UNIT - IV  Synchronization and Replication

Introduction

Generally time plays an important role in the applications. Example, there is a need of computers around the world of timestamp the electronic commerce transactions consistently.

Clocks, Events and process States:

A distributed system holds a process set \( P \), it contains \( N \) number of processes. These processes are kept separately on a single processor.

Every process \( p_i \) in \( P \) contains a state \( s_i \) which it sends as it executes.

Every process \( p_i \) does an action. This may be:

* Send Operation
* Receive
* Transforming

The history of process \( p_i \) is defined as the series of events which happens within it. It is given as:

\[
\text{History}(p_i) = h_i = <e^0, e^1, e^2, ...>
\]
Clocks:

Every computer has its own physical clock. These are the electronic devices. It will count the oscillations came in a crystal at a definite frequency. This count is divided and then stored in a counter register. These clock devices can be utilized inorder to generate interrupts within given intervals in the order.

The mean time of the real, physical time t for the process Pi, a software clock Ci(t) is generated as

\[ Ci(t) = \alpha \cdot Hi(t) + \beta \]

where \( Hi(t) \) - needs hardware clock value
\( \beta \) - offset

clock resolution

It is a period between the updates of the clock value.

clock skew and clock drift

computer clocks cannot be perfect, the difference between a set of readings of any two clocks termed as 'skew'. Crystal based clocks may have clock drift.
Clock drift:

It refers to the time count at different rates, so the corresponding clocks may diverge.

Clock drift rate

This is referred as the change in the offset between

- clock

Normal perfect reference clock per unit of time measured by the reference clock.

Synchronizing physical clocks

It must be synchronized the processes clocks $C_i$ with the external source of time. It is called as external synchronization. It is possible to calculate the interval between two events happened at different computers by appealing to their local clocks. This is done at the time of synchronizing the clocks $C_i$ with one another to a known degree of accuracy.

This is said to be interval synchronization.

These synchronization types are stated as.
1. **External Synchronization**

For a synchronization limit $D > 0$ and for a source $S$ of UTC time, it is stated that

$$|S(t) - C_i(t)| < D \quad \text{for } i = 1, 2, \ldots, N \text{ and for all real times } t \text{ to } t'$$

It implies that the clocks $C_i$ are accurate to within the bound $D$.

2. **Internal Synchronization**

For a synchronization limit $D > 0$, it is given as

$$|C_i(t) - C_j(t)| < D \quad \text{for } i, j = 1, 2, \ldots, N \text{ and for all real times } t \text{ to } t'$$

**Monotonicity**

This refers to a condition which a clock $C$ ever advances

$$t' \rightarrow C(t') > C(t)$$

**Example:**

Unix made facility. There is a hybrid resource correctness condition which may expect that there is a need of clock that obeys the monotonicity condition. This drift rate of the clock should be bounded between the synchronization points.
Synchronization in a synchronous system

The bounds are measured in a synchronous system for:
* The drift rate of clocks
* Maximum message transmission delay
* Execution time of every step in a process

One process transmits the time t on its local clock in a message m. So the time of receiving clocks is set to be t + T_{trans}. T_{trans} is the transmission time of a message. It must to agree both the clocks.

The transmission time for an asynchronous system is given as

\[ T_{trans} = m + x \]

where \( x \geq 0 \), here x value can be unknown.

Cristian’s method for synchronizing clocks

Cristian told to use a time server to get the signals from a source of UTC. A device is employed. This device and time server is connected with each other for synchronizing the computers externally.

Based on the clocks, the server process gives the time.
It is given as

If there is no upper bound on the message transmission delay in an asynchronous system then round trip times are short. It may be a small fraction of a second. Cristian said that this algorithm is probabilistic. It is possible to achieve the synchronization only when the short round trip times between client and server is obtained.

Cristian's method is affected by the problem along with all the services given by a single server. This single time server can be failed and so it render synchronization impossible temporarily.

