|  |
| --- |
| **UNIT IV SYNCHRONIZATION AND REPLICATION**  Introduction - Clocks, events and process states - Synchronizing physical clocks- Logical time and logical clocks - Global states – Coordination and Agreement – Introduction – Distributed mutual exclusion – Elections – Transactions and Concurrency Control– Transactions -Nested transactions – Locks – Optimistic concurrency control - Timestamp ordering – Atomic Commit protocols –Distributed deadlocks – Replication – Case study – Coda. |

**4.1 Clocks, events and process states**

* Let us consider how to order and timestamp the events that occur at a single process.
* Consider a distributed system consisting of a collection of

N processes Pi i =1, 2, 3 . . . N.

* Each process executes on a single processor, and the processors do not share memory.
* Each process **Pi has state Si** which in general, transforms as it executes.
* The process's state includes the values if all the variables within it. The state may also include the values of any objects in its local operating system environment that it affects, such as files.
* As each process Pi executes, **it takes a series of actions**, each of which is either a **message sends or receives operation, or an operation that transforms Pi state.**
* We **define an event** to be the occurrence of a single action that a process carries out as

if executes - a communication action or

a state-transforming section.

* The sequence of events within a single process Pi can be placed in a single, total ordering, using the relation.

—> i between the events.

ei —>i ei if and only if the event e occurs before e’ at Pi.

The history of a process Pi is defined as a series of events that take place within it, ordered by the relation

i history (Pi ) = h= <ei0, ei1,ei2 .>

**Clocks:**

* Each computer contains their own physical clock.
* These clocks are electronic devices that counts oscillations occurring in a crystal al a definite frequency, and that typically divide this count and store the result in a counter register.
* The operating system reads the node's hardware clock value H2(t), scales it and adds an offset so as to produce a software clock Ci( t ) = άH! ( t ) —> β that approximately measures real.

**Clock skew and clock drift:**

* Computer clocks will not be in perfect agreement. The **instantaneous difference between the readings of any two clocks is called clock skew.**
* Also, the crystal-based clocks may be subject to clock drift, which means that they count time at different rates.
* The oscillators are subject to physical variations with the consequence that their frequencies of oscillation differ.
* A clock's drift rate is the range in the offset between the clock and a nominal perfect reference clock per unit of time measured by the reference clock.
* For ordinary clocks based on a quartz crystal, this is about '' seconds/second - giving a difference of l sec every 10, 00,000 seconds or 11.6 days

**Skew between computer clocks in a distributed system**



**Figure 4.1**

**Coordinated universal time: (UTC)**

* It is an international standard for timekeeping, based on atomic time.
* UTC signals are synchronized and broadcast regularly from land based radio stations and satellites covering many parts of the world.
* Computers with receivers attached can synchronize their clocks with these timing signals.
* Computers may also receive the time to an accuracy of a few milliseconds over a telephone line.

**4.2 Synchronizing physical clocks.**

* In order to know at what time of day events occur at the processes in a distributed system, it is necessary to **synchronize the processes clock Ci with an Ci authoritative**, external source of time, this is **external synchronization**.
* If the **clocks are synchronized with one another to a known degree of accuracy**, then we can measure the interval between two events occurring at different computers **by appealing to their local clocks** - even though they are not synchronized externally known as **internal synchronization**.
* **Two modes of synchronization can be defined more closely as follows, over an interval of real time I.**

1. External Synchronization
2. Internal Synchronization

* **External synchronization**: For a synchronization bound D>0. and for a source S of **UTC time,**

**|*s(t) - Ci(t)| < D*** *, for I = 1, 2...N and for all real times t in I.*

* **Internal synchronization:** for a synchronization bound D>0. **|Ci (t) – Cj ( t )** for i, j = 1,2, 3, .. ., N and for all times t in I .

**Synchronization in a synchronous system.**

* In a synchronous system, **bounds are known for the drift rate of clocks**, the maximum message transmission delay and the time for execution.
* One process sends the time **t** on its local clock to the other in a message m.
* The receiving process could set its clock to the **time *t + Ttrans***

*where* ***Ttrans****. is the time taken* to transmit m between process.

* Unfortunately, *Ttrans is subject to variation and is unknown. In a synchronous system, by definition,* there is a lower bound (min) and an upper bound (max) on the time taken to transmit any message.
* Let the uncertainty in the message transmission time is u, so that **u= (max-min).**
* If its receiver sets its clock to be **t+min**, then the clock skew may be as much as **u**, since the message may in fact have taken time max to arrive.
* Similarly, if it sets its clock to **t+max**, the skew may again be as large as **u**. If. however, it sets its clock to the **half-way pit**, **t+(max+min)/2**, **then the skew is at most u/2.**
* In general, the **optimum bound** that can be achieved on clock skew when synchronizing **N clocks is u(l - 1/N).**

**Christian's method for synchronizing clocks:**

* Christian suggested the use of a time server, connected to a device that receives signals from a source UTC, to synchronize computers externally.
* Upon request, the server process **S** supplies the time according to its clocks,
* A process **p**  requests the time in a message **mr** and receives the time value **t** in the message **mi** .
* Process **p** records the **total round trip-time Tround** taken to send the request **mr** and receive the reply **mi** A simple **estimate of the time** to which **p** should set its clock is **t+Tround/2**,
* which assumes that the elapsed time is split equally before and after **S** placed **t** in **mi** If the value of the minimum transmission time min is known or can be conservatively estimated, We can determine the accuracy as follows:

**Clock synchronization using a time server**

**Figure 4.2**

* The earliest point at which **S** could have placed the time in **mi** was min after **p** dispatched **mr ,**The latest point at which it could have done **this was 'min' before mt arrived at p.**
* The time by **S's** clock when the reply message arrives is therefore in the range [**t + min, t+ Tround- min]** the width of this range is **-2 min**, **so the accuracy is +\_ (Tround/2-min)**

**The Berkeley algorithm:**

* In Berkeley algorithm, a coordinator computer is chosen to act as the master.
* This computer periodically polls the other computers whose clocks are to be synchronized, called slaves. The slaves send back their clock values to it.
* The master estimates their local clock times by observing the round-trip times and it averages the values obtained including its own clock's reading.
* Instead of sending the update current time back to the other computers, the master sends the amount by which each individual slave's clock requires adjustment.
* This can be a positive or negative value.
* The master takes a fault-tolerant average, that is, a subset of clocks is chosen that do not differ from one another by more than a specified amount, and the average is taken from only these clocks.
* If the master fails, then another computer can be elected to take over and function exactly as its predecessor.

