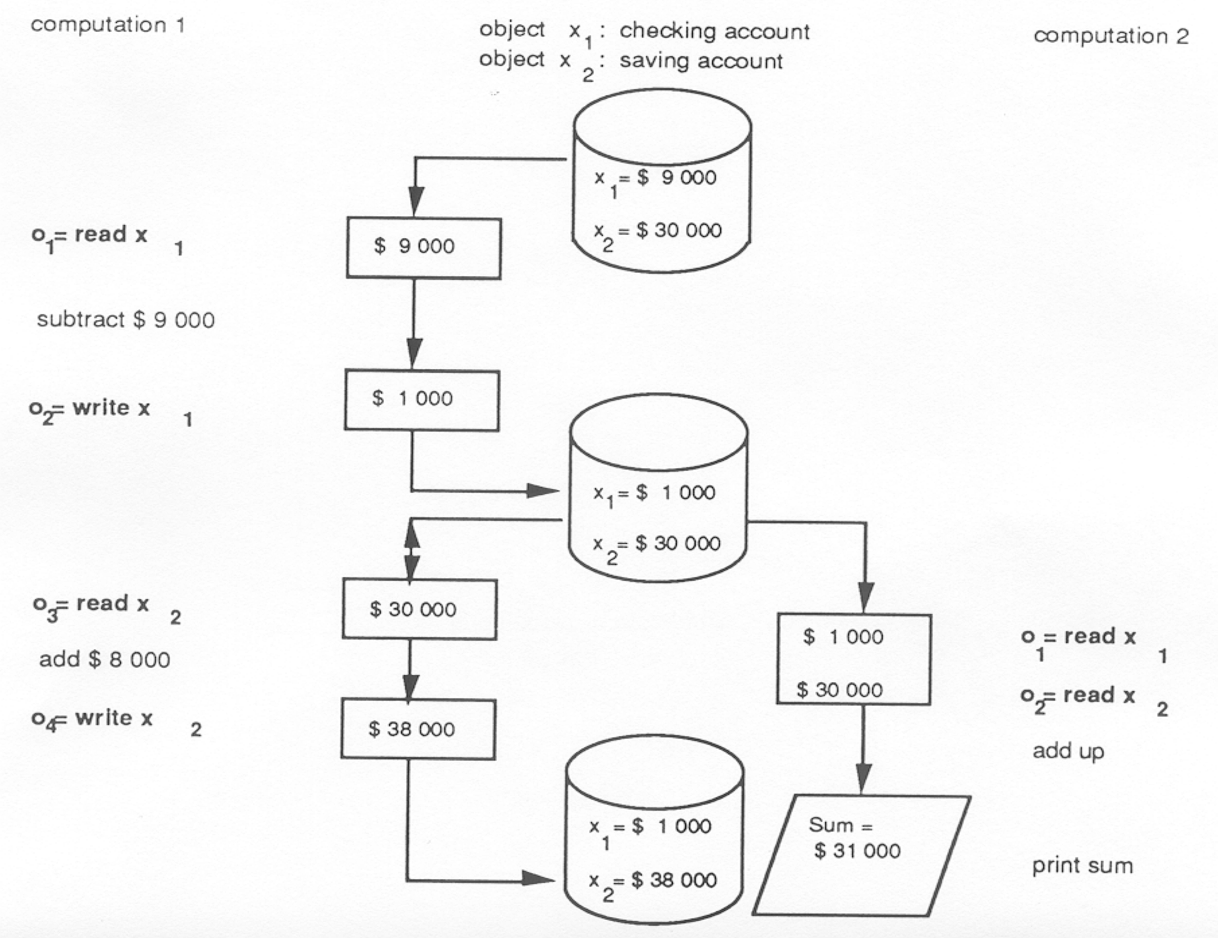
Distributed System Paradigms (37)

Incorrect execution of the schedule [$o_1^1(x_1)$, $o_2^1(x_1)$, $o_1^2(x_1)$, $o_2^2(x_2)$, $o_3^1(x_2)$, $o_4^1(x_2)$]



C_2 failed to deliver a correct result because it has become dependent on the effects of another computation C_1 although both of them were intended to be totally independent from each other.

