transaction 'T_j' propagates MWs that it's aware of all transactions that follows 'T_j' in the serialization graph (SG).

After recovery from failure, the DBS at site 'a' has two jobs to do: first it must bring each newly recovered copy 'X_a' up-to-date. This is easy to do with a copier transaction. The copier simply reads a quorum of copies of 'x' and writes in to all of those copies the most up do date value that it read. Version numbers can be used to determine this value.

Second, after a copy 'X_a' has been brought up-to-date,

the DBS should delete 'X_a' from the list of MWs on all copies so that, transactions which will come after update should not incur the overhead of QC. This entails sending a message to all sites, invalidating entries for X_a on their list of MWs.

Discussions

In missing writes algorithm the performance depends upon the frequency of switching between normal mode to failure mode. The supporting fact over available copies algorithm is that, it supports partitioning by paying cost over running special transactions.

Overall missing writes algorithm will reduce the cost of reads, if the communication failures are infrequent.

2.8 RESILIENT EXTENDED TRUE-COPY TOKEN SCHEME FOR A DISTRIBUTED DATABASE SYSTEM. [MIN082]

Introduction

In the true copy taken scheme true copy tokens are used to establish logical data. Among multiple physical copies, true copy tokens designate physical data copies that can be identified with the current logical data. Such physical data copies are called true copies. The concept of logical data is crucial in the new resiliency scheme, since resilient system operation can be realized if the continuity of

the logical data is preserved in the case of subsystem failures.

Environment

Transactions and operations are same as those described in chapter 1.

In this scheme, version numbers, assigned to the contents of logical components plays a key role. Initially each logical component contains versions zero, and each time a logical component value is updated, its version number is incremented by one, and this version number is assigned to each of the updates applied to the replicated physical components associated with the logical component. The read-set versions of a transaction are the versions read by the transaction and write-set versions of transaction are the versions created by the transaction.

The main feature in this algorithm is transaction buffer that supports the abortion of partially executed transaction without causing any ill effect to the systems.

When a transaction is allowed to access pending updates (updates created by transaction are pending until transaction issue a commit command) every transaction that has accessed the pending update must also be aborted, if the transaction that created the pending update is aborted. A

transaction buffer is provided for each transaction. Once a commit command is received, the updates in the transaction buffer are written into the database. The consistency constraints for updating database are same as described in Chapter 1.

Failure modes

The types of failures handled in this scheme are site crashes and message link failures. In both cases it is assumed that sites and message links simply cease to function, when they fail.

Algorithm

At first-true copy token scheme has been reviewed.

This scheme first establishes true copies that can be identified with the logical components, and performs locking over these true copies. When update request comes it must get locks over all true copies it needs.

A physical component contains either an exclusive copy, a shared copy or a void copy. Exclusive copies and shared copies are called true copies, and their data values are identical to the current data value of associated logical components. The content of void copy may be obsolete. Read-write accesses are allowed on exclusive copies and read only accesses are allowed on shared copies, but void copies

are not accessed for normal transaction processing. We assume that a true copy possesses a true copy token. Two types of locks namely share locks and exclusive locks are used over the true copies to realize consistent transaction processing. The locking must be two phase [ESWA76].

Resilient system operation

To describe this scheme we must clear on atomic update set (AUS), merge of atomic update set.

An AUS is a set of physical components that covers the complete set of logical components in the system, and it is always updated atomically. Multiple AUSs are provided so that at least one of them can survive under anticipated failures. In this algorithm we will consider only a fully replicated system where each site constitutes an atomic update. In general an AUS may span multiple sites.

An AUS is characterized as follows.

- A** An AUS is a set of physical components such that every logical component represented by at least one physical component in that AUS.
- B** If any physical component in an AUS is affected by the updates of a transaction, then the updates will be completely performed to the physical components in the AUS as long as the AUS remains alive.

C Updates of a transaction are committed to an AUS only if the read-set versions and the preceding write-set versions of the transactions are already committed to the AUS.

Merging of atomic update sets and Recovery set

The merge of AUSs is defined as the collection of the newest versions, relative to each logical component, found in those AUS. In principle, whenever a partitions are merged, all physical components in the new partition must be reinitialized by using the merge of AUSs in the new partition.

A recovery set is defined as the merge of two atomic update sets in the systems. It is stronger than AUS, i.e., it is defined even some of the updates are lost.

True copy generation

When all true copies for a logical components are lost, the logical components can not be accessed and no new versions can be created for it. We do not worry about those versions that are lost by the system failures because the transactions that created these are automatically aborted.

Once it is certain that all true copy tokens for some logical components are lost and we know its newest version surviving in the system, we can generate a true copy for it

by designating one of the physical copies of the newest version as the exclusive copy of the component.

Scheme

Once initiated, read operations must be applied on the shared copies. When the processing of transaction is completed, the exclusive copies of logical components, that the transaction wants to update must be exclusively locked. At this point shared locks held by the transaction can be released. Then exclusive copies are ready to be updated, the remote updates can be send to other sites. After updating those, the locks can be released. The set of updates created by the transaction must applied to each AUS only, if all of the read-set versions and preceding write-set versions of the transaction have been applied to AUS. Partitions can be merged by merge defined by AUSs. A failed AUS can be restored by using merge of AUSs.

Discussions

Functionally, the resilient extended true-copy token scheme can handle some problems in DDBS that could not handled by true copy token scheme.

This scheme does not employ log sub-systems and hence can support a total site crash. This feature is important for a system that includes a small site without a log sub-system that must tolerate total site crash.

By allowing transactions to access only true copies, system partitioning can be supported without any consistency problem. Merging of partitions can be performed by using the merge of the AUs in those partitions.

The new scheme allows us to add a new site to the system. The procedure for adding a new site to the system is logically identical to the site restoration procedure.

2.9 SEMANTICS BASED TRANSACTION MANAGEMENT TECHNIQUES FOR REPLICATED DATA [KUMA88]

Introduction

This algorithm is based on the semantics of transactions. Conventional multicopy algorithms have fast response time and more availability for read only transactions while sacrificing these goals for updates. This algorithm works well in the both retrieval and update environments by exploiting special application semantics by subdividing transactions into various categories, and utilizing commutativity property. For example in case where transactions issue additions and subtractions to the database, updates can be sent in any order, then after some time the database will be consistent.

Assumptions and environment

In this algorithm commutativity property of

transactions is exploited. Generally, replication provides multiple versions of same object at different sites for small duration. No writes should be processed during this time, i.e. in this algorithm data items may not represent same value. However read-only transactions always see a consistent database if they read data items from a single site for a restricted set of sites.

In this we assume full replication. Further it is assumed that a scheduler [BERN81] at each site serializes local transactions using two phase locking or any standard concurrency control mechanism, Here we view a transaction as a function, $F_R(x)$ which transforms an object 'X' to new value as follows.

$$X_{\text{new}} = F_R(X_{\text{old}})$$

where 'R' is a read vector (r-vector) or constants or other database objects (r_1, r_2, \dots, r_n) . In the special case where 'R' is a vector of all constants (c_i) the transaction is represented as $F_c(X)$ and 'c' is called a constant vector. Some examples of function on numeric data objects are:

$$F1_c(X) = c_1x;$$

$$F2_c(X) = X+c_1.$$

In this algorithm the transactions are divided into two categories.

- 1) Commute(C) type,
- 2) Not Self (NC) commute type.

There are transactions in which a vector consists of constants. Such transactions occur frequently in banking applications. For example, withdrawing 40 rupees from a bank account.

Consider the following functions:

$$F1_R(X) = X + r_1,$$

$$F2_R(Y) = Y - r_2.$$

If $r_1 = Y$, $r_2 = X$ then the final result depends on the order the two transactions will execute. On the other hand if r_1, r_2 replaced by constants c_1 and c_2 respectively then $F1$ and $F2$ will always commute. Here, examples are given to understand commute type and not self commute type of transactions.

Consider Simplified banking application.

Deposit: Add c_1 to account 'X'

Withdrawal: Subtract c_2 from account 'X'.

Add interest: Compute 5% of the amount in account 'X' and add it to 'X'.

In above the transactions can be represented as functions $F1, F2, F3$ respectively as follows.

$$F1_C(X) = X + c_1,$$

$$F2_c(X) = X - c_1$$

$$F3_c(X) = X + c_1X.$$

In above F1, F2 are self commute type and F3 is not self commute type.

Algorithm

The algorithm is presented based on the preanalysis of transactions. The preanalysis consists of first identifying all transactions which self commute and grouping them such that all pairs of transactions in a group also commute with each other.

Each site maintains a state vector (S-vector) NC_i, C_i where;

C_i : Number of C type transactions completed at site '1'.

NC_i : Number of NC type transactions completed at site '1'.