**The Network Time Protocol: (NTP)**

* The NTP defines architecture for a time service and a protocol to distribute time information over the internet.
* The NTP service is provided by a network of servers located across the Internet.
* Primary servers are connected directly to a time source such as a radio lock receiving UTC; secondary servers are synchronized, ultimately, with primary servers. **The servers are connected in a logical hierarchy called a synchronization subnet whose levels are called strata.**
* Primary servers occupy stratum 1, stratum 2 servers are secondary servers that are synchronized directly with primary servers; stratum 3 servers are synchronized with stratum *2 servers* and so on.
* The clocks belongings to servers with high stratum numbers are able to be less accurate than those with low stratum numbers, because errors are introduced at each level of synchronization.

**An example synchronization subnet in an NTP implementation**

**Figure 4.3**

**Messages exchanged between a pair of NTP peers**

**Figure 4.4**

**NTP servers synchronize with one another in one of three modes:**

* **Multicast mode** - It is intended for use on a high-speed LAN. One or more servers periodically multicasts the time to the servers running in other computers connected by the LAN, which set their clocks assuming a small delay.
* **Procedure call mode**- In this mode, one server accepts requests from other computers, which it processes by replying with its timestamp. This mode is suitable where higher accuracies are required than multicast mode.
* **Symmetric mode**- It is intended for use by the servers that supply time information in LANs and by the higher levels of the synchronization subnet, where the higher accuracies are to be achieved.
* A pair of server operating in symmetric mode exchange messages bearing timing information.
* Timing data are retained as part of an association between the servers that is maintained in order to improve the accuracy *of their synchronization over time.*

**4.3 Logical time and logical clocks**

* It is impossible to synchronize clocks perfectly across a distributed system. We cannot, in general, use physical time to find out the order of any arbitrary pair of events occurring within it.
* We can use a scheme that is similar to physical causality, but that applies in distributed systems, to order some of the events that occur at different processes.
* **This ordering is based on two simple points:**
* If two events occurred at the same process Pi (i = 1, 2, 3. . . N), then they occurred in the order in which Pi observes them.
* Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message.

The happened-before relation, denoted by is *—> defined as follows:*

HB1 If з process Pi : e—>i e1, then e->e1

HB2: For any message m, send (m) -> receive (m)

HB3: If e, e1 and e11 are events such that e—>e1 and e1—>e11, then e->e11.

The relation *—> is illustrated in figure , for the case of three processes; P 1, P2 and P3 in* Diagram.

**Events occurring at three processes**



**Figure 4.5**

* It can be seen that a->b, since the events occur in this order at process p| (a1 b) and similarly c->d. Furthermore b->c, since these events are involved in the sending and receiving message m(t), and similarly d->f. combining these relations, we also say that a->f.
* It can also be seen that not all events are related by the relation ->*. For example, a and e are not related with* ->*, since* they occur at different processes, and there is no chain of messages intervening between them.. So the events **a and e** are concurrent and it can be represented by **a || e.**

**Logical clocks:**

* Lamport invented a simple mechanism, by which the happened-before ordering can be captured numerically, called a logical clock.
* A lamport logical clock is a monotonically increasing software counter.
* Each process Pi keeps its own logical clock, Li which it uses to apply time stamps to events.
* The timestamp of event e at Pi. is denoted by Li (e). Processes update their logical clocks and transmit the values of their logical clocks in message as follows:

LC1: L. is incremented before each event is issued at process P*i : Li = Li+ 1*

LC2:

a) When a process P. sends a message m, it piggybacks on m the value t = Li.

b) On receiving (m, t), a process Pj computes Lj = max (Lj , t) and then applies LC1, before time-stamping the event receive (m).

**Lamport timestamps for the events**



**Figure 4.6**

* If e—>e1 then L(e) < L(e1). But the converse is not true. That is. If L(e) < L(e1), then we cannot infer that e->e1.
* Each processes P1, P2&P3 has its logical clock has its logical clock initialized to 0 the clock values given are immediately after the event to which they are adjacent. E.g. L(b)>L(e) but b || e.

**Vector clocks:**

**Vector clocks were developed to overcome the shortcoming of lamport's clocks**, the fact that from L(e) < L(e1) we cannot conclude that e->e1 A vector clock for a system of N processes array of N integers. Each process keeps its own vector clock vi

The rules for updating the clocks are as

VC1: Initially, Vi (j) = 0, for I, j=1, 2, …N.

VC2: Just before *pt* timestamps an event, it sets Vi[i] = Vi[i]+ 1

VC3: P. includes the value t = V [i] in every message it sends

VC4: when P. receives a timestamp t in a message, it sets

Vi[j] = max (Vi [ j], t[ j] ) , for j = 1, 2 ... N. Taking the component wise maximum of two vector timestamps in this way is known as a merge operation.

**Vector timestamps for the events**

****

**Figure 4.7**

**4.4 Global States:**

* The global state of a distributed system consists of the local state of each process, together with the messages that are currently in transit, i.e., that have been sent but not delivered.
* Knowing the global state of a distributed system may be useful in detecting the following.
* Distributed deadlock detection
* Distributed garbage object detection
* Distributed termination detection
* Distributed debugging.
* The global history is defined as the union of the individual process histories

**H=h0 Ụ h1 Ụ ….. Ụ hn-1**

* Mathematically any set of states of the individual processes can be taken to form a global state

**S =(s1 s2, ... sN).**

* The **notion of global state** can be graphically represented by what **is called as a cut.**
* The represents the last event that has been recorded for each process.
* Consider the events occurring at processes P1 and P2 as shown in figure, it has two cuts.
* The **leftmost cut is inconsistent** because, at P2 it includes the receipt of the message m1 but at P not includes the sending of that message. Which is showing an effect without a cause?
* The **rightmost cut is consistent**. It includes the sending but not the receipt of message m2 That is consistent with the actual execution because, the message may take some time to arrive.

**Cuts**



**Figure 4.8**

* A cut C is consistent, if for each event it contains, it also contains all the events that happened before that event:
* For all events **e є C, f e => = f є C**.A consistent global state is one that corresponds to a consistent cut.
* A run is a total ordering of all the events in a global history that is consistent with each local history's ordering, i(i=l, 2 ... N).
* A linearization a consistent run is an ordering of the events in a global history with this happened before relation on H.
* A state S1 is reachable from a state S if there is a linearization that passes through S and then S1.

**Snapshot algorithm of candy and lamport**

* This algorithm is used for determining global states of a distributed system.
* The goal of the algorithm is to record a set of process and channel states for a set of processes

Pi ( i=l , 2...N).

The algorithm assumes that:

* Neither channels nor processes fail;
* Communication is reliable so that every message sent is eventually received, exactly once.
* Channels are unidirectional. The graph of processes and channels is strongly connected.
* Any process may initiate a global snapshot at any time.
* The processes may continue their execution and sent and receive normal messages while the snap shot takes place.

