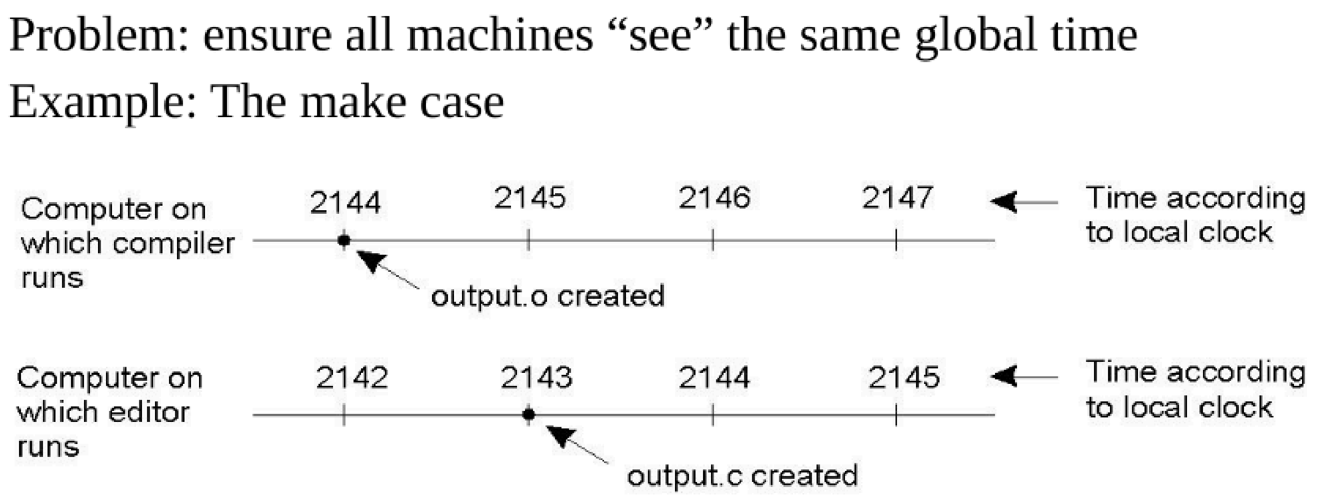
05.Synchronization

Synchronization in a Distributed System: an Introduction

The problem of synchronizing concurrent activities arises also in non-distributed systems
However, distribution complicates matters:

- Absence of a global physical clock
- Absence of globally shared memory
- Partial failures
  In these lectures, we study distributed algorithms for:
- Synchronizing physical clocks
- Simulating time using logical clocks & preserving event ordering
- Mutual exclusion
- Leader election
- Collecting global state & termination detection
- Distributed transactions
- Detecting distributed deadlocks
  Time plays a fundamental role in many applications:
- Execute a given action at a given time
- Time stamping objects/data/messages enables reconstruction of event ordering
  - File versioning
  - Distributed debugging
  - Security algorithms



Problem: ensure all machines “see” the same global time

Example: The make case

Computer on which compiler runs
2144
2145
2146
2147
Time according to local clock
output.o created
Computer on which editor runs
2142
2143
2144
2145
Time according to local clock
output.c created

Time

Up to 1940 time was measured astronomically: 1 second = 1/86400th of a mean solar day (the mean time interval between two consecutive transits of the sun), but the problem is that Earth is slowing down. Since 1948, time is measured physically: 1 second = 9,192,631,770 transitions of an atom of Cesium 133.

Synchronizing physical clocks

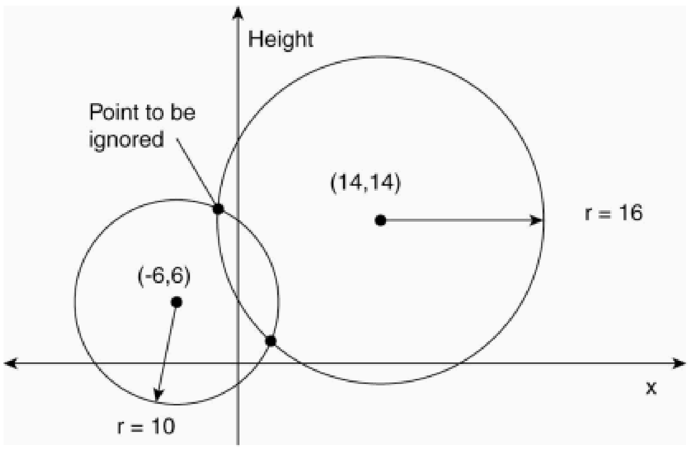
The PC use the GPS to synchronize clocks. In general clocks have a drifts, the clocks in pc doesn't measure time but it is like a metronome. You can do synchronization against the global clock or between nodes clocks. To do so you can go back(bad as all the algorithm assume advancing, instead you slow down the faster till the clocks are synchronzed) or you can go forward with one of the clock time.

GPS

Basic idea: get an accurate account of time as a side effect of GPS

How GPS works:

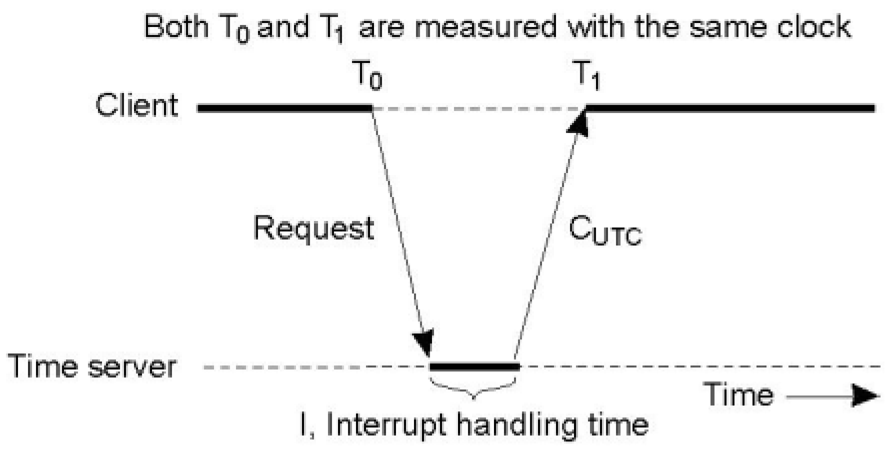
- Position is determined by triangulation from a set of satellites whose position is known
- Distance can be measured by the delay of signal
- But satellite and receiver clock must be in sync
- Since they are not we must take clock skew into account



We need the satellite to not skew in time, we connect them before leaving to avoid skewing. Usually they also use atomic clocks. The hypothesis is that the device clock is so synchronized with the satellite clock that the distance delay can be ignored. We have three incognita: x position, y position and time. We need at least three satellite to determine the position and the perfect time of the receiver. This is what you want. The precision is so high that the error is caused in the HW line length or delay.

Simple algorithms: Cristian’s (1989)

