

Distributed transactions

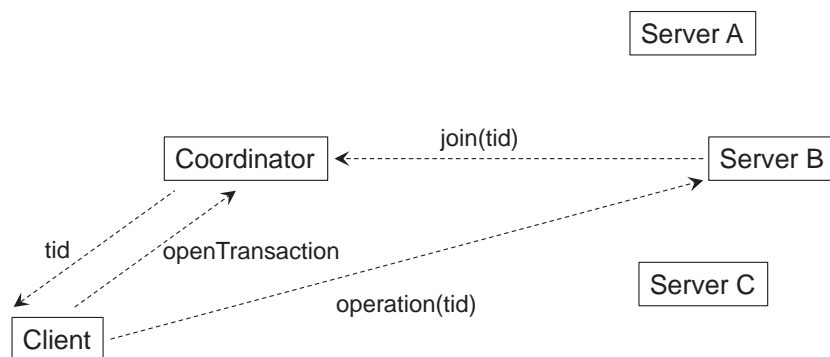
Johan Montelius and Vladimir Vlassov

Problem

- Several independent transaction servers should be coordinated in one transaction.
- How do we coordinate operations to guarantee serial equivalence?

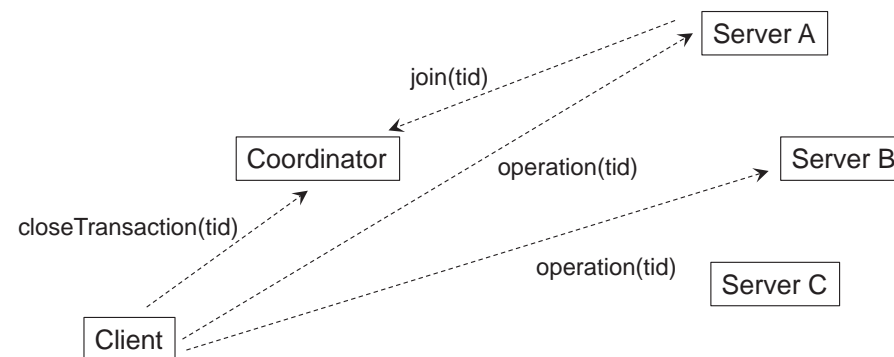
The architecture

transaction servers



The architecture

transaction servers





One-phase commit

- Client sends closeTransaction to coordinator.
- Coordinator tells participants to commit the transaction.
- Problem:
 - ?



Two-phase commit

- phase one (voting): ask participants to vote for commit or abort
 - if voting for commit, one has to be able to commit even after a node crash
 - if anyone aborts all must abort
- phase two (completion): inform all participants of the result



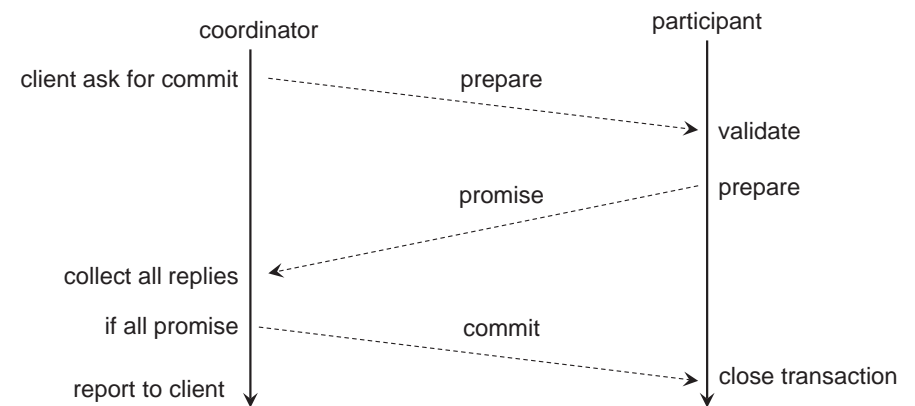
Consensus

Two-phase commit is a consensus protocol but:

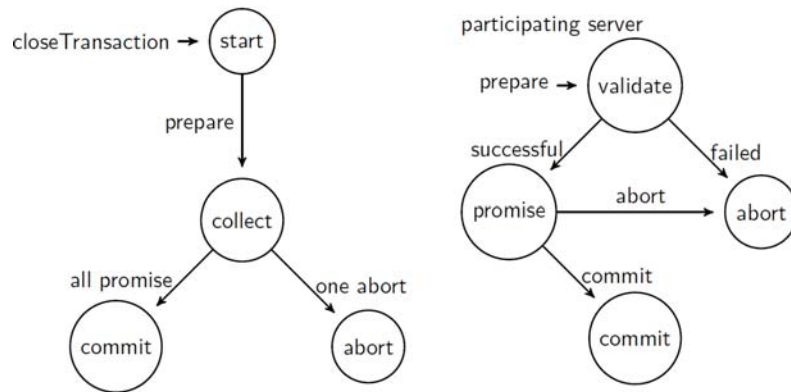
- all servers must vote
- if any server wants to abort then we abort



Two-phase commit



Two-phase commit



What if ..