**Chandy and Lamports ‘snapshot’ algorithm**

* For each process Pi the incoming and outgoing channels are used for receiving and sending messages respectively.
* The algorithm proceeds through the use of special marker messages.
* The marker has a dual role: as a prompt for the receiver to save its own state, if it has not already done so; and as a means of determining which messages to include in the channel state.
* The algorithm is defined through two rules;
  + - 1. The marker receiving rule
      2. The marker sending rule.
* **The marker sending rule** obligates processes to send a marker after they have recorded their state, but before they send any other messages.
* **The marker receiving rule** delegates a process that has not recorded its state to do so.
* In that case, this is the first marker that it has recorded. It notes which messages subsequently arrive on the other incoming channels.
* When a process that has already saved its state receives a marker on another channel, it records the state of that channel as the set of messages it received on it since it saved on its state.

**4.5 COORDINATION AND AGREEMENT**

**4.5.1 Introduction**

* This chapter introduces a collection of algorithms whose goals vary but that share an aim that is fundamental in distributed systems: for a set of processes to coordinate their actions or to agree on one or more values.
* The computers must be able to do so even where there is no fixed master-slave relationship between the components (which would make coordination particularly simple). The reason for avoiding fixed master-slave relationships is that we often require our systems to keep working correctly even if failures occur, so we need to avoid single points of failure, such as fixed masters.
* **Another important aim** is to consider failures, and how to deal with them when designing algorithms. Coping with failures is a subtle business, so we begin by considering some algorithms that tolerate no failures and progress through benign failures before exploring how to tolerate arbitrary failures
* Along the way, we encounter a fundamental result in the theory of distributed systems: even under surprisingly benign failure conditions, it is impossible to guarantee in an asynchronous system that a collection of processes can agree on a shared value
* **Distributed mutual exclusion is the extension to distributed systems of the familiar problem** of avoiding race conditions in kernels and multi-threaded applications. Since much of what occurs in distributed systems is resource sharing, this is an important problem to solve.
* It introduces the related but more general issue of how to **‘elect’** one of a collection of processes to perform a special role. We saw how processes synchronize their clocks to a designated time server. If this server fails and several surviving servers can fulfill that role, then for the sake of consistency it is necessary to choose just one server to take over.
* Coordination and agreement related to group communication is the ability to multicast a message to a group is a very useful communication paradigm, with applications from locating resources to coordinating the updates to replicated data.

**4.6 Distributed mutual exclusion**

* Distributed processes often need to coordinate their activities. If a collection of processes share a resource or collection of resources, then often mutual exclusion is required to prevent interference and ensure consistency when accessing the resources. This is **the *critical section* problem**, familiar in the domain of operating systems.
* In a distributed system, however, neither shared variables nor facilities supplied by a single local kernel can be used to solve it, in general. We require a solution to ***distributed mutual exclusion***: one that is based solely on message passing.
* It is useful to have a generic mechanism for distributed mutual exclusion at our disposal – one that is independent of the particular resource management scheme in question. We now examine some algorithms for achieving that.
* 4.8.1 Algorithms for mutual exclusion
* 4.8.2 The central server algorithm
* 4.8.3 A ring-based algorithm
* 4.8.4 An algorithm using multicast and logical clocks
* 4.8.5 Maekawa’s voting algorithm

**4.6.1 Algorithms for mutual exclusion**

* Consider a system of *N* processes *pi*, *i* = 1, 2,….*N*, that do not share variables. The processes access common resources, but they do so in a critical section. For the sake of simplicity, we assume that there is only one critical section. It is straightforward to extend the algorithms we present to more than one critical section.
* Assume that the system is asynchronous, that processes do not fail and that message delivery is reliable, so that any message sent is eventually delivered intact, exactly once.
* The application-level protocol for executing a critical section is as follows:
* ***enter()***// enter critical section – block if necessary
* ***resource Accesses()***// access shared resources in critical section
* ***exit()***// leave critical section – other processes may now enter
* Essential requirements for mutual exclusion are as follows:

**ME1**: (safety) At most one process may execute in the critical section(CS) at a time.

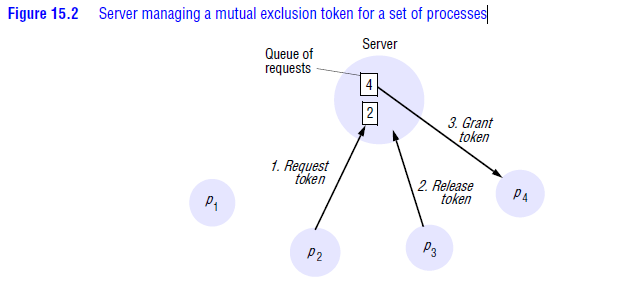
**ME2**: (liveness) Requests to enter and exit the critical section eventually succeed.

**ME3:**( → ordering) If one request to enter the CS happened-before another, then entry to CS is granted in that order.

* Evaluate the performance of algorithms for mutual exclusion according to the following criteria:
* ***bandwidth***consumed, which is proportional to the number of messages sent in each *entry* and *exit* operation;
* ***client delay***incurred by a process at each *entry* and *exit* operation;
* the algorithm’s effect upon the ***throughput***of the system. This is the rate at which the collection of processes as a whole can access the critical section, given that some communication is necessary between successive processes.
* We measure the effect using the ***synchronization delay***between one process exiting the critical section and the next process entering it; the throughput is greater when the synchronization delay is shorter.

**4.6.2 The central server algorithm**

* The simplest way to achieve mutual exclusion is to employ a server that grants permission to enter the critical section.
* To enter a critical section, a process sends a request message to the server and awaits a reply from it.
* Conceptually, the reply constitutes a token signifying permission to enter the critical section.
* If no other process has the token at the time of the request, then the server replies immediately, granting the token.
* If the token is currently held by another process, then the server does not reply, but queues the request.
* When a process exits the critical section, it sends a message to the server, giving it back the token.

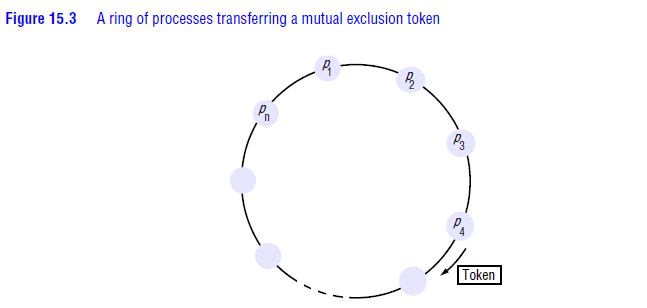


**Figure 4.12 Server managing a mutual exclusion token for set of processes**

* If the queue of waiting processes is not empty, then the server chooses the oldest entry in the queue, removes it and replies to the corresponding process.
* The chosen process then holds the token. In the figure4.12, we show a situation in which *p*2 ’s request has been appended to the queue, which already contained *p*4 ’s request. *p*3 exits the critical section, and the server removes *p*4 ’s entry and grants permission to enter to *p*4 by replying to it. Process *p*1 does not currently require entry to the critical section.
* Given our assumption that no failures occur, it is easy to see that the safety and liveness conditions are met by this algorithm. The reader should verify, however, that the algorithm does not satisfy property ME3.
* The server may become a performance bottleneck for the system as a whole.
* The synchronization delay is the time taken for a round-trip: a *release* message to the server, followed by a *grant* message to the next process to enter the critical section.

