Distributed transactions

Johan Montelius

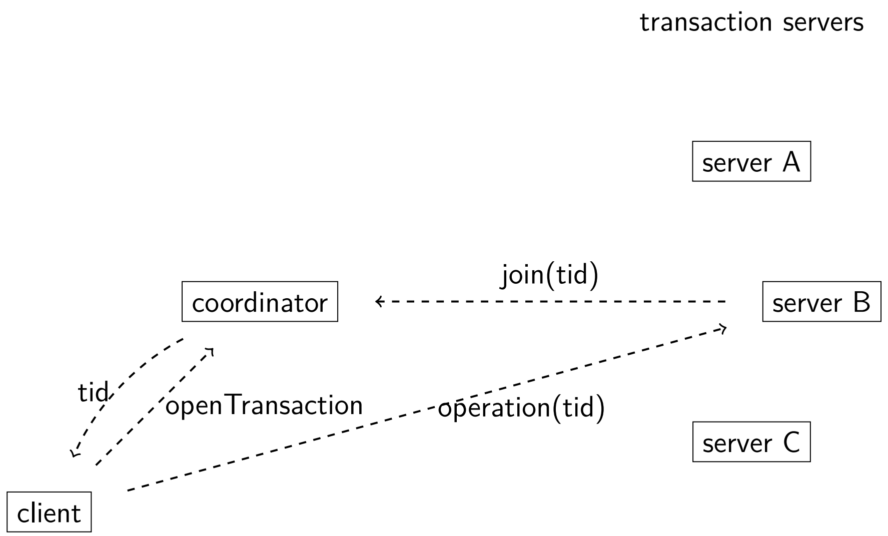
KTH

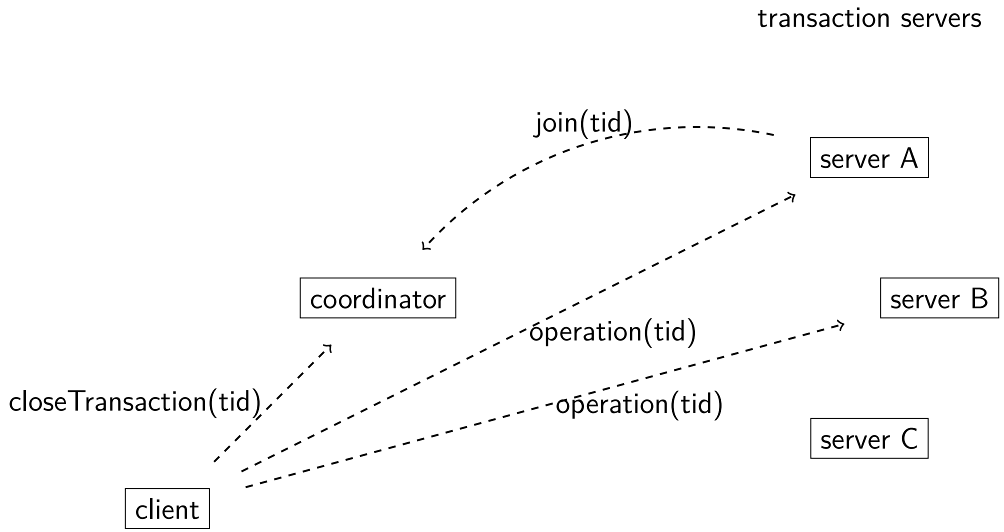
HT15

Problem

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

the architecture





one-phase commit

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

If one of the server says I want to abort
then that's the end because you need participation of the three servers
actually the one-phase doesn't even exist!

two-phase commit

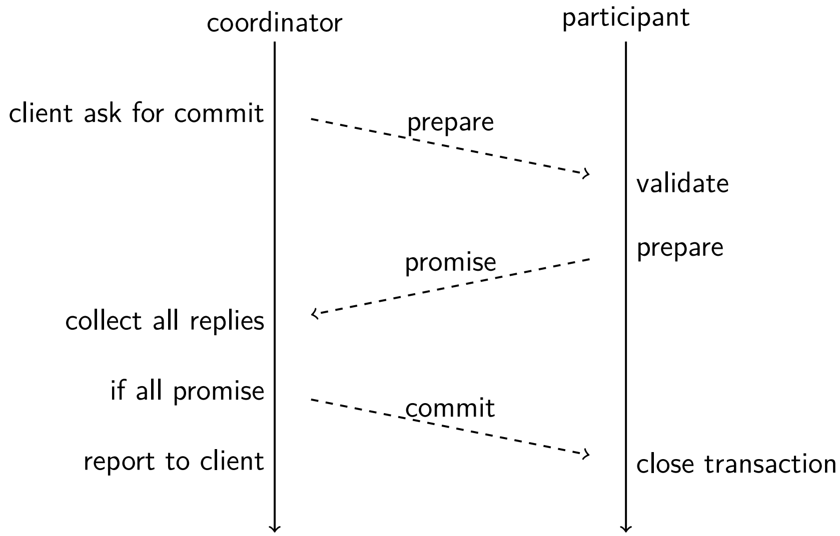
- phase one: ask participants to vote for commit or abort
  - if voting for commit one has to be able to commit even after a node crash
  - if anyone aborts all must abort
- phase two: inform all participants of the result