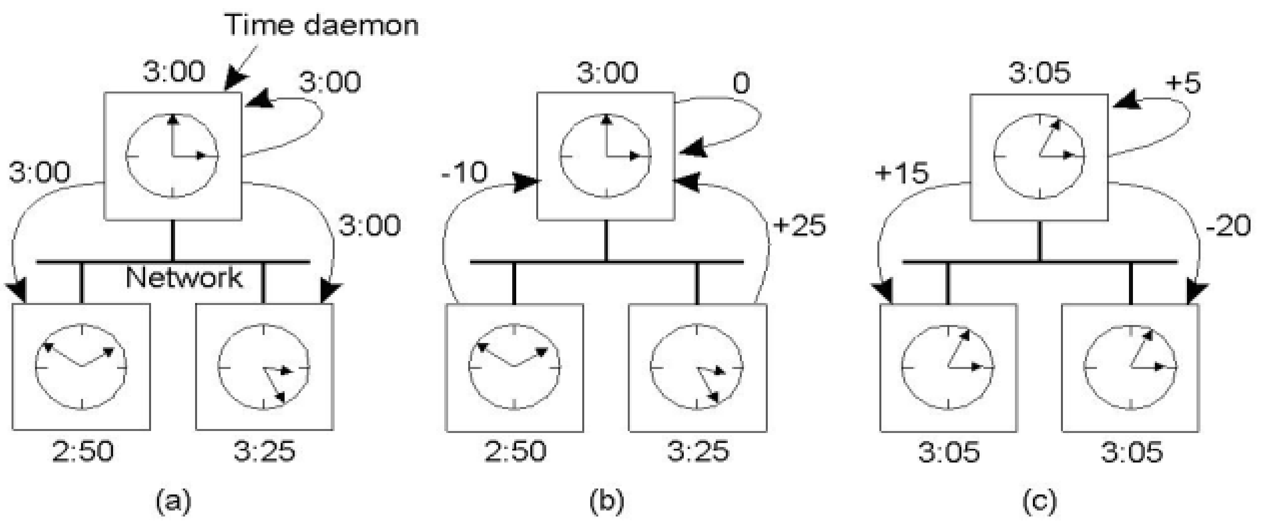
Problems with GPS is that it cost a lot and work only in open space.
The Cristian's algorithm only leverage the network: it uses the network time to synchronize the nodes. Usually it is used the server time clock with a minimum delay to take in count the time needed for the synchronization/message travel time. The assumption is that the network is omogeneous(time to go = time to come back).


Both T_0 and T_1 are measured with the same clock
T_0
T_1
Client
Request
C_{UTC}
Time server
Time
I, Interrupt handling time

$$T_{round} = T_1 - T_0 - I$$

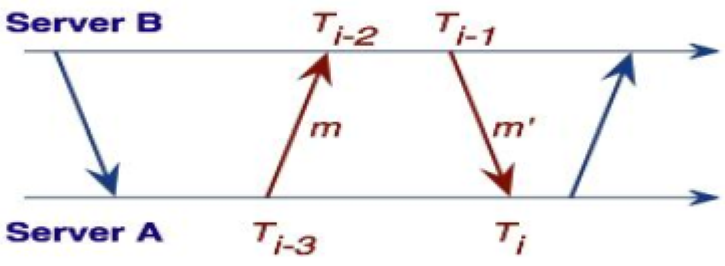
Simple algorithms: Berkeley (1989)

There is an active time server that ask periodically to the nodes what time is it and then compute an average and gives back the adjustement



Network Time Protocol (NTP)

All the machines on a large network are connected in a tree way and each machine ask the father for the perfect time. The machine on the root have it. When you start a machine you add the information of the NTP network that is in the same subnetwork/same country and depending on the network NTP use different protocol: in LAN NTP periodically send time update otherwise it operate similar to Cristian's but operate simmetrically: if a machine want ot synchronize with another one it send a message. This message contain the starting time(based on its own clock). The receiver use its own clock and answer to the message after some time. This operation is repeated a second time and then the host can calculate the time knowing the 4 number:



- $T_{i-2} = T_{i-3} + t + o$ and $T_i = T_{i-1} + t' – o$
- Total transmission time: $d_i = t + t' = T_{i-2} – T_{i-3} + T_i – T_{i-1}$
- $o_i = (T_{i-2} – T_{i-3} + T_{i-1} – T_i)/2$

- o = oi + (t’-t)/2
- $o_i – d_i/2 = o_i + (t' - t)/2 – t' \leq o \leq o_i + (t' - t)/2 + t = o_i + d_i/2$
  Thus o_i is an estimate of the offset and d_i is a measure of the accuracy of this estimate. We continue until the estimate is good enough. In practice NTP allows to synchronize in networks as large as the internet clocks with error to 10^{15} milliseconds.

Logical time: Scalar clocks and Vector clocks

Usually the order of happening is what is important in application for correctness. We care if two events are in causal order or sequence order. We want to respect the causal order(typically). If there is no intercation is not necessary to keep the event in synch as there isn't an interaction.

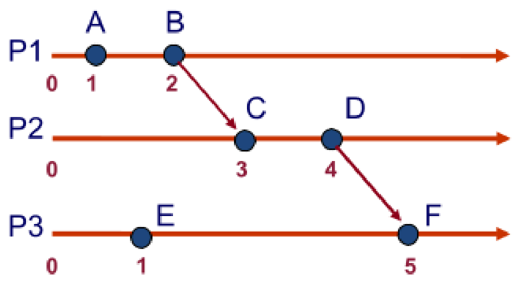
Scalar CLock

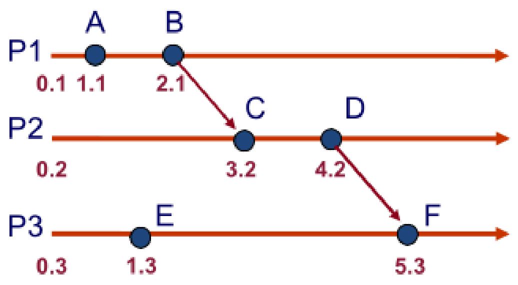
Events: things that happens in the distributed system and it is relevant for it
Let define the happens-before relationship $e \rightarrow e'$, as follows:

- If event e and e' occur in the same process and e occur before e' then e happens before e'.
- Sending of a message: if e is the sender of a message and e' is the receiver of the same message and e and e' happens in different process, e must happen before e'. Every type of communication require some time and order for emitting/receiving the communication.
- This relation must be transitive
  This relationship define a partial order between events as it cannot say anything about uncorrelated events/events that aren't communicating. If there is no interaction between events there can't be causality. The happens before relationship is the best approximation of causal relationship.

L. Lamport. "Time, clocks, and the ordering of events in a distributed system". Communications of the ACM, 21(7):558-565, July 1978

- Each process p_i keeps a logical scalar clock L_i
  - L_i starts at zero
  - L_i is incremented before p_i sends a message
  - Each message sent by p_i is timestamped with L_i
  - Upon receipt of a message, p_i sets L_i to: MAX(msg timestamp, L_i) + 1
- It can easily be shown, by induction on the length of any sequence of events relating two events e and e', that: $e \rightarrow e' \Rightarrow L(e) < L(e')$
- Note that only partial ordering is achieved. Total ordering can be obtained trivially by attaching process IDs to clocks





The only thing we can say is that if two events are in the happens before relationship one has a Lamport clock smaller than the other. If you order events in Lamport time you are assuring that the happens before relation it respected. Lamport clock define the global order. The acknowledgment are fundamental as they permit to know what message I can receive from each process as they also contain the Lamport clock. Message are FIFO. You have to acknowledge everyone(when you receive one you have to wait for each node to acknowlegde). Message and acknowledge MUST be in broadcast.A message is delivered to the application **only when it is at the highest in the queue** and **all its acks have been received**

Vector Clocks

Each process maintain in a vector a value for each process including itself.
The value in the vector refers the view on the events that happened in the processes. Initially the value is 0. When the process send a message attach a timestamp $t = V_i$ and increment V_i just before sending the message. When the process receive a message containing t it set $V_I = max(V_i[j], t[j])$ for all $j \neq i$ and then increments $V_i[i]$
Definitions (partial ordering):

- V = V' $\iff$ V[j] =V'[j], for all j
- V ≤ V' $\iff$ V[j]$\leq$V'[j], forall j

- V < V' $\iff$ V$\leq$ V' $\wedge$ V $\neq$ V'
- V || V' $\iff$ $\neg(V < V') \wedge \neg(V' < V)$
  An isomorphism between the set of partially ordered events and their timestamps (i.e., vector clocks)
  Determining causality:
- e $\rightarrow$ e' $\iff$ V(e) < V(e')
- e || e' $\iff$ V(e) || V(e')
  By looking only at the timestamps we are able to determine whether two events are causally related or concurrent.

