Transactions (2)

Solution: modeling computations executed by cooperating processes as *transactions* having *ACID* properties

C: Consistency by only allowing serializable execution of concurrent computations (atomicity of effect)

I: Isolation, i.e. realizing atomicity of effect also in the presence of a computation failure

D: Durability (permanence of effect), i.e.completed transactions can survive node/media crashes (commit)

A: Atomicity of failure (all or nothing - property), i.e. computations can be completely undone (abort)

ad C:

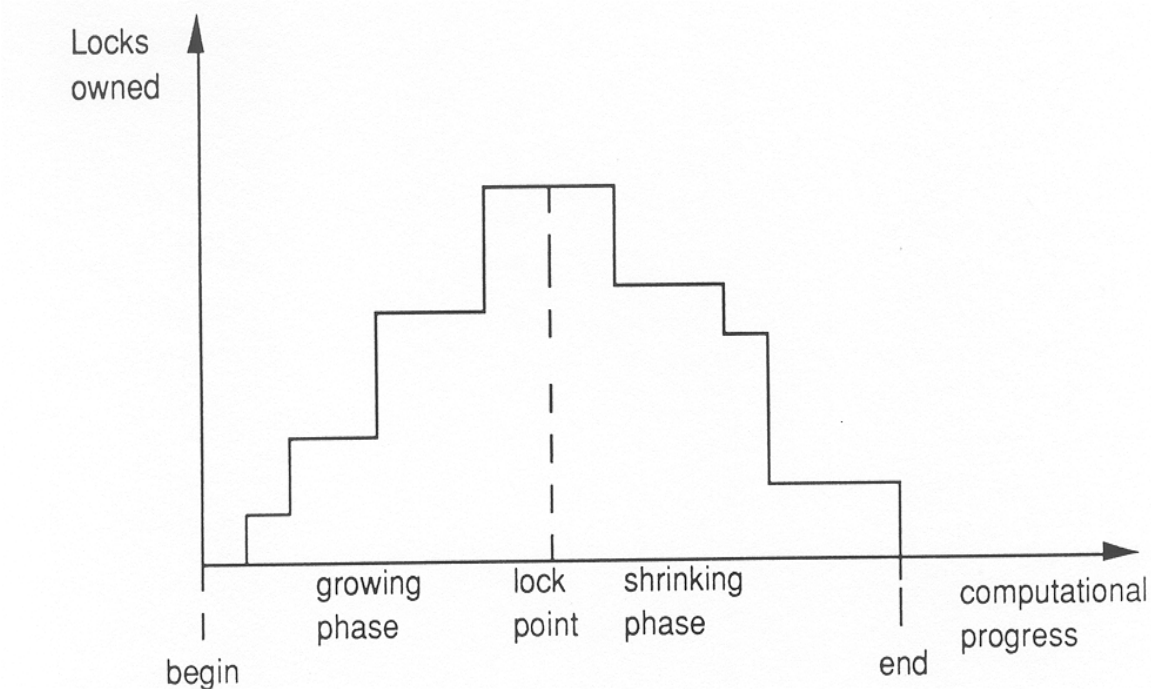
The two-phase lock protocol (2PL):

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

ad I:

Applying *Strict 2PL:*

Locks cannot be released until the transaction is completed.



Transactions (3)

ad D:

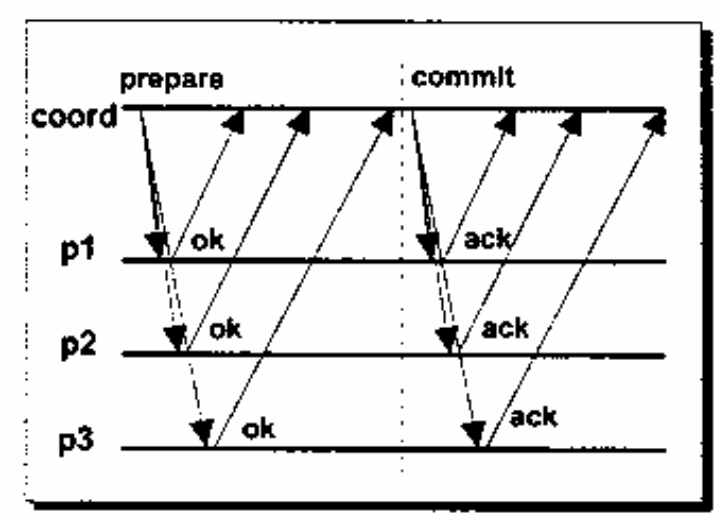
permanence of effect

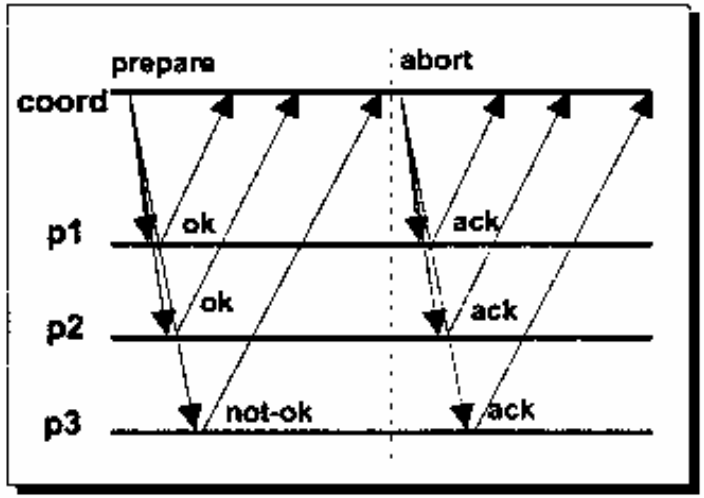
meaning that if a transaction completes successfully, the results of its operation in spite of subsequent node crashes will never be lost and cannot be aborted at any time later.

commit

meaning that the transaction has completed successfully and that its effects are made permanent.

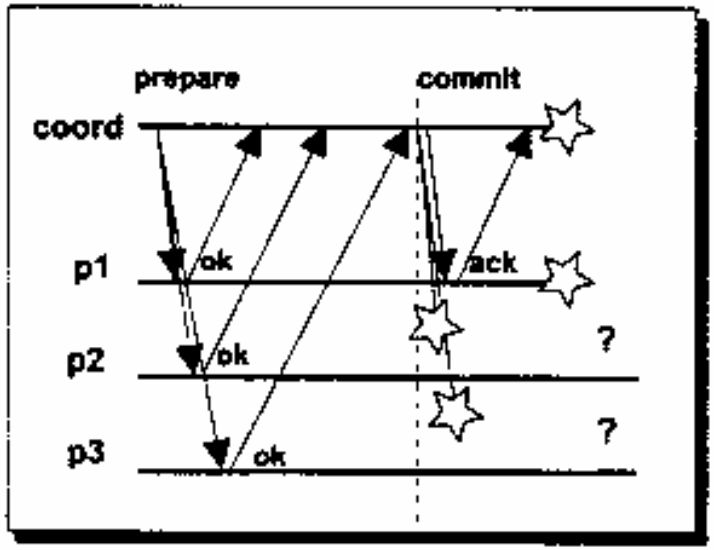
The two-phase commit protocol (2PC):



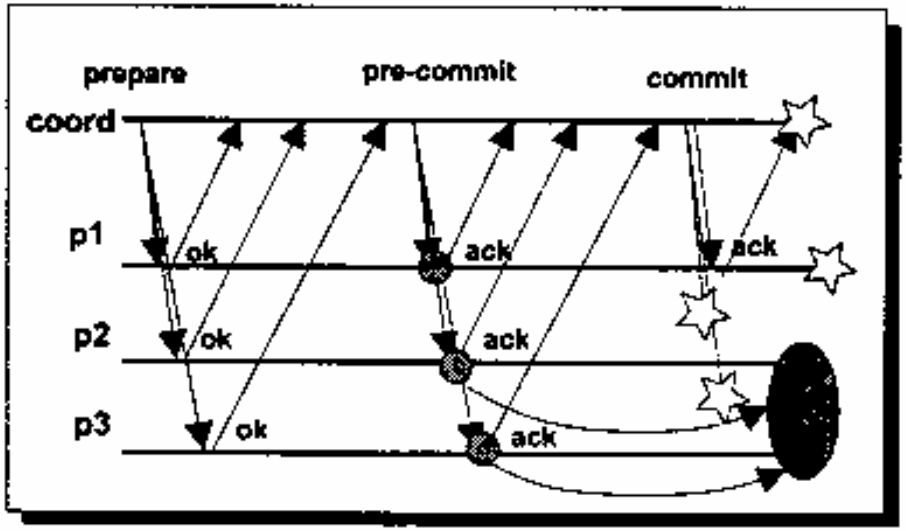


Transactions (4)

