4

SYNCHRONIZATION AND REPLICATION

4.1 INTRODUCTION TO SYNCHRONIZATION

Synchronizing data in a distributed system is an enormous challenge in and of itself. In single CPU systems, critical regions, mutual exclusion, and other synchronization problems are solved using methods such as semaphores. These methods will not work in distributed systems because they implicitly rely on the existence of shared memory.

Examples:

- ➢ Two processes interacting using a semaphore must both be able to access the semaphore. In a centralized system, the semaphore is stored in the kernel and accessed by the processes using system calls.
- ➢ If two events occur in a distributed system, it is difficult to determine which event occurred first.

Communication between processes in a distributed system can have unpredictable delays, processes can fail, messages may be lost synchronization in distributed systems is harder than in centralized systems because the need for distributed algorithms. The following are the properties of distributed algorithms:

- ❖ The relevant information is scattered among multiple machines.
- ❖ Processes make decisions based only on locally available information.
- ❖ A single point of failure in the system should be avoided.
- ❖ No common clock or other precise global time source exists

4.1.1 Issues in Clocks

Physical clocks in computers are realized as crystal oscillation counters at the hardware level. Clock skew(offset) is common problem with clocks. **Clock skew** is defined as the difference between the times on two clocks and **Clock drift** is the count time at different rates. **Clock drift rate** is the difference in precision between a prefect reference clock and a physical clock.

4.1.2 Synchronising clocks

There are two ways to synchronise a clock:

➢ **External synchronization**

- ∗ This method synchronize the process‟s clock with an authoritative external reference clock S(t) by limiting skew to a delay bound $D > 0$ - $|S(t) - Ci(t)| < D$ for all t.
- ∗ For example, synchronization with a UTC (Coordinated Universal Time)source.

➢ **Internal synchronization**

- ∗ Synchronize the local clocks within a distributed system to disagree by not more than a delay bound $D > 0$, without necessarily achieving external synchronization - $|Ci(t) - Cj(t)| < D$ for all i, j, t n
- ∗ For a system with external synchronization bound of D, the internal synchronization is bounded by 2D.

Checking the correctness of a clock

- ✓ If drift rate falls within a bound $r > 0$, then for any t and t‟ with $t‟ > t$ the following error bound in measuring t and t‟ holds:

  n $(1-r)(t‟-t) \leq H(t‟) - H(t) \leq (1+r)(t‟-t)$ n

- ✓ At this condition, no jumps in hardware clocks allowed
- ✓ The most frequently used conditions are:
  - ❖ Monotonically increasing
  - ❖ Drift rate bounded between synchronization points
  - ❖ Clock may jump ahead at synchronization points

Working of Computer timer:

- To implement a clock in a computer a counter register and a holding register are used.
- The counter is decremented by a quartz crystals oscillator.
- When it reaches zero, an interrupted is generated and the counter is reloaded from the holding register.

Clock skew problem

To avoid the clock skew problem, two types of clocks are used:

❖ Logical clocks : to provide consistent event ordering

❖ Physical clocks : clocks whose values must not deviate from the real time by more than a certain amount.

Software based solutions for synchronising clocks

The following techniques are used to synchronize clocks:

✓ time stamps of real-time clocks

✓ message passing

✓ round-trip time (local measurement)

Based on the above mentioned techniques, the following algorithmsprovides clock synchronization:

↑ Cristian‟s algorithm

↑ Berkeley algorithm

↑ Network time protocol (Internet)

4.1.3 Cristian's algorithm

Cristian suggested the use of a time server, connected to a device that receives signals from a source of UTC, to synchronize computers externally. Round trip times between processes are often reasonably short in practice, yet theoretically unbounded. The practical estimation is possible if round-trip times are sufficiently short in comparison to required accuracy.

Principle:

∗ The UTC-synchronized time server S is used here.

∗ A process P sends requests to S and measures round-trip time T_{round} .

∗ In LAN, T_{round} should be around 1-10 ms .

∗ During this time, a clock with a 10-6 sec/sec drift rate varies by at most 10-8 sec.

∗ Hence the estimate of T_{round} is reasonably accurate .

∗ Now set clock to t + ½ $T_{round.}$

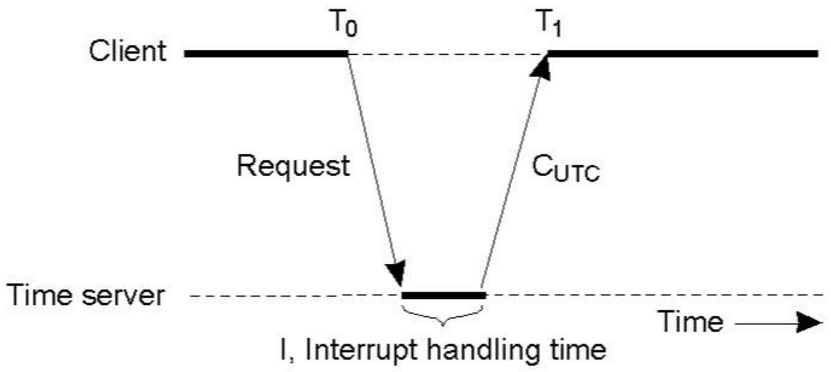


Fig 4.1 Cristian's algorithm

As per the given algorithm, the following results were found:

- ✓ Earliest time that S can have sent reply: t0 + min
- ✓ Latest time that S can have sent reply: t0 + T_{round} – min
- ✓ Total time range for answer: T_{round} - 2 * min
- ✓ Accuracy is ± (½T_{round} - min)

Problems in this system:

- Timer must never run backward.
- Variable delays in message passing / delivery occurs.

4.1.4 Berkeley algorithm

- Berkeley algorithm was developed to solve the problems of Cristian‟s algorithm.
- This algorithm does not need external synchronization.
- Master slave approach is used here.
- The master polls the slaves periodically about their clock readings.
- Estimate of local clock times is calculated using round trip.
- The average values are obtained from a group of processes.
- This method cancels out individual clock‟s tendencies to run fast and tells slave processes by which amount of time to adjust local clock
- In case of master failure, **master election algorithm** is used.

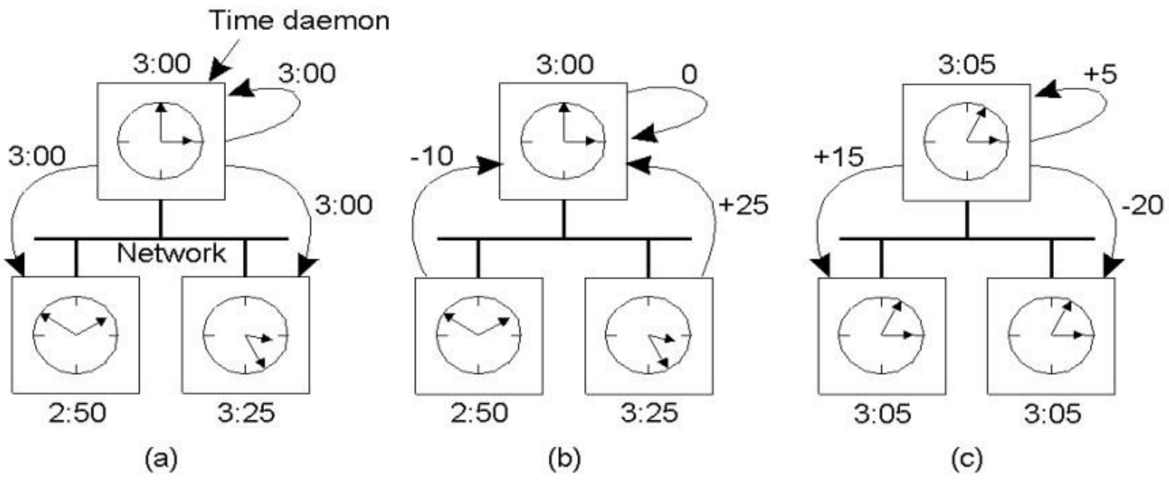


Fig 4.2: Berkeley Algorithm

In other words, the working of the algorithm can be summarized as,

- The time daemon asks all the other machines for their clock values.
- The machines answer the request.
- The time daemon tells everyone how to adjust their clock.

4.1.5 Network Time Protocol (NTP)

The Network Time Protocol defines architecture for a time service and a protocol to distribute time information over the Internet.

Features of NTP:

- To provide a service enabling clients across the Internet to be synchronized accurately to UTC.
- To provide a reliable service that can survive lengthy losses of connectivity.
- To enable clients to resynchronize sufficiently frequently to offset the rates of drift found in most computers.
- To provide protection against interference with the time service, whether malicious or accidental.

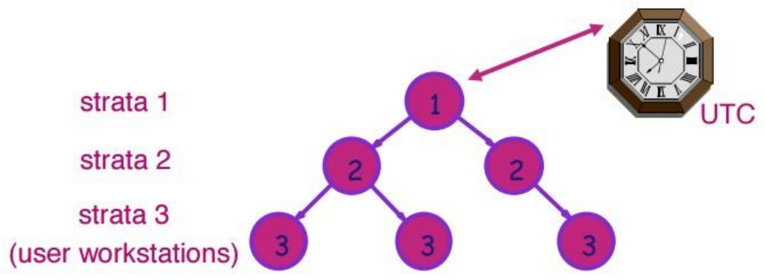


Fig 4.3: NTP strata

– The NTP service is provided b y a network of servers located across the Internet.

– Primary servers are connected directly to a time source such as a radio clock receiving UTC.

– Secondary servers are synchronized, with primary servers.

– The servers are connected in a logical hierarchy called a **synchronization subnet**.

– Arrows denote synchronization control, numbers denote strata. The levels are called **strata.**

Working:

↑ NTP follows a layered client-server architecture, based on UDP message passing.

↑ Synchronization is done at clients with higher strata number and is less accurate due to increased latency to strata 1 time server.

↑ If a strata 1 server fails, it may become a strata 2 server that is being synchronized though another strata 1 server.

Modes of NTP:

NTP works in the following modes:

❖ **Multicast:**

✓ One computer periodically multicasts time info to all other computers on network.

✓ These adjust clock assuming a very small transmission delay.

✓ Only suitable for high speed LANs; yields low but usually acceptable synchronization.

❖ **Procedure-call:**

✓ This is similar to Christian‟s protocol .

✓ Here the server accepts requests from clients.

✓ This is applicable where higher accuracy is needed, or where multicast is not supported by the network‟s hardware and software

❖ **Symmetric:**

✓ This is used where high accuracy is needed.

Working of Procedure call and symmetric modes:

- All messages carry timing history information.
- The history includes the local timestamps of send and receive of the previous NTP message and the local timestamp of send of this message
- For each pair i of messages (m, m‟) exchanged between two servers the following values are being computed

  - offseto_i : estimate for the actual offset between two clocks

  -delay d_i : true total transmission time for the pair of messages.

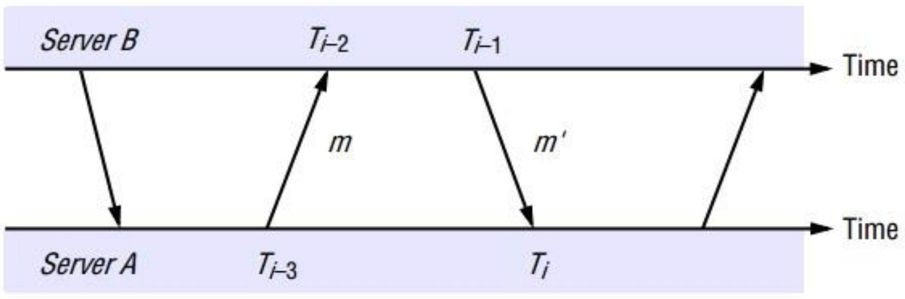


Fig 4.4: Message exchange between NTP

Delay and offset:

✓ Let o be the true offset of B‟s clock relative to A‟s clock, and let t and t‟ the true transmission times of m and m‟ (Ti , Ti-1 ... are not true time)

✓ The delay Ti-2 = Ti-3 + t + o………….. (1)

Ti = Ti-1 + t‟ – o ………………(2)

which leads to di = t + t‟ = Ti-2 - Ti-3 + Ti - Ti-1 (clock errors zeroed out à (almost) true d)

✓ Offset oi = ½ (Ti-2 – Ti-3 + Ti-1 – Ti) (only an estimate)

Implementing NTP

- ❖ Statistical algorithms based on 8 most recent pairs are used in NTP to determine quality of estimates.
- ❖ The value of oi that corresponds to the minimum di is chosen as an estimate for o .
- ❖ Time server communicates with multiple peers, eliminates peers with unreliable data, favors peers with higher strata number (e.g., for primary synchronization partner selection).
- ❖ NTP phase lock loop model: modify local clock in accordance with observed drift rate.
- ❖ The experiments achieve synchronization accuracies of 10 msecs over Internet, and 1 msec on LAN using NTP

4.2 LOGICAL TIME AND LOGICAL CLOCKS

- ✓ Synchronization between two processes without any form of interaction is not needed.
- ✓ But when processes interact, they must be event ordered. This is done by logical clocks.
- ✓ Consider a distributed system with n processes, p_1, p_2, …p_n.
- ✓ Each process p_i executes on a separate processor without any memory sharing.
- ✓ Each p_i has a state s_i. The process execution is a sequence of events.
- ✓ As a result of the events, changes occur in the local state of the process.
- ✓ They either send or receive messages.