To prevent this, is the goal of *concurrency control*

Distributed System Paradigms (38)

Serializability

A schedule S is *serializable* if it is computationally equivalent to at least one serial schedule S', i.e. if S produces the same output and leaves the object space in the same state as S'.

$C_k < C_l$ (C_l is *dependent* on C_k), if both computations contain at least one pair of conflicting operations such that $o_i^k(x) < o_j^l(x)$.

Let <* be the transitive closure of < with respect to all computations of a schedule S.
S is orderable w.r.t. its computations if S is acyclic with respect to <*, meaning that S does not contain any cycle $C_i <\ldots < C_j < \ldots C_i$.

S is *orderable* if and only if <* represents a partial order on the computations in S.

S is orderable --> S is serializable

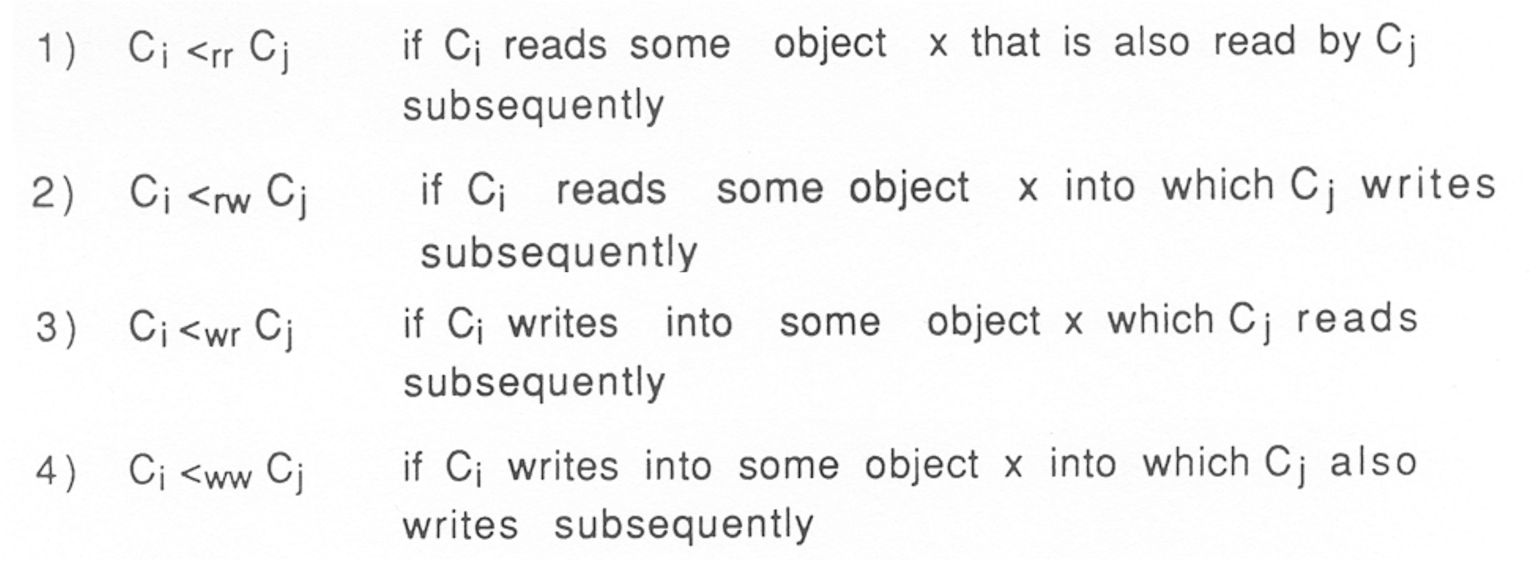
Recalling the previous example, we observe that $o_1^1(x_1) < o_1^2(x_1)$ and $o_2^2(x_2) < o_3^1(x_2)$. Hence, $C_1 < C_2 < C_1$ meaning that the corresponding schedule is not orderable.