The Berkeley algorithm

Gusella and Zatti gave an algorithms for achieving internal synchronization of the set of computers which runs Berkeley Unix.
It has to poll other computers named as slaves. The computers need to be synchronized. The master has to:

* Calculates the local clock times with help of Round Trip times.
* Averages the received values.

The accuracy of the protocol is based on a minimum round trip time among:

* Master
* Slaves

If any of the reading is large than this minimum then it will be deleted by the master. It has a duty of transmitting the amount using which every individual slave’s clock needs the adjustment.

It can be a positive or negative value. The algorithm removes the reading from faculty clocks, when the ordinary average has given. These types of clock may get a significant adverse effect.

If the master gets failed to work then another one is elected to do the functions exactly as its previous master.
The Network Time protocol (NTP)

It implies an architecture for a time service
→ a protocol for distributing the time information through internet.

Objectives and Features of NTP

1. To enable a service which makes the clients across the internet to be synchronized accurately to UTC.
2. To give a better service which can survive lengthy losses of connectivity.
3. To makes the clients to resynchronize onely to offset the rates of drift in various computers.
4. To give higher protection for the interference problem with the time service, whether it can be malicious or accidental.

There are three models to synchronize NTP servers.

i) Multicast Mode

This mode normally employed for a high speed LAN.

ii) Procedure call Mode

It is a mode which one server can have the request of others. Then it processes by replying with its timestamp.
iii) Symmetric mode:

This mode is used

* By the server which give time information in LAN's
* By higher level of synchronisation subnet

**Logical time and logical clocks.**

A physical time cannot be used to know the order of any arbitrary pair of events happened within it.

This type of ordering is depending upon two suggestions. They are

1. When two events happened at the same time (process)\( P_i \) \( (i=1,2,3\ldots N) \), they occurred in the order in which \( P_i \) observer them.

2. When the message is transmitted, transmission should take place first before the reception.

These two suggestions are accepted by Lamport and it is said to be happened before relation:

Happened before relation:

It is represented by \( \rightarrow \)

This is stated as
HB1: If this process \( P_i: e \rightarrow e' \) then \( e \rightarrow e' \).

HB2: Transmit\((m)\) \(\rightarrow\) receive\((m)\) for any type of message

Transmission\((m)\) \(\rightarrow\) refers the message transmission

Receive\((m)\) \(\rightarrow\) refers message reception

HB3: If \( e, e', e'' \) are events such that
\[ e \rightarrow e', e' \rightarrow e'' \] then \( e \rightarrow e'' \).

Here, HB1 or HB2 can be applied between \( e_i \) and \( e_{i+1} \).

The relation \( \rightarrow \) is given for a set of processes \( P_1, P_2, \) and \( P_3 \).

It is given as
Logical clocks

The happened before ordering can be obtained numerically by a concept told by Lamport. It is said to be logical clock. Logical clock is monotonically increasing software counter.

The timestamp of event e at pi is given as \( L_i(e) \), the happened before relation \( \rightarrow_h \) is captured by updating the logical clocks of the processes and transmitting these values in the messages. It is given as

\[ L_C1: L_i \text{ is incremented before every event is given at process } p_i. \text{ It is stated as } \]
\[ L_i = L_i + 1 \]

\[ L_C2: (i) \text{ when a process } p_i \text{ transmits a message } m, \text{ it piggybacks on } m \text{ the value } t \text{ is given as } \]
\[ t = L_i \]

\[ (ii) \text{ The } L_i \text{ value is calculated by a process when getting } (m, t). \text{ It is given as } \]
\[ L_i = \max (L_i, t) \]

\[ L_C1 \text{ is applied before time stamping the event receive } (m). \]
Vector clocks:

These are developed by manner 10 n: 1t refers an array of N integers for a system of N processes. Every process holds its own vector clock v_i for the purpose of time stamping local events. Here the processes piggyback vector timestamps on the message which they transmitted. There are some rules to update the clock, they are:

Vc1 : v_i[0] = 0 for i = 1, 2, ..., N at initial stage.

Vc2 : p_i sets v_i[i] = v_i[i] + 1 before it time stamps.

Vc3 : when p_i transmits a message, it must to add the value t = v_i.