4.2.1 Lamport Ordering of Events

The partial ordering obtained by generalizing the relationship between two process is called as **happened-before relation or causal ordering or potential causal ordering**. This term was coined by Lamport. Happens-before defines a partial order of events in a distributed system. Some events can"t be placed in the order.

- We say $e \rightarrow_i e''$ if e happens before e" at process i.
- $e \rightarrow e''$ is defined using the following rules:
  - ∗ Local ordering: $e \rightarrow e''$ if $e \rightarrow_i e''$ for any process i
  - ∗ Messages: send(m) → receive(m) for any message m
  - ∗ Transitivity: $e \rightarrow e''''$ if $e \rightarrow e''$ and $e'' \rightarrow e''''$

4.2.2 Logical Clocks

A Lamport logical clock is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock.

- ✓ Lamport clock L orders events consistent with logical happens before ordering.

  If $e \rightarrow e''$, then $L(e) < L(e'')$

- ✓ But the converse is not true.

- ✓

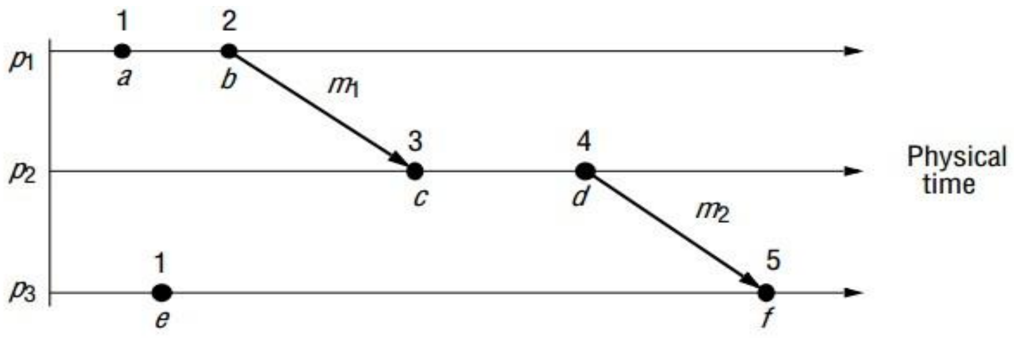


Fig 4.5: Lamport's timestamps for the three process

Lamport's Algorithm

* Assume that each process i keeps a local clock, L_i. There are three rules:

  1. At process i, increment L_i before each event.
  2. To send a message m at process i, apply rule 1 and then include the current local time in the message: i.e., send(m,L_i).
  3. To receive a message (m,t) at process j, set $L_j = \max(L_j,t)$ and then apply rule 1 before time-stamping the receive event.

* The global time L(e) of an event e is just its local time. For an event e at process i, $L(e) = L_i(e)$.

Totally ordered logical clocks

- Many systems require a total-ordering of events, not a partial-ordering.
- Use Lamport"s algorithm, but break ties using the process ID.
  - $L(e) = M * L_i(e) + i$

  Where M = maximum number of processes

Vector clocks

The main aim of vector clocks is to order in such a way to match causality. The expression, $V(e) < V(e")$ if and only if $e \rightarrow e"$, holds for vector clocks, where V(e) is the vector clock for event e. For implementing vector clock, label each event by vector V(e) $[c_1, c_2 \ldots, c_n]$. $c_i$is the events in process i that causally precede e.

- Each processor keeps a vector of values, instead of a single value.
- VC_i is the clock at process i; it has a component for each process in the system.

  $VC_i[i]$ corresponds to P_i„s local "time".

  $VC_i[j]$ represents P_i„s knowledge of the "time" at P_j(the # of events that P_i knows have occurred at Pj
- Each processor knows its own "time" exactly, and updates the values of other processors" clocks based on timestamps received in messages.

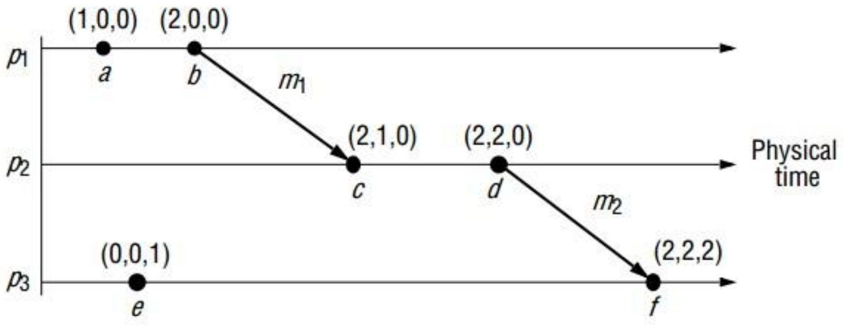


Fig 4.6: Vector Timestamps

Vector timestamps have the disadvantage, compared with Lamport timestamps, of taking up an amount of storage and message payload that is proportional to N, the number of processes.

4.3 GLOBAL STATES

This section checks whether a property is true in a distributed systems. This is checked for the following problems:

- ✓ distributed garbage collection
- ✓ deadlock detection
- ✓ termination detection
- ✓ Debugging

The global state of a distributed system consists of the local state of each process, together with the messages that are currently in transit, that is, that have been sent but not delivered.

Distributed Garbage Collection

– An object is considered to be garbage if there are no longer any references to it anywhere in the distributed system.

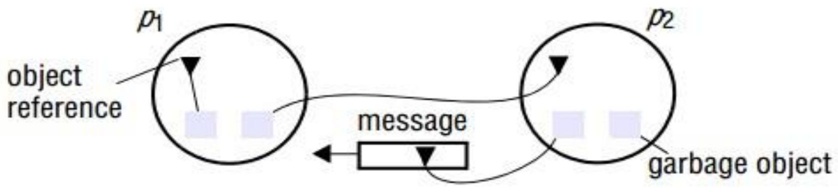


Fig 4. 7: Distributed Garbage Collection

– To check that an object is garbage, verify that there are no references to it anywhere.

– The process p1 has two objects that both have references :

- ✓ One has a reference within p1 itself
- ✓ p2 has a reference to the other.

– Process p2 has one garbage object, with no references to it anywhere in the system.

– It also has an object for which neither p1 nor p2 has a reference, but there is a reference to it in a message that is in transit between the processes.

Distributed deadlock detection:

– A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and where there is a cycle in the graph of this „waits-for" relationship.

– In the following figure, processes p1 and p2 are each waiting for a message from the other, so this system will never make progress.

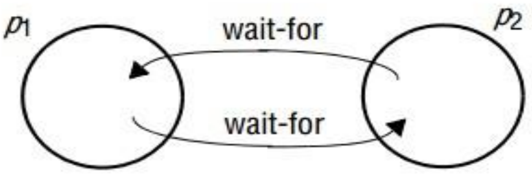


Fig 4.8: Distributed deadlock detection

Termination detection

- Detecting termination is a problem is to test whether each process has halted.
- Find whether a process is either active or passive.
- A passive process is not engaged in any activity of its own but is prepared to respond with a value requested by the other.
- The phenomena of termination and deadlock are similar:
  - A deadlock may affect only a subset of the processes in a system, whereas all processes must have terminated
  - The pro n a deadlock cycle: a deadloc ther action, for which another in any activity.

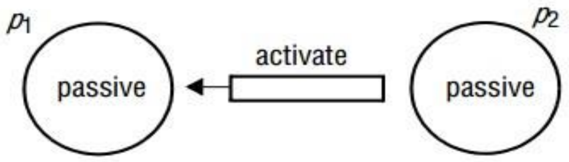


Fig 4.9: Termination of process

Distributed debugging:

Consider an application with process pi and a variable x, where i=1, 2, …, N. As the program executes the variables may change the value. The value must be within the range. The relationships between variables must be calculated only for the variables which executes at same time.

4.3.1 Global States and Consistency cuts (Distributed Snapshots)

Distributed Snapshot represents a state in which the distributed system might have been in. A snapshot of the system is a single configuration of the system.

A distributed snapshot should reflect a consistent state. A global state is consistent if it could have been observed by an external observer. For a successful Global State, all states must be consistent

- If we have recorded that a process P has received a message from a process Q, then we should have also recorded that process Q had actually send that message.
- Otherwise, a snapshot will contain the recording of messages that have been received but never sent.
- The reverse condition (Q has sent a message that P has not received) is allowed.

The notion of a global state can be graphically represented by what is called a **cut**. A cut represents the last event that has been recorded for each process.

The history of each process if given by:

$$history(p_i) = h_i = < e_i^0, e_i^1, e_i^2, ... >$$

Each event either is an internal action of the process. We denote by s_i^kt he state of process p_i immediately before the kth event occurs. The state $s_i$in the global state S correspondin g to the cut C is that of p_i immediately after the last event processed by p_i in the cut – ei^{ci}. The set of events ei^{ci} is called the frontier of the cut.

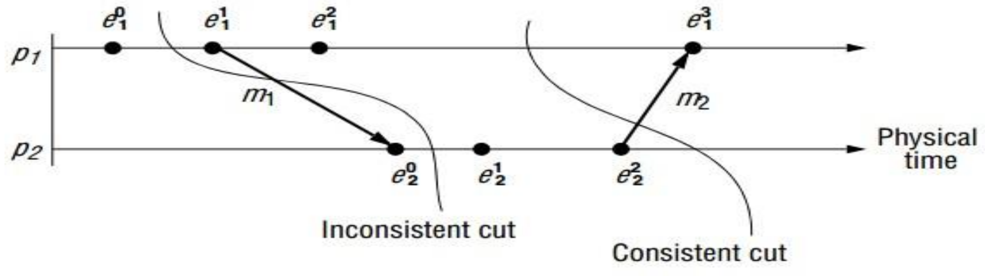


Fig 4.10: Types of cuts

In the above figure, the $<e_1^1, e_2^0>$and $<e_2^2, e_1^3>$ are the frontiers.The $<e_1^1, e_2^0>$ is inconsistent because at p2 it includes the receipt of the message m1 , but at p1 it does not include the sending of that message. The frontier $<e_2^2, e_1^3>$ is consistent since it includes both the sending and the receipt of message m1 and the sending but not the receipt of message m2.

A **consistent global state** is one that corresponds to a consistent cut. We may characterize the e execution of a distributed s yste m as a series of transitions between global states of the system:

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$$

A **run** is a total ordering of all the events in a global history that is consistent with each local history‟s ordering. A **linearization** or consistent run is an ordering of the events in a global history that is consistent with this happened-before relation o on H. A linearization is also a run.

4.3.2 Global state predicates, stability, safety and liveness

- Detecting a condition such as deadlock or termination amounts to evaluating a global state predicate.
- A **global state predicate** is a function that maps from the set of global states of processes in the system
- One of the useful characteristics of the predicates associated with the state of an object being garbage, of the system being deadlocked or the system being terminated is that they are all stable: once the system enters a state in which the predicate is True, it remains True in all future states reachable from that state.
- By contrast, when we monitor or debug an application we are often interested in non-stable predicates, such as that in our example of variables whose difference is supposed to be bounded.