Distributed System Paradigms (39)

- orderability is only a sufficient not a necessary condition for serializability.

Read/Write semantics

For each pair (C_i,C_j) of dependent computations



1) $C_i <_{rr} C_j$ if C_i reads some object x that is also read by C_j subsequently

2) $C_i <_{rw} C_j$ if C_i reads some object x into which C_j writes subsequently

3) $C_i <_{wr} C_j$ if C_i writes into some object x which C_j reads subsequently

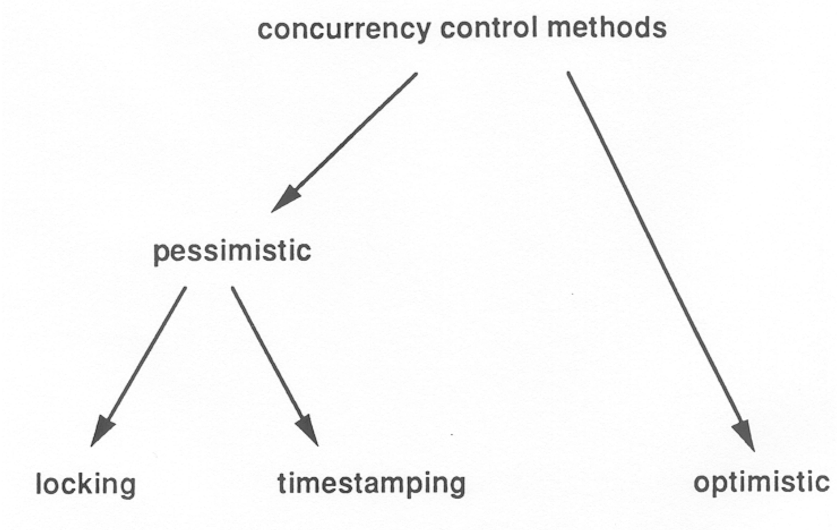
4) $C_i <_{ww} C_j$ if C_i writes into some object x into which C_j also writes subsequently

---> it suffices to care that $<^*_{rw} \cup <^*_{wr} \cup <^*_{ww}$ is orderable in order to ensure serializability

Distributed System Paradigms (40)

concurrency control methods

Classification of basic concurrency control methods



Locking

Two locks are in conflict, if both are locks on the same object and at least one of them is a *writelock*.