Blocking of 2PC:



Three-phase Commit (3PC):



Transactions (5)

ad A:

atomicity of failure

which ensures that if a transaction is interrupted by a failure, any effects (state changes on objects) produced by the failed transaction will be undone so that it appears as if it never has been executed.

abort

denotes the undoing of all effects of a transaction by restoring the respective recovery line by means of local backward error mechanisms.

cascading abort

which denotes the abort of all affected transactions as a consequence of a preceding abort. Obviously, the occurrence of a cascading abort would violate the isolation property or the durability property

Local backward error mechanisms

Undo rule

ensures that in case of a transaction abort those recovery points of the affected objects that have been established at the beginning of the transaction can be restored.

Redo rule

ensures that the objects written by a transaction can be updated according to its results.

Transactions (6)

Intentions lists (No Undo/Redo)

- Active transactions are not allowed to update current object states.
- All the changes that a transaction wants to make to object states are stored in a list.
- This list is written out to non-volatile storage.
- If the transaction commits, the affected objects are modified according to the „intentions list“.
- In case of an abort, the list is simply deleted.

Shadowing (No Undo/ No Redo)

Idea:
to organize the data structures on stable storage such that a single atomic write to it effects that a transaction is committed and for all modified objects of the transaction the new state is stably stored.

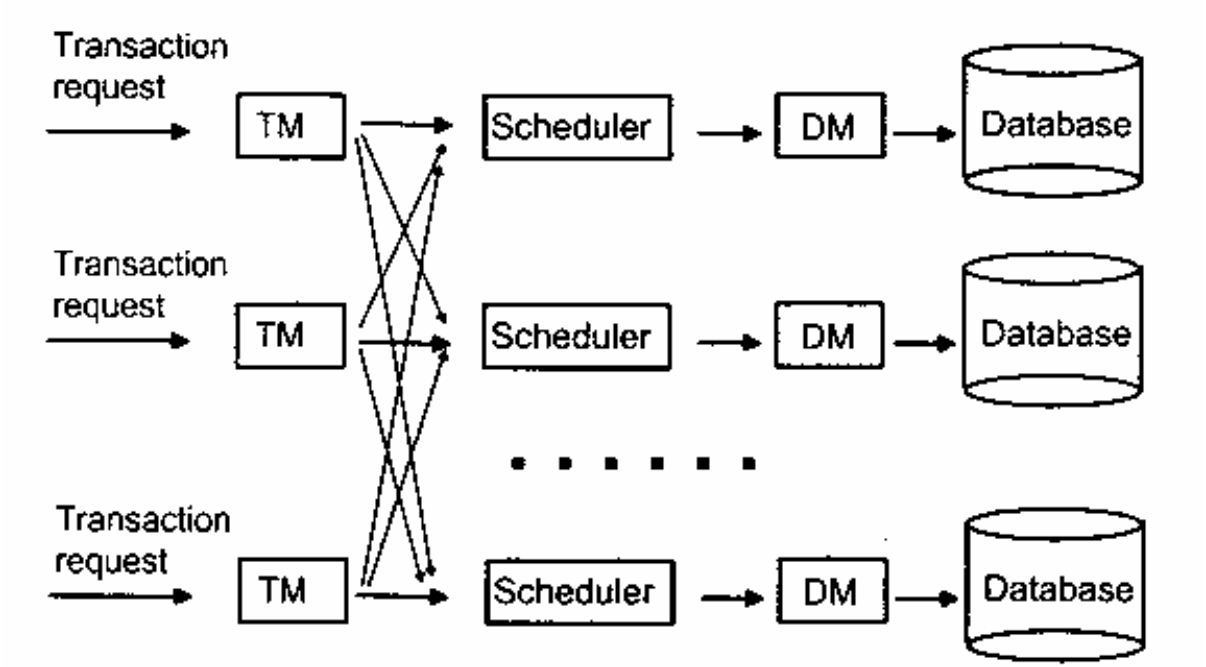
- The location (address) of the recovery points of the affected objects, (the state obtained by its last commit), stored in the active directory, is copied in a directory called scratch copy.
- There exists a master record which has a bit indicating which of the two directories is the current one.
- Each write operation on an object results in changing the corresponding entry in the scratch copy to point to the address of the new object version.
- The commitment of the transaction is triggered by a so-called atomic pointer swap meaning that the bit in the master record is simply complemented.