**4.6.3 A ring-based algorithm**

* One of the simplest ways to arrange mutual exclusion between the *N* processes without requiring an additional process is to arrange them in a logical ring.
* This requires only that each process *pi* has a communication channel to the next process in the ring, *p*(*i* + 1)*mod N* .
* The idea is that exclusion is conferred by obtaining a token in the form of a message passed from process to process in a single direction –clockwise, say – around the ring.
* The ring topology may be unrelated to the physical interconnections between the underlying computers.

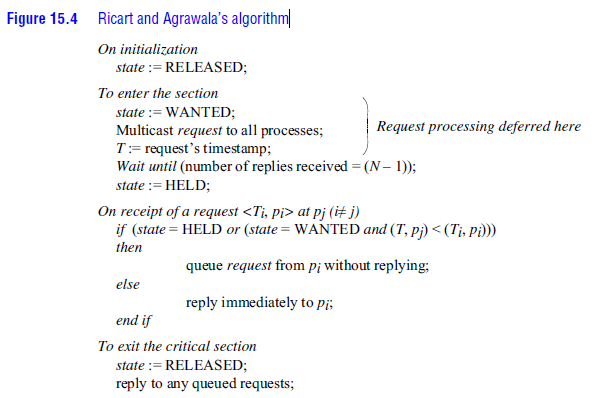


**Figure 4.13 A ring of processes transferring a mutual exclusion token**

* If a process does not require entering the critical section when it receives the token, then it immediately forwards the token to its neighbor.
* A process that 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.
* It is straightforward to verify that the conditions ME1 and ME2 are met by this algorithm, but that the token is not necessarily obtained in happened-before order.
* This algorithm continuously consumes network bandwidth (except when a process is inside the critical section): the processes send messages around the ring even when no process requires entry to the critical section. The delay experienced by a process requesting entry to the critical section is between 0 messages and *N* messages.
* To exit the critical section requires only one message. The synchronization delay between one process’s exit from the critical section and the next process’s entry is anywhere from 1 to *N* message transmissions.

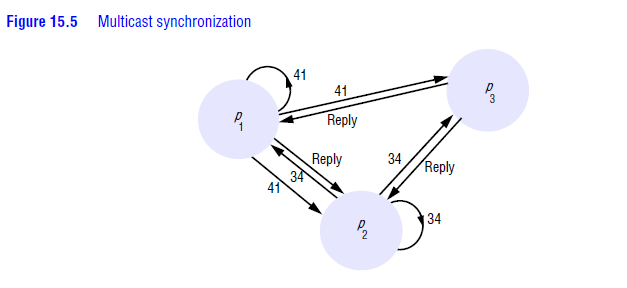
**4.6.4 An algorithm using multicast and logical clocks •**

* Ricart and Agrawala [1981] developed an algorithm to implement mutual exclusion between *N* peer processes that is based upon multicast.
* The basic idea is that 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 conditions under which a process replies to a request are designed to ensure that conditions ME1*–*ME3 are met.



**Figure 4.14 Ricart and Agrawala algorithm**

* The processes *p*1 ,*p*2 ,…. *pN* bear distinct numeric identifiers.
* They are assumed to possess communication channels to one another, and each process *pi* keeps a Lamport clock, updated according to the rules LC1 and LC2 of logical clock.
* Messages requesting entry are of the form <*T, pi* >, where *T* is the sender’s timestamp and *pi* is the sender’s identifier.
* Each process records its state of being outside the critical section (*RELEASED*), wanting entry (*WANTED*) or being in the critical section (*HELD*) in a variable *state*.
* If a process requests entry and the state of all other processes is*RELEASED*, then all processes will reply immediately to the request and the requester will obtain entry.
* If some process is in the state *HELD*, then that process will not reply to requests until it has finished with the critical section, and so the requester cannot gain entry in the meantime.
* If two or more processes request entry at the same time, then whichever process’s request bears the lowest timestamp will be the first to collect *N* – 1 replies, granting it entry next.
* If the requests bear equal Lamport timestamps, the requests are ordered according to the processes’ corresponding identifiers.
* This algorithm achieves the safety property ME1. If it were possible for two processes *pi* and *pj*(*i*, *j* ) to enter the critical section at the same time, then both of those processes would have to have replied to the other.
* But since the pairs *<Ti ,pi>*are totally ordered, this is impossible. We leave the reader to verify that the algorithm also meets requirements ME2 and ME3*.*
* Let us assume that *p*3 is not interested in entering thecritical section, and that *p*1 and *p*2 request entry concurrently.
* The timestamp of *p*1’srequest is 41, and that of *p*2 is 34. When *p*3 receives their requests, it replies immediately.

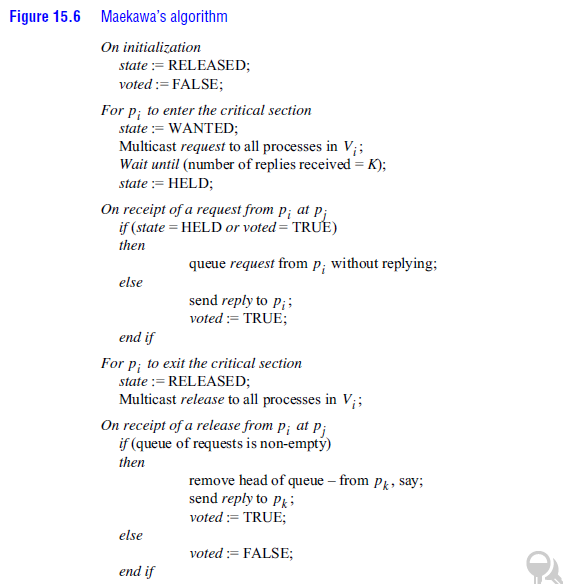


**Figure 4.15 Multicast Synchronization**

* When *p*2 receives *p*1’s request, it finds that its own request has the lower timestamp and so does not reply, holding *p*1 off.
* However, *p*1 finds that *p*2 ’s request has a lower timestamp than that of its own request and so replies immediately.
* On receiving this second reply, *p*2 can enter the critical section. When *p*2 exits the critical section, it will reply to *p*1 ’s request and so grant it entry.
* Gaining entry takes 2(*N* – 1) messages in this algorithm: *N* – 1 to multicast the request, followed by *N* – 1 replies. Only one message is required for the request; the total is then *N* messages.
* It is thus a more expensive algorithm, in terms of bandwidth consumption, than the algorithms just described.
* However, the client delay in requesting entry is again a round-trip time.
* The advantage of this algorithm is that its synchronization delay is only one message transmission time.
* Both the previous algorithms incurred a round-trip synchronization delay.