C_i, NC_i are counters which are advanced each time a new transaction is performed at a site. C and NC transactions observe different protocols for processing.

A C transaction

1 Performs updates to local copies and commits upon compilation (A scheduler at each site guarantees serializability among local transactions).

- 2 After commit; the corresponding C-vector, the transaction name, and the data item names are send to all remote sites.

An NC type transaction

- 1 Form a quorum of sites by locking a majority of copies of accessed data items.
- 2 Selects the objects at the site with the highest value of NC_i for updating.
- 3 Performs updates to copies at the chosen site.
- 4 Computes S-vector with respects to one c-type transaction, and execute it at the other sites in the quorum.
- 5 Release locks and spools the S-vector and the object name to all sites not in the quorum.

The spooler program runs at each site and performs the following actions.

- 1 It accepts an update message from a transaction and ensures it is transmitted reliably to all other sites.
- 2 It receives messages from other sites and runs them as transaction at the local site.
- 3 Updates the state vector.

Discussions

In this the transactions are divided into C type and NC type. This requires a special preanalysis procedure. The authors [KUMA88] have given provisions to deal with integrity. This technique ensures correctness, though not serializability, and takes advantage of fact that several versions of each object exist in a multicopy environment. Deadlock may happen. The authors do not mention the case of site crash and partitioning problem. Overall this algorithm works better in case where, C-type of transactions are more frequent than NC type.

2.10 CONCLUSION

In this chapter we have discussed the algorithms briefly, without missing an essentials. Examples have been given to illustrate certain algorithms in greater clarity. following this overview of algorithms, we proceed to compare replica control algorithms in Chapter 3.

CHAPTER 3

COMPARISON OF ALGORITHMS

3.1 INTRODUCTION

A number of algorithms are presented in chapter 2. There are differences among these in terms of crash recovery, number of messages transferred, partitioning behavior and so on. In this chapter the various approaches are first grouped into different categories and later after, these are compared in the light of various criteria such as number of messages required to accomplish an update, site failures and partition failures, by pointing differences among them. The algorithms can be divided into groups based on the techniques used for updating the database. The algorithms are classified as follows.

A Token based

- 1 Concurrency control and consistency of multiple copies of data in distributed 'INGERS' (Primary copy algorithm).
- 2 Resilient extended true-copy token scheme for a distributed database system (True copy token scheme).

B Voting based

- 1 A majority consensus approach to concurrency control for multi copy databases (majority consensus).

2 Quorum consensus algorithm.

C Locking based

1 An algorithm for concurrency control and recovery in replicated distributed databases (Available copies).

2 Missing writes algorithm.

D Semantics based:

1 Semantics based transaction management techniques for replicated data (Semantics based).

In this chapter the properties such as failure handling are discussed group wise. However Deadlock detection is not dealt by us. An algorithm for this has been proposed by Badal [BADA86]. To compare the above algorithms the different criteria are given below.

1 **Cost:** Cost in general, is taken to the number of messages required to be transmitted for meeting a single processing requirement. Communication between any two sites can be termed as one message. The cost of accomplishing an update includes computation and communication costs. Here we neglect the computation cost. In most cases these are considered negligible compared to the communication costs.

2 **Site failures:** There are many reasons for the site to

fail. In case of site crashes different algorithms follow different procedures. Recovery procedures are compared in this heading.

- 3 **Partitioning behavior:** As mentioned earlier, sites may get partitioned into groups such that these groups can not communicate among themselves. Some algorithms support network partition heading.
- 4 **Read write ratio:** From the previous chapter it is clear that, in some algorithms reads are processed faster than writes. This property is discussed under this heading.
- 5 **Dead locks, and others:** Some algorithms need to run special recovery procedures to recover. This increases complexity. Some suffer from Deadlocks. These complexities, and among other are discussed in this part.

3.2 COST COMPARISON

For cost comparison certain assumptions are made. When update request comes it requires no messages to initiate update. There are 'N' sites in the system. After execution, the updates can be propagated through any path. To reduce complexity different notations are used depending on the algorithm. This part can be explained by cost evaluation and comparison which is discussed below.

A Cost evaluation

1 Majority consensus

This algorithm is based on the majority voting. Each update request must collect a majority number of votes to accomplish it. In the evaluation the term site is used instead of DSMP.

1A No conflicts with other update requests, no site failures, no rejections

To achieve a consensus: inter site messages : $N/2$.

To notify the home site set of acceptance : $N-1$.

Total number of messages (M_m) = $(N/2) + (N-1) = 3(N/2) - 1$.

So, the minimum number of messages to accomplish update = $3(N/2) - 1$.

1B Conflicts occur, no site failures, no rejections

In case of conflicts votes of more than $(N/2)$ sites may be required to resolve a request. Each additional site requires an additional message. In the worst case it requires $(N-1)$ site messages. Then the maximum number of messages required in worst case conditions are $2N-2$.

If rejections will be more, than number of messages will increase.

2 Quorum consensus

In this algorithm each update collects read quorum to read, and write quorum to write. For simplicity, assume that read write quorums are same for update request.

If the number of votes in the quorum = 'V', then the number of sites communicated by each update request varies depends on the site, where the update request originated. So, the number of messages required to get quorum varies from request to request. So we take

$$R = QV$$

Where

R --> Number of messages needed to get quorum (the number of sites).

Q --> A factor such that : $0 < Q < 1$.

V --> Number of votes.

2A No conflicts, No rejections, No failures

To achieve consensus. : R-1

To notify the DBMP the set of acceptance. : N-1

Total number of messages(Qm) : R+N-2.

2B Conflicts occur, No rejections, No failures

If there are conflicts at some site, then the request proceeds to another sites to get quorum. Consider the case where, there is a only one conflict.

Then

Number of messages required = Q_m+1 .

Similarly in the worst case condition the number of messages:

$$= Q_m+(N-R)$$

$$= R+N-2+N-R \text{ (Substituting } M = R+N-2\text{)}.$$

$$Q_w = 2N-2.$$

2C Conflicts, rejections, and failures, occur

If a request is rejected then it has to resubmit again.
The minimum number of messages required, if it submits one
time = Q_w

Because, it may be rejected by home site.

The maximum number of messages needed, if it resubmits one
time = $Q_w + Q_w$

If the request is resubmitted 'K' number of times then

$$\text{Minimum number of messages} = (K+1)(2N-2)$$

$$\text{Maximum number of messages} = (2N-2).$$

Here also, when the update request is rejected, then it is
recomputed again.

So, the computation cost increases by increasing number of
rejections.

3 Available copies algorithm

This algorithm uses locking principle. To read, a transaction can read at any site. To write it must update all replicas.

3A When sites never fails

Messages to lock the copies of all sites : N-1

Messages to inform the update and unlock
the database copies : N-1

Total number of messages = $(2N-2)$

So, for any update it requires $(2N-2)$ messages to accomplish update.

3B Crashes occur

In case of crashes, then, the DBS runs status transactions. These transactions update and remove each variable depending upon the type. When crash occurs then it must be detected (Assumption). Then, the EXCLUDE transaction excludes all failed copies from each site.

The number of messages required to exclude all available copies : $(N-1)$

To recover from failure INCLUDE transaction includes all copies into sites directories.

The number of messages required to recover all copies: $(N-1)$

Total number of messages required for one crash and recovery : $2(N-1)$

When crash occurs than the request is aborted. It has to resubmit again.

Another $(2N+1)$ messages required to accomplish update. So number of messages depends upon the number of crashes.

If there are 'K' crashes then, to recover, this algorithm requires

$$= 2K(N-1) + K(2N+1) \text{ number of messages.}$$

The first term is for recovery and second is for resubmission.

3C Read-Write Ratio

The number of messages depends upon number of read requests and write requests. In this algorithm reads require no messages. It can simply lock the nearest data item (if it is available) then it reads.

If read-write ratio is 'R' then, the number of messages
 $= R(2N) = 2NR$

So the number of messages depends up on the variable 'R'. If 'R' = 0.1, then write requests = 10; read requests =

90. In total, the number of messages are directly proportional to coefficient of read-write (R).

3D Deadlock

In this algorithm deadlock may occur. So, each deadlock requires backup of one update. It has to be resubmitted again.

If it is resubmitted one time the number of messages (total number of messages for that update) = $(2N+1) + (2N+1)$

To resolve the deadlock, number of messages needed depends upon the type of deadlock resolution algorithm [BADA86].

4 Primary copy algorithm

This algorithm is based on the primary copy. In cost evaluation we generalize this for distributed system. The number of messages in this algorithm depends on the network structure. If two sites want to communicate, they may require $(N-1)$ number of messages.

Now we introduce one variable called network variable which depends on the structure of the network and location of two sites that wants to communicate.