4.3.3 Snapshot algorithm of Chandy and Lamport

Assumptions of the algorithm

- The algorithm to determine global states records process states locally and the states are collected by a designated server.
- No failures in c y.
- Unidirectional (
- There is always
- Global snapsho
- No process acti

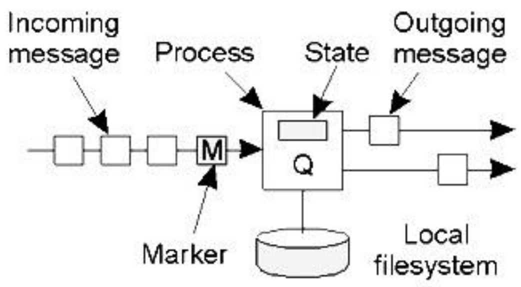


Fig 4.11: Process and organization in distributed systems

Algorithm:

- If a process Q receives the marker requesting a snapshot for the first time, it considers the process that sent the marker as its predecessor.

➢ When Q completes its part of the snapshot, it sends its predecessor a DONE message.

➢ By recursion, when the initiator of the distributed snapshot has received a DONE message from all its successors, it knows that the snapshot has been completely taken.

4.4 COORDINATION AND AGREEMENT

When more that one process run in an execution environment, they need to coordinate their actions. To solve problems in coordination avoid fixed master-salve relationship to avoid single points of failure for fixed master. Distributed mutual exclusion for resource sharing can also be used.

When a collection of process share resources, mutual exclusion is needed to prevent interference and ensure consistency. In this case, there is no need for shared variables or facilities are provided by single local kernel to solve it. This solution is based on message passing. The main issue to be addressed while implementing the above method is the failure.

4.4.1 Failure Assumptions and failure Detectors

✓ The message passing paradigm is based on reliable communication channels.

✓ But there can be process failures (i.e.) the whole process crashes.

✓ The failure can be detected when the object/code in a process that detects failures of other processes.

✓ There are two types of failure detectors: unreliable failure detector and reliable failure detector.

✓ The following are features of the unreliable failure detectors:

- unsuspected or suspected (i.e.) there can be no evidence of failure
- each process sends ``alive" message to everyone else
- not receiving ``alive" message after timeout
- This is present in most practical systems

✓ The following are features of the reliable failure detectors:

- unsuspected or failure
- They are present in synchronous system

4.5 DISTRIBUTED MUTUAL EXCLUSION

The mutual exclusion makes sure that concurrent process access shared resources or data in a serialized way. If a process, say Pi , is executing in its critical section, then no other processes can be executing in their critical sections.

Distributed mutual exclusion provide critical region in a distributed environment.

4.5.1 Algorithms for Distributed Mutual Exclusion

Consider there are N processes and the processes do not fail. The basic assumption is the message delivery system is reliable. So the methods for the critical region are:

- ✓ **enter() :** Enter the critical section block if necessary
- ✓ **resourceAccesses():**Access the shared resources
- ✓ **exit():** Leaves the critical section. Now other processes can enter critical section.

The following are the requirements for Mutual Exclusion (ME):

- ➢ **[ME1] safety**: only one process at a time
- ➢ **[ME2] liveness**: eventually enter or exit
- ➢ **[ME3] happened-before ordering:** ordering of enter() is the same as HB ordering

The second requirement implies freedom from both deadlock and starvation. Starvation involves fairness condition.

Performance Evaluation:

The following are the criteria for performance measures:

- ❖ **Bandwidth consumption**, which is proportional to the number of messages sent in each entry and exit operations.
- ❖ The **client delay** incurred by a process at each entry and exit operation.
- ❖ **Throughput of the system**: Rate at which the collection of processes as a whole can access the critical section.

4.5.2 Central Sever Algorithm

- ✓ This employs the simplest way to grant permission to enter the critical section by using a server.
- ✓ A process sends a request message to server and awaits a reply from it.

- ✓ If a reply constitutes a token signifying the permission to enter the critical section.
- ✓ If no other process has the token at the time of the request, then the server replied immediately with the token.
- ✓ If token is currently held by another process, then the server does not reply but queues the request.
- ✓ Client on exiting the critical section, a message is sent to server, giving it back the token.

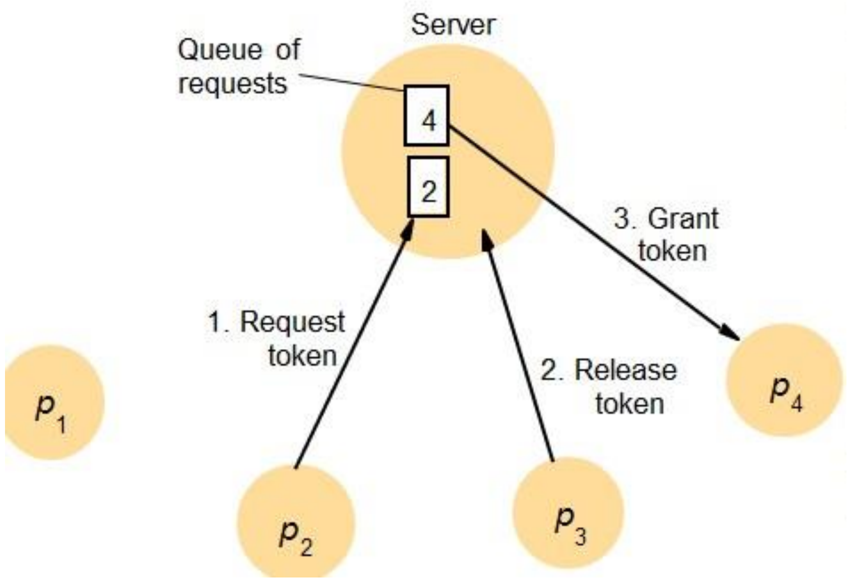


Fig 4. 12: Central Server Algorithm

The central server algorithm fulfils ME1 and ME2 but not ME3 (i.e.) safety and liveness is ensured but ordering is not satisfied. Also the performance of the algorithm is measured as follows:

- ➢ Bandwidth: This is measured by entering and exiting messages. Entering takes two messages (request followed by a grant) which are delayed by the round-trip time. Exiting takes one release message, and does not delay the exiting process.
- ➢ Throughput is measured by synchronization delay, round-trip of a release message and grant message.

4.5.3 Ring Based Algorithm

- ❖ This provides a simplest way to arrange mutual exclusion between N processes without requiring an additional process is arrange them in a logical ring.

- ❖ Each process pi has a communication channel to the next process in the ring as follows: p(i+1)/mod N.
- ❖ The unique token is in the form of a message passed from process to process in a single direction clockwise.
- ❖ If a process does not require to enter the CS when it receives the token, then it immediately forwards the token to its neighbor.
- ❖ A process requires the token waits until it receives it, but retains it.
- ❖ To exit the critical section, the process sends the token on to its neighbor.

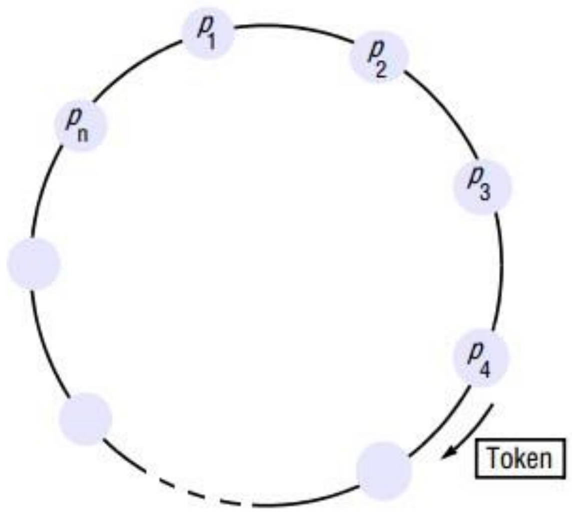


Fig 4.13: Ring based algorithm

This algorithm satisfies ME1 and ME2 but not ME (i.e.) safety and liveness are satisfied but not ordering. The performance measures include:

- ➢ **Bandwidth:** continuously consumes the bandwidth except when a process is inside the CS. Exit only requires one message.
- ➢ **Delay:** experienced by process is zero message(just received token) to N messages(just pass the token).
- ➢ **Throughput:** synchronization delay between one exit and next entry is anywhere from 1(next one) to N (self) message transmission.

4.5.4 Multicast Synchronisation

- ▪ This exploits mutual exclusion between N peer processes based upon multicast.
- ▪ Processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message.

- The condition under which a process replies to a request are designed to ensure ME1 ME2 and ME3 are met.
- Each process pi keeps a Lamport clock. Message requesting entry are of the form<T, pi>.
- Each process records its state of either RELEASE, WANTED or HELD in a variable state.
  - If a process requests entry and all other processes is RELEASED, then all processes reply immediately.
  - If some process is in state HELD, then that process will not reply until it is finished.
  - If some process is in state WANTED and has a smaller timestamp than the incoming request, it will queue the request until it is finished.
- If two or more processes request entry at the same time, then whichever bears the lowest timestamp will be the first to collect N-1 replies.

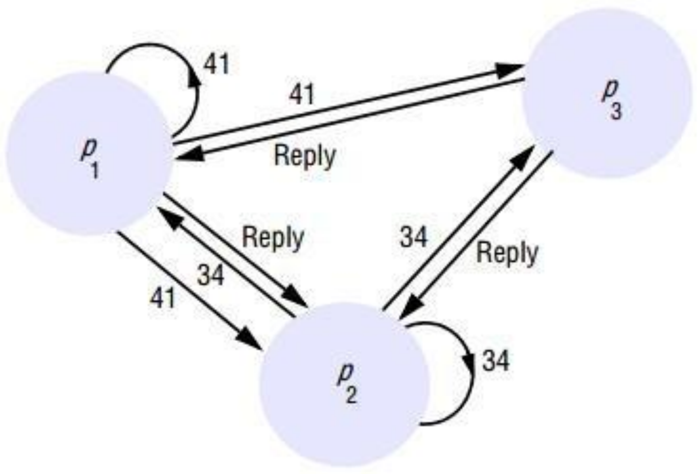


Fig 4.13: Multicast Synchronisation

- In the above figure, P1 and P2 request CS concurrently.
- The timestamp of P1 is 41 and for P2 is 34.
- When P3 receives their requests, it replies immediately.
- When P2 receives P1"s request, it finds its own request has the lower timestamp, and so does not reply, holding P1 request in queue.
- However, P1 will reply. P2 will enter CS. After P2 finishes, P2 reply P1 and P1 will enter CS.
- Granting entry takes 2(N-1) messages, N-1 to multicast request and N-1 replies.

Performance Evaluation:

- Bandwidth consumption is high.
- Client delay is again 1 round trip time
- Synchronization delay is one message transmission time.

4.5.5 Maekawa's Voting Algorithm

- In this algorithm, it is not necessary for all of its peers to grant access. Only need to obtain permission to enter from subsets of their peers, as long as the subsets used by any two processes overlap.
- Think of processes as voting for one another to enter the CS. A candidate process must collect sufficient votes to enter.
- Processes in the intersection of two sets of voters ensure the safety property ME1 by casting their votes for only one candidate.
- A voting set Vi associated with each process pi.
- There is at least one common member of any two voting sets, the size of all voting set are the same size to be fair.
- The optimal solution to minimizes K is K~sqrt(N) and M=K.
- The algorithm is summarized as follows:

  $V_i \subseteq \{P_1, P_2, \dots P_N\}$

  Such that for all i, j = 1, 2, … N

  $Pi \in V_i$

  $V_i \cap Vj \neq \phi$

  $|V_i| = K$

  Each process is contained in M of the voting set V_i

Maekawa's Voting Algorithm

```
On initialization
        state := RELEASED;
        voted := FALSE;
For p_i to enter the critical section
        state := WANTED;
        Multicast request to all processes in $V_i$;
        Wait until (number of replies received = K);
        state := HELD;
On receiptof a request from$p_i$ at $p_j$
        if (state = HELD orvoted = TRUE)
        then
            queue request from $p_i$ without replying;
        else
            send reply to $p_i$;
            voted := TRUE;
        end if
For $p_i$ to exit the critical section
        state := RELEASED;
        Multicast release to all processes in $V_i$;
On receiptof a release from$p_i$ at $p_j$
        if (queue of requests is non-empty)
        then
            remove head of queue – from $p_k$, say;
            send reply to $p_k$;
            voted := TRUE;
        else
            voted := FALSE;
        end if
```

– The ME1 is met.