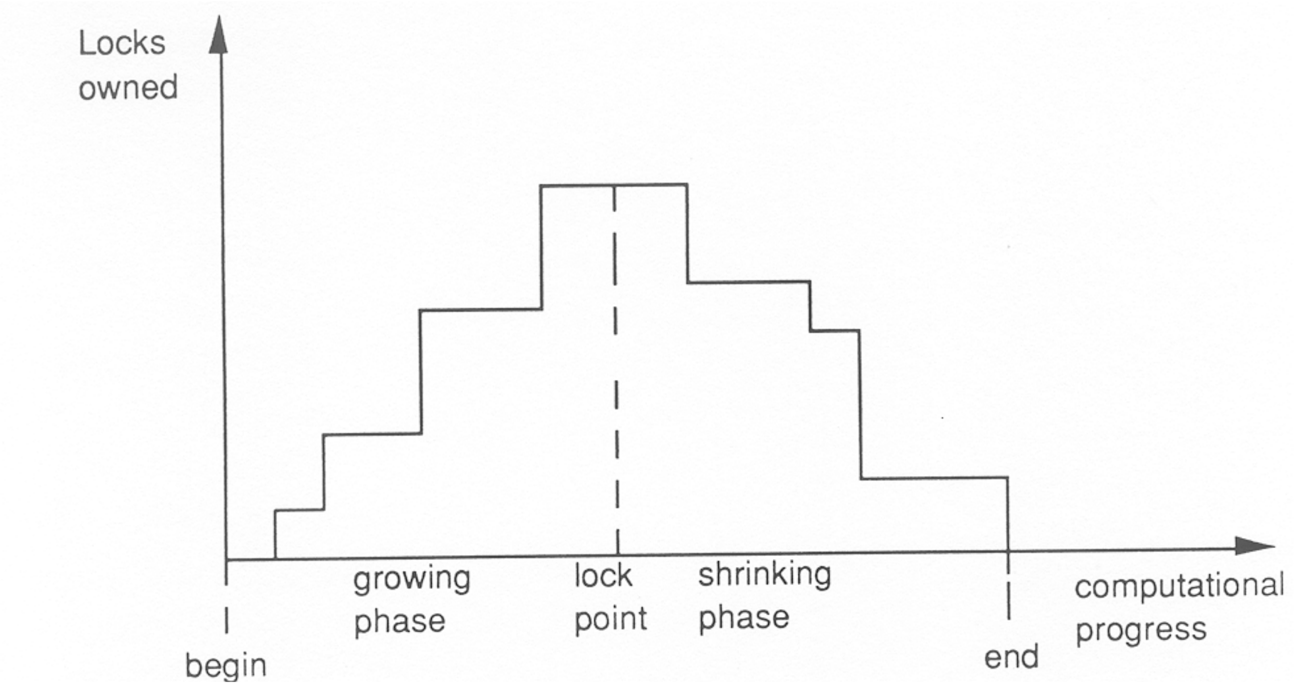
Theorem (Esweran):

S is serializable, if

1) at no time during the execution of S two computations do own conflicting locks and
2) once a computation releases a lock, it can never acquire additional locks again.

Distributed System Paradigms (41)

The two-phase lock protocol (2PL):



For any pair of computations with C < C', C reaches its lock point when C' is still in its growing phase.

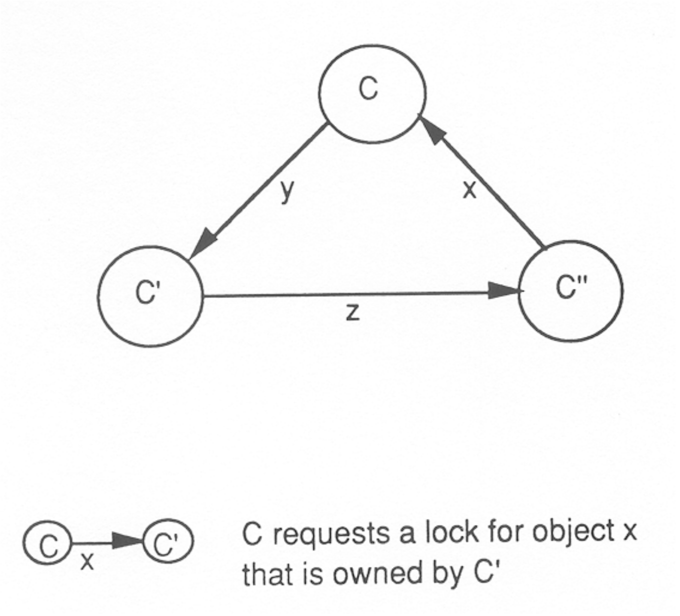
--> C can never become dependent on C'.

--> S is acyclic w.r.t. <* --> S is orderable

The serialization order produced by 2PL can be determined by the order in which the scheduled computations reach their lock point.

Distributed System Paradigms (42)

Conflict graph for detecting deadlocks:



C requests a lock for object x that is owned by C'

Timestamping

Timestamps may be generated by concatenating the local time (sequence nr.) with the unique node id.

- computations are ordered w.r.t.their object access according to their timestamps assigned
- a serialization order is selected a priori and a schedule is forced to obey this order, i.e. in the case of conflicting operations those computations that attempt an out-of-order access are invalidated.

By definition, the resulting schedule is serializable.