In the worst case the maximum number of messages required to get lock: = $(N-1)$

The number of messages required to get lock from particular site: $= V(N-1)$. ($0 \leq V \leq 1$)

We can say the variable 'V' takes the values between zero to one. When locked data item has found at home site, then $V = 0$. In the worst case, $V = 1$. After getting lock it sends its updates to all sites. For this, this requires $(N-1)$ number of messages.

So, the total number of messages required to get lock on single data item and sending updates $= (N-1) + V(N-1)$
 $= (N-1)(V+1)$

The number of messages required for locking, when a request which contains more than one variable (the 'V' varies from variable to variable).

$$\begin{aligned} &= V_1(N-1) + V_2(N-2) + \dots + V_n(N-1) \\ &= V_t(N-1) \quad (\text{Where } V_t = V_1 + V_2 + \dots + V_n) \end{aligned}$$

including messages for updating $= (V_t+1)(N-1)$

5 Missing writes algorithm

This is a combination of quorum consensus and available copies algorithms. For both, cost has already been evaluated. We can describe this algorithm by variable 'U' in which the site failures occur.

The number of messages required: $= (1-U)A + U Q$

A --> Number of messages required in available copies algorithm

Q --> Number of messages required in quorum consensus

U --> Variable that failures occur.

Suppose $U = 0.01$, means failures occur one in hundred.

6 Resilient extended true copy token scheme

In evaluating number of messages the true copy token, and resilient methods take same number of messages.

In this true copies owns tokens. For simplicity, assume that there are 'T' number of tokens in the system. For each update request

To get locks on true copies : T

To send updates all sites : N-1

Total number of messages = $T+(N-1)$

One can not, however guarantee that, since each update request gets locks on true copies in 'T' number of messages. Because, the tokens are spanned over entire system. So, we have to introduce a coefficient(S) which depends on the distribution of the tokens. The maximum number of messages (then $S = 1$) required to get lock for each update request is N-1.

In total, we can say that the number of messages required to get lock over all varies from 'T' to (N-1)

For read request these can lock any copy which contain token, then reads. So, the number of messages required for reads are little more flexible.

Semantics based

It is based on the semantics of the transactions. The number of messages depends on the ratio of commute and not self commute type. If the transaction is C type it requires no messages. If it is NC type it has to get lock of majority of copies.

For each update request:

Messages to get locks over majority of copies : $(N/2)-1$

Messages for sending updates to all sites : $N-1$

So, for each update request if it is NC type; the number of messages = $3N(N/2)-2$

For the commute types no messages required. The total cost of number messages required = $3WN(N/2)-2$

where W is the ratio of commute type to not commute type

$$(W = NC_n / C_n)$$

B Cost comparison

For cost comparison, consider group by group. In token based algorithm the number of messages depends on the location of site and network structure (depends on whether the network is fully or partially connected). In, primary

copy method the number of messages depends on the location of primary copy. If we know a particular site gets number of update requests, then we can locate more of primary copies on that site, resulting in overall reduction of number of messages, so the cost depends on the design considerations. But, in the resilient scheme, because of increasing resiliency its response is slightly delayed, i.e. it has to update the AUsS consistently. Dead lock detection considered as another overhead. Compared to other algorithms, this requires less number of messages.

In the majority consensus, the number of messages required for transactions depends on the number of sites. In the case of conflicts, rejections will be more. So, the transactions has to be resubmitted again which will increase the computation cost. Quorum consensus will reduce the number of messages by weighted voting scheme. Here the number of messages depend on the number of votes the update request has to collect. Some sites may have more number of votes (weight is high). In this case, the number of messages depends on the home site. Quorum consensus takes less number of messages, as compared to majority consensus. It pays equal price to reads and writes, which is not the case with a locking and token based algorithms.

In locking based, the cost of algorithm strictly depends on the number of write requests. For write requests

it requires $(N-1)$ messages to lock all copies, which is not in the case of voting based algorithms. If the read requests are more than write requests this can be considered for implementation. Same is the case with missing writes algorithm.

In the semantics based, locking is used. It requires pre-analysis of transactions which increases complexity of running extra algorithms to do this. This is not designed for complex updates. In the case of this algorithm we can say that, if commute type of transactions are more in number, it requires less number of messages.

3.3 SITE FAILURES

Here, we consider the failures are detectable. Failures can be detected by time out, missing acknowledgments and some other techniques. This detection part is supported by communication network.

In the case of token based algorithms, the ability to tolerate failures depends on various factors. In primary copy method, site which contains the primary copy is lost, there is no way to recover from such type of failures. So, this method is vulnerable to failures of site which contains primary copy. In extended true copy token scheme, tokens are assigned for more number of copies for each data item. If the token is lost, then there is no way to assign a token. Sites which are not having tokens can fail and recover. In

resilient scheme this drawback is removed by main maintaining AUs, which are updated consistently. So this scheme is vulnerable to more number of site failures.

In majority consensus the recovery needs no special provision. The recovery of sites can process parallelly with updates. When site failure is detected the update request automatically ignores that site. Time stamps are used for this purpose. Same is the case with quorum consensus algorithm. A copy of 'x' that was down and therefore missed some writes will not have the largest version number. Therefore the transactions will automatically ignores that data items (sites). Quorum consensus needs large number of copies to tolerate a given number of failures. The quorum consensus needs three copies to tolerate one failure, five copies to tolerate two failures and so forth, in particular case two copies are no help at all. With the two copies this can not even tolerate one failure [BERN87].

In available copies algorithm, the site failures require execution of status transactions. [BERN84]. In this sites must fail infrequently. It requires $2(N-2)$ number of messages for recovery of single site. This is lot of work. When sites fail too often, quorum consensus, missing writes algorithm are better options [BERN84]. But, when sites fail frequently the switching will be more, which will increase extra burden of changing modes.

In semantics based one, if the site fails, the author [KUMA88] has given no mechanism for recovery. We can say that it is vulnerable to failure of majority of sites. Because, it uses majority principle for locking.

3.4 PARTITIONING BEHAVIOUR

At first when a partition occurs it must be detected. Managing a partition is a notoriously hard problem. Typically the cause of a partition failures cannot be known by sites themselves. At best, a site may be able to identify the other sites in its partition. But for the sites outside of its partition it will not be able to distinguish between the case in which those sites are simply isolated from it and the case in which those sites are down. In addition low response from other sites can cause the network to appear partitioned.

In token based algorithm, which is a resilient technique for items sharing distributed resources, the primary activity. In case of partition failure, only the partition containing the primary copy can access the data item. All updates are simply forward after recovery for execution. This approach works well only if site failures are distinguishable from network failures. If primary site for data item fails then new primary can be elected [EAGA86]. For discussion of election process see [GARG82]. The problem arises in case of two copies of an object. In

this situation any network partition can make both copies inaccessible. In the token based, each item has a token associated with it, permitting the update to access the item. In the event of network partition, only the group containing the token will be able to access the item. The major weakness in this scheme is that the accessibility is lost, if the token is lost as a result of site or communication media failure. Resilient scheme removes this drawback. By providing AUsSs, it can recover in case of partition. If the token is lost it can recover that token by identifying recent data item from AUsSs.

In voting based algorithm, the quorum constraints ensure that an item can not be read in one partition and return in another. Hence the read write conflicts cannot occur between partitions. Another constraint (number of votes to be collected are greater than half of total votes) ensures that writes cannot happen in parallel, or if the system is partitioned the writes cannot occur in two different partitions at the same time. So, the data base is consistent in the case of partition.

In locking based, the available copies algorithm does not support partition. It assumes that partition occurs rarely. In missing writes, at the time of failure it follows quorum consensus. But when it changes mode every site must maintain several files about the information of

missing updates and their values and so on. These files can grow faster. In all missing writes algorithm tolerates partitioning.

About semantics based one, the behavior in case of partition requires special analytical study in which the author [KUMA88] has not mentioned. It is possible to make this algorithm (if not) consistent with slight modifications, because it uses majority principle.

3.5 READ WRITE RATIO

Basically replication introduces the major difference between read requests and write requests in execution part. Here some algorithms treats these as equal in terms of number of messages. Ensuring availability and robustness in case of failures by the replica control algorithm constitutes an optimization problem. Some designers put stress on availability by reducing robustness, and others vice versa.

For both reads and writes in primary copy method, takes same number of messages. But, in the token based approach, number of tokens for each data item are greater in number, which increases accessibility for reads by reducing number of messages. Similarly is the case with the resilient scheme.

In voting based approach the read and write requests are considered to be equal, with respect to number of messages. A request has to collect quorums. The graph in which the ratio of reads and writes versus total number of messages shown in figure 4.1.