– If two processes can enter CS at the same time, the processes in the intersection of two voting sets would have to vote for both.

– The algorithm will only allow a process to make at most one vote between successive receipts of a release message.

– This algorithm is deadlock prone.

– If three processes concurrently request entry to the CS, then it is possible for p1 to reply to itself and hold off p2; for p2 rely to itself and hold off p3; for p3 to reply to itself and hold off p1.

– Each process has received one out of two replies, and none can proceed.

– If process queues outstanding request in happen-before order, ME3 can be satisfied and will be deadlock free.

Performance Evaluation:

- Bandwidth utilization is 2sqrt(N) messages per entry to CS and sqrt(N) per exit.
- Client delay is the same as Ricart and Agrawala‟salgorithm, one round-trip time.
- Synchronization delay is one round-trip time which is worse than R&A

4.5.6 Fault Tolerance

❖ The reactions of the algorithms when messages are lost or when a process crashes is fault tolerance.

❖ None of the algorithm that we have described would tolerate the loss of messages if the channels were unreliable.

❖ The ring-based algorithm cannot tolerate any single process crash failure.

❖ Maekawa‟salgorithm can tolerate some process crash failures: if a crashed process is not in a voting set that is required.

❖ The central server algorithm can tolerate the crash failure of a client process that neither holds nor has requested the token.

4.6 ELECTIONS

An algorithm for choosing a unique process to play a particular role is called an election algorithm.

The requirements for election algorithms:

- **E1(safety):** a participant pi has$elected_i$ = ⊥ or elected P where ⊥ indicates a value that is not defined.
- **E2(liveness):** All processes Pi participate in election process and eventually set$elected_i$!=⊥ or crash.

4.6.1 Ring based Election Algorithm

- All the processes arranged in a logical ring.
- Each process has a communication channel to the next process.
- All messages are sent clockwise around the ring.
- Assume that no failures occur, and system is asynchronous.
- The ultimate goal is to elect a single process coordinator which has the largest identifier.

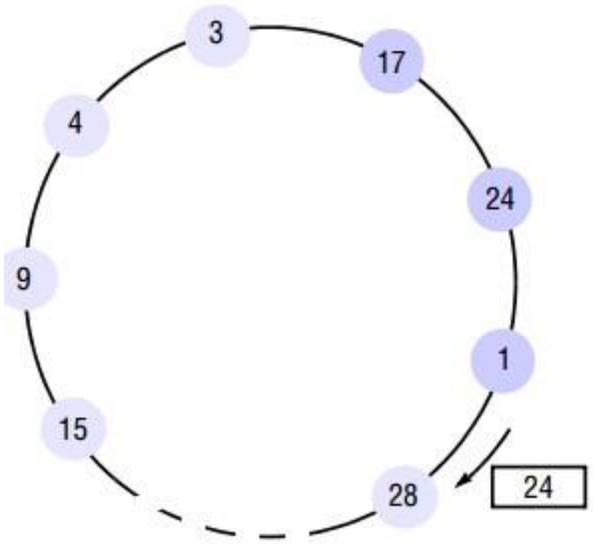


Fig 4.14: Election process using Ring based election algorithm

Steps in election process:

1. Initially, every process is marked as non-participant. Any process can begin an election.
2. The starting processes marks itself as participant and place its identifier in a message to its neighbour.
3. A process receives a message and compares it with its own. If the arrived identifier is larger, it passes on the message.
4. If arrived identifier is smaller and receiver is not a participant, substitute its own identifier in the message and forward if. It does not forward the message if it is already a participant.
5. On forwarding of any case, the process marks itself as a participant.

6. If the received identifier is that of the receiver itself, then this process" s identifier must be the greatest, and it becomes the coordinator.
7. The coordinator marks itself as non-participant set elected_i and sends an elected message to its neighbour enclosing its ID.
8. When a process receives elected message, marks itself as a non-participant, sets its variable elected_i and forwards the message.

The election was started by process 17. The highest process identifier encountered so far is 24.

Requirements:

- E1 is met. All identifiers are compared, since a process must receive its own ID back before sending an elected message.
- E2 is also met due to the guaranteed traversals of the ring.
- Tolerates no failure makes ring algorithm of limited practical use.

Performance Evaluation

- If only a single process starts an election, the worst-performance case is then the anti-clockwise neighbour has the highest identifier. A total of N-1 messages is used to reach this neighbour. Then further N messages are required to announce its election. The elected message is sent N times. Making 3N-1 messages in all.
- Turnaround time is also 3N-1 sequential message transmission time.

4.6.2 Bully Algorithm

- This algorithm allows process to crash during an election, although it assumes the message delivery between processes is reliable.
- Assume that the system is synchronous to use timeouts to detect a process failure and each process knows which processes have higher identifiers and that it can communicate with all such processes.
- In this algorithm, there are three types of messages:
    - **Election message:** This is sent to announce an election message. A process begins an election when it notices, through timeouts, that the coordinator has failed. T=2Ttrans+Tprocess From the time of sending
    - **Answermessage:** This is sent in response to an election message.
    - **Coordinatormessage**: This is sent to announce the identity of the elected process.

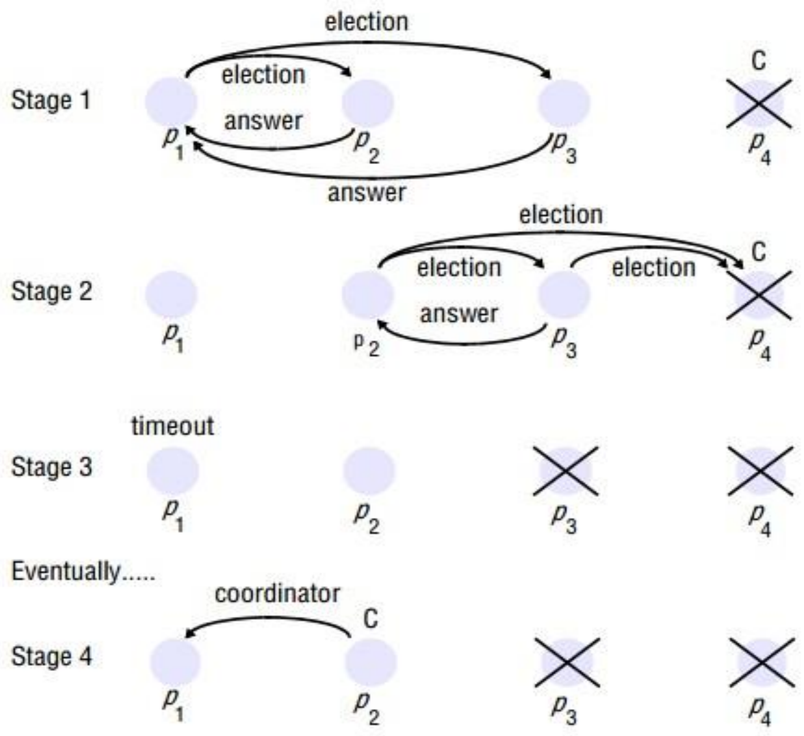


Fig 4.15: Stages in Bully Algorithm

Election process:

- The process begins a election by sending an election message to these processes that have a higher ID and awaits an answer in response.
- If none arrives within time T, the process considers itself the coordinator and sends coordinator message to all processes with lower identifiers.
- Otherwise, it waits a further time T'' for coordinator message to arrive. If none, begins another election.
- If a process receives a coordinator message, it sets its variable elected_i to be the coordinator ID.
- If a process receives an election message, it send back an answer message and begins another election unless it has begun one already.

Requirements:

∗ E1 may be broken if timeout is not accurate or replacement.

∗ Suppose P3 crashes and replaced by another process. P2 set P3 as coordinator and P1 set P2 as coordinator.

∗ E2 is clearly met by the assumption of reliable transmission.

Performance Evaluation

- Best case the process with the second highest ID notices the coordinator‟s failure. Then it can immediately elect itself and send N-2 coordinator messages.
- The bully algorithm requires O(N^2) messages in the worst case - that is, when the process with the least ID first detects the coordinator‟s failure. For then N-1 processes altogether begin election, each sending messages to processes with higher ID.

4.7 TRANSACTION AND CONCURRENCY CONTROL

Fundamentals of transactions and concurrency control

- ✓ A transaction is generally atomic.
- ✓ The state of the transaction being done is not visible. If it is not done completely, any changes it made will be undone. This is known as rollback.
- ✓ Concurrency is needed when multiple users want to access the same data at the same time.
- ✓ Concurrency control (CC) ensures that correct results for parallel operations are generated.
- ✓ CC provides rules, methods, design methodologies and theories to maintain the consistency of components operating simultaneously while interacting with the same object.

A Transaction defines a sequence of server operations that is guaranteed to be atomic in the presence of multiple clients and server crash.

All concurrency control protocols are based on serial equivalence and are derived from rules of conflicting operations:

- Locks used to order transactions that access the same object according to request order.
- Optimistic concurrency control allows transactions to proceed until they are ready to commit, whereupon a check is made to see any conflicting operation on objects.
- Timestamp ordering uses timestamps to order transactions that access the same object according to their starting time.

Synchronisation without transactions

- Synchronization without transactions are done with multiple threads.
- The use of multiple threads is beneficial to the performance. Multiple threads may access the same objects.

- In Java, Synchronized keyword can be applied to method so only one thread at a time can access an object.
- If one thread invokes a synchronized method on an object, then that object is locked, another thread that invokes one of the synchronized method will be blocked.

Enhancing Client Cooperation by Signaling

- The clients may use a server as a means of sharing some resources.
- In some applications, threads need to communicate and coordinate their actions by signaling.

Failure modes in transactions

- The following are the failure modes: Writes to permanent storage may fail, either by writing nothing or by writing wrong values.
- Servers may crash occasionally.
- There may be an arbitrary delay before a message arrives.

4.8 TRANSACTIONS

- Transaction originally from database management systems.
- Clients require a sequence of separate requests to a server to be atomic in the sense that:
  - They are free from interference by operations being performed on behalf of other concurrent clients.
  - Either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes**.**

Atomicity

– All or nothing: a transaction either completes successfully, and effects of all of its operations are recorded in the object, or it has no effect at all.

  - **Failure atomicity**: effects are atomic even when server crashes
  - **Durability:** after a transaction has completed successfully, all its effects are saved in permanent storage for recover later.

Isolation

Each transaction must be performed without interference from other transactions. The intermediate effects of a transaction must not be visible to other transactions.

4.8.1 Concurrency Control

Serial Equivalence

- ✓ If these transactions are done one at a time in some order, then the final result will be correct.
- ✓ If we do not want to sacrifice the concurrency, an interleaving of the operations of transactions may lead to the same effect as if the transactions had been performed one at a time in some order.
- ✓ We say it is a serially equivalent interleaving.
- ✓ The use of serial equivalence is a criterion for correct concurrent execution to prevent lost updates and inconsistent retrievals.

Conflicting Operations

- ✓ When we say a pair of operations conflicts we mean that their combined effect depends on the order in which they are executed. E.g. read and write
- ✓ There are three ways to ensure serializability:
  - ➢ Locking
  - ➢ Timestamp ordering
  - ➢ Optimistic concurrency control

Process 1	**Process 2**	**Conflict**	**Reason**
Read	Read	No	-
Read	Write	Yes	The result of these operations dependon their order of execution.
Write	Write	Yes	The result of these operations depend on their order of execution.