**4.6.5 Maekawa’s voting algorithm**

* Maekawa [1985] observed that in order for a process to enter a critical section, it is not necessary for all of its peers to grant it access.
* Processes need only obtain permission to enter from *subsets* of their peers, as long as the subsets used by any two processes overlap.
* We can think of processes as voting for one another to enter the critical section. A ‘candidate’ process must collect sufficient votes to enter.
* Processes in the intersection of two sets of voters ensure the safety property ME1, thatat most one process can enter the critical section, by casting their votes for only one candidate.
* Maekawa associated a *voting set Vi* with each process *pi* ( *i*= 1,2,….. *N* ), where *Vi€{ p*1, *p*1 ,…. *pN*}. The sets *Vi* are chosen so that, for all *i*, *j* = 1, 2,…..*N* :
* ***pi €Vi***
* *Error! Bookmark not defined.***– there is at least one common member of any two voting sets**
* **|*Vi |*= *K* – to be fair, each process has a voting set of the same size**
* **Each process *pj*is contained in *M* of the voting sets *Vi* .**



**Figure 4.16 Maekawa’s algorithm**

* To obtain entry to the critical section, a process *pi* sends *request* messages to all *K* members of *Vi*(including itself).
* *pi*cannot enter the critical section until it has received all *K reply* messages. When a process *pj*in*Vi* receives *pi* ’s*request* message, it sends a *reply* message immediately, unless either its state is *HELD* or it has already replied (‘voted’) since it last received a *release* message.
* Otherwise, it queues the request message (in the order of its arrival) but does not yet reply.
* When a process receives a *release* message, it removes the head of its queue of outstanding requests (if the queue is nonempty) and sends a *reply* message (a ‘vote’) in response to it.
* To leave the critical section, *pi* sends *release* messages to all *K* members of *Vi*(including itself).
* This algorithm achieves the safety property, ME1. If it were possible for two processes *pi* and *pj*to enter the critical section at the same time, then the processes in *Vi n Vj ≠ Φ* would have to have voted for both.
* But the algorithm allows a process to make at most one vote between successive receipts of a *release* message, so this situation is impossible.
* Unfortunately, the algorithm is deadlock-prone. Consider three processes, *p*1 , *p*2 and *p*3 , with *V*1 ={*p*1 ,*p*2 } , *V*2={ *p*2, *p*3} and *V*3={ *p*3, *p*1} . If the three processes concurrently request entry to the critical section, then it is it is possible for *p*1 to reply to itself and hold off *p*2 , for *p*2 to reply to itself and hold off *p*3 , and for *p*3 to reply to itself and hold off *p*1
* Each process has received one out of two replies, and none can proceed.
* The algorithm can be adapted [Sanders 1987] so that it becomes deadlock-free.
* In the adapted protocol, processes queue outstanding requests in happened-before order, so that requirement ME3 is also satisfied.

**Fault tolerance**

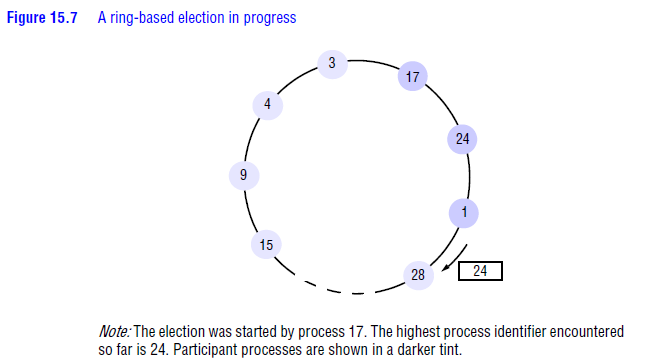
* The main points to consider when evaluating the above algorithms with respect to fault tolerance are:
* What happens when messages are lost?
* What happens when a process crashes?
* None of the algorithms that we have described would tolerate the loss of messages, if the channels were unreliable. Maekawa’s algorithm can tolerate some process crash failures: if a crashed process is not in a voting set that is required, then its failure will not affect the other processes.

**4.7 Elections**

* An algorithm for choosing a unique process to play a particular role is called an *election algorithm*.
* We say that a process *calls the election* if it takes an action that initiates a particular run of the election algorithm. An individual process does not call more than one election at a time, but in principle the *N* processes could call *N* concurrent elections.
* At any point in time, a process *pi* is either a *participant* – meaning that it is engaged in some run of the election algorithm – or a *non-participant* – meaning that it is not currently engaged in any election.
* An important requirement is for the choice of elected process to be unique, even if several processes call elections concurrently. For example, two processes could decide independently that a coordinator process has failed, and both call elections.
* Without loss of generality, we require that the elected process be chosen as the one with the largest identifier. The ‘identifier’ may be any useful value, as long as the identifiers are unique and totally ordered.
  + Each process *pi* ( *i*= 1, 2,…. *N* ) has a variable *electedi*, which will contain the identifier of the elected process. When the process first becomes a participant in an election it sets this variable to the special value ‘ ┴ ’ to denote that it is not yet defined.
* Our requirements are that, during any particular run of the algorithm:
* **E1: (safety) A participant process *pi* has *electedi*= ┴ or *electedi*= *P*,where*P*is chosen as the non-crashed process at the end of the run with the largest identifier.**
* **E2: (liveness) All processes *pi* participate and eventually either set *electedi*≠ ┴– or crash**
* We measure the performance of an election algorithm by its total network bandwidth utilization (which is proportional to the total number of messages sent), and by the *turnaround time* for the algorithm: the number of serialized message transmission times between the initiation and termination of a single run.
* 4.9.1 A ring-based election algorithm
* 4.9.2 The bully algorithm
* 4.9.3 TYPES OF MESSAGE

**4.7.1 A ring-based election algorithm**

* The algorithm of Chang and Roberts is suitable for a collection of processes arranged in a logical ring.
* Each process *pi* has a communication channel to the next process in the ring, *p*(*i* + 1)*mod N* , and all messages are sent clockwise around the ring.
* 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 neighbor.