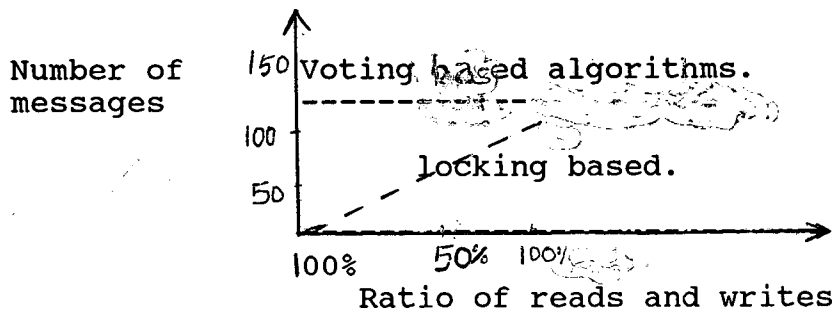


Figure 4.1

In locking based approach the reads require no messages, but for write it has to get locks over all copies. So read-write ratio effects number of messages. The graph is shown in figure 4.1.

In semantics based approach the flexibility is not there in terms of reads and writes. The number of messages depends on the ration of commute and not commute type of transactions. The graph is shown in figure 4.2.

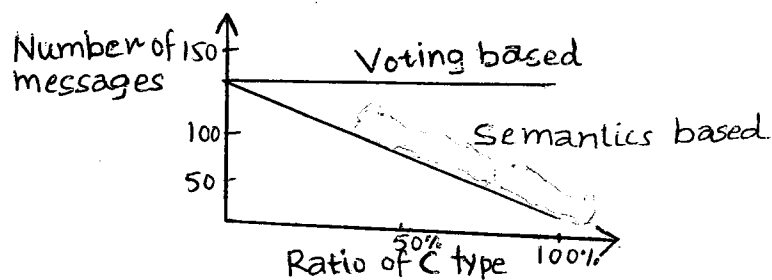


Figure 4.2

3.6 DEADLOCK DETECTION AND OTHERS

In this section we discuss the properties of algorithms with respect to dead locks, inclusions of other site among others.

In resilient scheme, as compared to other token based algorithms, the concept of atomic update sets had been introduced. This AUsS had to updated atomically, which requires extra burden and may delay the response. Deadlocks algorithms, which will delay the responses of update requests. Addition of new site poses no problem, which follows the recovery procedure.

In majority consensus, if there are conflicts, the number of rejections increases. Time stamps maintenance also requires extra storage and communication costs. Addition of new site is not a problem. But, in quorum consensus all copies of each data item must be known in advance. A known copy of 'x' can recover, but a new copy of 'x' cannot be created, because it could alter the definition of x's quorums, in principle, one can change the weights of sites, while DBS is running. But, this requires special synchronization technique. It uses version numbers which need extra storage requirement.

In available copies, recovery requires running of status transactions, which will increase the complexity.

Dead locks may occur frequently.

Finally, in semantics based approach, deadlock may happen, and it requires pre-analysis of transactions.

3.7 CONCLUSION

In this chapter, first we divided the various approaches into different categories to facilitate the comparison. Then, we compared these categories of algorithms.

Token based algorithms reduces the number of messages, but increases the vulnerability in case of failures. it is a balanced scheme between number of messages and complexity. Voting based approaches, are robust as compared to the others. These support the partitioning of system. But, in this case cost of reads, writes is same. In the case of more read requests, and less number of site failures however locking based algorithms are preferred.

CHAPTER 4

AN ALGORITHM BASED ON EXECUTING REQUESTS AFTER VISITING MAJORITY OF SITES

4.1 INTRODUCTION

A number of replica control algorithms, described in the literature based on the voting, locking, tokens and semantics of update requests are presented in chapter 2, and a comparison is presented in the chapter 3. The basic characteristic of some of these algorithms is that each update request is executed first (independent of other requests) and, there is a possibility that the update request may be rejected. In case of rejections the request is resubmitted again. In the voting based algorithms also, each update request has no guarantee that it will not be rejected. The reason behind this is that in these algorithms when request arrives, it is executed and the base variables and update variables [THOM79] with time stamps are sent for voting.

The basic philosophy of serializability theory is that, if a set of transactions $\{T_0, T_1, \dots, T_n\}$ try to update the database, the execution sequence is correct, if they execute and updates one after another, and the database remains consistent.

Based on the above algorithms, we have explored an alternating approaches to reduce the overhead without distroying the database consistency. Our approach is based on the notion that, when an update request comes, we can identify the update variables without executing it and send these to other sites for majority approval, instead of sending new update values. In systems like banking and railway reservation systems the most update requests are single or double line (SQL) statements. So, identifying the reads and writes in the update request is not a major problem for the system. In the next section, we describe the update technique which will take less number of messages and rejections, as compared to previous algorithms. About consistency, we guarantee that this algorithm preserves mutual and internal consistency which is described in chapter 2.

4.2 ENVIRONMENT

We assume an environment in which copies of database are accessible at a number of database sites. This algorithm further assumes that the architecture of a distributed database management systems (DDBMS), is same as described[BERN81]. In this, each site is a computer, running one or both of the following software modules: a transaction manager (TM) and a data manager (DM). The TMs supervise interactions between users and the DDBMS while DMs manage

actual database. A network is a computer-to-computer communication system. The network is assumed to perfectly detect failures when they occur. In addition we assume that between any pair of sites the network delivers messages in the order, they were sent.

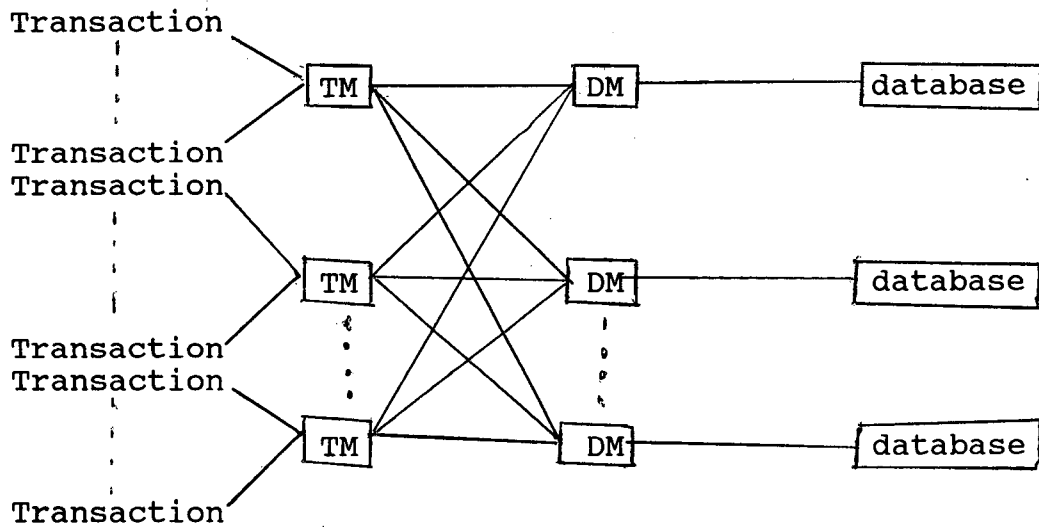


Figure 4.1

Users interact with the DDBMS by executing transactions. Transactions may be on-line queries expressed in a self contained query language, or application programs written in general purpose programming languages. However we do assume that transactions represent complete and correct computations; each transaction, if executed alone on an initially consistent database, would terminate, produce correct results, and leave the database

consistent. In this algorithm the concept of request is used. The structure of DDBMS is shown in figure 4.1.

From users perspective, the database consist of a collection of a logical data items, denoted by x, y, z . We leave granularity of logical data items unspecified; in practice, they may be files, records, etc. A logical database state is an assignment of values to the logical data items composing a database.

In this algorithm we are considering fully replicated database, i.e each data item is stored reduntantly at multiple sites. A transaction is a set of operations. An operation is an activity that manipulates data. There are three types of operations: read operations, write operations and local computations. The operations that can be performed on a replicated data items are read, which returns its value, and write, which changes its value.

Read and write operations either logical or physical are used to access the components. Local computations can transform the data read by read operation and supply the transformed data to write operations. Further more data may be passed in the from of messages between two physical operations that occur at different sites. Message links, connecting sites are used to send messages.

Physical read operation $read [x_i]$, returns the current

content of physical component x_i and physical write operation $write [x_i]$ updates the content of physical component of x_i . We can say a logical read operation $read [x_i]$ corresponds to physical read operation for some 'i'. A logical write operation $write [x]$ corresponds to the set of physical write operations $write [x_1], write [x_2], \dots, \dots$, each of which writes some data value $write [x]$.

Update Requests

A transaction 'T' is modeled as a sequence of read and write operations. Because reads and writes are responsible for changing the state of database. When the update request comes to a particular site, it identifies the reads and writes, needed for the execution of that update request. Then, the site prepares 'request' which has to be sent to other sites for majority approval. The request contains the data items to be locked, request number, and the ~~request~~ ~~data~~. The various terms are explained in further sections.