4.9 NESTED TRANSACTIONS

- ❖ Nested transactions extend the 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 transaction are called **sub-transactions.**

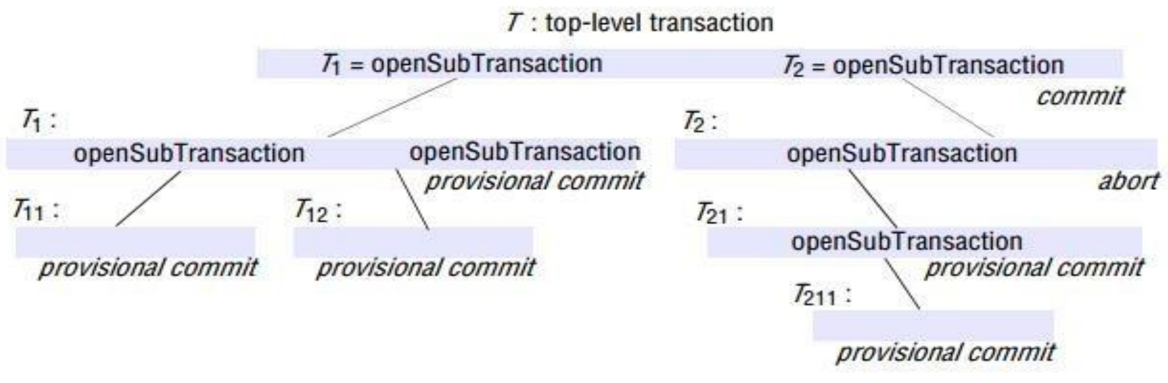


Fig 4.16: Nested Transactions

- ❖ T is a top-level transaction that starts a pair of sub-transactions, T1 and T2.
- ❖ The sub-transaction T1 starts its own pair of sub-transactions, T11 and T22.
- ❖ Also, sub-transaction T2 starts its own sub-transaction, T21, which starts another sub-transaction, T211.
- ❖ A sub-transaction appears to be atomic to its parent with respect to transaction failures
  and to concurrent access.
- ❖ Sub-transactions at the same level can run concurrently, but their access to common objects is serialized.
- ❖ Each sub-transaction can fail independently of its parent and of the other sub-transactions.
- ❖ When a sub-transaction aborts, the parent transaction can choose an alternative sub-transaction to complete its task.
- ❖ If one or more of the sub-transactions fails, the parent transaction could record the fact and then commit, with the result that all the successful child transactions commit.
- ❖ It could then start another transaction to attempt to redeliver the messages that were not sent the first time.

Advantages of Nested Transactions

- ▪ Sub- transactions at same level can run concurrently.
- ▪ Sub- transactions can commit or abort independently.

The rules for committing of nested transactions are:

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

4.10 LOCKS

❖ A simple example of a serializing mechanism is the use of exclusive locks.

❖ Server can lock any object that is about to be used by a client.

❖ If another client wants to access the same object, it has to wait until the object is unlocked in the end.

❖ There are two types of locks: Read and Write.

❖ Read locks are shared locks (i.e.) they do not bring about conflict.

❖ Write locks are exclusive locks, since the order in which write is done may give rise to a conflict.

Lock	**Read conflict**	**Write conflict**
None	No	No
Read	No	Yes
Write	Yes	Yes

❖ An object can be read and write.

❖ From the compatibility table, we know pairs of read operations from different transactions do not conflict.

❖ So a simple exclusive lock used for both read and write reduces concurrency more than necessary. (Many readers/Single writer).

The following rules could be framed:

1. If T has already performed a read operation, then a concurrent transaction U must not write until T commits or aborts.
2. If T already performed a write operation, then concurrent U must not read or write until T commits or aborts.

Lock implementation

- The granting of locks will be implemented by a separate object in the server called the lock manager.
- The lock manager holds a set of locks, for example in a hash table.
- Each lock is an instance of the class Lock and is associated with a particular object.
- Each instance of Lock maintains the following information in its instance variables:
  - ∗ the identifier of the locked object
  - ∗ the transaction identifiers of the transactions that currently hold the lock
  - ∗ a lock type

Two phase Locking

The basic two-phase locking (2PL) protocol states:

- ✓ A transaction T must hold a lock on an item x in the appropriate mode before T accesses x.
- ✓ If a conflicting lock on x is being held by another transaction, T waits.
- ✓ Once T releases a lock, it cannot obtain any other lock subsequently.

In two phase locking, a transaction is divided into two phases:

- ✓ A growing phase (obtaining locks)
- ✓ A shrinking phase (releasing locks)

  This lock ensures conflict serializability.**Lock-point** is the point where the transaction obtains all the locks.With 2PL, a schedule is conflict equivalent to aa serial schedule ordered by the lock-point of the transactions.

Strict Two Phase Locking

- ➢ Because transaction may abort, strict execution are needed to prevent dirty reads and premature writes, which are caused by read or write to same object accessed by another earlier unsuccessful transaction that already performed an write operation.

➢ So to prevent this problem, a transaction that needs to read or write an object must be delayed until other transactions that wrote the same object have committed or aborted.

➢ The rule in Strict Two Phase Locking:

↑ Any locks applied during the progress of a transaction are helduntil the transaction commits or aborts.

$$D = \begin{bmatrix} T1 & T2 \\ S(A) & \\ R(A) & \\ & S(A) \\ & R(A) \\ & X(B) \\ & R(B) \\ & W(B) \\ & Commit \\ X(C) & \\ R(C) & \\ W(C) & \\ Commit & \end{bmatrix}$$

Fig 4. 16: Strict two phase locking between T1 and T2

1. When an operation accesses an object within a transaction:

    (a) If the object is not already locked, it is locked and the operation proceeds.

    (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.

    (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.

    (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)

2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction

A transaction with a read lock that is shared by other transactions cannot promote its read lock to a write lock, because write lock will conflict with other read locks.

4.10.1 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 – there is an edge from node T to node U when transaction T is waiting for transaction U to release a lock.

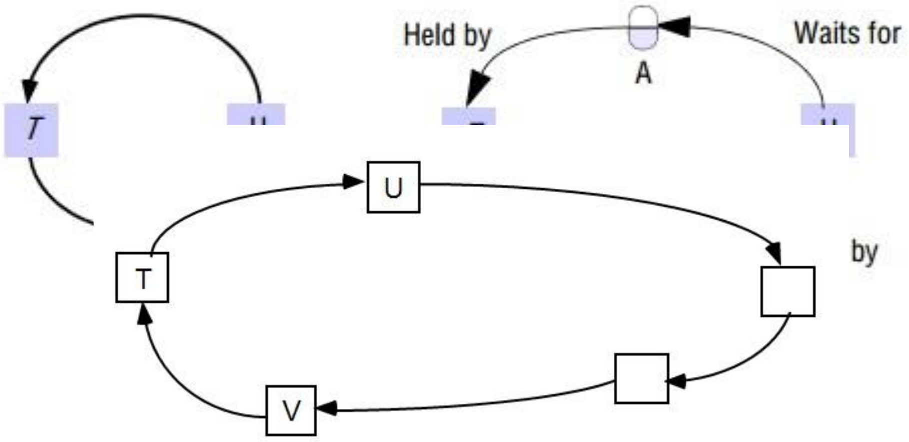


Fig 4.18: A cycle in wait for graph

Deadlock prevention:

- Simple way is to lock all of the objects used by a transaction when it starts.
- It should be done as an atomic action to prevent deadlock. a. inefficient, say lock an object you only need for short period of time. b.
- Hard to predict what objects a transaction will require.
- Judge if system can remain in a Safe state by satisfying a certain resource request. Banker‟s algorithm.
- Order the objects in certain order.
- Acquiring the locks need to follow this certain order.

Safe State

- ❖ System is in safe state if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes is the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with j <i.
- ❖ If a system is in safe state, then there are no deadlocks.
- ❖ If a system is in unsafe state, then there is possibility of deadlock.
- ❖ Avoidance is to ensure that a system will never enter an unsafe state.

Deadlock Detection

- Deadlock 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.
  - o If lock manager blocks a request, an edge can be added. Cycle should be checked each time a new edge is added.
  - o One transaction will be selected to abort in case of cycle. Age of transaction and number of cycles involved when selecting a victim
- Timeouts is commonly used to resolve deadlock. Each lock is given a limited period in which it is invulnerable. After this time, a lock becomes vulnerable.
  - – If no other transaction is competing for the object, vulnerable object remained locked. However, if another transaction is waiting, the lock is broken.

Disadvantages:

- Transaction aborted simply due to timeout and waiting transaction even if there is no deadlock.
- Hard to set the timeout time

4.11 OPTIMISTIC COCURRENCY CONTROL

The locking and serialization of transaction has numerous disadvantages.:

- ✓ Lock maintenance represents an overhead that is not present in systems that do not support concurrent access to shared data. Locking sometimes are only needed for some cases with low probabilities.
- ✓ The use of lock can result in deadlock. Deadlock prevention reduces concurrency severely. The use of timeout and deadlock detection is not ideal for interactive programs.
- ✓ To avoid cascading aborts, locks cannot be released until the end of the transaction. This may reduce the potential for concurrency.

- It is based on observation that, in most applications, the likelihood of two clients‟ transactions accessing the same object is low.
- Transactions are allowed to proceed as though there were no possibility of conflict with other transactions until the client completes its task and issues a closeTransaction request.
- When conflict arises, some transaction is generally aborted and will need to be restarted by the client.
- Each transaction has the following phases:
  - **Working phase:**
    - Each transaction has a tentative version of each of the objects that it updates.
    - This is a copy of the most recently committed version of the object.
    - The tentative version allows the transaction to abort with no effect on the object, either during the working phase or if it fails validation due to other conflicting transaction.
    - Several different tentative values of the same object may coexist.
    - In addition, two records are kept of the objects accessed within a transaction, a read set and a write set containing all objects either read or written by this transaction.
    - Read are performed on committed version (no dirty read can occur) and write record the new values of the object as tentative values which are invisible to other transactions.
  - **Validation phase:**
    - When closeTransaction request is received, the transaction is validated to establish whether or not its operations on objects conflict with operations of other transaction on the same objects.
    - If successful, then the transaction can commit.
    - If fails, then either the current transaction or those with which it conflicts will need to be aborted.
  - **Update phase:**
    - If a transaction is validated, all of the changes recorded in its tentative versions are made permanent.
    - Read-only transaction can commit immediately after passing validation.
    - Write transactions are ready to commit once the tentative versions have been recorded in permanent storage.

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- that is, any transactions that had not yet committed at the time this transaction started.
- Each transaction is assigned a number when it enters the validation phase (when the client issues a closeTransaction).
- Such number defines its position in time.
- A transaction always finishes its working phase after all transactions with lower numbers.
- That is, a transaction with the number Ti always precedes a transaction with number Tj if i < j.
- The validation test on transaction Tv is based on conflicts between operations in pairs of transaction Ti and Tv, for a transaction Tv to be serializable with respect to an overlapping transaction Ti, their operations must conform to the below rules.

Rule No	**T_v**	**T_1**	**Rule**
1	Write	Read	T_i must not read objects written by T_v
2	Read	Write	Tvmust not read objects written by T_i
3	Write	Write	T_i must not read objects written by $T_v$and $T_v$must not read objects written by Ti.

Types of Validation

* **Backward Validation:** checks the transaction undergoing validation with other preceding overlapping transactions- those that entered the validation phase before it.
* **Forward Validation:** checks the transaction undergoing validation with other later transactions, which are still active

Backward Validation	**Forward Validation**

Backward validation of transaction T_v
boolean valid = true;
for (intT_i = startTn+1; T_i<= finishTn; T_i++){
if (read set of T_v intersects write set of T_i) valid = false;
}
	Forward validation of transaction T_v
	boolean valid = true;
	for (intT_{id} = active1; T_{id}<= activeN; T_{id}++){
	if (write set of T_v intersects read set of T_{id}) valid = false;
	}

Starvation

- When a transaction is aborted, it will normally be restarted by the client program.
- There is no guarantee that a particular transaction will ever pass the validation checks, for it may come into conflict with other transactions for the use of objects each time it is restarted.

  The prevention of a transaction ever being able to commit is called starvation.

4.12 TIMESTAMP ORDERING

- Timestamps may be assigned from the server‟s clock or a counter that is incremented whenever a timestamp value is issued.
- Each object has a write timestamp and a set of tentative versions, each of which has a write timestamp associated with it; and a set of read timestamps.
- The write timestamps of the committed object is earlier than that of any of its tentative versions, and the set of read timestamps can be represented by its maximum member.
- Whenever a transaction‟s write operation on an object is accepted, the server creates a new tentative version of the object with write timestamp set to the transaction timestamp. Whenever a read operation is accepted, the timestamp of the transaction is added to its set of read timestamps.
- When a transaction is committed, the values of the tentative version become the values of the object, and the timestamps of the tentative version become the write timestamp of the corresponding object.
- Each request from a transaction is checked to see whether it conforms to the operation conflict rules.
- Conflict may occur when previous done operation from other transaction Ti is later than current transaction Tc. It means the request is submitted too late.