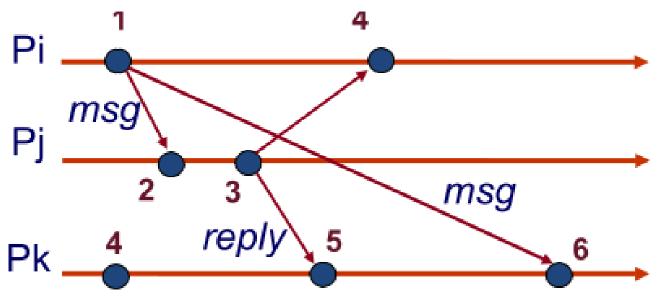
Vector clocks for causal delivery

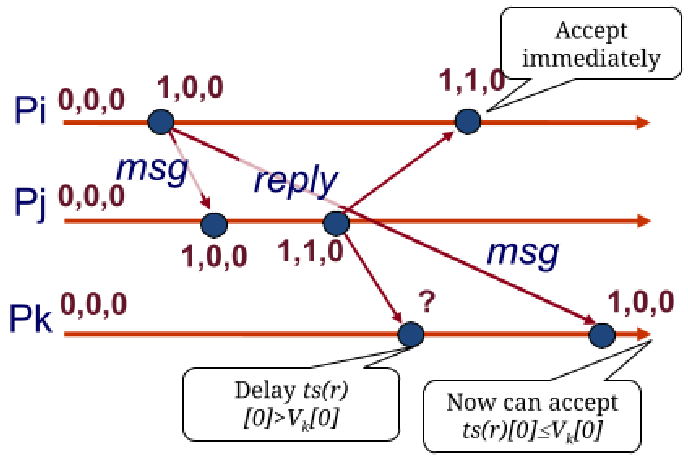
We can use a variation of vector clock where you only increment the position when you send a message, when you receive a message you only merge and don't consider internal events.

A slight variation of vector clocks can be used to implement causal delivery of messages in a totally distributed way

Example: bulletin boards

- Messages and replies sent (using reliable, FIFO ordered, channels) to all the boards in parallel
- Need to preserve the ordering only between messages and replies
- Totally ordered multicast is too strong
  - If M1 arrives before M2, it does not necessarily mean that the two are related
- Using vector clocks:
  - Variation: Increment clock only when sending a message. On receive, just merge, not increment
  - Hold a reply until the previous messages are received:
    - $ts(r)[j] = V_k[j]+1$
    - $ts(r)[i] \leq V_k[i]$ for all $i \neq j$





Mutual Exclusion

Algorithm that tries to avoid interference between two activities and tries to guarantee consistency on shared resource access.
Requirements:

- Safety property: At most one process may execute in the critical section at a time

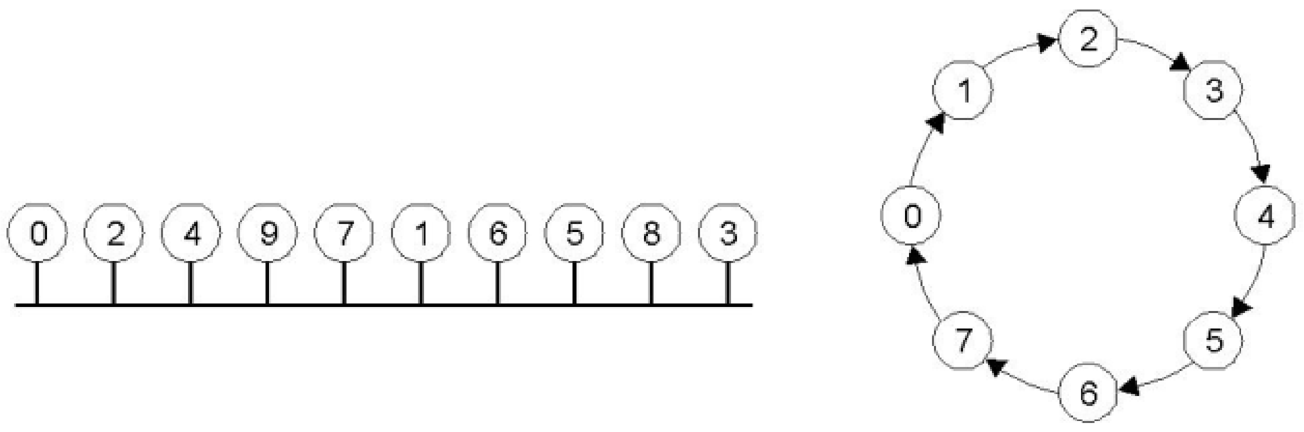
- Liveness property: All requests to enter/exit the critical section eventually succeed (no deadlock, no starvation)
- Optional: If one request happened-before another, then entry is granted in that order
  We assume reliable channels and processes.
  We can assure this in various ways:
- Centralize the coordination as we let the centralize controller to take action or ask it to have access to the process. This is bad as there is only one point of failure: the centralized controller
- Mutual exclusion with scalar clock

Mutual exclusion with scalar clock

To request access to a resource:

- A process P_i multicasts a resource request message m, with timestamp T_m , to all processes (including itself)
  Upon receipt of m, a process P_J:
- If it does not hold the resource and it is not interested in holding the resource, P_j sends an acknowledgment to P_i
- If it holds the resource, P_j puts the requests into a local queue ordered according to T_m (process ids are used to break ties)
- If it is also interested in holding the resource and has already sent out a requests, P_j compares the timestamp Tm with the timestamp of its own requests
  - If Tm is the lowest one, P_j sends an acknowledgement to P_i, otherwise it put the request into the local queue above
  If Tm is the lowest one, P_j sends an acknowledgement to P_i, otherwise it put the
  request into the local queue above
  On releasing the resource, a process P_i acknowledges all the requests queued while using the resource
  A resource is granted to P_i when its request has been acknowledged by all the other processes
  This protocol is fair as it use the Lamport clock so it is fair, it doesn't create deadlock.

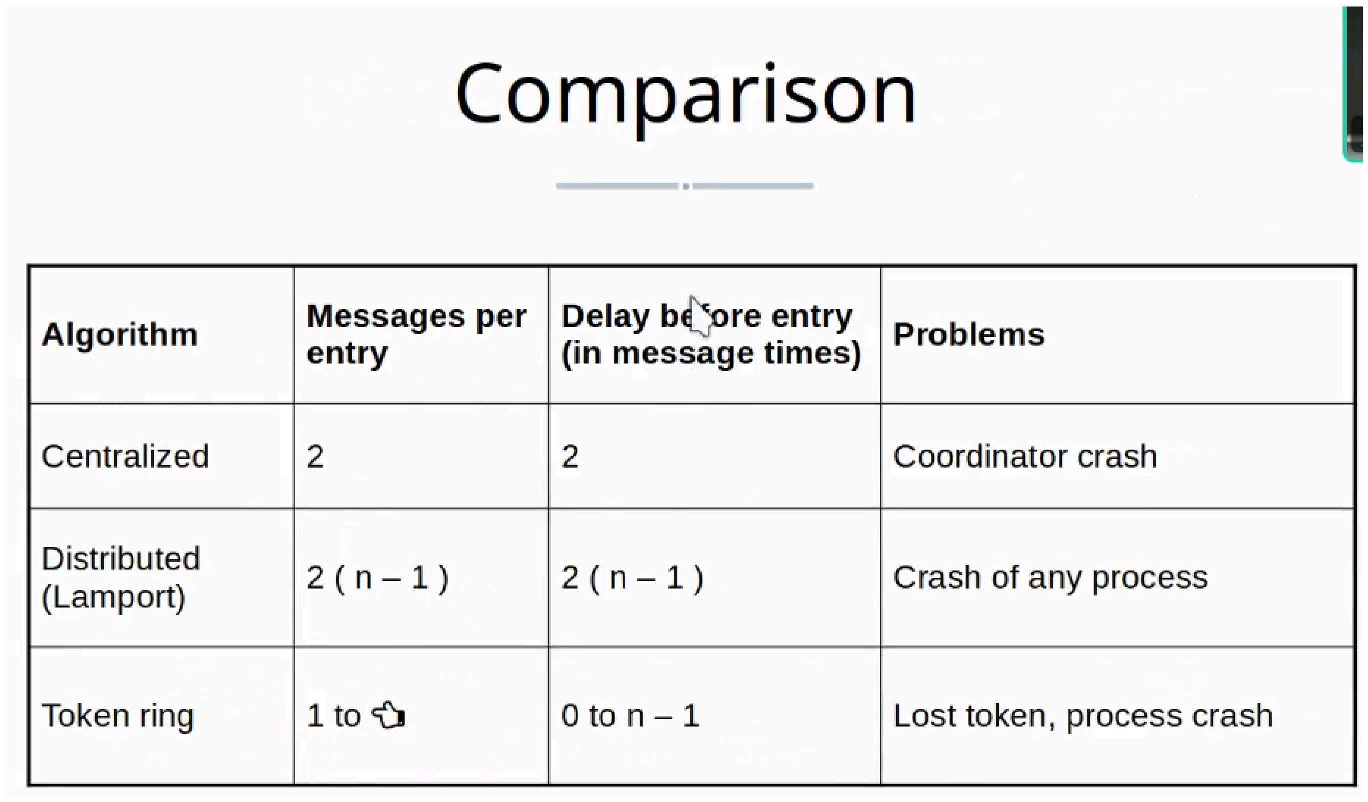
A token ring solution



(a) (b)