Variants:

- invalidations can be omitted if both conflicting operations represent writes (Thomas Write Rule)
- delay the processing of operations to wait for operations with smaller timestamps (conservative timestamping).

Distributed System Paradigms (43)

- timestamps are not assigned a priori, but when the first conflict between two computations occurs (dynamic timestamping)

Pro´s:

- simple algorithm
- due to the a priori selected order no deadlocks can occur

Con´s:

- much more pessimistic leading to unnecessary invalidated computations due to the a priori ordering
- using invalidation instead of blocking could be more expensive
- writes can only be made effective after the respective computation has terminated

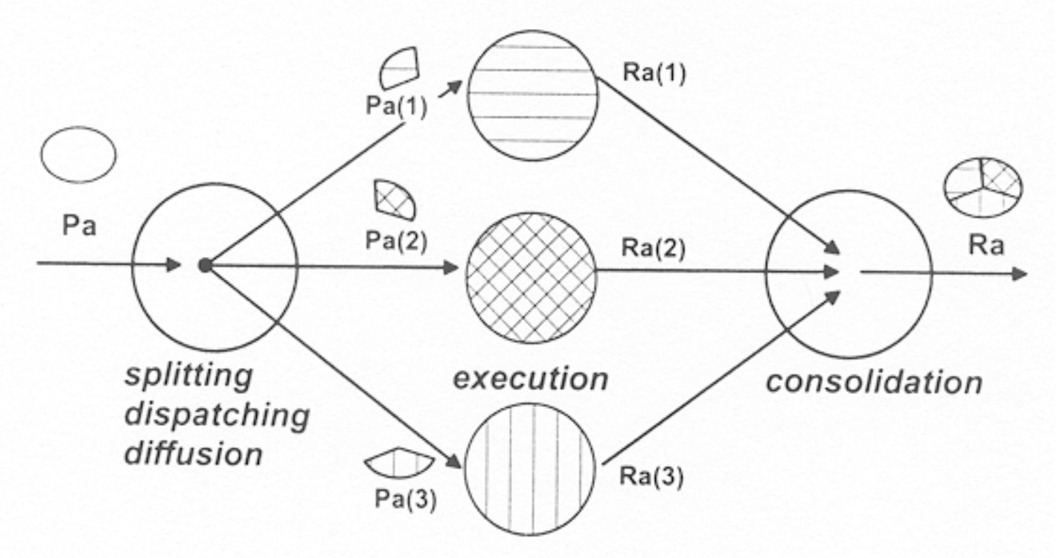
Models of Distributed Computing (3)

3. Classes of distributed activities

Coordination

It addresses the necessary steps to execute actions on several nodes that contribute to a common goal.