Rule	**T_c**	**T_i**	**Description**
1	Write	Read	T_c must not write an object that has been read by any T_i where $T_i > T_c$this requires that $T_c >=$ the maximum read timestamp of the object.
2	Write	Write	T_c must not write an object that has been written by any T_i where $T_i > T_c$this requires that $T_c >$ the write timestamp of the committed object.

3	Read	Write	T_c must not read an object that has been written by any T_i where $T_i > T_c$ this requires that $T_c >$ the write timestamp of the committed object.

Timestamp ordering by read rule:

```
if ( $T_c >$ write timestamp on committed version of D) {
    let $D_{selected}$ be the version of D with the maximum write timestamp $\leq T_c$
    if ($D_{selected}$ is committed)
            perform read operation on the version $D_{selected}$
    else
            Wait until the transaction that made version $D_{selected}$ commits or aborts
            then reapply the read rule
} else
    Abort transaction $T_c$
```

Timestamp ordering by write rule

```
if ($T_c \geq$ maximum read timestamp on D&&
    $T_c >$ write timestamp on committed version of D)
            perform write operation on tentative version of D with write timestamp $T_c$
else /* write is too late */
    Abort transaction $T_c$
```

Multi-version timestamp ordering

- ✓ In multi-version timestamp ordering, a list of old committed versions as well as tentative versions is kept for each object.
- ✓ This list represents the history of the values of the object.
- ✓ The benefit of using g multiple versions is that read operations that arrive too late need not be rejected.
- ✓ Each version has a read timestamp recording the largest timestamp of any transaction that has read from it in addition to a write timestamp.
- ✓ As before, whenever a write operation is accepted, it is directed to a tentative version with the write timestamp of the transaction.

- ✓ Whenever a read operation is carried out, it is directed to the version with the largest write timestamp less than the transaction timestamp.
- ✓ If the transaction timestamp is larger than the read timestamp of the version being used, the read timestamp of the version is set to the transaction timestamp.
- ✓ When a read arrives late, it can be allowed to read from an old committed version, so there is no need to abort late read operations.
- ✓ In multi-version timestamp ordering, read operations are always permitted, although they may have to wait for earlier transactions to complete, which ensures that executions are recoverable.
- ✓ There is no conflict between write operations of different transactions, because each transaction writes its own committed version of the objects it accesses.
- ✓ The rule in this ordering is

Tcmust not write objects that have been read by any Ti where Ti >Tc

Multi-version Timestamp ordering write rule

```
if (read timestamp of DmaxEarlier” Tc)
perform write operation on a tentative version of D with write timestamp Tc
else abort transaction Tc.
```

4.13 ATOMIC COMMIT PROTOCOL

- ➢ The atomicity property of transactions demands when a distributed transaction comes to an end, either all of its operations are carried out or none of them.
- ➢ In the case of a distributed transaction, the client has requested operations at more than one server.
- ➢ A transaction comes to an end when the client requests that it 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 of the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they have carried it out. This is **one phase atomic commit protocol.**

Two Phase Commit Protocol

- The **two-phase commit protocol** is designed to allow any participant to abort its part of a transaction.
- Due to the requirement for atomicity, if one part of a transaction is aborted, then the whole transaction must be aborted.
- In the first phase of the protocol, each participant votes for the transaction to be committed or aborted.
- Once a participant has voted to commit a transaction, it is not allowed to abort it.
- In the second phase of the protocol, every participant in the transaction carries out the joint decision.
- If any one participant votes to abort, then the decision must be to abort the transaction.
- If all the participants vote to commit, then the decision is to commit the transaction.
- The following are the operations in two phase commit protocol:
  - **canCommit?(trans)→Yes / No**: Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.
  - **doCommit(trans):**Call from coordinator to participant to tell participant to commit its part of a transaction.
  - **doAbort(trans):** Call from coordinator to participant to tell participant to abort its part of a transaction.
  - **haveCommitted(trans, participant):** Call from participant to coordinator to confirm that it has committed the transaction.
  - **getDecision(trans) →Yes / No:** Call from participant to coordinator to ask for the decision on a transaction when it has voted Yes but has still had no reply after some delay. Used to recover from server crash or delayed messages

Two phase commit protocol

Phase 1 (voting phase):

1.The coordinator sends a canCommit? request to each of the participants in the transaction.

2. When a participant receives a canCommit? request it replies with its vote (Yes or No) to the coordinator. Before voting Yes, it prepares to commit by saving objects in permanent storage. If the vote is No, the participant aborts immediately.

Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).

(a)If there are no failures and all the votes are Yes, the coordinator decides to commit the transaction and sends a doCommitrequest to each of the participants.

(b)Otherwise, the coordinator decides to abort the transaction and sends doAbortrequests to all participants that voted Yes.

4. Participants that voted Yes are waiting for a doCommitor doAbortrequest from the coordinator. When a participant receives one of these messages it acts accordingly and, in the case of commit, makes a have Committedcall as confirmation to the coordinator.

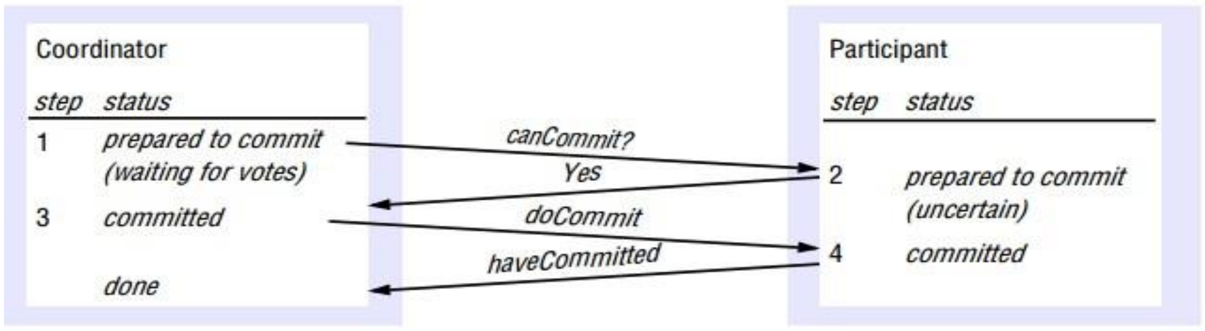


Fig 4.16: Two phase commit protocol

Two phase commit for nested transactions

- 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.
- The following are the operations are allowed:
  - openSubTransaction(trans) →subTrans: Opens a new subtransaction whose parent is trans and returns a unique subtransaction identifier.
  - getStatus(trans)→committed, aborted, provisional: Asks the coordinator to report on the status of the transaction trans. Returns values representing one of the following: committed, aborted or provisional.
- When a sub-transaction completes, it makes an independent decision either to commit provisionally or to abort.
- A coordinator for a sub-transaction will provide an operation to open a sub-transaction, together with an operation enabling that coordinator to enquire whether its parent has yet committed or aborted.

Hierarchic Two phase commit protocol

- ∗ In this approach, the two-phase commit protocol becomes a multi-level nested protocol.
- ∗ The coordinator of the top-level transaction communicates with the coordinators of the sub-transactions for which it is the immediate parent.
- ∗ It sends canCommit? messages to each of the latter, which in turn pass them on to the coordinators of their child transactions.
- ∗ canCommit?(trans, subTrans) →Yes / No: Call from coordinator to coordinator of child sub-transaction to ask whether it can commit a sub-transaction subTrans. The first argument, trans, is the transaction identifier of the top-level transaction. Participant replies with its vote, Yes / No.

Flat Two phase commit protocol

- ✓ In this approach, the coordinator of the top-level transaction sends canCommit? messages to the coordinators of all of the sub-transactions in the provisional commit list.
- ✓ During the commit protocol, the participants refer to the transaction by its top-level TID.
- ✓ Each participant looks in its transaction list for any transaction or sub-transaction matching that TID.
- ✓ A participant can commit descendants of the top-level transaction unless they have aborted ancestors.
- ✓ When a participant receives a canCommit? request, it does the following:
- ✓ If the participant has any provisionally committed transactions that are descendants of the top-level transaction, trans, it:
  - – checks that they do not have aborted ancestors in the abortList, then prepares
    to commit (by recording the transaction and its objects in permanent storage);
  - – aborts those with aborted ancestors;
  - – sends a Yes vote to the coordinator.
- ✓ If the participant does not have a provisionally committed descendent of the top level transaction, it must have failed since it performed the sub- transaction and it sends a No vote to the coordinator

4.14 DISTRIBUTED DEADLOCKS

- A cycle in the global wait-for graph (but not in any single local one) represents a distributed deadlock.
- A deadlock that is detected but is not really a deadlock is called a **phantom deadlock.**
- Two-phase locking prevents phantom deadlocks; autonomous aborts may cause phantom deadlocks.
- Permanent blocking of a set of processes that either compete for system resources or communicate with each other is deadlock.
- No node has complete and up-to-date knowledge of the entire distributed system. This is the cause of deadlocks.

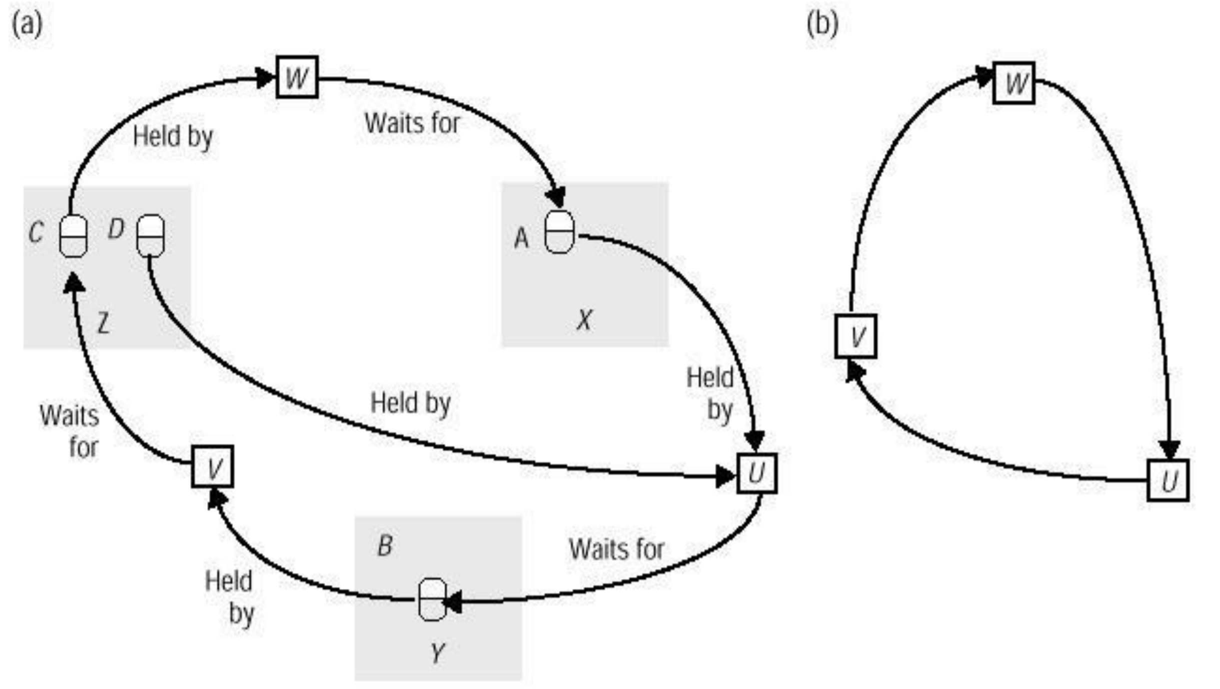
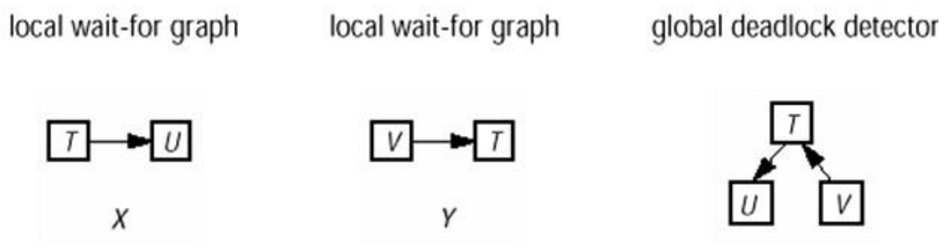


Fig 4.17: Distributed deadlocks and wait for graphs

Types of distributed deadlock

- **Resource deadlock :**Set of deadlocked processes, where each process waits for a resource held by another process (e.g., data object in a database, I/O resource on a server)
- **Communication deadlocks:** Set of deadlocked processes, where each process waits to receive messages (communication) from other processes in the set.