Transactions (7)

Write-ahead logging (Undo/Redo)

- Active transactions are allowed to update current object states.
- Updates are stored on a log (similar to intentions lists), but now only to redo committed transactions in case of a node crash (Recall: committed object states are guaranteed, i.e. cannot be aborted later)
- To permit a necessary Undo operation in case of a transaction failure, so-called Before-images (object state before the update) also are stored on a log. The same applies to commit records.
- If the system restarts after a node crash, the log is consulted. Updates made by committed transactions (identified due to the corresponding commit record on the log) are redone, all others are undone.

Architecture of a distributed transactional data base system



Transactions (7a)

Conclusion

With (classical) transactions, the problem of the domino effect was solved by enforcing isolation between different transactions, e.g. by means of strict 2PL.

---> in general works no longer, if transactions must cooperate

---> looking for a model for distributed systems in general, i.e. including cooperating processes:
the isolation property must be given up (serializability as consistency criterion may still work))

---> there still exists a priori defined recovery line per action, but cascading aborts become possible

---> it needs damage assesssment before doing recovery (looking for the respective recovery line by checking which actions have to be aborted)

---> classical data base transaction are no adequate model to support generic applications on the OS level

The more general problem to be solved is:

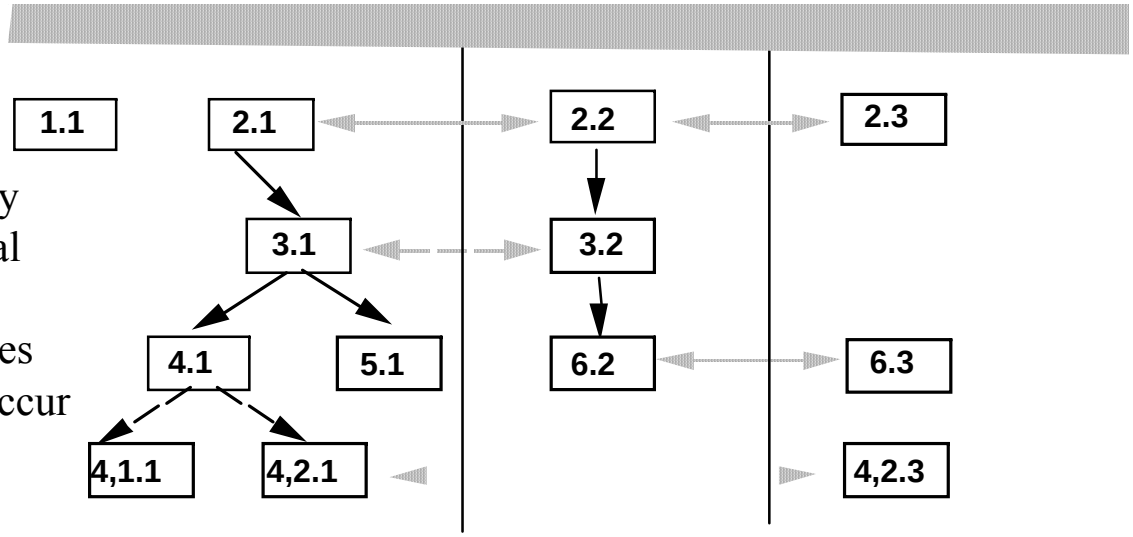
How to execute the joint activity of cooperating computations (actions)?