4.3 ASSUMPTIONS AND TERMINOLOGY

Here we consider a fully replicated database. When ever a transaction is received, it is the job of local concurrency controller (TM) to deal with it. So, we assume that every site contains local concurrency controller. Here, unique time stamps are assigned to each request [transaction]. Every request finds the consistent state of

the database at particular time 't', updates it and leaves the consistent state of the database. Every site follows the same methods to preserve internal consistency. Another assumption is that rejections will not occur. Every conflict is ordered by this algorithm. In this algorithm the words transaction and update requests are used interchangeably with no difference.

The following terms are used to present this algorithm. These are request, request variables, Request wait for graph, Request number (time stamp), locking table majority and request messages.

Request: It contains the request variables, request number, request table and visiting sites. It is prepared by home site of each request (when update request comes), and sent to majority of sites.

Request Variables: when update request comes to particular site, the variables are identified. These are (data items) forming the read set and write set.

Request Wait For Graph [RWFG]: When a request visits particular site, it puts its Request number into the RWFG of that site (according to rules) and it copies the request numbers to its own RWFG. This has to be maintained by every site.

Request Number (RN): This is a time stamp, assigned by

the home site assigns.

Allocation of time stamps is described in another section.

Locking Table (LT): Every site has to maintain this table. When the request comes to any site, to visit majority of sites, it puts its request variables in to that table. This is used to identify conflicting operations.

Majority : In [THOM79] 'Majority' word is used. Majority means the number of sites that each request must visit.

If 'N' is a number of sites then

Majority = $(N+1)/2 + 1$ (if N is odd)

= $(N/2)+1$ (if N is even)

Request Message Table (RMT): This table is maintained at every site. After visiting a majority of sites, before coming to home site, the request(R_m) sends messages to all other sites which are ordered before ' R_m ', about R_m 's ordering. These messages are stored in RMT of particular site.

4.4 INTER SITE COMMUNICATION

Update requests made by sites must be communicated among the sites to visit majority of sites. There are two possible communication methodologies (Figure 4.2).

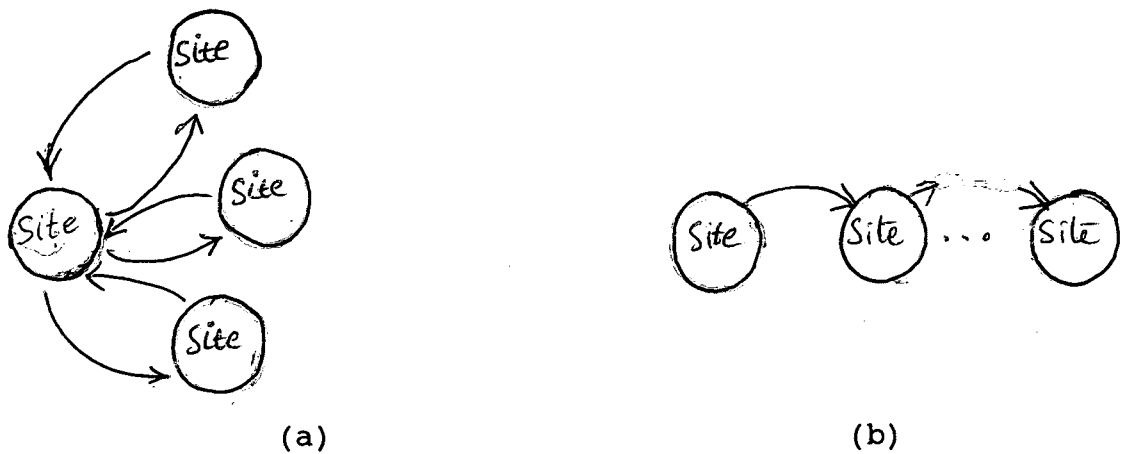


Figure 4.2

1. **Broadcast:** After receiving the update request (Figure 4.2a), then that site transmits that request to visit majority. After visiting sites, the request must return to the original site.
2. **Daisy Chaining:** The site receiving the request (Figure 4.2b), decides the order and the request is passed to next site with incrementing the number of sites visited. After visiting the majority of sites, the request is returned to original site.

Use of a broadcast discipline allows requests to be resolved with minimum delay at the possible expense of extra messages. On the other hand daisy chaining results in resolution with the minimum number of messages at the expense of relatively high delay. In practice, the choice of communication discipline should be based upon the performance requirements for the database system as well as the characteristics of the underlying communication system. In this algorithm the discussion is based on the daisy

chaining communication. For broadcast type of communication this algorithm is slightly modified.

4.5 TIME STAMPS

In this algorithm time stamps are used to resolve conflicts. Conflicting operations are defined in the Chapter 1. The properties that time stamps use for this purpose should have uniqueness and monotonicity (i.e., successively generated time stamps should have a high value)

All time stamps in the distributed system come ultimately from update requests. The question of when one by whom should the request be time stamped, arises.

Here, it is assumed that, each site has access to a local, monotonically increasing clock, but there is no common clock accessible to all sites. A time stamp generated by a site 'S' is a pair (T,S) where T is a time obtained from the local site clock. T is called the C-part (for clock) of the time stamp and S is called the S-part (for site) of the time stamp.

Equality, greater than, and less than for time stamps can be defined as follows.

Let, $T_1 = (C_1, S_1)$ and $T_2 = (C_2, S_2)$.

Equality (=) : $T_1 = T_2$ if and only if $C_1 = C_2$ and $S_1 = S_2$.

Greater than (>) : $T_1 > T_2$ if and only if $C_1 > C_2$ or $C_1 = C_2$

and $S_1 > S_2$.

Less than : $T_1 < T_2$ if and only if $C_1 < C_2$ or $C_1 = C_2$ and $S_1 < S_2$.

4.6 FAILURE ASSUMPTIONS

The components of a distributed DBS can fail in many ways. Here we assume that site failures are clean : when site fails, it simply stops running; when the site recovers, it knows that it failed and initiates a recovery procedure. We do not consider failures of type, in which a site continues to run but performs incorrect actions.

We assume that a site failures are detectable: when the site is down, then the other sites detect that fact.

4.7 ALGORITHM

In this algorithm time stamps will play a role in ordering of transactions. Here, the term ordering is used. When we say, R_i is ordered after R_j ($R_j \rightarrow R_i$), it means that R_i is executed after R_j , irrespective of time stamps.

In each request (R_i), when it completes visiting majority sites, broadcasts the RWFG of this request to all sites which are in the request.

When the update request comes to site 'Si' then that follows the following actions.

- 1 The site prepares the 'request' then sends it to the

other sites. It assigns the time stamp and puts its number into RWFG of that site (it assigns $N=1$). Then, this request is sent to visit the majority of sites. After visiting the majority of sites it must come back to this originating site (home site).

Request (R_i) = (RN, L, RWFG, N)

where,

RN--->Request number

L---->Locking variables.

RWFG---->Request wait for graph (TWFG) (The ordering of request in which R_i is executed after).

N---->Number of visiting sites. This is a counter which is incremented at every site. This is used to test whether a request has visited the majority of the sites or not.

- 2 The request (R_m) comes to site 'A'. Let, R_i ($i=1, \dots, k$) to be requests of the request table A. Set $N=N+1$; $i=i+1$. Got 2A.
- 2A Test whether locking variables of ' R_m ' are conflicting with ' R_i '. If yes, then go to 2B, otherwise go to 2C.
- 2B The request number ' R_m ' is ordered after R_i ($R_i \rightarrow R_m$) in the RWFG if $i \neq k$ then $i=i+1$; go to 2A, else go to 2C.
- 2C $i = i + 1$; go to 2A.
- 2D Store the request number of R_m , locking variables in site 'A' after R_1, \dots, R_k , if $N=M$ (Majority), then R_m 's

RWFG will be sent to every home site of ' R_i ' such that R_i belongs to RWFG.

After visiting the majority of sites, the request (R_m) comes to home site, it will not be executed until it gets RWFG messages, or updates from all requests, which are ordered before ' R_m '. In broadcast type after receiving the RWFG from all sites then the home site (R_i) sends the RWFG to all request (R_j) home sites, if R_j belongd to RWFG, and it will not execute until it receives the RWFGs or updates from all requests which are order before ' R_i '.

When request message comes from particular request (R_i), to R_j then these two are tested whether cycles are formed. Cycle means in the RWFG of R_j , R_i is ordered before, and in the RWFG of R_i , R_j is ordered before. These can be removed, by ordering according to time stamps. when the request receives messages or updates from all R_i 's which are ordered before R_m , then R_m can execute the database.

After receiving the updates of request (R_m), then this request, and its locking variables will be deleted from all sites, and RWFG's.