VCA : at the time of getting a time stamp t in a message, it sets v_j[i] = max(v_j[i], t[i]) for i = 1, 2, ..., N.

For a vector clock v_i, v_i[i] refers the number of events that p_i has time stamped. v_i[i] (i = 2) refers the number of events which have occurred at p_i.

Global states:

All of the concurrency issues which are faced in a tightly coupled system-like

* Mutual Exclusion
* Deadlock
* Starvation
The design strategies in these areas are very difficult to implement. Because there is no global state to the system.

**Global state and consistent cuts:**

Actually a global state is said to be the combined state of all the processes, when the processes are having correct synchronized clocks, it is acceptable that time at which every process would record it’s state.

Let the general system \( \mathcal{X} \) of \( N \) processes be

\[ P_i = (i = 1, 2, \ldots, N) \]

A series of events happened at every process the characteristics of every process is given by it’s history. This is stated as

\[ \text{History (} P_i \text{)} = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle \]

Any finite prefix of the process's history is given as

\[ h_i^k = \langle e_i^0, e_i^1, \ldots, e_i^k \rangle \]

The global history \( \mathcal{G} \) is created a global state. This is given as

\[ \mathcal{G} = (s_1, s_2, \ldots, s_N) \]
A cut of the system execution is a subset of its global history which is a union of prefixes of process histories. This is represented as

\[ C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_n} \]

**Consistent global state:**

This relates to a consistent cut. The distributed system execution is given as series of transitions among global states of the system.

\[ S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \]

one event can be happened at some single process in the system in every transition.

**Snapshot algorithm of Chandy and Lamport**

This algorithm is used to find the global state of the distributed system. It is aimed to record a set of:

1. Process
2. Channel State

For a collection of processes. This algorithm keeps state locally at process. It does not employ any method to collect the global state at one site.

Here, not all the processes, the state which the processes collected has to be send to a designated collector process.
Some of the assumptions used in this algorithm are:

i) Neither channel nor processes fail, so every transmitted message is obtained exactly once.

ii) The channels are undirectional, so the FIFO ordered message delivery is possible.

iii) It must be given as a function of the graph of the processes and the channels.

iv) A global snapshot can be initiated by any process at any time.

v) When the snapshot is taken, it is allowed for the process to continue their execution:
   - Sending messages.
   - Receiving messages.

Coordination and Agreement

There are many algorithms to implement the mutual exclusion for a set of processes in order to coordinate their access to the shared resource.

In an asynchronous system, there is no timing assumption. But in case of synchronous system, there are bounds on the maximum message transmission delay on:

i) The turn to execute every step of a process

ii) The clock drift rates.
The timeouts are used to find the process crashes with help of synchronous assumptions. There are some problems in the distributed systems. They are:

* Mutual Exclusion
* Resource sharing

Failure assumptions and Failure detectors

When the network components are affected by the failures, a reliable communication protocol is used by the processes. This protocol will mask these types of failures.

Example:

* Retransmission of corrupted information
* Router failure between 2 networks

Failure detector

It is said to be a service which can process the queries related to a process whether it has been failed or not. This is done by an object which will be local to every process. This process runs a failure detection algorithm along with its counterparts at other processes.

Types of failure detectors:

1. Reliable failure detector
2. On reliable failure detector
1. Reliable Failure detector

It is a type of detector which detects the failures accurately. This passes the responses for the queries of processes in terms of

* unsuspected
* Failed

2. Unreliable failure detector

It should not produce the accurate result of identifying the failures. This may give responses in terms of

* unsuspected
* Failed

Both will not conclude whether the process has been crashed or not. If the response is 'Unsuspected' then it means that detector holds an evidence to prove that the process has not failed. If it is 'Suspected' then it means that detector has a clue to say that the processes have been failed.
Distributed Mutual Exclusion

In a distributed system, however, neither shared
variable nor facilities supplied by a single local kernel
can be used to solve it, in general. We require to
distributed mutual exclusion, one that is based solely
on message passing.