- A participating server crashes before making a promise
- A participating server crashes after having promised
- The coordinator crashes before asking for a promise
- The coordinator crashes but you have made a promise

Two-phase commit can be suspended waiting for a crashed coordinator

If we know our peers

Assume that the participants know each other.

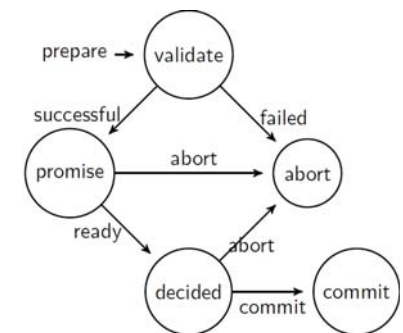
If the coordinator crashes:

- and no participant was told to commit, then it is safe to abort
- if one participant was told to commit, then we should all commit

What if the coordinator and one participant has crashed and none of the surviving participants have received a commit message?

Three-phase commit

- If the coordinator crashes and one node is still in the promised state we know that the coordinator has not ordered a commit - we can thus abort.
- If the coordinator crashes and all nodes are in the decided state they agree to commit.



Relies on perfect failure detectors - and that we know who is in the group.



Concurrency control

- locking
- optimistic
- timestamp



The danger of locking

Assume we implement *strict two-phase locking* and need to take the locks for *foo*, *bar* and *zot*.

What does it mean and what should we do?



Avoid or handle

You can either avoid deadlocks or detect them.

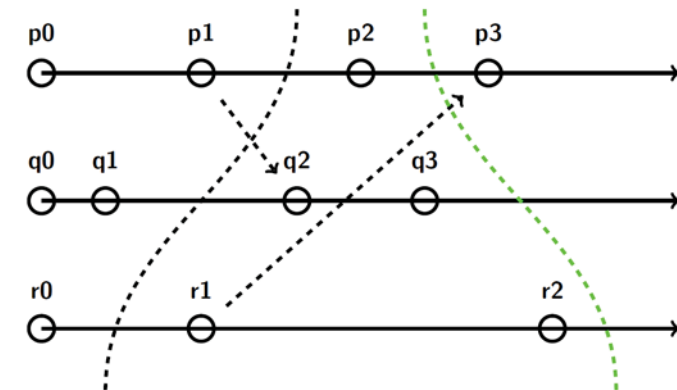
We are in a deadlock if T is waiting for S that is waiting for... that is waiting for T.

- A set of processes is deadlocked when each process in the set is waiting for an event which can only be caused by another process in that set

Examine the state and look for circular dependencies.



A distributed state





Deadlock detection

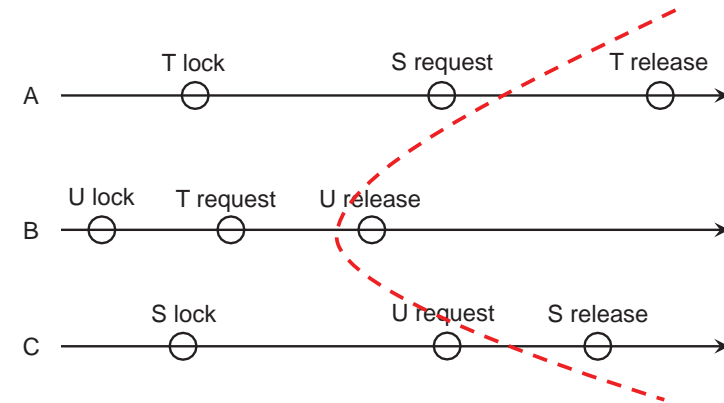
What if:

- server A reports: S is waiting for T
- server B reports: T is waiting for U
- server C reports: U is waiting for S

Deadlock detected, let's do something



Phantom deadlock



Detection

How do we detect deadlocks?



Optimistic concurrency control

Transactions should be validated in a total order.

What if transaction T is validated at A and transaction S at B?



Timestamp order

A global timestamp that all transaction servers agree to.



Summary

Distributed transactions

- a global total order of transactions
- if one server needs to abort, then all should abort

Two-phase commit

- coordinator asks participants to prepare
- participants promise to commit (or aborts)
- coordinator directs participants to commit

Distributed deadlock

- hard to prevent
- simpler to detect

Concurrency control

- locks
- optimistic
- timestamp