3 EXAMPLE

The following example will clarify the algorithm.

Example 1: Consider the five site network.

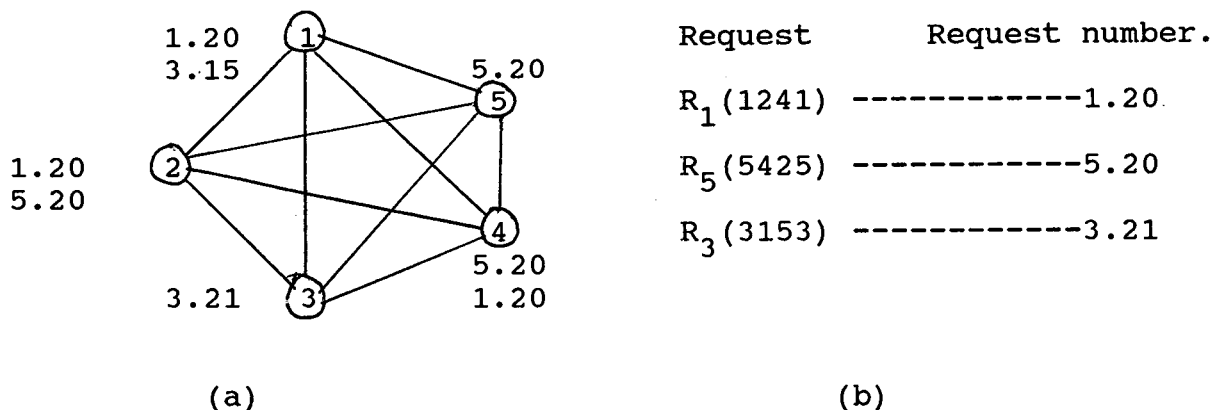


Figure 4.2

and further assume requests, as shown in figure 4.2(b). In this the first part of time stamp shows the site number, and the second part shows the clock (Two digit). Here, number of sites in the majority is three. The path of each request is shown in the brackets (Figure 4.2(b)). Let us assume that the transmission time between two sites is one unit. The following table (Figure 4.3) clears the transmission of requests. All three requests are of conflicting type.

Requests/Time -->	2.20	2.21	2.22	2.23	2.24
1.20	1	2	4	1	
5.20	5	4	2	5	
3.21		3	1	5	1

This can be explained as follows. When request R_1 visits site number 2 it puts the 1.20 at site 1 (RWFG) and visits site 4. But at this site request R_5 has already visited.

So, it puts the request number after R_5 , then it stores the R_5 in its RWFG such that ' R_5---R_1 ', similarly for all others. Now at this point it has got majority, then it sends the message ' $R_5-->R_1$ ' to site 5. When requests gets majority the request tables of R_1, R_3 contain the order shown below.

```

R1-----5.20 ----> 1.20
R5-----1.20 ----> 5.20
R3-----1.20 ----> 5.20 ----> 3.21
                                ↗
                                1.20 (Here, R3 will execute after R1, R5)

```

R_1 will not execute until it receives the message or update from R_5 , and the same is for others. When R_1 receives the message ' $1.20--->5.20$ ' from R_5 , cycle is formed then R_1 orders according to time stamps. At the end the orders are as follows.

```

R1----1.20---->1.20
R5----1.20---->5.20
R3----1.20---->3.21
                    ↗
                    5.20

```

(Here, R_3 receives no messages, so it executes after receiving updates of R_1, R_5)

The above order is consistent. In this R_1 will execute, then after receiving updates of R_1, R_5 will execute, then R_3 .

4.9 SITE CRASHES AND PARTITIONING BEHAVIOUR

In this it is assumed that when site failures occur then those are detected. If site fails then each update request ignores that site without stopping normal processing. When a site recovers a message is communicated to all sites. It can recover data by copying data from other sites.

When partitioning occurs, then the partition which is having majority of sites can process update requests.

4.10 WHY THE ALGORITHM WORKS

The way the requests visit majority of sites, sending of messages to other sites to resolve the cycles, and time stamps are the basis of this algorithm.

The mutual exclusion necessary to make preservation of internal consistency is achieved by each individual site by making request to visit majority of the sites. In particular, concurrency control is achieved by the concept of visiting majority that ensures that the majority sites for any two update requests have at least one site common and this algorithm ensures that conflicting requests are ordered one after another.

In this algorithm, if two up-date requests conflict at a time only one request sees the consistent state of

database. After receiving updates of one request the other will execute, which will ensure the one copy version of database.

A formal proof of the correctness of the algorithm is not described; however, the rest of this section informally argues for its correctness. In particular, to establish correctness of algorithm we claim that:

A This algorithm ensures serializability; i.e. conflicting transaction will execute one by one.

B This algorithm ensures 1-serializability.

A This algorithm ensures 1-serializability

Proof: Here we use the concept of serializability. A definition of serializability and conflicting operations used here is the same as in Chapter 1. Assume that the update requests in $SG(R)$ by this algorithm is (R_1, \dots, R_n) , we have to show, $SG(R)$ is acyclic.

A1 If R_1, R_2 are conflicting update requests. Then $R_1 \rightarrow R_2$ or $R_2 \rightarrow R_1$ but not both.

Proof: Suppose assume that two update requests R_1, R_2 orders the transactions individually as follows.

$$R_1 : R_2 \rightarrow R_1 \qquad R_2 : R_1 \rightarrow R_2$$

But at the site they get the majority R_1 sends the

message ' $R_2 \rightarrow R_1$ ' to R_2 and; R_2 sends the message ' $R_1 \rightarrow R_2$ ' to R_1 .

With respect to time stamps the final ordering will be done. So the final execution will be either $R_1 \rightarrow R_2$ or $R_2 \rightarrow R_1$, not both.

A2 Let (R_1, R_2, \dots, R_n) be the final ordering of this algorithm. Then R_n will execute after all.

Proof : If $R_1 \rightarrow R_2$ is a final ordering then R_2 executes after receiving the updates of R_1 (From A1).

Similarly for $R_2 \rightarrow R_3$. So by transitivity, ' R_n ' will execute after receiving the updates of all (R_1, \dots, R_{n-1}) .

Suppose by contradiction assume that, $SG(R)$ has a cycle over

$$R_1 \rightarrow R_2 \rightarrow R_3 \dots R_n \rightarrow R_1$$

The above is violated the rules of the algorithm, in which R_n, R_1 sends their orders to each other, then cycle is resolved on the order of time stamps. So, either $R_n \rightarrow R_1$ or $R_1 \rightarrow R_n$ will present in the $SG(R)$.

So, $SG(R)$ is acyclic.

Hence, the execution sequences produced by this algorithm is serializable.

B The execution sequences produced by this algorithm is 1-copy serializable.

Proof : In order to prove 1-copy serializability, we must prove that, the execution sequences produced by this algorithm follows the following two rules. (From Chapter 1).

(a) If R_j reads data item 'X' from R_i then $R_i \ll R_j$

(b) If R_j reads 'X' from R_i , R_k writes 'X' (i, j, k distinct), and $R_i \ll R_k$, then $R_j \ll R_k$.

Part (a) can be proved easily, R_j reads data item 'X' from R_i , it means both are conflicting type. In this algorithm each request is visiting majority of sites, in which R_i and R_j are ordered as $R_i \ll R_j$.

Part (b) R_i, R_j and R_k are update requests.

The following orders are true.

$$R_i \ll R_k; R_j \dashrightarrow R_i$$

If R_i reads the value of R_j , according to part (a) it executes after receiving the updates of R_j . Suppose assume that R_k writes 'X' in between R_j and R_i . Then according to algorithm if R_j and R_k are conflicting type they must have ordered one after (i.e., $R_i \dashrightarrow R_k$ or $R_k \dashrightarrow R_i$), and R_k also conflicting type.

$$\text{So, the order will be } R_j \dashrightarrow R_k \dashrightarrow R_i \quad (1)$$

But from (b) R_j reads the value written by R_i . Equation (1) contradicts with this.

So, the executions produced by this algorithm is 1-copy serializable.

4.11 CONCLUSION

In voting type approaches the update request is executed first, then the update variables are sent to other sites for majority approval. In quorum consensus approach the majority is replaced by quorum. In these approaches, if the update requests are conflicting type, then the rejections are more in number.

In this algorithm, rejections will not be there but it is assumed that, when update request comes, we must identify the variables, it reads and writes. This is little more complex in case of large update requests. But, in the case of banking, railway reservation system, these variables can be identified with less effort.

Overall, this algorithm works well in case of conflicts, and needs less number of messages than voting approaches. The main advantage is, rejections will not occur and it is robust to site crashes and network partitioning.