Processes are logically arranged in a ring, regardless of their physical connectivity

- At least for the purpose of mutual exclusion
  Access is granted by a token that is forwarded along a given direction on the ring
  A process not interested in accessing the resource forwards the token
- Resource access is achieved by retaining the token
- Resource release is achieved by forwarding the token
  This approach is not fair as it can violate the happens before relationship



Comparison

Algorithm	Messages per entry	Delay before entry (in message times)	Problems
Centralized	2	2	Coordinator crash
Distributed (Lamport)	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to [illegible]	0 to n – 1	Lost token, process crash

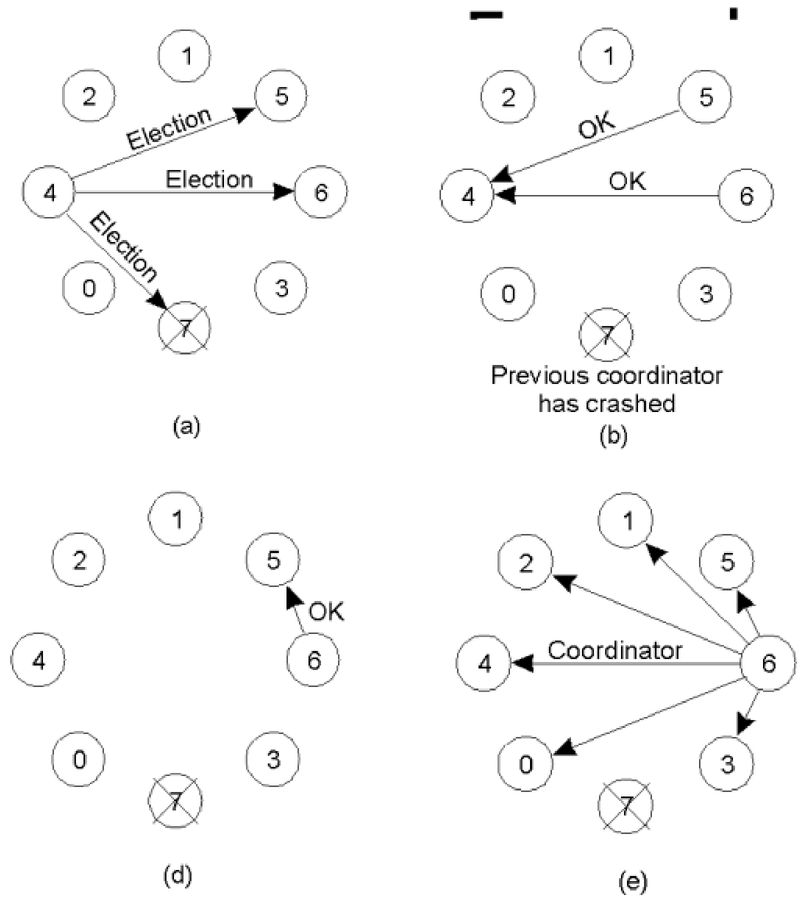
Leader Election

We need to guarantee to have only one leader and that everyone is informed on who is the leader. The system must be closed and the various nodes need to be distinguishable by an ID. If the leader crash we elect as leader the one with the highest ID.

The bully election algorithm

Additional assumptions

- Reliable links
- It is possible to decide who has crashed (synchronous system)
  Algorithm:
- When any process P notices that the actual coordinator is no longer responding requests it initiates an election
- P sends an ELECT message, including its ID, to all other processes with higher IDs
- If no-one responds P wins and sends a COORD message to the processes with lower IDs
- If a process P’ receives an ELECT message it responds (stopping the former candidate) and starts a new election (if it has not started one already)
- If a process that was previously down comes back up, it holds an election
  - If it happens to be the highest-numbered process currently running it wins the election and takes over the coordinator’s job (hence the name of the algorithm)

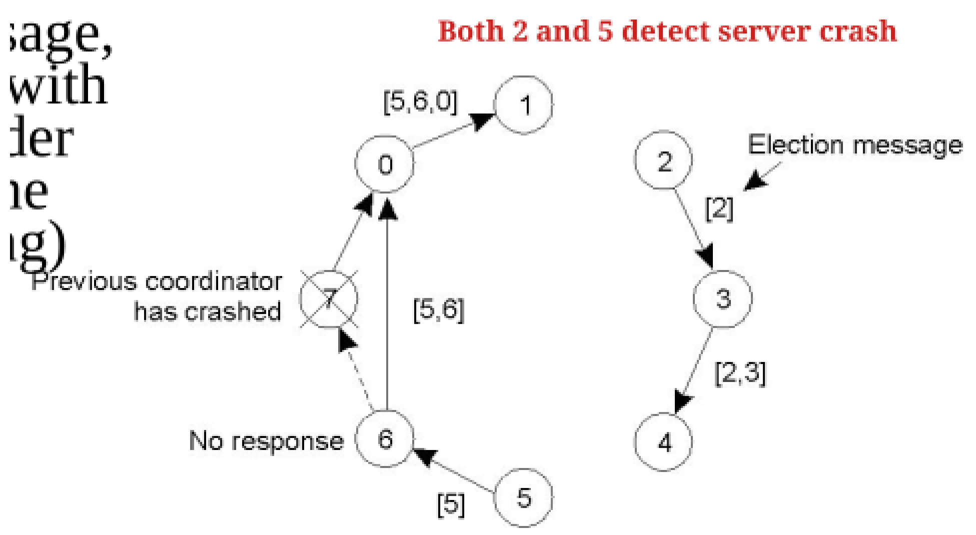


- The leader, 7, fails
  - Process 4 (noticing the failure of 7) holds an election
  - Process 5 and 6 respond, telling 4 to stop
  - Now 5 and 6 each hold an election
  - 6 tells 5 to stop
  - 6 wins and tells everyone
- When 7 comes back it holds an election and wins