Requirements for implementing mutual exclusion

1. Mutual Exclusion

For shared resources which can be used by any
concurrent process, only one process should employ the
resources at any time. It means that a process which has
been granted the resources have to release it, before
it can be granted to another process.

2. No Starvation:

when a process is granted the resource eventually
release it, each request have to be eventually granted.

In case of single processor systems, this mutual
exclusion is implemented by using

- semaphores
- monitors
- locks.
Distributed mutual exclusion algorithms are classified into two types:

1. Symmetric algorithm

   A client process wants to enter a critical section required to consult other client processes.

2. Token based algorithms

   Only a process possessing a token can enter a critical section. When some of the process tries to enter it transmits a request to the token holder and waits till it gets the token.

Algorithm for Mutual Exclusion:

There is a system of holding \( N \) processes \( P_i \), \( i = 1, 2, 3 \ldots N \), which do not share variables. This system is asynchronous, that processes do not fail. That message delivery is reliable so the message can be transmitted is eventually delivered intact, exactly once. An application level protocol is given as:

1. \( \text{enter}(i) \)

   It is used to enter into the critical section block, if needed.

2. \( \text{resource access}(i) \)

   It is employed to use the shared resources in the critical section.
3. exit w

This is used to leave the critical section, so that any other process can be entered.

Evaluating the performance of algorithm for mutual Exclusion

This evaluation is made based on the factors

* Bandwidth
* Client delay
* Through put

Elections

An algorithm for choosing a unique process to play a particular role is called an election algorithm. For example, in a variant of our `central-server` algorithm for mutual exclusion, the `server` is chosen from among the processes \( p_i, i = 1, 2, \ldots, N \) that need to use the critical section.

A Ring based Election algorithm

Each process \( p_i \) has a communication channel to the next process in the ring, \( p_i = (i + 1) \mod N \), and all messages are sent clockwise around the ring. We assume that no failure occurs, and that the system is asynchronous.
The goal of this algorithm is to elect a single process called the coordinator, which is the process with the largest identifier.

Initially, every process is marked as a non-participant in an election. Any process can begin an election. It proceeds by marking itself as a participant, placing its identifier in an election message and sending it to its clockwise neighbour.

**Working of the algorithm**

If a process gets the token, it verifies if it needs to enter a critical section and acts as:

* If it likes to enter a critical section then it maintains the token. Also it enters the critical section.
  Then the token is passed along the ring to its neighbor process.

* Else if does not want to enter a critical section, it sends the token to its neighbor process. Suppose if any of the process does not want to enter in critical section then the token is simply circulating around the ring.
**The Bully algorithm**

This algorithm is found by Garola-Medina when a process realizes that the coordinator is no longer responding to the request message, an election is initiated. A process may have an election as in the given way.

1. P transmits an election message to all other processes which are having higher numbers.
2. If any one of the other processes does not transmit its reply then it will be decided that P wins in the election. So now P becomes the coordinator.
3. If any one of the higher up’s responses then it takes over. Also the process P’s job is done.

There are three types of messages are used while implementing this algorithm. They are:

- **Election** → It informs an election
- **Answer** → This refers the response of an election message
- **Coordinator** → This is passed to inform the identity of the elected process.

Generally a process may initiate its election when it comes to know that the coordinator has failed. Many number of processes can able to identify this situation at the same time.
A reliable failure detector is developed because of the synchronous system. The maximum message transmission delay = \( T_{\text{trans}} \). Maximum delay for processing a message = \( T_{\text{proc}} \). So, an upper bound on total elapsed time is given as

\[
T_{\text{elap}} = 3T_{\text{trans}} + T_{\text{proc}}
\]

**Step 1**

**Step 2**

Eventually

Coordinator

www.studentsfocus.com
Transactions and Concurrency Control

Transactions:

In some situations, clients require a sequence of separate requests to a server to be atomic in the sense that:

1) they are free from interference by operations being performed on behalf of other concurrent clients, and

2) either all of the operations must be completed successfully, or they must have no effect at all in the presence of server crashes.