CHAPTER - 5

CONCLUSIONS

The advent of distributed system has added a new aspect to fault tolerance. To increase fault tolerance replicated data is stored redundantly at multiple locations. This report contains the discussion and comparison various approaches to replication. After that new technique is presented.

The first we discussed replication issues, advantages and overhead of replication. Correctness criteria for concurrency control and replication control is presented.

Then, various approaches are described in brief. After these are divided into different categories based on the central idea. The approaches are compared in groups with respect to number of messages required for each update request, site crashes, partitioning behaviour.

At last, we presented an algorithm based on the executing update requests after visiting majority of sites. In this, we use time stamps to order the update requests.

Overall, the replicated distributed system, provides adequate performance in case of failures. Different algorithms can be proposed depending on the type of environment.

In broadcast environment the new technique exhibited less delay, by reducing number of messages in case of conflicts and, it is robust to site failures and network partitioning.

BIBLIOGRAPHY

- [AHAM87] Ahamad, M., Ammar, M. Performance Characterization of Quorum Consensus Algorithms For Replicated Data. In Proceedings of the Symposium on Reliability in Distributed Software and Database System. (1987), pp, 161-168.
- [ALLC83] Allchin, J.E. A Suite of Rubust Algorithms for Maintaining Replicated Data Using Weak Consistency Conditions. In Proceedings of the 3rd Symposium in Distributed Software and Database Systems. 1983, pp,47-56.
- [ALSB76] Alsberg, P.A., Belford, G.G., Day, J.D., Grapa, E. Multi-copy Resiliency Techniques. Technical report CAC Document No.202, Center for Advanced Computation, University Illinois at Urbana-Champaign, May, 1976.
- [BADA86] Badal., D.J. The Distributed Deadlock Detection Algorithm. ACM Transactions on Computer Systems. Nov.1986, pp.,320-327.
- [BARB84] Barbara, D., Garcia-Molina, H. The Vulnerability of Voting Mechanisms. In Proc. 4th Symp. on Reliability in Distributed Software and Database Systems, pages 45-53. IEEE, Silver Spring, MD, 1984.
- [BERN77] Bernstein, P.A., Rothie, J.B., Goodman, N., Papadimitriou, C.A. The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case). IEEE Trans. on Software Engg. Vol.SE-4, No.3, May 1978, pp.154-168.
- [BERN81] Berstein, P.A., Goodman, N. Concurrency Control in Distributed Database Systems. ACM Computing Surveys. Vol.13, No.2, June 1981, pp.185-221.
- [BERN83] Bernsteain, P.A., Goodman, N. Concurrency Control and Recovery for Replicated Distributed Databases. TR-20-83, Center for Research in Computing Technology, Harward Univ., July 1983.
- [BERN83a] Bernstein, P.A., Goodman, N. The Failure and Recovery For Replicated Databases. In Proceedings of the 2nd Annual Symposium on Principles of Distributed Computing, Montreal, August, 1983, pp, 114-121.
- [BERN84] Bernstein, P.A., Goodman, N. An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. ACM Trans. on Database Systems, 9(4), December 1984, pp. 596-615.

- [BERN85] Bernstein, P.A., Goodman, N. Serializability Theory for Replicated Databases. Journal of Computer and System Sciences 31, (1985), pp. 355-374.
- [BERN87] Bernstein, P.A., Goodman, N. Serializability Theory for Replicated Databases. Journal of Computer and System Sciences 31, (1985), pp. 355-374.
- [BERN87] Bhargava, B. Transaction Processing and Consistency Control of Replicated Copies During Failures in Distributed Databases. Journal of Management Information Systems. Vol.4, No.2, Fall 1987, pp.93-112.
- [BHAR88] Bhargava, B., Ng, P., A dynamic Majority Determination Algorithm for Reconfiguration of Network Partitions. Information Sciences. 46, pp.27-45.
- [BHAR90] Bhargava, B., Browne, S. Hybrid Value/Event Representation of Replicated Objects. Department of Computer Sciences. CST-TR-967, Purdue University. March, 1990.
- [BHAR90] Bhargava, B., Lian, S. Typed Token Approach for Database Processing During Network Partitioning. In Conference on Management of Data, (COMAD90) December 1990, New Delhi, India.
- [BLAU83] Blaustein, S.T., Garcia, H., Ries, D.R., Chilenskes, R.M., Kaufman, C.W. Maintaining Replicated Databases Even in the Presence of Network Partitions. In Proceedings of the IEEE 16th Electrical and Aerospace Systems Conference (Washington, D.C., Sept.), IEEE, Newyork, pp.,353-360.
- [BLAU84] Blaustein, B.T., Kaufman, C.W. Updating Replicated Data During Communication Failures. In Proc. 11th International VLDE Conference, Stockholm, August 1985.
- [BLOC87] Block, J.J., Deniels, D.S., Spector, A.Z. A Weighted Voting Algorithm For Replicated Directories. 1987, To Appear in Journal of ACM.
- [BIRM85] Birman, K. Replication and Fault-Tolerance in the Isis System. In Proceedings of the 10th ACM Symposium on Operating System Principles, December, 1985.
- [BREI82] Breitweiser, H., Leszak, M.A. Distributed Transaction Processing Protocol Based on Majority Consensus. In Proceedings 1st ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Aug.1982, pp.224-237.

- [DANI83] Daniels, D., Spector, A.Z. An Algorithm For Replicated Directories. In Proceedings 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, (Aug. 1983), pp.104-114.
- [DAVI83] Davidson, S.B., Garcia-Monila, H., Skeen, D. Consistency in Partitioned Networks. ACM Computing Surveys, Vol.17, No.3, September 1985.
- [DROS88] Drost, N.J., Leeuwen J.V. Assertional Verification of a Majority Consensus Algorithm for Concurrency Control in Multicopy Databases. In Lecture Notes in Computer Science (No.335), 1988.
- [EAGE81] Eager, D.L. Rubust Concurrency Control in Distributed Databases. Technical report CSRG#135, Computer System Research Group, University of Toronto, October, 1981.
- [EAGE83] Eager, D.L., Savcik, K.C. Achieving Robustness in Distributed Database Systems, ACM Trans. Database Syst.8,3 (Sept.1983), 354-381.
- [ELAB85] El Abbadi, A., Skeen, D., Cristian, F. An Efficiency Fault-Tolerent Protocol for Replicated Data Management. In Proc. 4th ACM SIGACT-SIGMOD Symp. on Principles on Database Systems, Portland, Oregon, March 1985, pp.215-228.
- [ELAB86] El Abbadi, A., Toueg, S. Availability in Partioned Replicated Databses. In proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems. Cambridge, MA, March, 1986, pp.240-251.
- [ELAB87] El Abbadi, A., Toueg, S. Maintaining Availability in Partitioned Replicated Databases. Tech. Rep.TR-87-857, Dept. of Computer Science, Cornell University, Ithacea, N.Y., 1987.
- [ELLI77] Ellis, C.A. Consistency and Correctness of Duplicate Database Systems. 6th Symp. Operating System Principles, Nov.1977, pp.67-84.
- [ELLI77a] Ellis, C.A. A Robust Algorithm for Updating Dublicate Databases. In Proc. 2nd Berkeley Workshop Distributed Data management and Computer Networks., 1977, pp.1146-1158.
- [ESWA76] Eswaran, K.P., Gray, J.N. Lorie,R.A., Taiger, T.L. The Notions of Consistency and Predicate Locks in a Database System. Comm. ACM 19(11), November, 1976, pp.624-633.

- [GARC79] Garcia-Molina, H. Performance of Update Algorithms for Replicated Data in a Distributed Database. Tech. Rep. STAN-CS-744, Department of Computer Science, Stanford University, June 1979.
- [GARC82] Garcia-Molina, H. Elections in a Distributed Computing System, IEEE Trans. on Computers C-31(1): January, 1982, 48-59.
- [GARC82a] Garcia-Molina, H. Reliability issues for Fully Replicated Distributed Databases, IEEE Computer, 15, 9 (Sept.1982), pp.34-42.
- [GARC82b] Garcia-Molina, H., Wiederhold, G. Read-Only transactions in Distributed Database. ACM Transactions on Database Systems, Vol.7, No.2, June 1982, pp.209-234.
- [GARC83a] Garcia-Molina, H., Barbara, D. How to Assign Votes in a Distributed System. Technical Report TR 311-3/1983, Department of Electrical Engineering and Computer Science, Princeton University, 1983.
- [GARC86] Garcia-Molina, H. The Future of Data Replication. In 5th Symp. on Reliability in Distributed Software and Database Systems, IEEE, Los Angeles, January, 1986, pp.13-19.
- [GARD79] Gardarin, G., Lebaux, P. Centralized Control Update Algorithm for Fully Redundant Distributed Databases. In Proc. 1st Int'l Conf. on Distributed Computing Systems, IEEE, October, 1979, pp.699-705.
- [GARD80] Gardarin, G., Chu, W.W.A. Distributed Control Algorithm for Reliably and Consistently Updating Replicated Databases. IEEE Trans. on Computers C-29(12):December, 1980, pp.1060-1068.
- [GELE79] Gelencse, E., Savick, K. Analysis of update synchronization for Multicopy Databases. IEEE Trans. Computer. C-28, 10(Oct 1979), pp.737-747.
- [GIFF79] Gifford, D.K. Weighted Voting for Replicated Data. In Proc. 7th ACM SIGOPS Symp. on Operating System Principles, Pacific Grove, CA, December, 1979, pp.150-159.
- [HERL84] Herlihy, M.P. Replication Methods for Abstract Data Types. Ph.D. Thesis, Massachusetts Institute of Technology, May 1984.
- [HERL86] Herlihy, M.A. Quorum-Consensus Replication Method for Abstract Data Types. ACM Trans. on Computer Systems 4(1):February, 1986, pp.32-53.

