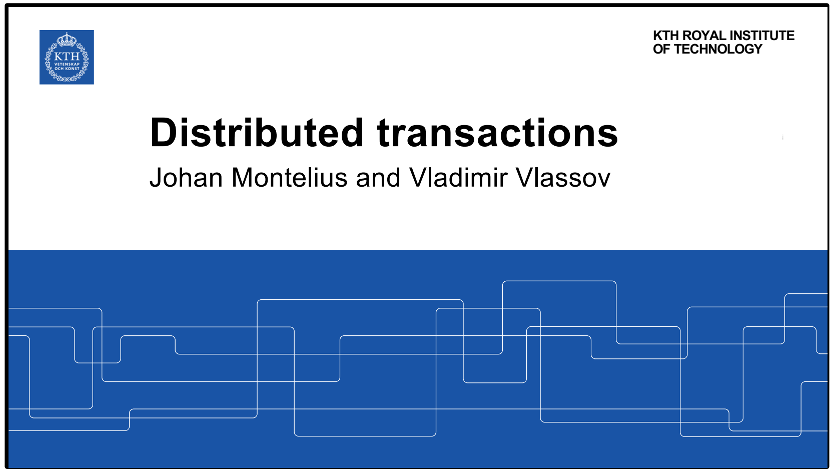
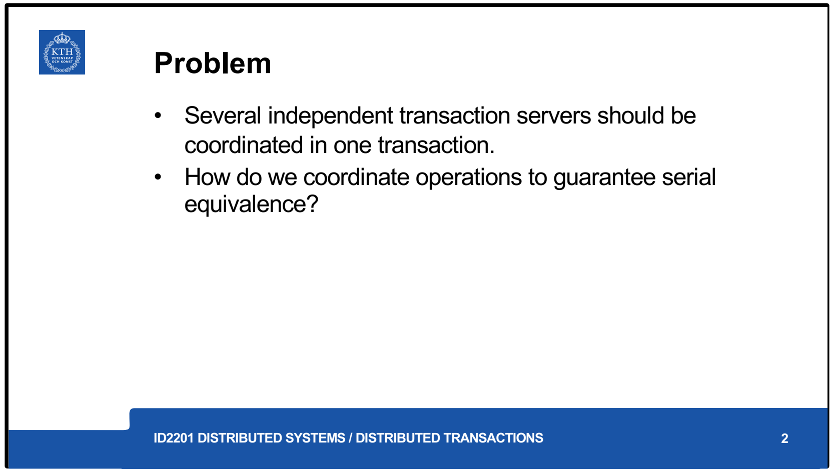


KTH ROYAL INSTITUTE
OF TECHNOLOGY

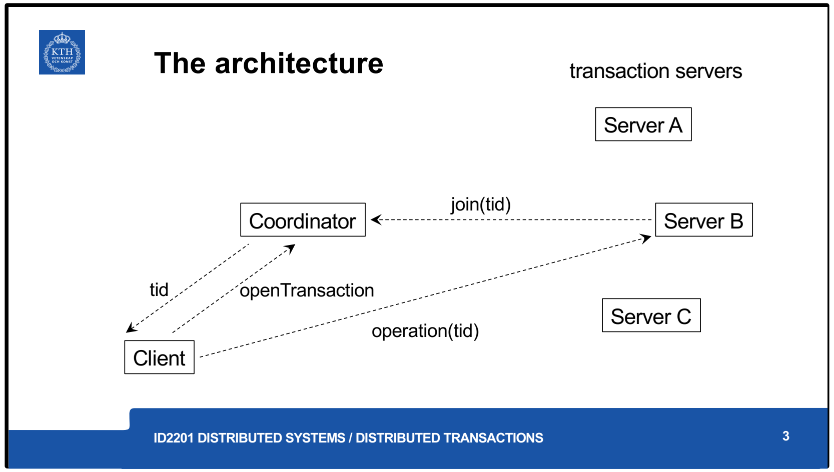
Distributed transactions

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

Server A

Coordinator

join(tid)