**Figure 4.16 A ring-based election in process**

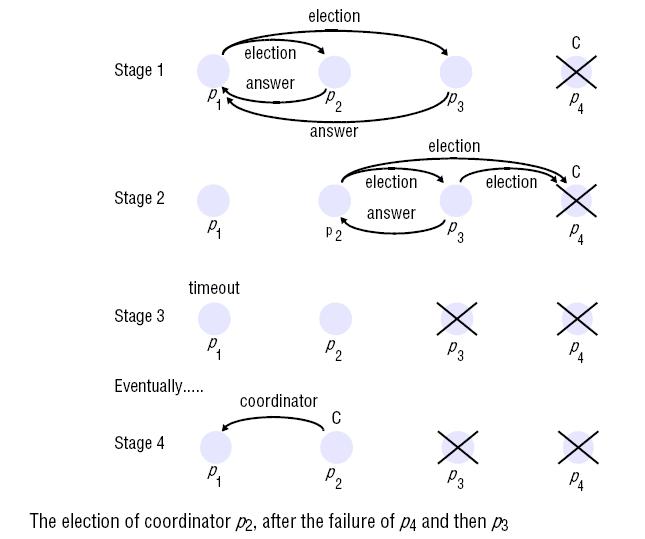
* When a process receives an *election* message, it compares the identifier in the message with its own.
  + If the arrived identifier is greater, then it forwards the message to its neighbor.
  + If the arrived identifier is smaller and the receiver is not a *participant*, then it substitutes its own identifier in the message and forwards it; but it does not forward the message if it is already a *participant*.
  + On forwarding an *election* message in any case, the process marks itself as a *participant*.
* If, however, the received identifier is that of the receiver itself, then this process’s identifier must be the greatest, and it becomes the coordinator.
* The coordinator marks itself as a *non-participant* once more and sends an *elected* message to its neighbour, announcing its election and enclosing its identity.
* It is easy to see that condition E1 is met.
  + All identifiers are compared, since a process must receive its own identifier back before sending an *elected* message.
  + For any two processes, the one with the larger identifier will not pass on the other’s identifier. It is therefore impossible that both should receive their own identifier back.
* Condition E2 follows immediately from the guaranteed traversals of the ring (there are no failures).
* Note how the *non-participant* and *participant* states are used so that duplicate messages arising when two processes start an election at the same time are extinguished as soon as possible, and always before the ‘winning’ election result has been announced.
* If only a single process starts an election, then the worst-performing case is when its anti-clockwise neighbour has the highest identifier.
  + A total of *N* – 1 messages are then required to reach this neighbour, which will not announce its election until its identifier has completed another circuit, taking a further *N* messages.
  + The *elected* message is then sent *N* times, making 3*N* – 1 messages in all.
  + The turnaround time is also 3*N* – 1 , since these messages are sent sequentially.
  + The *election* message currently contains 24, but process 28 will replace this with its identifier when the message reaches it.
* While the ring-based algorithm is useful for understanding the properties of election algorithms in general, the fact that it tolerates no failures makes it of limited practical value.
* However, with a reliable failure detector it is in principle possible to reconstitute the ring when a process crashes.

**4.7.2 The bully algorithm**

* The bully algorithm [Garcia-Molina 1982] allows processes to crash during an election, although it assumes that message delivery between processes is reliable, it uses timeouts to detect a process failure.
* The bully algorithm, on the other hand, assumes that each process knows which processes have higher identifiers, and that it can communicate with all such processes.

**4.7.3 TYPES OF MESSAGE**

* Election message - announce an election.
* Answer message - response to an election message.
* Coordinator message - announce the identity of the elected process.
* Since the system is synchronous, we can construct a reliable failure detector. There is a maximum message transmission delay, *Ttrans* , and a maximum delay for processing a message *Tprocess* .
* Therefore, we can calculate a time *T* = 2*Ttrans* + *Tprocess* that is an upper bound on the time that can elapse between sending a message to another process and receiving a response.
* If no response arrives within time *T*, then the local failure detector can report that the intended recipient of the request has failed.
* The process that knows it has the highest identifier can elect itself as the coordinator simply by sending a ***coordinator* message** to all processes with lower identifiers.
* A process with a lower identifier can begin an election by sending an ***election* message** to those processes that have a higher identifier and awaiting ***answer* messages** in response.
* If none arrives within time *T*, the process considers itself the coordinator and sends a *coordinator* message to all processes with lower identifiers announcing this.
* If the process waits a further period *T* ‘for a *coordinator* message to arrive from the new coordinator. If none arrives, it begins another election.
* If a process *pi* receives a *coordinator* message, it sets its variable *elected i* to the identifier of the coordinator contained within it and treats that process as the coordinator.
* If a process receives an *election* message, it sends back an *answer* message and begins another election – unless it has begun one already.
* When a process is started to replace a crashed process, it begins an election. If it has the highest process identifier, then it will decide that it is the coordinator and announce this to the other processes.
* Thus it will become the coordinator, even though the current coordinator is functioning. It is for this reason that the algorithm is called the ‘bully’ algorithm.



**Figure 4.17**

* There are four processes, *p*1 –*p*4 . Process *p*1 detects the failure of the coordinator *p*4 and announces an election (stage 1 in the figure). On receiving an *election* message from *p*1 , processes *p*2 and *p*3 send *answer* messages to *p*1 and begin their own elections; *p*3 sends an *answer* message to *p*2 , but *p*3 receives no *answer* message from the failed process *p*4 (stage 2). It therefore decides that it is the coordinator. But before it can send out the *coordinator* message, it too fails (stage 3). When *p*1 ’s timeout period *T*􀁣 expires (which we assume occurs before *p*2 ’s timeout expires), it deduces the absence of a *coordinator* message and begins another election. Eventually, *p*2 is elected coordinator (stage 4).

**4.8TRANSACTION 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.
  1. **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.

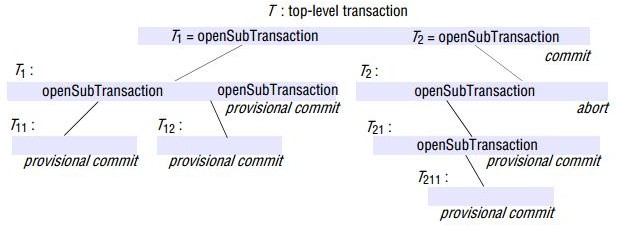
* + 1. **Concurrency Control Serial Equivalence**
       - If these transactions are done one at a time in some order, then the final result will be correct.
       - If we do not want to sacrifice the concurrency, an interleaving of the operations of transactions may lead to the same effect as if the 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. |

* 1. **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.
  1. **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:

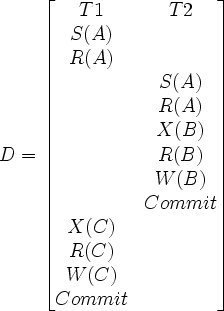
* A transaction T must hold a lock on an item x in the appropriate mode before T accesses x.
* If a conflicting lock on x is being held by another transaction, T waits.
* Once T releases a lock, it cannot obtain any other lock subsequently. In two phase locking, a transaction is divided into two phases:
* 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.