- [HERL87] Herlihy, M.P. Availability vs. Concurrency : Atomicity Mechanisms For Replicated Data. ACM Trans. Comput. Syst.4,3(Aug.1987), pp.249-274.
- [HERL88] Herlihy, M.P., Weihl, W.E. Hybrid Concurrency Control for Abstract Data Types. In Proceedings of the 7th ACM SIMOD-SIGACT Symposium on Principles of Database Systems (PGDS) (March 1988), pp.201-210.
- [HERL90] Herlihy, M. Apolizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. ACM Trans. on Database Systems Vol.15, No.1, March 1990, pp.96-124.
- [HEVN88] Hevner., A.R., Arune Rao. Distributed Data Allocation Strategies. A Chapter in 'Advances in Computers 1988.
- [JAJ087] Jajodia, S., Meadows, C.A. Mutual Consistency in Decentralized Distributed Systems. In Proceedings of 3rd Int. Conf. on Data Engineering. Los Angles, Februrary, 1987, pp.396-404.
- [JAJ087a] Jajodia, S., Mutchler, D. Dynamic Voting. In Proceedings of ACM Int. Conf. on Management of Data (SIGMOD) Sanfrancisco, May 1987.
- [JAJ090] Jajodia, S., Mutchler, D. Dynamic Voting Algorithms For a Maintaining the Consistency of a Replicated Database. ACM Transactions on Database Systems. Vol.15, No.2, June 1990, pp.230-280.
- [JOHN75] Johnson, P.R., Thomas, R.H. The Maintenance of Duplicate Databases. Tech. Rep. RFC677C31507, Network Working Group, January, 1975.
- [JOSE86] Joseph. T. Low Cost Management of Replicated Data. Department of Computer Science, Cornell University, Ph.D. Dissertation. Jan. 1986.
- [JOSE86] Joseph, T.A., Birman, K.P. Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems. ACM Trans. on Computer Systems 4(1):Februray, 1986, pp.54-70.
- [KANE79] Kaneko, A. Logical Clock Synchronization Method for Duplicated Database Control. In Ist Int'l Conf. Distributed Computing Systems, Huntsville, Ala., Oct. 1979, pp.601-611.
- [KOON86] Koon, T., Ozsu, M.T. Performance Comparison of Resilient Concurrency Control Algorithms for Distributed Databases. In Proc. Int' Conf. on Data Engineering, IEEE, Los Angles, Feb., 1986, pp.565-573,

- [KOHL81] Kohler, W.H. A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. ACM Computing Surveys, Vol.13, No.12, June 1981.
- [KORTH86] Korth., H.F., Silberschatz., A. Database Systems Concepts. Mc Graw-Hills Book Company, 1986.
- [KUMA88] Kumar, A., Stonebraker, M. Semantics Based Transaction Management Techniques for Replicated Data. ACM SIGACT-SIGNOD 1988.
- [LAMP78] Lamport, L. Time, Clocks and Ordering of Events in a Distributed System. Comm. ACM21(7):July 1978, pp.558-565.
- [MINO82] Mincura, T. Wiederhold, G.R. Resilient Extended True Copy Token Schemes for a Distributed Database Systems. IEEE Trans. on Software Engineering, Vol.SE-8, No.3, May 1982.
- [NDE85] Noe, J.D., Proudfool, A., Pu, C. Replication in Distributed Systems: The Eden Experinece. Technical Report 85-08-06, Department of Computer Science, Univesity of Wshington, September 1985.
- [PAPA79] Papadimitriou,C.H. Serializability of Concurrent Database Updates. Journal of the ACM 26(4): October, 1979, pp.631-653.
- [PAPA86] Papadimitriou, C.H. The Theory of Concurrency Control Computer Science Press, Rockville, MD, 1986.
- [PART88] Paris, J.F., Long, D.D.E. Efficient Dynamic Voting Algorithms. In Proc. 4th IEEE Int. Conf. on Data Engg. LA, Feb, 1988.
- [PARK83] Parker Jr., D.S., Popek, G.J., Rudisin, G., Stoughton, A. Walker, B.J., Walton B., Chow, J.M., Edwards, D., Kiser, S., Kline, C. Detection of Mutural Inconsistency in Distributed Systems. IEEE Trans. on Software Engineering SE-9(3): pp.240-247, May, 1983.
- [PU86] Pu, C., Jerre, D., Proudfoot, A. Regeneration of Replicated Objects : A Technique and Its Eden Implimentation. In Proceedings International Conference on Data Engineering, IEEE Computer Society, 1986.
- [REED83] Reed, D.P. Implimenting Atomic Actions on Decentralized Data. ACM Transactions on Computer Systems. Vol,1.No.1, Feb.1983.

- [ROTH77] Rothie, J.B., Goodman, N., Bernstein, P.A. The redundant Update Methodology of SDD-1: A System for Distributed Databases (The Fully a Redundant Case), Rep. No.CCA-77-02, Computer Corporation of America, 1977.
- [ROYH77a] Rothnie, J.B., Goodman, N., A Study of Updating in a Redundant Distributed Database Environment. Computer Corp. America, Cambridge, MA, Tech. Rep. CCA-77-01, Feb.15, 1977.
- [SARI86] Sarin, S.K. Rubust Algorithm Design in Highly Available Distributed Databases. In Proc. of the 5th Symposium on Reliability in Distributed Software and Database System. Los Angles, January 1986, pp.87-94.
- [SCHL83] Schlicting, R.S., Scheneider, F. Fail Stop Processors : An Approach to Designing Fault Tolerant Distributed Computer Systems. ACM Transactions on Computer Systems.1, 3(1983), pp.22-228.
- [SELI80] Selinger, P.G. Replicated Data in Distributed Databases. I.W. Draffen and F. Poole,Eds. Cambridge University Press, Cambridge 1980.
- [SKEE84] Skeen, D., Wright, D. Increasing Availability in Partitioned Networks. In Proc. of the 3rd SIGACT-SIGMOD Symposium on Principles of Database Systems, New York, 1984.
- [STON79] Stonbraker, M. Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES. IEEE Trans. on Software Engineering 3(3):May, 1979, pp.188-194.
- [TANG88] Tang, J., Natarjan, N.A. Formal Model for Pessimistic Schemes for Managing Replicated Databases, Tech., Report, Dept. of Computer Science, Pennsylvania State University 1988.
- [TANG89a] Tang, J., Natarajan, N.A. Static Pessimistic Scheme for Handling Replicated Databases. In Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Portland, Oregon.
- [TAKA79] Takagi, A. Concurrent and Reliable Updates of Distributed Databases. MIT Laboratory for Computer Science, Report, MIT/LCS/TM-144, Nov., 1979.
- [THOM78] Thomas, R.F.A. Solution to Concurrency Control Problem for Multicopy Databases. In Proc. Spring COMPCON, Feb.28-March 3, 1978.

- [THOM79] Thomas, R.H. A Majority Consensus Approach to Concurrency control for Multiple Copy Databases. ACM Trans. on Database Systems 4(2):June 1979, pp.180-209.
- [TONG88] Tong, Z., Kain, R.Y. Vote Assignments in Weighted Voting Mechanisms. In Proc. of Seventh Symposium on Reliable Distributed Systems. October 1988, pp.138-143.
- [WEIH84] Wehl, W. Specification and Implementation of Atomic Data Types, Ph.D. Thesis, MIT, March, 1984.
- [WUU84] Wu, G.T.J., Bernstein, A.J. Efficient Solutions to the Replicated Log and Dictionary Problems. In Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing. Vancouver, Aug. 1984, pp.233-242. Also Appears in ACM Operating System Review. Vol.20, No.1, January 1986, pp.57-66.
- [YANN84] Yanakakis, M. Serializability by Locking. ACM, 31, April 1984, pp.227-234.