A transaction applies to recoverable objects and is intended to be atomic. It is often called an atomic transaction. There are two aspects:

All or nothing

A transaction either completes successfully, and the effects of all of its operations are recorded in the objects, or it has no effect at all. This all-or-nothing effect has two further aspects of its own:

* Failure atomicity
* Durability

Isolation: Each transaction must be performed without interference from other transactions.
A transaction is either successful or it is aborted in one of two ways - the client abort it or the server abort it.

* Service actions related to process crashes
* Client actions related to server process crashes.

Concurrent control

Two well known problems of concurrent transactions in the context of the banking example - the 'lost update' problem and the 'inconsistent retrievals' problem.

The lost update problem:

The lost update problem is illustrated by the following pair of transactions on bank account A, B and C, whose initial balances are $100, $200 and $300 respectively. Transaction T transfers an amount from account A to account B. Transaction U transfers an amount from account C to account B.

Inconsistent retrievals

The balances of the two bank accounts, A and B, are both initially $200. The result of branch total includes the sum of A and B as $300, which is wrong. This is an illustration of the 'inconsistent retrievals' problem.
Serial Equivalence:

An interleaving of the operations of transactions in which the combined effect is the same as if the transactions had been performed one at a time in some order is a serially equivalent interleaving.

Conflicting Operations:

For any pair of transactions, it is possible to determine the order of pairs of conflicting operations on objects accessed by both of them.

Recoverability from abort:

Servers must record the effects of all committed transactions and none of the effects of aborted transactions. They must therefore allow for the fact that a transaction may abort by preventing it affecting other concurrent transactions if it does so.

*Dirty reads:

This problem is caused by the interaction between a read operation in one transaction and an earlier write operation in another transactions on the same object.
* Recoverability of transactions
* Cascading aborts
* Premature aborts (writes)
* Strict executions of transactions
* Tentative versions

Nested Transactions

Nested transactions extend the above transaction model by allowing transactions to be composed of other transactions. Thus several transactions may be started from within a transaction, allowing transactions to be regarded as modules that can be composed as required.

The outermost transaction in a set of nested transactions is called the top-level transaction. Transactions other than the top-level transactions are called subtransactions.

Nested transactions have the following advantages:

1. Subtransactions at one level may run concurrently with other subtransactions at the same level in the hierarchy.
5. Subtransactions can commit or abort independently.

Rules for committing of nested transactions:

* A transaction may commit or abort only after
  its child transactions have completed.
* When a subtransaction completes, it makes an
  independent decision either to commit provisionally or
  to abort. If decision to abort is final.
* When a parent aborts, all of its subtransactions
  are aborted.
* When transaction aborts, the parent can decide
  whether to abort or not.
* If the top-level transaction commits, then all of the
  subtransactions that have provisionally committed
  can commit too, provided that none of their ancestors,
  has aborted.

Locks

A simple example of a serializing mechanism is
the use of exclusive locks. In this locking scheme,
the server attempts to lock any object that is about
 to be used by any operations of a client transaction’s.
If a client requests access to an object that is already locked due to another client's transactions, the request is suspended and the client must wait until the object is unlocked.

**Lock implementation**

The granting of locks will be implemented by a separate object in the server that we call the lock manager. The lock manager holds a set of locks, each lock is an instance of the class Lock, and is associated with a particular object.

```
public class Lock {
    private Object object;
    private Vector holders;
    private LockType lockType;

    public synchronized void acquire(Transaction trans, LockType aLockType) {
        while (!)
            try {
                wait();
            } catch (InterruptedException e) {
            }
    }
```
if (holders. is empty()) {
    holders. add element (trans);
    lock. type = a lock. type;
} else if (an other transaction hold the lock share it *) {
    if (in this transaction not a holder *) holders. add element (trans);
    else if (in this transaction a holder but needs a more exclusive lock *)
        lock. type. promote (trans);
}

public synchronized void release (trans to trans) {
    holders. remove element (trans);
    notify. all (trans);
}

Each instance of lock maintains the following information in its instance variables:
- the identifier of the located object
- the transaction identifiers of the transactions that currently hold the lock
- a lock type
Locking rules for nested transactions:

The aim of a locking scheme for nested transactions is to serialize access to objects so that:

1. Each set of nested transactions is a single entity that must be prevented from observing the partial effects of any other set of nested transactions.