A ring-based algorithm

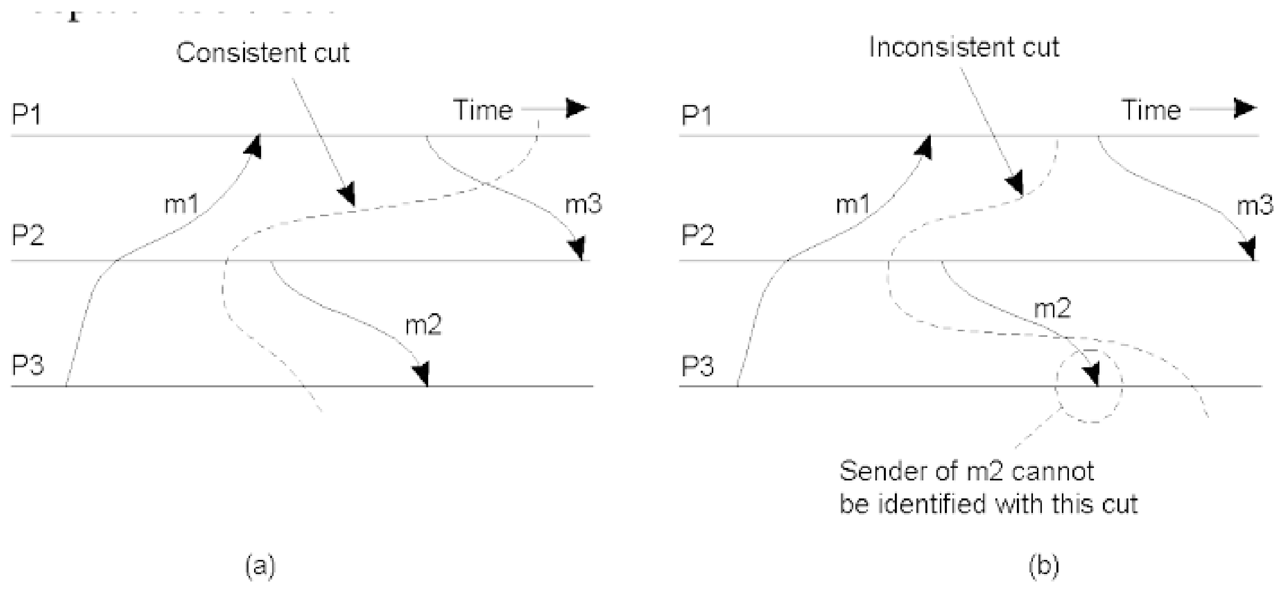
Assume a (physical or logical) ring topology among nodes
When a process detects a leader failure, it sends an ELECT message containing its ID to the closest alive neighbor
Upon receipt of the election message a process P:

- If P is not in the message, add P and propagate to next alive neighbor
- If P is in the list, change message type to COORD, and re-circulate
  On receiving a COORD message, a node considers the process with the highest ID as the new leader (and is also informed about the remaining members of the ring)
  Multiple messages may circulate at the same time
- Eventually converge to the same content



Collecting Global State and Termination Detection

Nobody can have a global view of the system as it requires to see multiple processes and instantly.
In a real distributed system we can use a tool called Cut and use a distributed snapshot in which we take picture of different part of the system at different time and use them to have a "global view". This collection of picture is acceptable if the channels have different speed and the situation represented can happen.



Formally a cut of a system S composed of N processes $p_1, \ldots, p_n$ can be defined as the union of the histories of all its processes up to a certain event
$C = h_1^{k_1} \cup h_2^{k_2} \cup \ldots \cup h_n^{k_n}$
where $h_i^{k_i} = < e_i^0, e_i^1 \ldots e_i^{ki} >$
A cut C is consistent if for any event e it includes, it also includes all the events that happened before e. Formally: $\forall e, f : e \in C \land f \rightarrow e \Rightarrow f \in C$

Distributed snapshot

Assume FIFO, reliable links/nodes + strongly connected graph

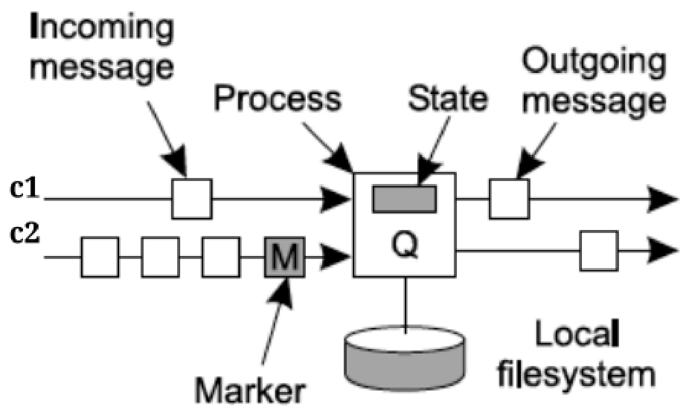
Any process p may initiate a snapshot by

- Recording its internal state
- Sending a token on all outgoing channels.
  - This signals a snapshot is being run
- Start recording a local snapshot
  - I.e., record messages arriving on every incoming channel

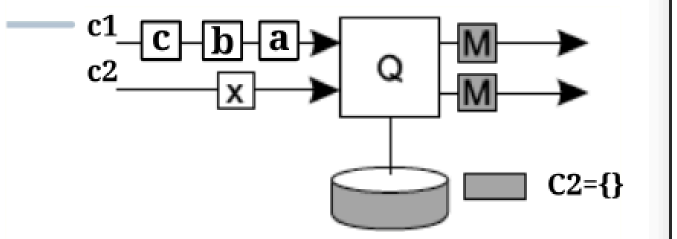
Upon receiving a token, a process q

- If not already recording local snapshot
  - Records its internal state
  - Sends a token on all outgoing channels
  - Start recording a local snapshot (see above)
- In any case stop recording incoming message on the channel the token arrived along

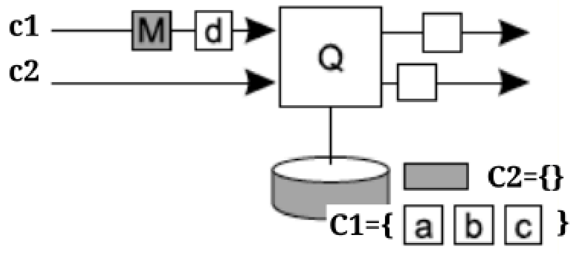
- Recording messages
  - If a message arrives on a channel which is recording messages, record the arrival of the message, then process the message as normal
  - Otherwise, just process the message as normal
- Each process considers the snapshot ended when tokens have arrived on all its incoming channels
  - Afterwards, the collected data can be sent to a single collector of the global state



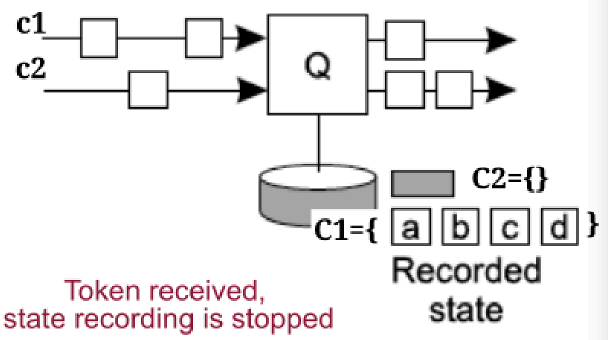
Normal processing, first marker about to be received



Token just received for the first time, state saved, token forwarded to outgoing links, begin recording messages



Recording of messages from the incoming links from which a token has not been received, yet



Token received, state recording is stopped

Characterizing the observed state

Theorem
The distributed snapshot algorithm selects a consistent cut
Proof
Let e_i and e_j be two events occurring at p_i and p_j, respectively, such that $ei \rightarrow e_j$
Suppose e_j is part of the cut and e_i is not. This means e_j occurred before p_j saved its state, while e_i occurred after p_i saved its state
If p_i = p_j this is trivially impossible, so suppose $pi \neq p_j$
Let $m_1, \ldots m_h$ be the sequence of messages that give rise to the relation $e_i \rightarrow e_j$
If e_i occurred after p_i saved its state then p_i sent a marker before e_i, so ahead of $m_1, \ldots m_h$
By FIFO ordering over channels and by the marker propagating rules it results that p_j received a marker ahead of $m_1, \ldots m_h$
By the marker processing rule it results that p_j saved its state before receiving $m_1, \ldots m_h$, i.e., before e_j occurred, which contradicts our initial assumption
A consistent Cut is a good way to bring back the system to a working state if the system crashed.
The distributed snapshot algorithm does not require blocking of the system as the collecting is interleaved with processing. We need to identify the various marker of the snapshots to allow the distinction of different snapshot on disk.

Termination detection

Each job need to have terminated its process and there are no messages in transit. A possible solution is take a distributed snapshot and see if all the process are idle and the channels are empty.
A possibility to check if a distributed algorithm as terminated is to use diffusing computation where initially all processes are idle except the init process. As process is activated only by a message sent to it. Termination condition: when processing is complete at each node, and there are no more messages in the system.