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

1. When an operation accesses an object within a transaction:
   1. If the object is not already locked, it is locked and the operation proceeds.
   2. If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   3. If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   4. 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

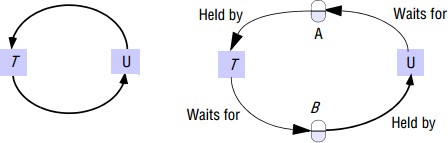
A transaction with a read lock that is shared by other transactions cannot promote its read lock to a write lock, because write lock will conflict with other read locks.

* + 1. **Deadlocks**

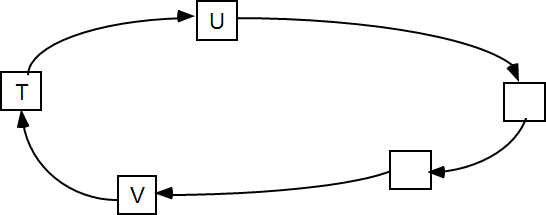


***Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock.***

* + - * 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 4.17: Wait for graph**



**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 <P1, P2, …, Pn> of ALL the processes is the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, 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.
  + If lock manager blocks a request, an edge can be added. Cycle should be checked each time a new edge is added.
  + 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
  1. **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 close Transaction 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** | **Tv** | **T1** | **Rule** |
| 1 | Write | Read | Ti must not read objects written by Tv |
| 2 | Read | Write | Tvmust not read objects written by Ti |
| 3 | Write | Write | Ti must not read objects written by Tvand Tvmust 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 Tv boolean valid = true;  for (intTi = startTn+1; Ti<= finishTn; Ti++){  if (read set of Tv intersects write set of Ti) valid = false;  } | Forward validation of transaction Tv boolean valid = true;  for (intTid = active1; Tid<= activeN; Tid++){  if (write set of Tv intersects read set of Tid) 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.

* 1. **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** | **Tc** | **Ti** | **Description** |
| 1 | Write | Read | Tc must not write an object that has been read by any Ti where Ti>Tcthis requires that Tc>= the maximum read timestamp of the object. |
| 2 | Write | Write | Tc must not write an object that has been written by any Ti where Ti>Tcthis requires that Tc> the write timestamp of the committed object. |

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | Read | Write | Tc must not read an object that has been written by any Ti where Ti>Tcthis requires that Tc>the write timestamp of the committed object. |

**Timestamp ordering by read rule:**

if ( Tc> write timestamp on committed version of D) {

let Dselected be the version of D with the maximum write timestamp ≤ Tc if (Dselected is committed)

perform read operation on the version Dselected

else

Wait until the transaction that made version Dselected commits or aborts then reapply the read rule

} else

Abort transaction Tc

**Timestamp ordering by write rule**

if (Tc ≥ maximum read timestamp on D&&

Tc> write timestamp on committed version of D)

perform write operation on tentative version of D with write timestamp Tc else /\* write is too late \*/

Abort transaction Tc

**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.

* 1. **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.**

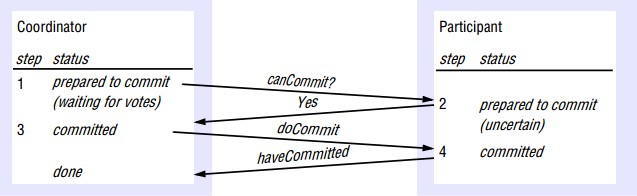
**Two Phase Commit Protocol**

* + - 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).

1. 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.
2. 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**

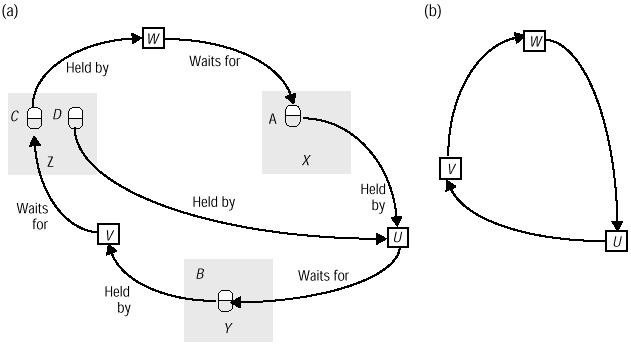
* + - In this approach, the two-phase commit protocol becomes a multi-level nested protocol.
    - The coordinator of the top-level transaction communicates with the coordinators of the sub-transactions for which it is the immediate parent.
    - It sends canCommit? messages to each of the latter, which 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);

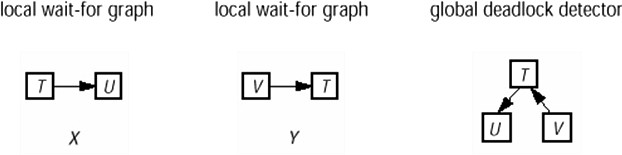
* + aborts those with aborted ancestors;
  + sends a Yes vote to the coordinator.
* If the participant does not have a provisionally committed descendent of the top level transaction, it must have failed since it performed the sub- transaction and it sends a No vote to the coordinator
  1. **DISTRIBUTED 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.

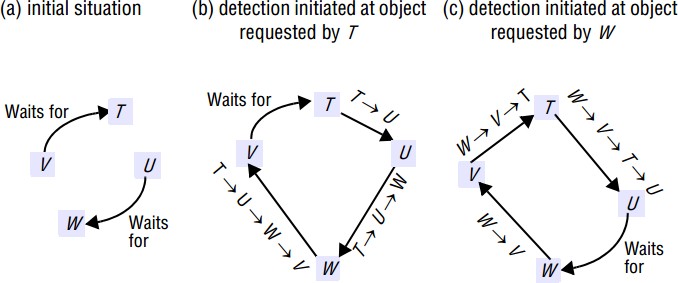
* + - **Detection:** receive probes and decide whether deadlock has occurred and whether to forward the probes.
    - When a server receives a probe TU and finds the transaction that U is waiting for, say V, is waiting for another data item elsewhere, a probe TUV is forwarded.
    - **Detection:** 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.

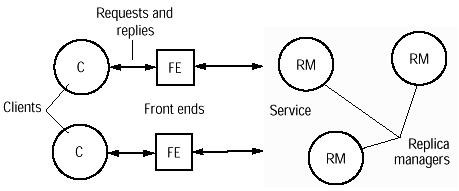


**Fig 4.17: Initiated probes**

* 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.
  1. **REPLICATION**

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

* + - Replication transparency
    - Consistency
    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.
  1. **CASE STUDY: CODA**

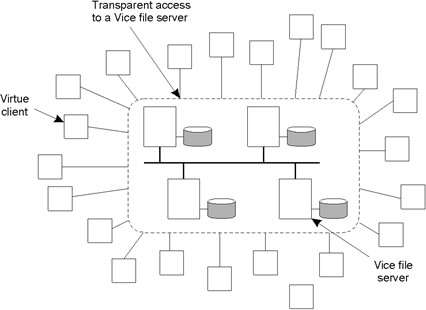
Coda is a distributed file system. Coda has been developed at Carnegie Mellon University (CMU) in the 1990s, and is now integrated with a number of popular UNIX-based operating systems such as Linux.