2. Each transaction within a set of nested transactions must be prevented from observing the partial effects of the other transactions in the set.

Deadlocks:

Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock. A wait-for graph can be used to represent the waiting relationships between current transactions. In a wait-for graph, the nodes represent transactions and the edges represent wait-for relationships between transactions.
Lock Compatibility

<table>
<thead>
<tr>
<th>For one object</th>
<th>Lock Requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock already set</td>
<td>read</td>
</tr>
<tr>
<td>none</td>
<td>read</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
</tr>
<tr>
<td>write</td>
<td>OK</td>
</tr>
</tbody>
</table>

Deadlock prevention

One solution is to prevent deadlocks. An apparently simple but not very good way to overcome deadlocks is to lock all of the objects used by a transaction when it starts. This would need to be done as a single atomic step so as to avoid deadlocks for this stage. Such a transaction cannot run into deadlock with other transactions, but it unnecessarily restricts access to shared resources.

Upgrade locks:

CORBA's concurrency control service introduces a third type of lock, called upgrade, the use of which...
is intended to avoid deadlock. A transaction with an upgrade lock on a data item is permitted to read that data item, but this lock conflicts with any upgrade locks set by other transactions on the same data item. This type of lock cannot be set implicitly by the use of a read operation, but must be requested by the client.

**Deadlock detection**

Deadlocks may be detected by finding cycles in the wait-for graph. Having detected a deadlock, a transaction must be selected for abortion to break the cycle.

**Timeouts:**

Lock timeouts are a method for resolution of deadlocks that is commonly used. Each lock is given a limited period in which it is invulnerable. After that time, a lock becomes vulnerable.
Increasing concurrency in locking schemes

Even when locking rules are based on the conflicts between read and write operations and the granularity of which they are applied is as small as possible, there is still some scope for increasing concurrency. There are two approaches used:

1. Quasi version locking - the setting of exclusive locks is delayed until a transaction commits.

2. Hierarchical locks - mixed granularity locks are used.

Optimistic concurrency control

Kung and Robinson identified a number of inherent disadvantages of locking and proposed an alternative optimistic approach to the serialization of transactions. Each transaction has the following phases:

* Working Phase
* Validation Phase
* Update Phase
Validation of transactions

Validation uses the read-write conflict rules to ensure that the scheduling of a particular transaction is serially equivalent with respect to all other overlapping transactions in any transactions that had not yet committed at the time this transaction started.

The validation of transactions can be classified into two types, as follows:

- Backward validation
- Forward validation

Time stamp ordering

In concurrency control schemes based on timestamp ordering, each operation in a transaction is validated when it is carried out. If the operation cannot be validated, the transaction is aborted immediately and can then be restored by the client. Each transaction is assigned a unique time stamp value when it starts.

The timestamp decides its position in the time sequence of transactions. Requests from transactions can be totally ordered according to
Their timestamps. The basic timestamp ordering rule is based on operation conflicts and is very simple.

A transaction's request to write an object is valid only if that object was last read and written by earlier transactions. A transaction's request to read an object is valid only if that object was last written by an earlier transaction.

**Timestamp ordering write rule:**

\[
\text{if } (T \geq \text{maximum read timestamp on } X) \text{ then write timestamp on committed version of } X
\]

perform write operation on tentative version of \( X \)

with write timestamp \( T_e \)

else

Abort transaction \( T_e \)

**Timestamp ordering read rule:**

\[
\text{if } (T \geq \text{write timestamp on committed version of } X) \text{ then }
\]

let \( D \) selected be the version of \( X \) with the maximum write timestamp \( \leq T_e \)

\[
\text{if } (D \text{ selected is committed) then }
\]

Perform read operation on the version \( D \) selected

else wait until the transaction that made version \( D \) selected commits or aborts

then reapply read rule if else

Abort transaction \( T_e \)
Atomic commit protocols

A transaction comes to an end when the client requests that a transaction be committed or aborted. A simple way to complete the transaction in an atomic manner is for the coordinator to communicate the commit or abort request to all the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they had carried it out. This is an example of a one-phase atomic commit protocol.