Dijkstra-Scholten termination detection

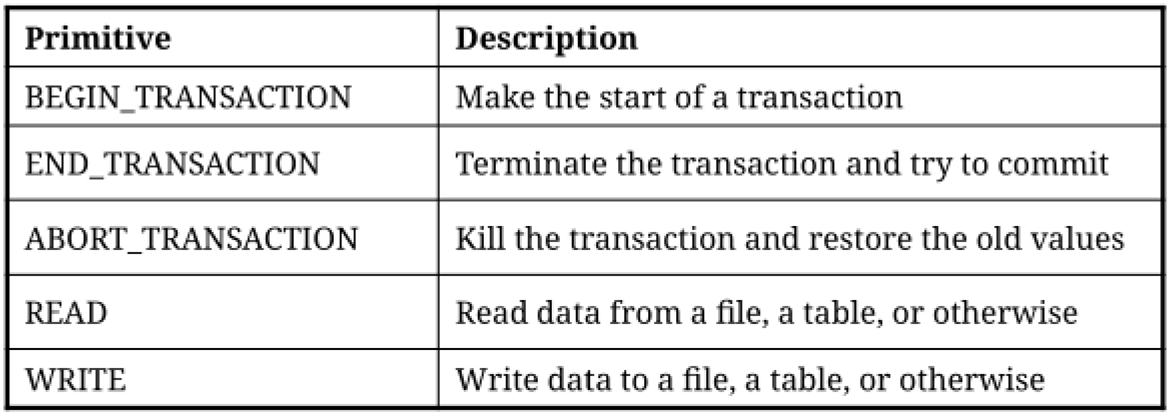
Works for diffusing computations
Key concepts:

- Create a tree out of the active processes
- When a node finishes processing and it is a leaf, it can be pruned from the tree
- When only the root remains, and it has completed processing, the system has terminated
  Each node keeps track of nodes it sends a message to (those it may have

woken up), its children
If a node was already awake when the message arrived, then it is already part of the tree, and should not be added as a child of the sender
When a node has no more children and it is idle, it tells its parent to remove it as a child
The computation can considered completed when the top node children have completed execution and the top node complete execution. **There is the need to not create cycle in the tree**

Distributed transactions and Detecting distributed deadlocks

A transaction(group of operations that respect ACID properties and terminate with abort or commit, either the transaction commit all the operations or cancel all the operations results)

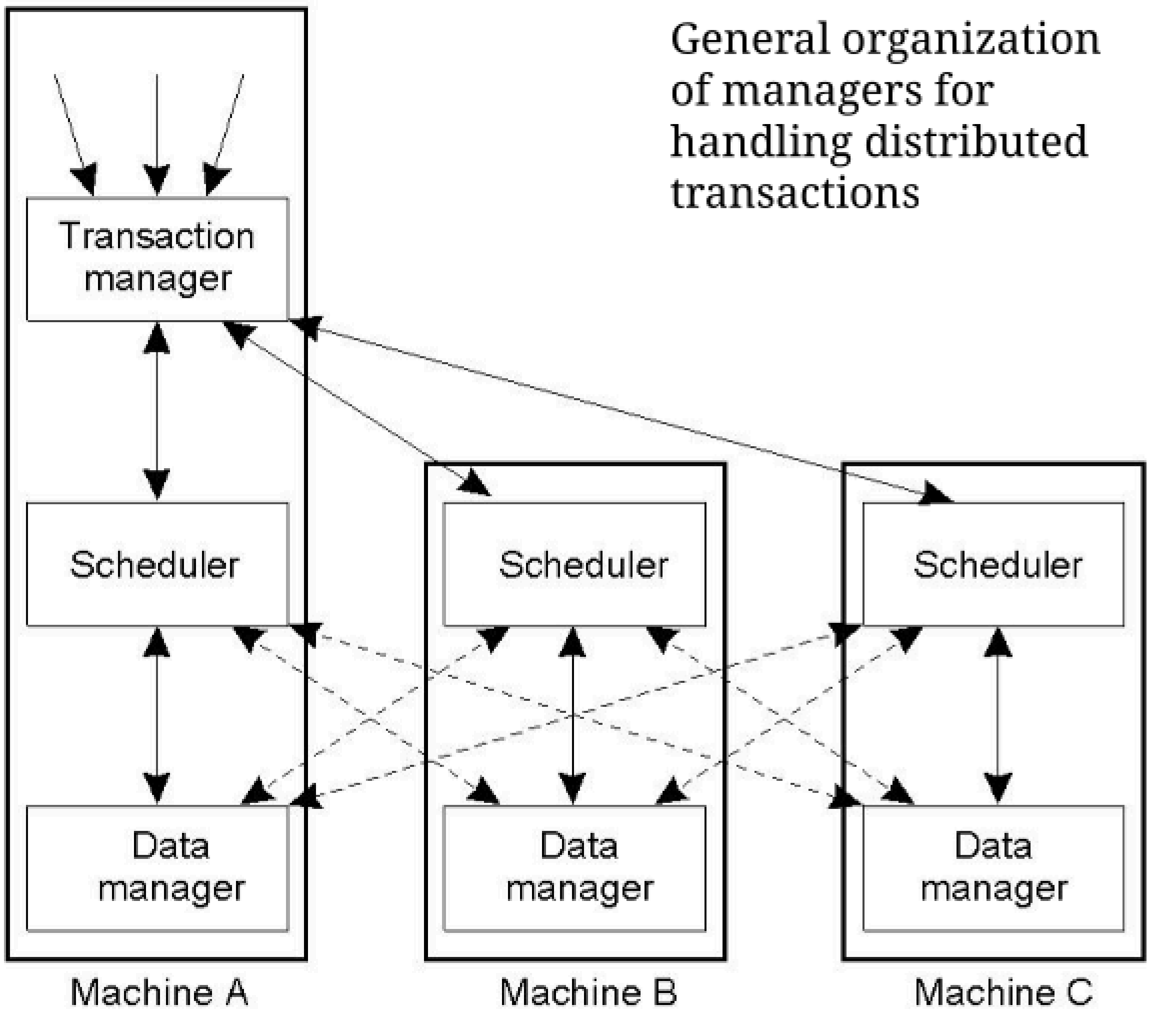


Primitive	Description
BEGIN_TRANSACTION	Make the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

All-or-nothing (ACID properties):

- Atomic: to the outside world, the transaction happens indivisibly
- Consistent: the transaction does not violate system invariants
- Isolated (or serializable): concurrent transactions do not interfere with each other
- Durable: once a transaction commits, the changes are permanent
  A distributed transaction can be:
- Flat:ACID transactions as defined earlier
- Nested: Constructed from sub-transactions. Sub-transactions can be undone once committed (if their parent transaction aborts): durability applies only to top-level transactions. Sub-transactions conceptually operate on a private copy of the data:
  - If it aborts the private copy disappears

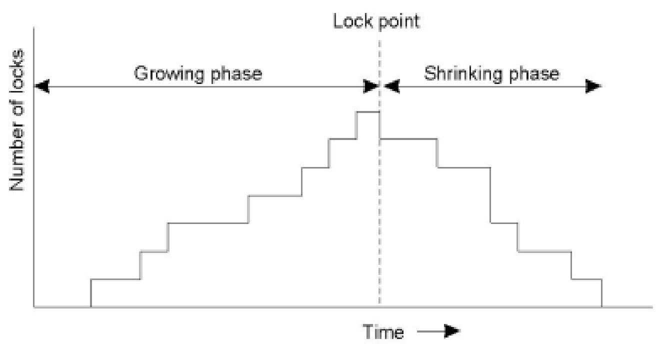
  - If it commits, the modified private copy is available to
  - the next sub-transaction
    Typically, each sub-transaction runs on a different host, providing a given service
- Distributed: Accounts for data distribution. Essentially flat transactions on distributed data. Instead, nested transactions are hierarchical. Need distributed locking. The transaction manager involves all the machines at the same time.
  Atomicity can be guaranteed easily by having an index mapping the logical file on block on disk and use an approch called private workspace: when you need to modify a file you create a copy of the index and operate on it adding new blocks and modify this new block until you want to commit and discard teh old index. If you want to roll-back you simply discard the copy(pessimistic approach). Writeahead log work on the file directly and use a log(transaction that made the change, which file/block and the old and new value) in the case you need to roll-back the changes(optimistic approach)
  Concurrency can be controlled distinguishing three parts:
- Data manager: where you put the private workplace or the
- Transaction manager: Transforms high-level operations in scheduling requests
- Scheduler: schedule operations such that two transactions don't interfere with each others. Manage the locks/unlocks
  Each distributed transaction has a transaction manager, there can be one or more scheduler for each machine and a single data manager for machine. If there are multiple copies of data on multiple machines all of them need to be updated by the transaction.