To solve this problem means to find a solution for the recovery of such joint, distributies activities!

Atomic actions (4)

The damage assessment graph (DAG) for atomic actions

A *Recovery Unit (RU)* represents the set of all object states that are produced by a certain atomic action on one node.



The postulate of serializability considerably simplifies the structure of a DAG. The local DAGs adopt a tree structure.
The possibility for backward expansion does not need to be checked because it cannot occur anyhow.

The actions 4,1 and 4,2 are nested with respect to action 4 in the example above. Hence, they can abort without automatically affecting their father. The other way round does not hold because of the atomicity property, i.e. whenever

RU 4 aborts all of its sons are included.

i.j : recovery unit of action i located on node j

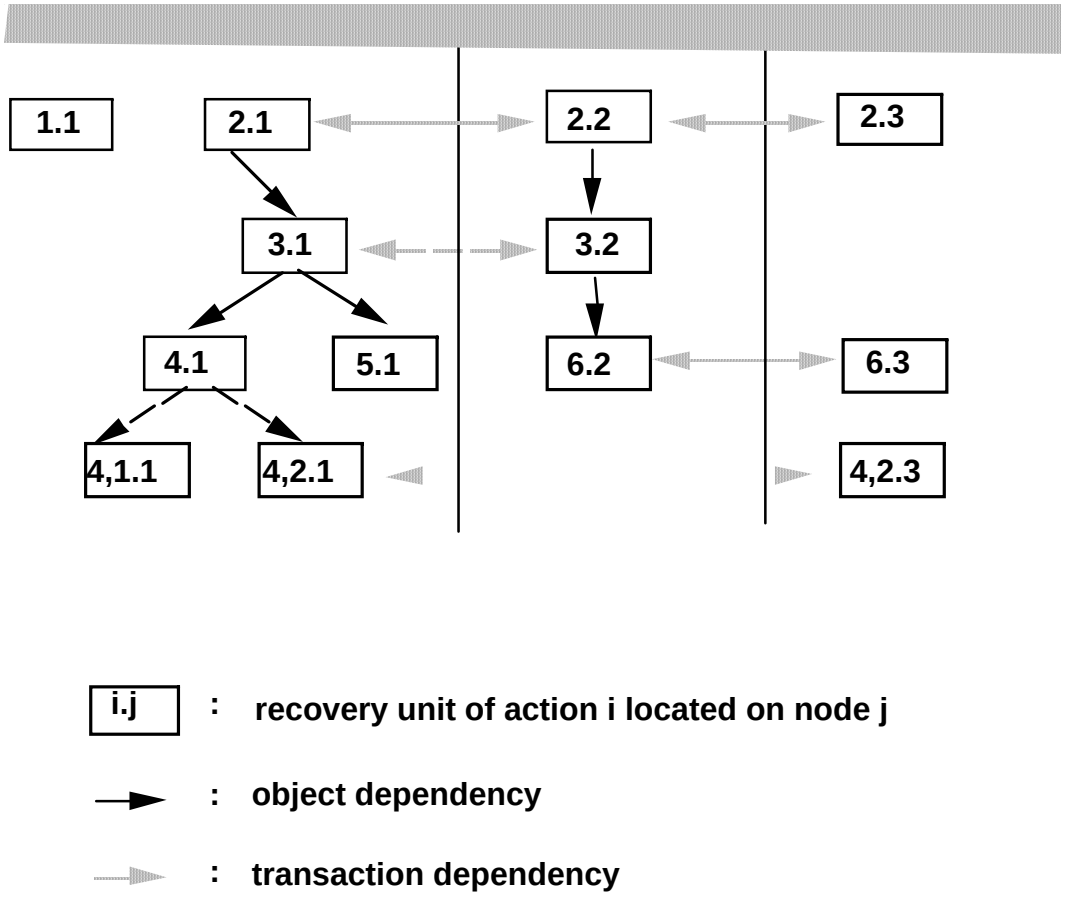
: object dependency

: transaction dependency

Atomic actions (5)

The damage assessment (chase) protocol, executed to identify the abort set, illustrated by an example:

Assume the RM (recovery manager) of node 1 gets notice to abort RU 3.1. Consequently, the whole subgraph emanating from the root vertex 3.1 must be contained in the abort set. The abort message to be generated includes the identifiers 3 and 4,2. This results in the abort of the RUs 3.2 and 6.2 at node 2, which then generates a further abort message for action 6. In receiving both messages, node 3 aborts the RUs 6.3 and 4,2.3 and sends no further chase message which concludes the protocol.