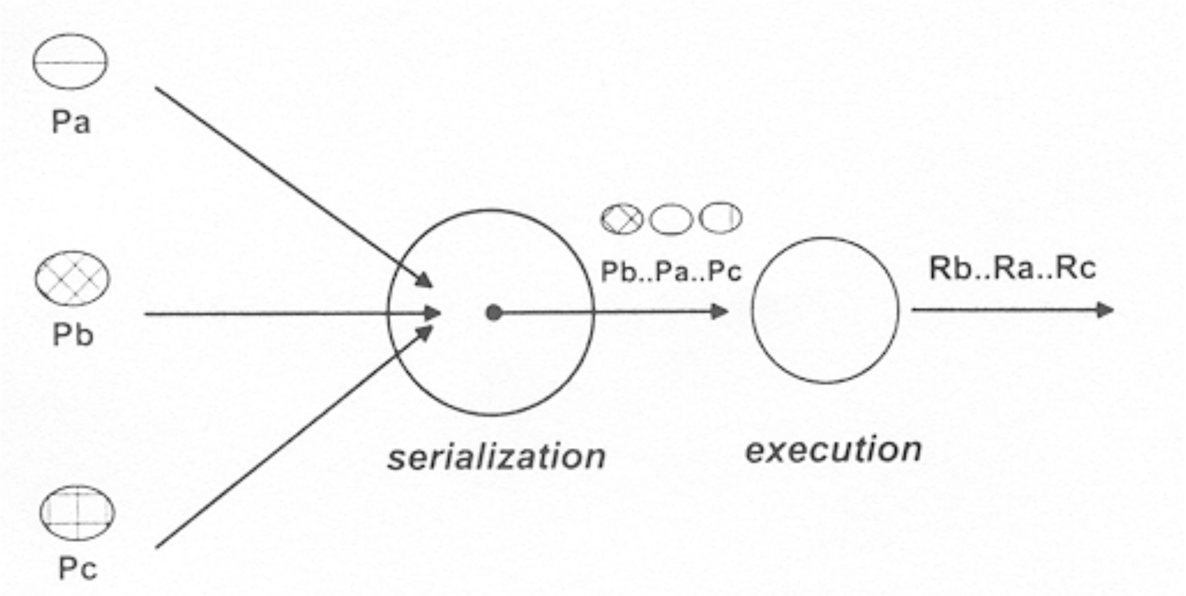
Flow Diagram of Coordination Activities



Models of Distributed Computing (4)

Sharing
It addresses the necessary steps to ensure the correct execution of actions using shared resources.

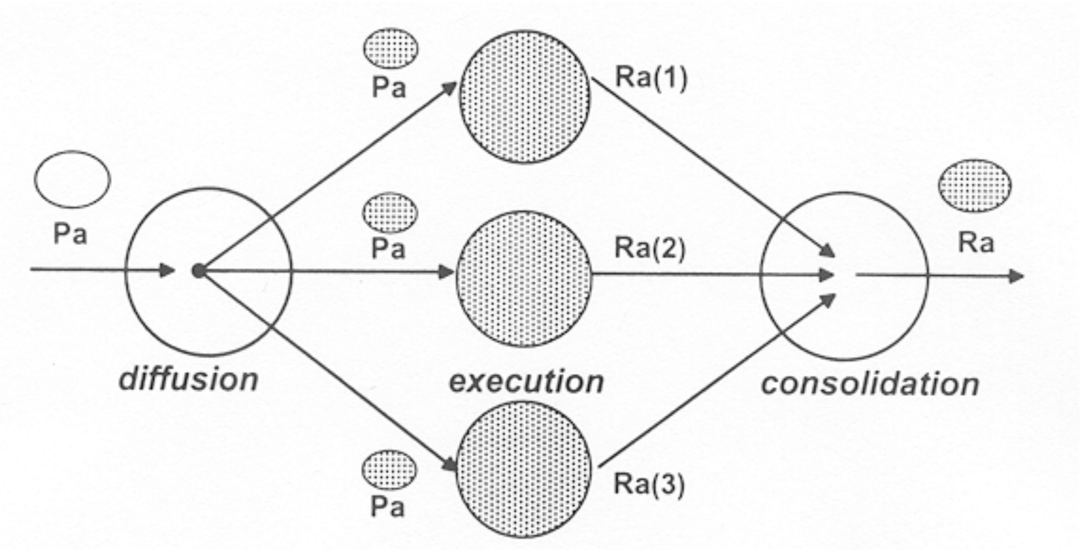
Flow Diagram of Sharing Activities



Models of Distributed Computing (5)

Replication
It addresses the necessary steps to execute the same set of actions on different nodes such that results are identical.

Flow Diagram of Sharing Activities



active replication: all participants execute the same set of actions in the same order
passive *replication:* a primary participant only executes the set of actions, the others (backups) only *log* them and receive state updates (*checkpoints*) from the primary.

omissive fault model: only one result is delivered (used for ensuring *availability*)
value fault model: only the correct result is delivered (determined by *majority voting*)

Models of Distributed Computing (6)

Combining Activities

Example Flow Diagram
(e.g. a distributed database, made of replicated fragments residing on several nodes, accessed by several users)