**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**

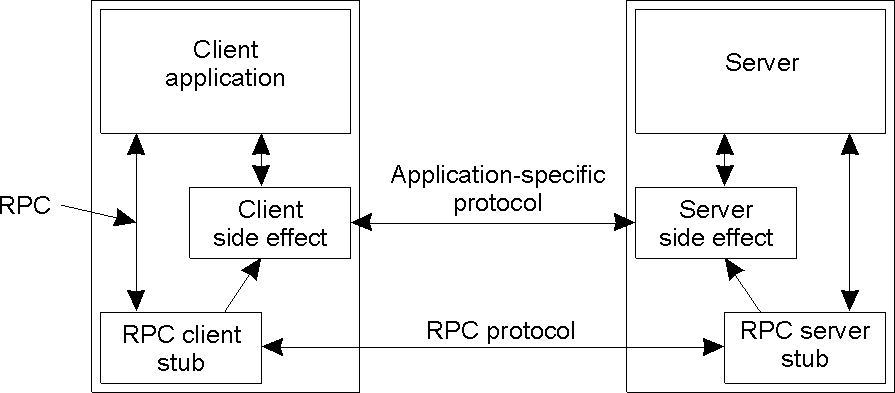
* The clients cache entire files locally.
* Cache coherence is maintained by the use of callbacks. Clients dynamically find files on server and cache location information.
* For security, token-based authentication and end-to-end encryption is used.



**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**

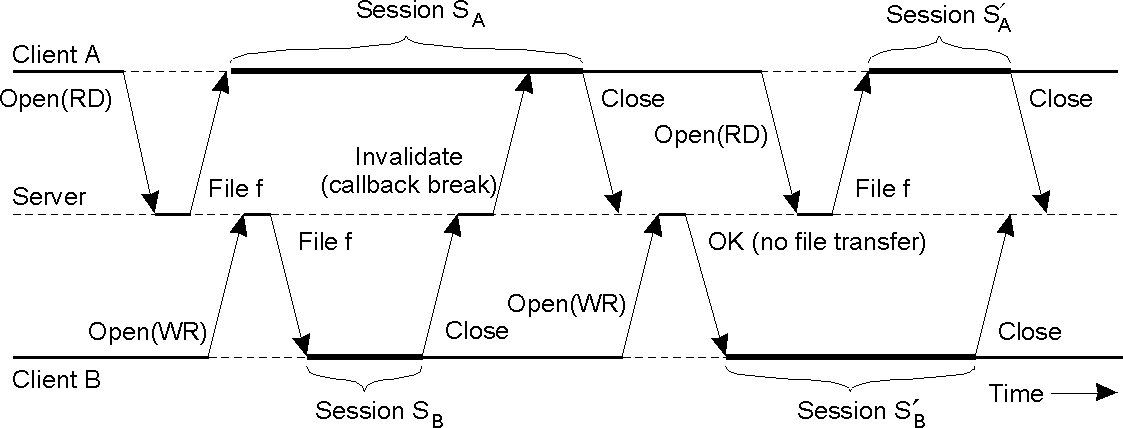
Coda maintains distinction between client and server processes. The clients are known as Venus processes and Server as Vice processes. The threads are non preemptive and operate entirely in user space. Low-level thread handles I/O operations.

**Caching in Coda**

* Cache consistency in Coda is maintained using **callbacks.**
* The Vice server tracks all clients that have a copy of the file and provide

**callback promise.** The tokens are obtained from vice server.

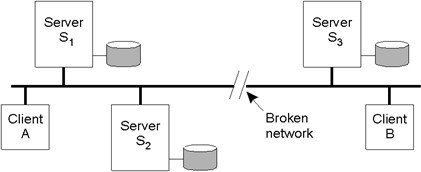
* It guarantee that Venus will be notified if file is modified
* Upon modification Vice server send invalidate to clients.



**Fig 4.20: Caching in Coda**

**Server Replication in Coda**

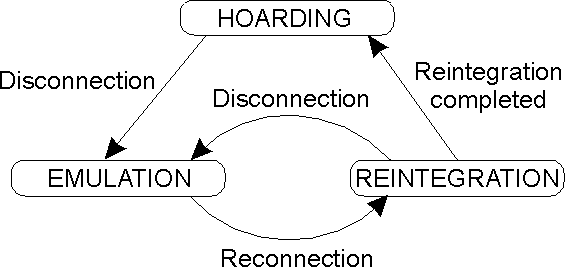
* The basic unit of replication is termed as volume.
* The Volume Storage Group (VSG) is set of servers that have a copy of a volume.
* The Accessible Volume Storage Group (AVSG) is set of servers in VSG that the client can contact .
* The Coda uses vector versioning:
  + One entry for each server in 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

1. **Define clock skew.**

**REVIEW QUESTIONS**

**PART – A**

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

1. **Define clock drift.**

Clock drift is the count time at different rates.

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

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

1. **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.
2. **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.
3. **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.

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

* time stamps of real-time clocks
* message passing
* round-trip time (local measurement)

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

 Cristian‟s algorithm

 Berkeley algorithm

 Network time protocol (Internet)

1. **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.

1. **What is NTP?**

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

1. **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
  + offsetoi : estimate for the actual offset between two clocks
  + delay di : true total transmission time for the pair of messages.

1. **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.

1. **Define logical clock.**

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

1. **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.

1. **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.

1. **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.

1. **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.

1. **What is consistent cut?**

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

1. **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.

1. **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.

1. **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.

1. **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

1. **What are the features of the reliable failure detectors?**

* Unsuspected or failure
* They are present in synchronous system

1. **Define distributed mutual exclusion.**

Distributed mutual exclusion provide critical region in a distributed environment.

1. **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

1. **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.

1. **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.

1. **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.

1. **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.

1. **What is election algorithm?**

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

1. **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

1. **What is bully algorithm?**

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

1. **What are the messages in Bully algorithm?**

There are three types of messages:

1. 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
2. Answermessage: This is sent in response to an election message.
3. Coordinatormessage: This is sent to announce the identity of the elected process.
4. **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.

1. **List the methods to ensure serializability.**

There are three ways to ensure serializability:

* Locking
* Timestamp ordering
* Optimistic concurrency control

1. **Give the advantages of nested transactions.**

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

1. **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.

1. **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

1. **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.

1. **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.

1. **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.

1. **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.

1. **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.
2. **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.

1. **Define phantom deadlock.**

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

1. **What is replication?**

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

1. **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.

1. **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.