Local and Global Wait for Graphs



Edge Chasing

- ❖ When a server notes that a transaction T starts waiting for another transaction U, which is waiting to access a data item at another server, it sends a probe containing ⟨T→U⟩ to the server of the data item at which transaction U is blocked.
- ❖ **Detection:** receive probes and decide whether deadlock has occurred and whether to forward the probes.
- ❖ When a server receives a probe ⟨T→U⟩ and finds the transaction that U is waiting for, say V, is waiting for another data item elsewhere, a probe ⟨T→U→V⟩ is forwarded.
- ❖ **Detection:** receive probes and decide whether deadlock has occurred and whether to forward the probes.

  When a server receives a probe ⟨T→U⟩ and finds the transaction that U is waiting for, say V, is waiting for another data item elsewhere, a probe ⟨T→U→V⟩ is forwarded.
- ❖ **Resolution:** select a transaction in the cycle to abort

Transaction priorities

- ✓ Every transaction involved in a deadlock cycle can cause deadlock detection to be initiated.
- ✓ The effect of several transactions in a cycle initiating deadlock detection is that detection may happen at several different servers in the cycle, with the result that more than one transaction in the cycle is aborted.

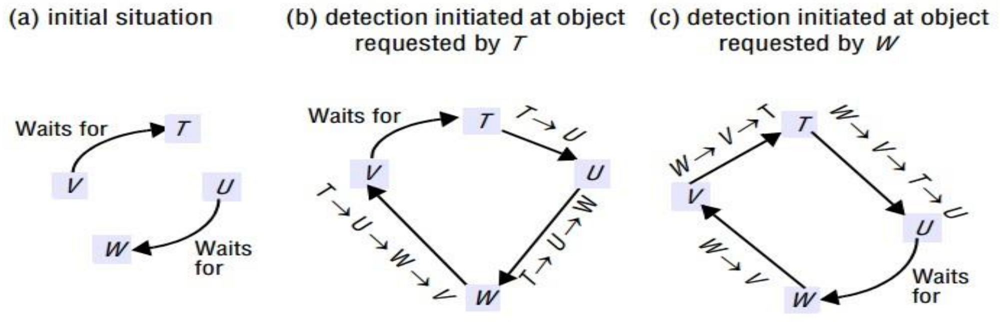


Fig 4.17: Initiated probes

- ✓ Consider transactions T, U, V and W, where U is waiting for Wand V is waiting for T.
- ✓ At about the same time, T requests the object held by U and W requests the object held by V.
- ✓ Two separate probes, <T →U > and <W →V >, are initiated by the servers of these objects and are circulated until deadlocks are detected by each of the servers.
- ✓ The cycle is <T →U→W →V →T>and, the cycle is <W →V →T →U →W >.
- ✓ In order to ensure that only one transaction in a cycle is aborted, transactions are given priorities in such a way that all transactions are totally ordered.
- ✓ In order to ensure that only one transaction in a cycle is aborted, transactions are given priorities in such a way that all transactions are totally ordered.
- ✓ The problem is that the order in which transactions start waiting can determine whether or not a deadlock will be detected.
- ✓ The above pitfall can be avoided by using as scheme in which coordinators save copies of all the probes received on behalf of each transaction in a **probe queue.**
- ✓ When a transaction starts waiting for an object, it forwards the probes in its queue to the server of the object, which propagates the probes on down hill routes.

4.14 REPLICATION

Replication in distributed systems enhances the performance, availability and fault tolerance. The general requirement includes:

- Replication transparency
- Consistency

4.14.1 System Model

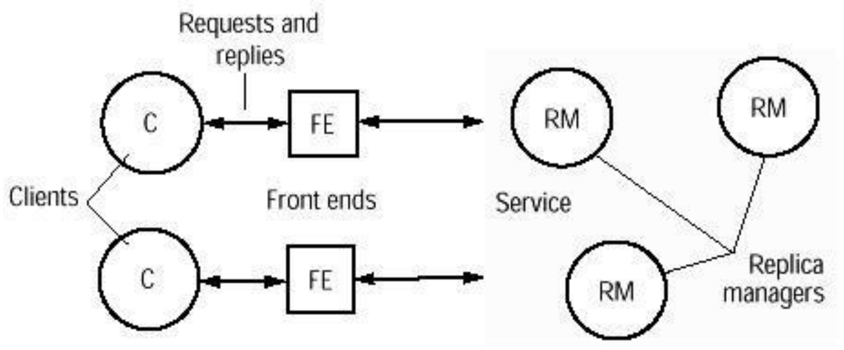


Fig 4.18: Architecture for replication management

- ➢ The data in a system consist of a collection of items called objects.

- An object could be a file, say, or 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 behavior that are tied to some degree of consistency by the system‟s operation.
- The system models include replica managers which are components that contain the replicas on a given computer and perform operations upon them directly.
- Each client‟s requests are first handled by a component called a front end.
- The role of the front end is to communicate by message passing with one or more of the replica managers, rather than forcing the client to do this itself explicitly.
- It is the vehicle for making replication transparent.
- A front end may be implemented in the client‟s address space, or it may be a separate process.

Phases in request processing:

- **Issuance of request:** The front end issues the request to one or more replica managers:
  – either the front end communicates with a single replica manager, which in turn communicates with other replica managers

  – or the front end multicasts the request to the replica managers.
- **Coordination:**
  - The replica managers coordinate in preparation for executing therequest consistently.
  - They agree, if necessary at this stage, on whether the request is to be applied.
  - They also decide on the ordering of this request relative to others.
  - The types of ordering includes: FIFO ordering, casual ordering and total ordering.
- **Execution:** The replica managers execute the request – perhaps tentatively: that is, in such a way that they can undo its effects later.
- **Agreement:** The replica managers reach consensus on the effect of the request – if any – that will be committed.
- **Response:** One or more replica managers respond to the front end.

4.15 CASE STUDY: CODA

Coda is a distributed file system. Coda has been developed at Carnegie Mellon University (CMU) in the 1990s, and is now integrated with a number of popular UNIX-based operating systems such as Linux.

Features of Coda File System (CFS):

- CFS main goal is to achieve high availability.
- It has advanced caching schemes.
- It provide transparency

Architecture of Coda

- The clients cache entire files locally.
- Cache coherence is maintained by the use of callbacks. Clients dynamically find files on server and cache location information.
- For security, token-based authentication and end-to-end encryption is used.

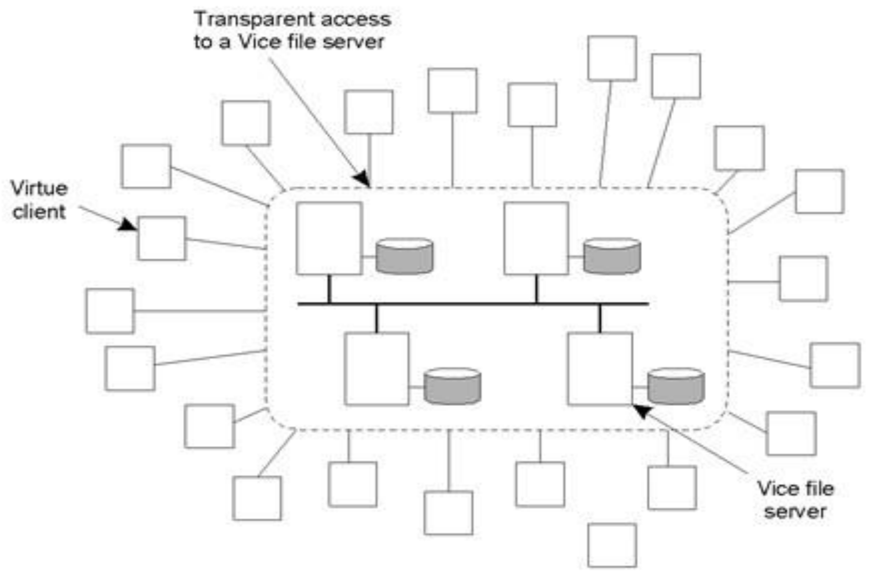


Fig 4.18: Coda file systems

- It is a principle of the design of Coda that the copies of files residing on servers are more reliable than those residing in the caches of clients.
- It is possible to construct a file system that relies entirely on cached copies of files in client.
- But such systems will have poor QOS.

- The Coda servers exist to provide the necessary quality of service.
- The copies of files residing in client caches are regarded as useful only as long as their currency can be revalidated against the copies in server.
- Revalidation occurs when disconnected operation ceases and the cached files are reintegrated with those in the servers.

Communication in Coda

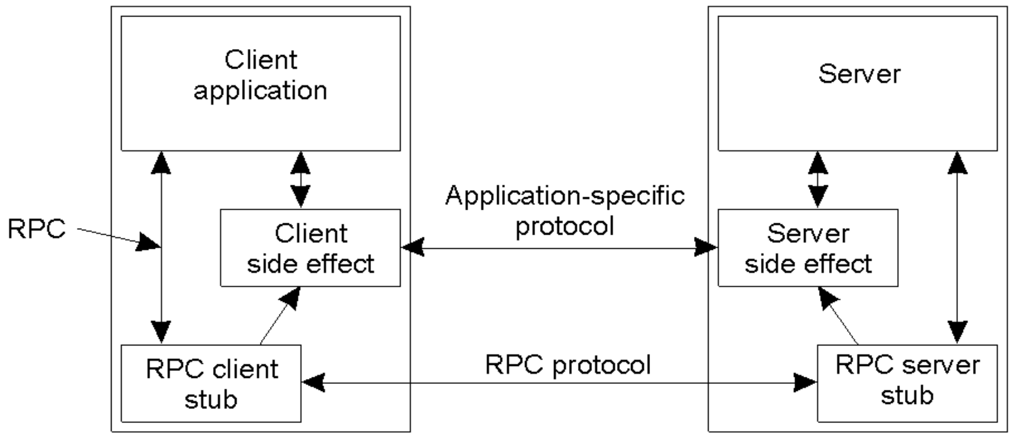


Fig 4.19: Communication in Coda

❖ Coda uses RPC2: a sophisticated reliable RPC system.

❖ The RPC2 start a new thread for each request, server periodically informs client it is still working on the request.

❖ RPC2 is useful for video streaming.

❖ RPC2 also has multicast support

❖ A side effect is a mechanism by which the client and server can communicate using an application-specific protocol.

❖ Coda uses side effect mechanism.

❖ Coda servers allow clients to cache whole files.