Consensus

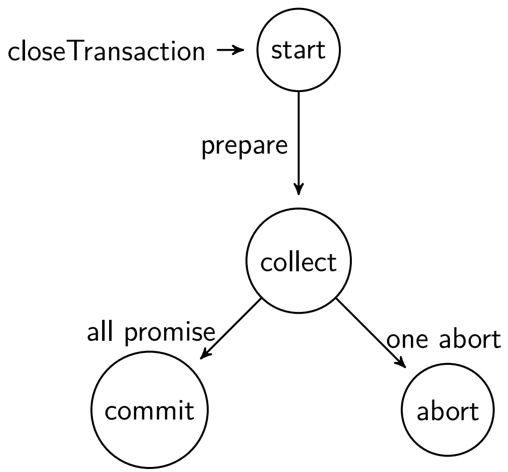
Two-phase commit is a consensus protocol but:

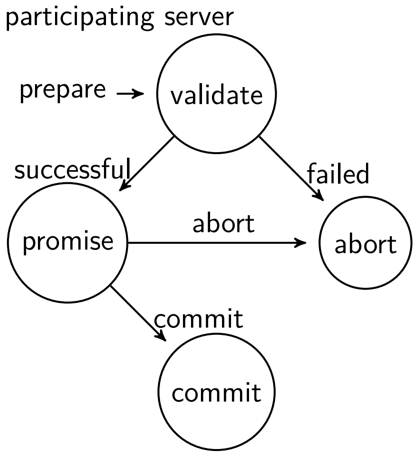
- all servers must vote
- if any server wants to abort then we abort

Two-phase commit



Two-phase commit





what if ..

It's very robust!

- 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

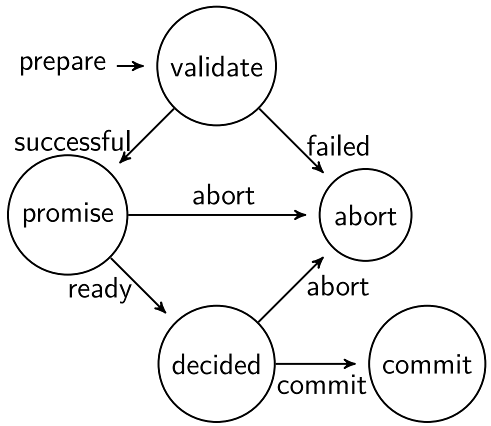
Assume that the participants know each other.

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 in the *promised state* and coordinator crashes, and no non-crashed participant is in the *decided state* then abort, otherwise commit.
- If in the *decided state* and coordinator crashes then commit

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

It works but it is actually never used —‘

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?

First grab all the locks, and then if you have all of them commit before you close (strict), or if you don't have them all release them.
With the order of the locks, for example, if you want the locks you need to grab them in alphabetical order. ordering them solve the problem/

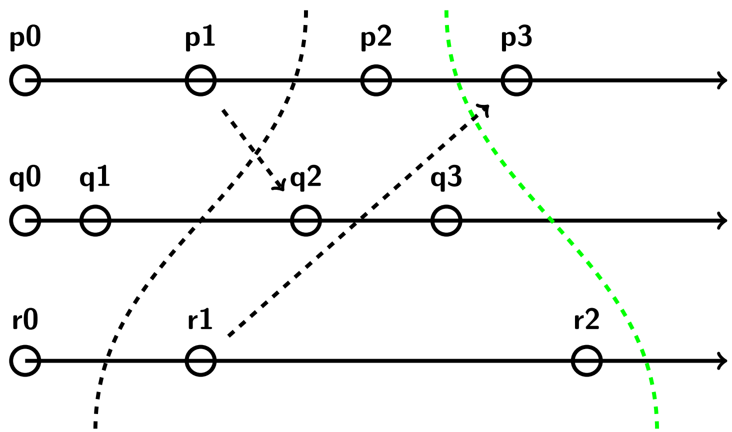
avoid or handle

You can either avoid dead-locks or detect them.

We are in a dead-lock if T is waiting for S that is waiting for... that is waiting for T.

Examine the state and look for circular dependencies.

a distributed state



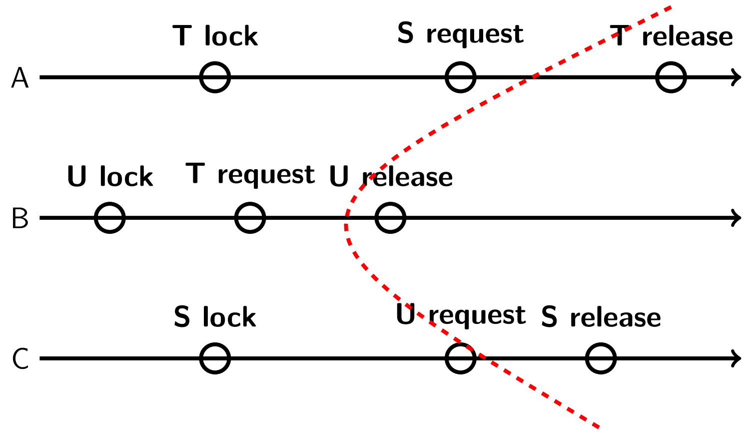
deadlock detection

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?

With a marker ! you pass the marker and if you receive a marker it means that there is a circle then it means that there is a deadlock ! You abort and you break the circle ! penser à la petite mise en scène du prof !

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