The one-phase atomic commit protocol is inadequate because, in the case when the client requests a commit, it does not allow a server to revoke a unilateral decision to abort a transaction.

The two-phase atomic commit protocol is designed to allow any participant to abort its part of transaction. Due to the requirement for atomicity, if one part of a transaction is aborted, then the whole transaction must also be aborted.

Two-phase commit protocol

There is no connection between the coordinator and the participants apart from the participants informing the coordinator when they join the transaction. A client’s request commit (or abort) a transaction is directed
To the coordinator, if the client request abort Transaction, or if any of the transactions is aborted by one of the participants, the coordinator informs the participants immediately. It is when the client asks the coordinator to commit the transaction that two-phase commit protocol comes into use.

Operations for two-phase commit protocol:

- Can Commit (Trans) → Yes/No
- doCommit (Trans)
- doAbort (Trans)
- haveCommitted (Trans, Participant)
- getDecision (Trans) → Yes/No

Two-phase commit protocol for nested transaction

The outermost transaction in a set of nested transactions is called the top-level transaction. Transactions other than the top-level transaction are called sub-transactions. If the top-level transaction is Ti, Ti1, Ti2, Ti3, Ti4, Ti5, and T12 are sub-transactions. T1 and T2 are child transactions of Ti, which is referred to as their parent.
Distributed Deadlocks

Deadlocks can arise within a single server when locking is used for concurrency control. Servers must either prevent or detect and resolve deadlocks. Using timeouts to resolve possible deadlocks is a clumsy approach — it is difficult to choose an appropriate timeout interval, and transactions are aborted unnecessarily. With deadlock detection schemes, a transaction is aborted only when it is involved in a deadlock.

Most deadlock detection schemes operate by forming cycles in the transaction wait-for graph. In a distributed system involving multiple servers being accessed by multiple transactions, a global wait-for graph can in theory be constructed from the local ones. There can be a cycle in the global wait-for graph that is not in any single local one (i.e., there can be a distributed dead-lock).
Detection of a deadlock requires a cycle to be found in the global transaction wait-for graph that is distributed among the servers that were involved in the transactions.

Example:

Server Y: U → V (added when U requests b.withdraw(30))
Server Z: V → W (added when V requests c.withdraw(30))
Server X: W → U (added when W requests a.withdraw(30))

Phantom deadlock:

A deadlock that is 'detected' but is not really a deadlock is called a phantom deadlock. In distributed deadlock detection, information about wait-for relationship between transactions is transmitted from one server to another. If there is a deadlock, the necessary information will eventually be collected in one place and a cycle will be detected.

Edge chasing

A distributed approach to deadlock detection uses a technique called edge chasing or path pushing. In this approach, the global wait-for graph is not constructed, but
Each of the servers involved has knowledge about some of its edges. The server attempts to find a cycle by forwarding messages, called probes, which follows the edges of the graph throughout the distributed system. A probe message consists of transaction wait-for relationship representing a path in the global wait-for graph.

Edge-chasing algorithms have three steps:

* initiation
* detection
* resolution

Replication:

Replication is a technique for enhancing services. The motivations for replication are to improve a service's performance, to increase its availability, or to make it fault tolerant.

* Performance enhancement
* Increased availability
* Fault tolerance
System model and Group Communication

An object could be a file, say on a Java object. But each such logical object is implemented by a collection of physical copies called replicas. The replicas are physical objects, each stored at a single computer, with data and behaviour that are tied to some degree of consistency by the system's operation. The replicas of a given object are not necessarily identical, at least not at any particular point in time. Some replicas may have received updates that others have not received.

**System model:**

The model involves replicas held by distinct, which are components that contain the replicas on a given computer and perform operations upon them directly. This general model may be applied in a client-server environment, in which case a replica manager is a server.