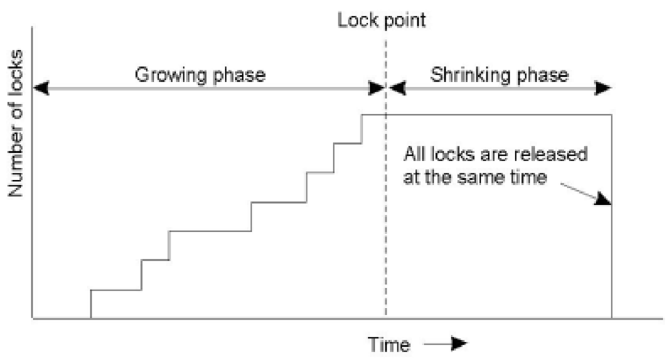


To permit the concurrent execution either the transactions are serial or we need to execute them in a way that the final result is equal as if they are serial.

Two-Phase Locking (2PL)

When a process need to access a piece of data it requires the lock on the data. Once a lock on some data is released it cannot be reacquired, so in a transaction there is a phase where you acquire all locks and a phase where you release all locks.This approach might create deadlock as the acquire phase is a growing phase.





Implementing 2PL:

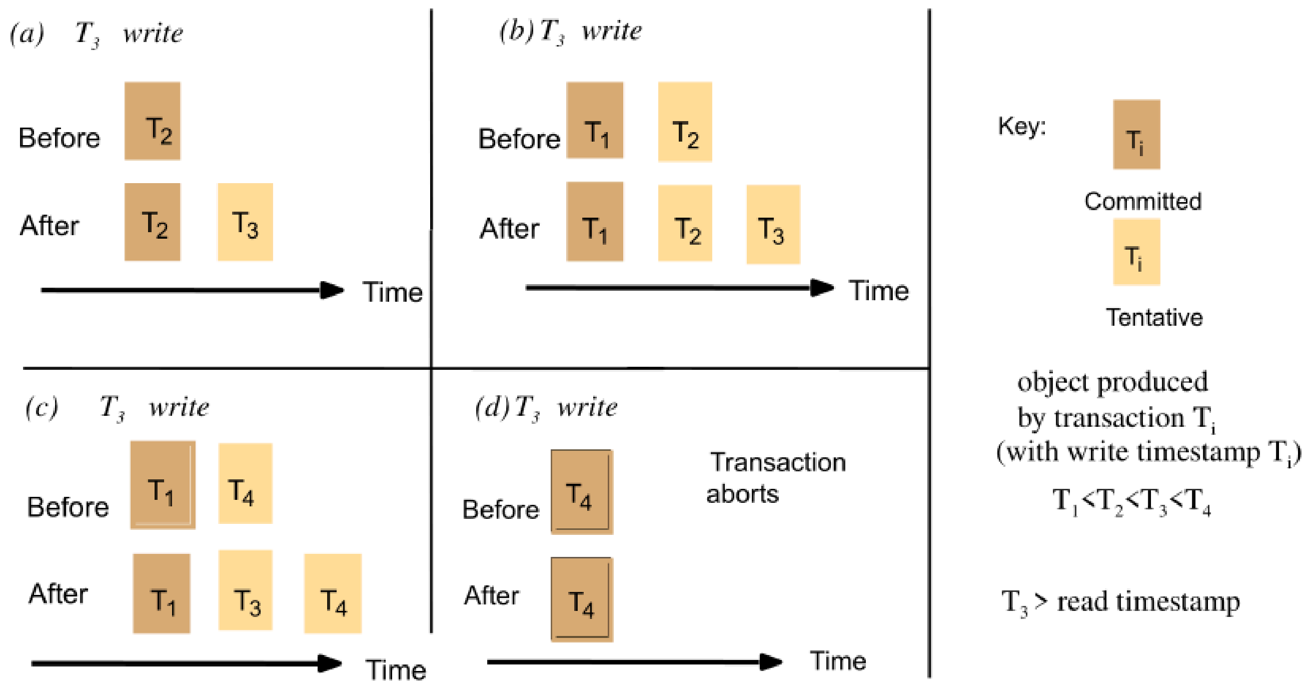
- Centralized 2PL:The transaction manager contacts a centralized lock manager, receives lock grant, interacts directly with the data manager, then returns the lock to the lock manager
- Primary 2PL: Multiple lock managers exist. Each data item has a primary copy on a host: The lock manager on that host is responsible for granting locks. The transaction manager is responsible for interacting with the data managers
- Distributed 2PL: Assume data may be replicated on multiple hosts.The lock manager on a host is responsible for granting locks on the local replica and for contacting the (local) data manager

Pessimistic Timestamp Ordering

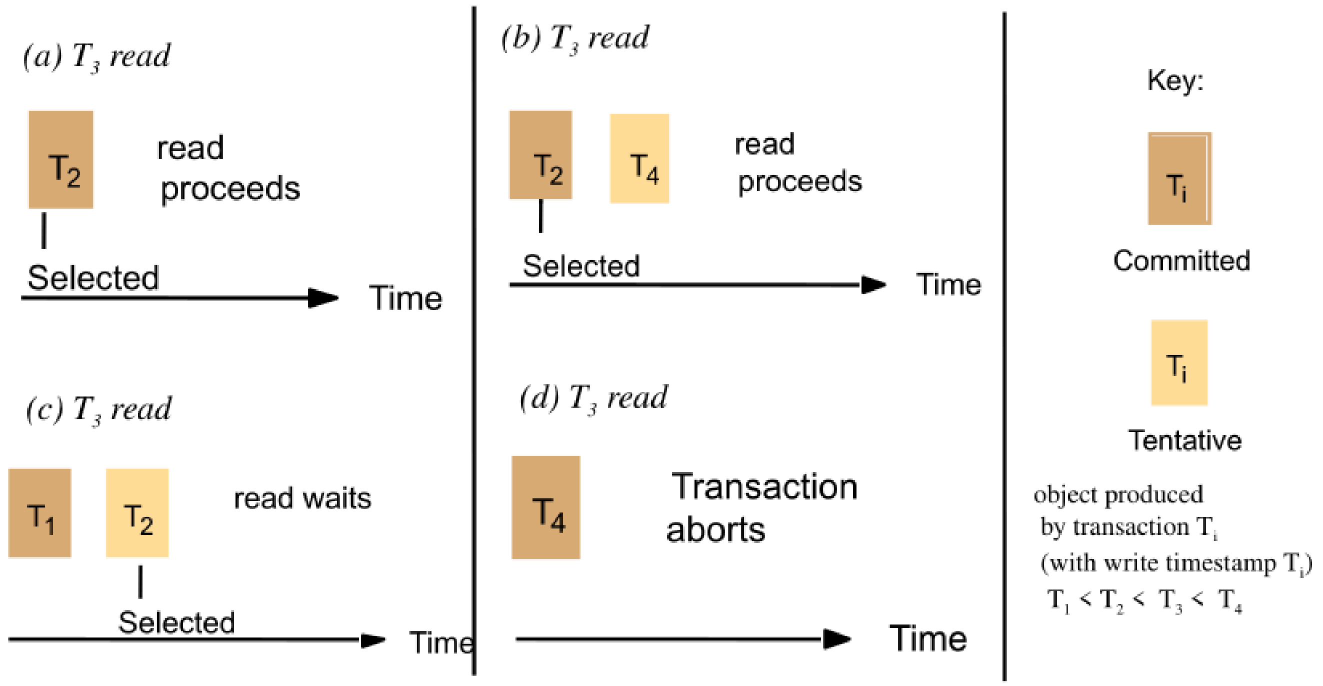
Tries to order the transaction in a way they seem to be serial.
Each transaction has a timestamp derived by a Lamport clock(to be fair, it only need to be different). Write operations on a data item x are recorded in tentative versions, each with its
own write timestamp $ts_{wr}(x_i)$, until commit is performed. Each data item x has also a read timestamp $ts_{rd}(x)$: That of the last transaction which read x.
The scheduler operates as follow:

- When receives write(T,x) at time=ts
  - If $ts > ts_{rd}(x)$ and $ts > ts_{wr}(x)$ perform tentative write x_i with timestamp $ts_{wr}(x_i)$
    else abort T since the write request arrived too late
- Scheduler receives read(T,x) at time=ts
  - If $ts > ts_{wr}(x)$
  Let x_{sel} be the latest version of x with the write timestamp lower than ts
  If x_{sel} is committed perform read on x_{sel} and set $ts_{rd}(x) = max(ts, ts_{rd}(x))$
  else wait until the transaction that wrote version x_{sel} commits or abort then reapply the rule
  else abort T since the read request arrived too late
  Aborted transactions will reapply for a new timestamp and simply retry
  Deadlock-free

Write operations and timestamps



Read operations and timestamps



Optimistic Timestamp Ordering

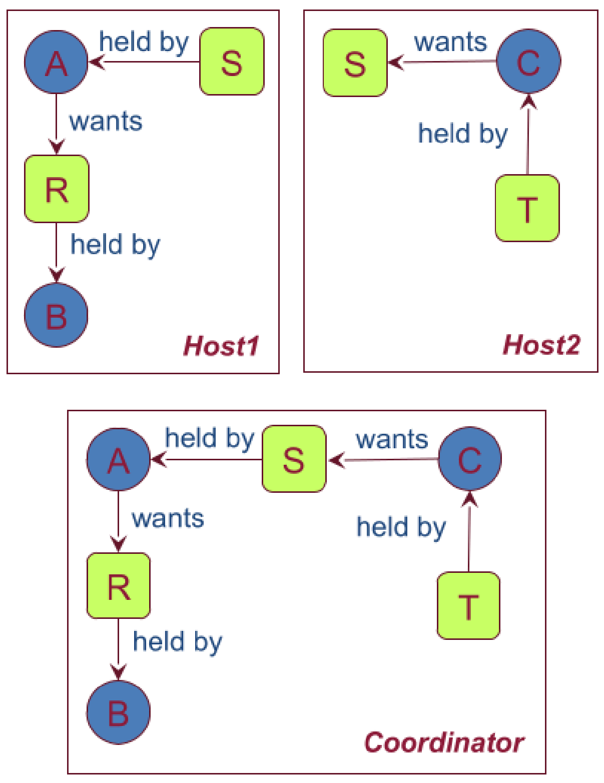
The idea is based on the assumption that conflicts are rare so instead of caring during the execution of transactions if they interact you let them execute and then watch what variable are touched and at what time. You abort all the transactions that operated on the same resource even if the transaction would commit. This is deadlock free, allows maximum parallelism. Under heavy load, there may be too many rollbacks.

Distributed deadlocks

Same concept as in conventional systems: but worse to deal with, since in a distributed system resources are spread out
Approaches:

- Ignore the problem: most often employed, actually meaningful in many settings
- Detection and recovery: typically by killing one of the processes
- Prevention
- Avoidance: never used in (distributed) systems, as it implies a priori knowledge about resource usage
  Distributed transactions are helpful: to abort a transaction (and perform a rollback) is less disruptive than killing a process

Centralized deadlock detection



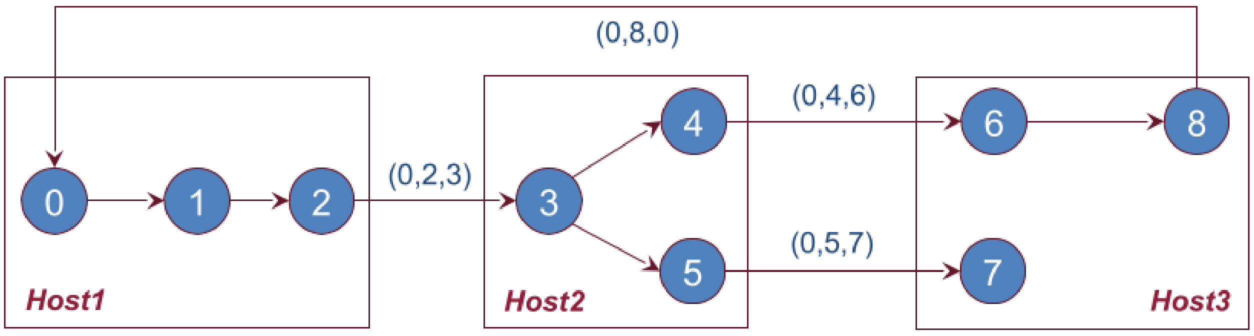
Each machine maintains a resource graph for its own resources and reports it to a coordinator
Options for collecting this information:

- Whenever an arc is added or deleted, a message is sent to the coordinator with the update
- Periodically, every process sends a list of arcs added or deleted since the last update
- Coordinator can request information on-demand
  None works well, because of false deadlocks
- For instance, if B releases R and acquires T and the coordinator receives data from host 2 before receiving data from host 1

Distributed deadlock detection

There is no coordinator in charge of building the global wait-for graph
Processes are allowed to request multiple resources simultaneously
When a process gets blocked, it sends a probe message to the processes holding resources it wants to acquire

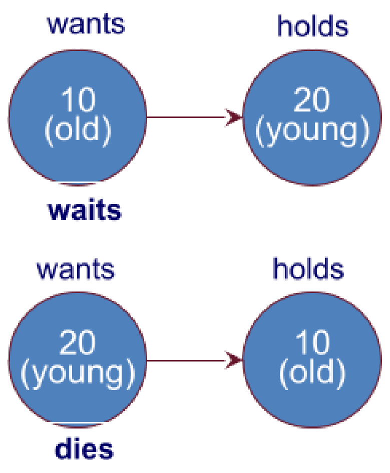
- Probe message: (initiator, sender, receiver)
- A cycle is detected if the probe makes it back to the initiator



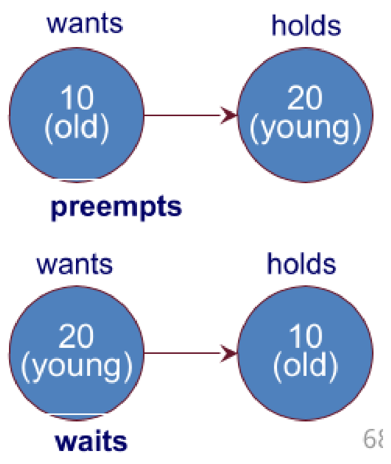
When a deadlock is detected we can recover from it either killing the initiator(more than one process may be unnecessarily aborted if more than one initiator detects the loop). Alternatively the initiator picks the process with the higher identifier and kills it. Requires each process to add its identifier to the probe.

Distributed prevention

It make deadlock impossible by design still using a locking mechanism. The idea is that if a process want something that is locked by another process you wait only if your timestamp is lower than the one of the process that is using the resource otherwise you get aborted. Following chain of waiting processes, the timestamps will always increase (no cycles). In principle, could have the younger wait. The younger process continue to apply-get killed until is old enough



If a process can be preempted, i.e., its resource taken away, an alternative can be devised (wound-wait algorithm):Preempting the young process aborts its transaction, and it may immediately try to
reacquire the resource, but will wait. In wait-die, young process may die many times before the old one releases the resource: here, this is not the case


wants
holds
10 (old)
20 (young)
preempts
wants
holds
20 (young)
10 (old)
waits