❖ Modifications by other clients are notified through invalidation messages require multicast RPC. The modifications can be done in any one of the following ways:

  – Sending an invalidation message one at a time

  – Sending invalidation messages in parallel

Processes in Coda

Coda … ts are known as Venus pr… and operate entirely in u…

Caching in

*

* … and provide

*

*

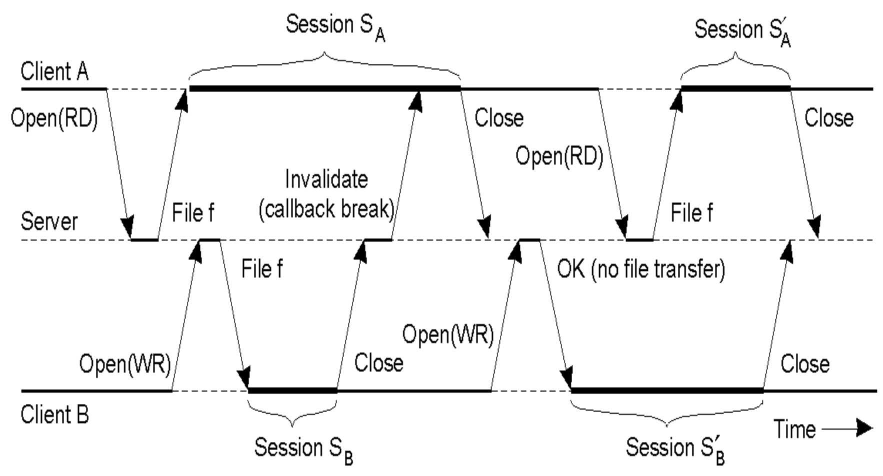


Fig 4.20: Caching in Coda

Server Replication in Coda

- ✓ The basic unit of replication is termed as volume.
- ✓ The Volume Storage Group (VSG) is set of servers that have a copy of a volume.
- ✓ The Accessible Volume Storage Group (AVSG) is set of servers in VSG that the client can contact .
- ✓ The Coda uses vector versioning:
    - ➢ One entry for each server in VSG
    - ➢ When file updated, corresponding version in AVSG is updated

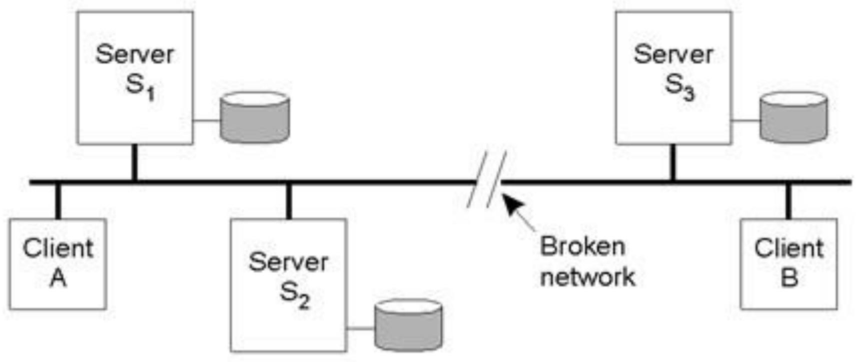


Fig 4.21: Server Replication

- ✓ In the above figure, the versioning vector at the time of partition is :[1,1,1]
- ✓ Client A updates file : versioning vector in its partition: [2,2,1]
- ✓ Client B updates file: versioning vector in its partition: [1,1,2]
- ✓ After the partition is repaired, compare versioning vectors. There will be a conflict.

Fault tolerance

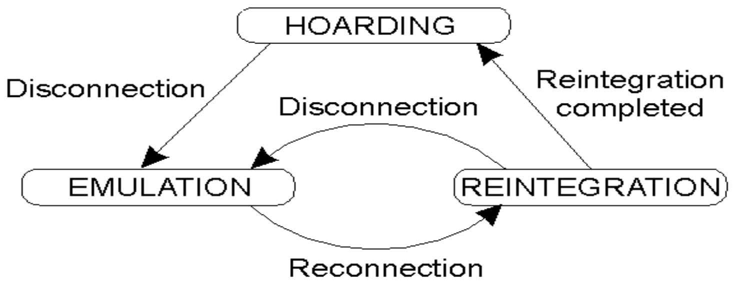


Fig 4.22: Fault Tolerance

The following are the fault tolerating actions:

- **HOARDING**: File cache in advance with all files that will be accessed when disconnected
- **EMULATION:** when disconnected, behavior of server emulated at client
- **REINTEGRATION:** transfer updates to server; resolves conflicts

REVIEW QUESTIONS

PART – A

1. **Define clock skew.**

   Clock skew is defined as the difference between the times on two clocks.

2. **Define clock drift.**

   Clock drift is the count time at different rates.

3. **What is Clock drift rate ?**

   Clock drift rate is the difference in precision between a prefect reference clock and a physical clock.

4. **What is External synchronization?**

   ∗ This method synchronize the process‟s clock with an authoritative external reference clock S(t) by limiting skew to a delay bound D > 0 - |S(t) - Ci(t) | < D for all t.

   ∗ For example, synchronization with a UTC (Coordinated Universal Time)source.

5. **What is Internal synchronization?**

   ∗ Synchronize the local clocks within a distributed system to disagree by not more than a delay bound D > 0, without necessarily achieving external synchronization - |Ci(t) - Cj(t)| < D for all i, j, t n

   ∗ For a system with external synchronization bound of D, the internal synchronization is bounded by 2D.

6. **Give the types of clocks.**

   Two types of clocks are used:

   ❖ Logical clocks : to provide consistent event ordering

   ❖ Physical clocks : clocks whose values must not deviate from the real time by more than a certain amount.

7. **What are the techniques are used to synchronize clocks?**

   ✓ time stamps of real-time clocks

   ✓ message passing

   ✓ round-trip time (local measurement)

8. **List the algorithms that provides clock synchronization.**

   - Cristian‟s algorithm
   - Berkeley algorithm
   - Network time protocol (Internet)

9. **Give the working of Berkley's algorithm.**

   The time daemon asks all the other machines for their clock values. The machines answer the request. The time daemon tells everyone how to adjust their clock.

10. **What is NTP?**

    The Network Time Protocol defines architecture for a time service and a protocol to distribute time information over the Internet.

11. **Give the working of Procedure call and symmetric modes:**

    - All messages carry timing history information.
    - The history includes the local timestamps of send and receive of the previous NTP message and the local timestamp of send of this message
    - For each pair i of messages (m, m‟) exchanged between two servers the following values are being computed

      - offseto$_i$: estimate for the actual offset between two clocks
      - delay d$_i$: true total transmission time for the pair of messages.

12. **Define casual ordering.**

    The partial ordering obtained by generalizing the relationship between two process is called as happened-before relation or causal ordering or potential causal ordering.

13. **Define logical clock.**

    A Lamport logical clock is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock.

14. **What is global state?**

    The global state of a distributed system consists of the local state of each process, together with the messages that are currently in transit, that is, that have been sent but not delivered.

15. **When do you call an object to be a garbage?**

    An object is considered to be garbage if there are no longer any references to it anywhere in the distributed system.

16. Define distributed deadlock.

A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and where there is a cycle in the graph of this „waits-for‟ relationship.

17. What is distributed snapshot?

Distributed Snapshot represents a state in which the distributed system might have been in. A snapshot of the system is a single configuration of the system.

18. What is consistent cut?

A consistent global state is one that corresponds to a consistent cut.

19. Define run.

A run is a total ordering of all the events in a global history that is consistent with each local history‟s ordering.

20. What is linearization?

A linearization or consistent run is an ordering of the events in a global history that is consistent with this happened-before relation o on H.

21. What is global state predicate?

A global state predicate is a function that maps from the set of global states of processes in the system.

22. What are the features of the unreliable failure detectors?

- unsuspected or suspected (i.e.) there can be no evidence of failure
- each process sends ``alive" message to everyone else
- not receiving ``alive" message after timeout
- This is present in most practical systems

23. What are the features of the reliable failure detectors?

- Unsuspected or failure
- They are present in synchronous system

24. Define distributed mutual exclusion.

Distributed mutual exclusion provide critical region in a distributed environment.

25. What are the requirements for Mutual Exclusion (ME)?

[ME1] safety: only one process at a time

[ME2] liveness: eventually enter or exit

[ME3] happened-before ordering: ordering of enter() is the same as HB ordering

26. What are the criteria for performance measures?

Bandwidth consumption, which is proportional to the number of messages sent in each entry and exit operations.

The client delay incurred by a process at each entry and exit operation.

Throughput of the system: Rate at which the collection of processes as a whole can access the critical section.

27. What is ring based algorithm?

This provides a simplest way to arrange mutual exclusion between N processes without requiring an additional process is arrange them in a logical ring.

28. What is mutual synchronisation?

This exploits mutual exclusion between N peer processes based upon multicast. Processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message.

29. What is Maekawa's Voting Algorithm?

In this algorithm, it is not necessary for all of its peers to grant access. Only need to obtain permission to enter from subsets of their peers, as long as the subsets used by any two processes overlap.

30. What is election algorithm?

An algorithm for choosing a unique process to play a particular role is called an election algorithm.

31. What is Ring based Election Algorithm?

➢ All the processes arranged in a logical ring.

➢ Each process has a communication channel to the next process.

➢ All messages are sent clockwise around the ring.

➢ Assume that no failures occur, and system is asynchronous.

➢ The ultimate goal is to elect a single process coordinator which has the largest identifier

32. What is bully algorithm?

This algorithm allows process to crash during an election, although it assumes the message delivery between processes is reliable.

33. What are the messages in Bully algorithm?

There are three types of messages:

a. Election message: This is sent to announce an election message. A process begins an election when it notices, through timeouts, that the coordinator has failed. T=2Ttrans+Tprocess From the time of sending

b. Answermessage: This is sent in response to an election message.

c. Coordinatormessage: This is sent to announce the identity of the elected process.

34. Define transaction.

A Transaction defines a sequence of server operations that is guaranteed to be atomic in the presence of multiple clients and server crash.

35. List the methods to ensure serializability.

There are three ways to ensure serializability:

- Locking
- Timestamp ordering
- Optimistic concurrency control

36. Give the advantages of nested transactions.

- Sub- transactions at same level can run concurrently.
- Sub- transactions can commit or abort independently.

37. Give the rules for committing of nested transactions.

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

38. What are the information held by the locks?

Each instance of Lock maintains the following information in its instance variables:

* the identifier of the locked object
* the transaction identifiers of the transactions that currently hold the lock
* a lock type

39. What is two phase locking?

The basic two-phase locking (2PL) protocol states:

- ✓ A transaction T must hold a lock on an item x in the appropriate mode before T accesses x.
- ✓ If a conflicting lock on x is being held by another transaction, T waits.
- ✓ Once T releases a lock, it cannot obtain any other lock subsequently.

40. Define deadlock.

Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock.

41. Give the disadvantages of locking and serialization.

- ✓ Lock maintenance represents an overhead that is not present in systems that do not support concurrent access to shared data. Locking sometimes are only needed for some cases with low probabilities.
- ✓ The use of lock can result in deadlock. Deadlock prevention reduces concurrency severely. The use of timeout and deadlock detection is not ideal for interactive programs.
- ✓ To avoid cascading aborts, locks cannot be released until the end of the transaction. This may reduce the potential for concurrency.

42. What is multi version time stamp ordering?

In multi-version timestamp ordering, a list of old committed versions as well as tentative versions is kept for each object. This list represents the history of the values of the object. The benefit of using g multiple versions is that read operations that arrive too late need not be rejected.

43. What is Hierarchic Two phase commit protocol?

* In this approach, the two-phase commit protocol becomes a multi-level nested protocol.
* The coordinator of the top-level transaction communicates with the coordinators of the sub-transactions for which it is the immediate parent.

44. What is Flat Two phase commit protocol?

- ✓ In this approach, the coordinator of the top-level transaction sends canCommit? messages to the coordinators of all of the sub-transactions in the provisional commit list.
- ✓ During the commit protocol, the participants refer to the transaction by its top-level TID.
- ✓ Each participant looks in its transaction list for any transaction or sub-transaction matching that TID.

45. Define phantom deadlock.

A deadlock that is detected but is not really a deadlock is called a phantom deadlock.

46. What is replication?

Replication in distributed systems enhances the performance, availability and fault tolerance.

47. What is Coda?

Coda is a distributed file system. Coda has been developed at Carnegie Mellon University (CMU) in the 1990s, and is now integrated with a number of popular UNIX-based operating systems such as Linux.

48. What are the fault tolerating actions?

- HOARDING: File cache in advance with all files that will be accessed when disconnected
- EMULATION: when disconnected, behaviour of server emulated at client
- REINTEGRATION: transfer updates to server; resolves conflicts

PART - B

1. Explain the clocking in detail.
2. Describe Cristian"s algorithm.
3. Write about Berkeley algorithm
4. Brief about NTP.
5. Explain about logical time and logical clocks.
6. Describe global states.

7. Brief about distributed mutual exclusion.
8. Write in detail about election algorithms.
9. Explain transactions and concurrency control.
10. Describe nested transaction and its issues.
11. What is optimistic concurrency control?
12. Write in detail about timestamp ordering.
13. Describe atomic commit protocol.
14. Explain distributed deadlocks.
15. Describe replication.
16. Write about Coda.