The set of replica manager may be static or dynamic. In dynamic system, new replica manager may appear. This is not allowed in a static system. In static system, replica managers do not crash, but they may cease operating for an indefinite period.
Group Communication:

Groups are useful for managing replicated data, and in other systems where processes cooperate towards a common goal by receiving and processing the same set of multicast messages.

Role of the Group Membership Service:

A group membership service has four main tasks, as follows:

* Providing an interface for group membership changes
* Implementing a failure detector
* Notifying members of group membership changes
* Performing group address expansion
View delivery:

The programmer's life is made harder or easier according to the guarantees that apply when the system delivers view to the group members. For each group g, the group management service delivers to any member process peg a series of views \( V_0(g), V_1(g), V_2(g), \ldots \) etc.

Some basic requirements for view delivery are as follows:

* Order
* Integrity
* Non-triviality

View-synchronous group communication

A view-synchronous group communication system makes guarantees additional to those above about the delivery ordering of view notification with respect to the delivery of multicast messages. View synchronous communication extends the reliable multicast semantics to take account of changing group views. The guarantees provided by view synchronous group communication are as follows:

* Agreement
* Integrity
* Validity (closed groups)
The Coda file system is a descendant of NFS that aims to address several requirements that NFS does not meet, particularly the requirement to provide high availability despite disconnect operation. The design requirement for Coda was derived from experience with NFS at CMU and elsewhere, involving its use in large scale distributed systems on both local and wide area communication networks.

Coda aims to meet all three of these requirements under the general heading of constant data availability. The aim was to provide users with the benefits of a shared file repository, but to allow them to rely entirely on local resources when the repository is partially or totally inaccessible.

The Coda Architecture

Coda runs what it calls 'Venus' processes at the client computers and 'vice' processes at file server computers, adopting the NFS terminology. The vice processes are what we have called replica managers. The Venus processes are a hybrid of front ends and replica managers. The Venus processes are a hybrid of front ends and replica managers.
The set of servers holding replica of a file volume is known as the volume storage group (VSG). At any instant, a client wishing to open a file in such a volume can access some subset of the VSG, known as the available volume storage group (AVSG).

Coda file access proceeds in a similar manner to NFS, with cached copies of files being supplied to the client computer by any one of the servers in the current AVSG. In coda, disconnected operation is said to occur when the AVSG is empty. This may be due to network or server failures, or it may be a consequence of the deliberate disconnection of the client computer, as in the case of laptop.

The replication strategy:

Coda replication strategy is optimistic - it allows modification of files to proceed when the network is partitioned or during disconnected operation. It relies on the attachment to each version of a file of a coda version vector (CVV). A CVV is a vector of timestamps, one element for each server in the relevant VSG.

Each element of the CVV is an estimate of the number of modification performed on the version of the file (i.e., held at) the corresponding server.
The advantages deriving from the replication of some or all file volumes on multiple servers are:

1. The files in a replicated volume remain accessible to any client that can access at least one of the replicas.
2. The performance of the system can be improved by shunting some of the load of servicing client requests on a replicated volume between all of the servers that hold replicas.

**Update semantics**

The currency guarantees offered by coda when a client opens a file are weaker than for APs, reflecting the optimistic update strategy.

**Accepting replica**

The strategy used on open and close to access the replicas of a file in a variant of the read-one/write-all approach. On open, if a copy of the file is not present in the local cache, the client identifies a preferred server from the Avail for the file. The preferred server may be chosen at random, or on the basis of performance criteria such as physical proximity or server load.
Cache Coherence:

The cache currency guarantees stated above mean that the Venus process at each client must detect the following events within 7 seconds of their occurrence.

1. Enlargement of an AVSG
2. Shrinking of an AVSG
3. A lost callbox event

Disconnected Operations

When disconnected operations end, a process of reintegration begins. For each cached file or directory that has been modified, created or deleted during disconnected operation, Venus executes a sequence of update operations to move the AVSG replicas identical to the cached copy. Reintegration proceeds top-down from the root of each cached volume.