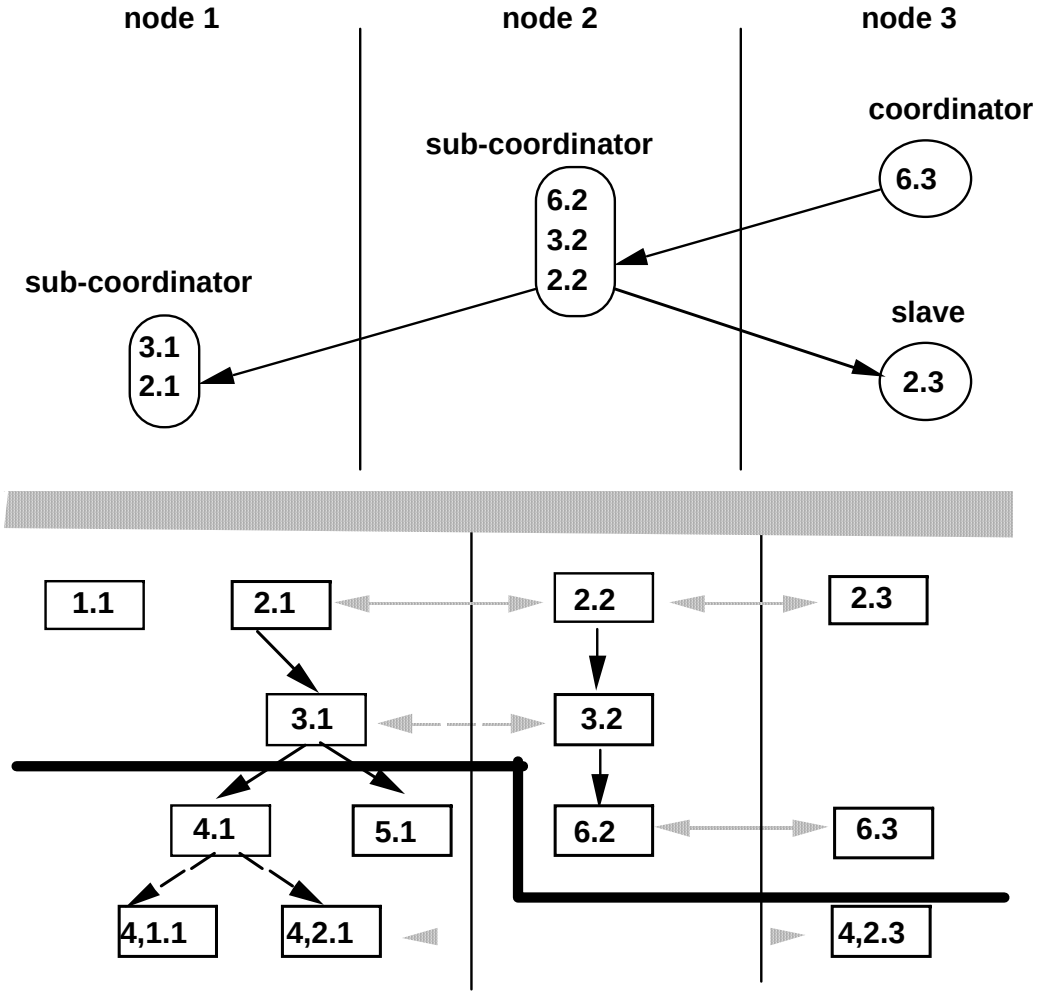


Atomic actions (6)

The commit protocol

The main task in moving forward the commit line is the execution of a Commit Protocol (CP). First, the commit set has to be determined.

Example: node 3 wants to commit action 6 and acts as coordinator



Resulting commit line:
actions above that line belong to the commit set