Server B

tid

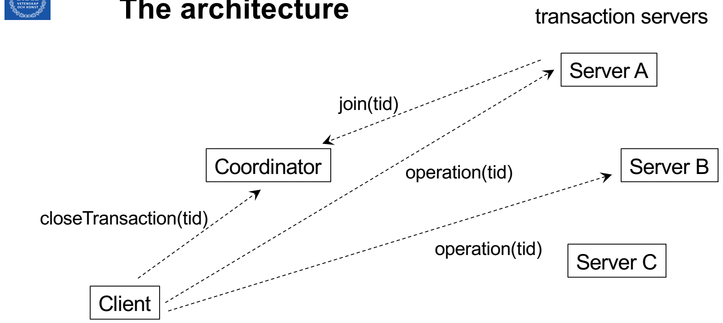
openTransaction

Server C

operation(tid)

Client

The architecture



transaction servers

Server A

join(tid)

Coordinator

operation(tid)

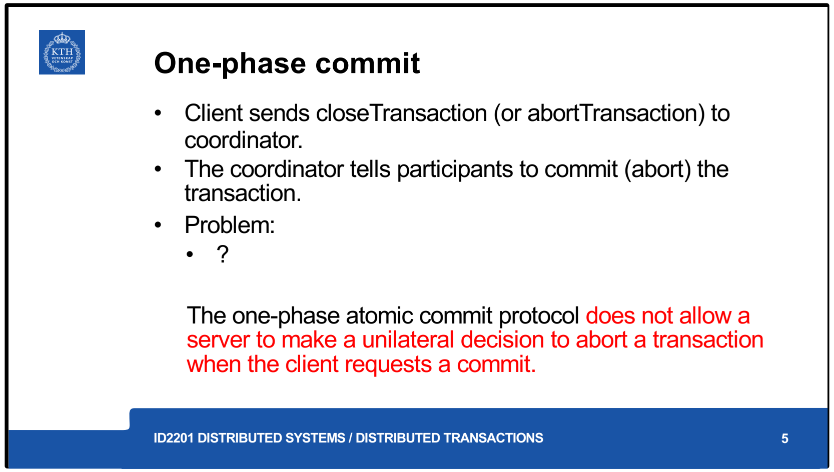
Server B

closeTransaction(tid)

operation(tid)

Server C

Client

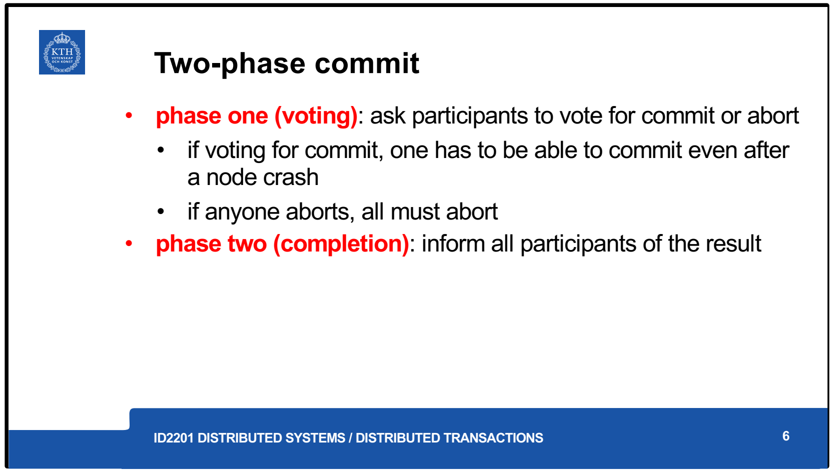


One-phase commit

- Client sends closeTransaction (or abortTransaction) to coordinator.
- The coordinator tells participants to commit (abort) the transaction.
- Problem:
  - ?

The one-phase atomic commit protocol does not allow a server to make a unilateral decision to abort a transaction when the client requests a commit.

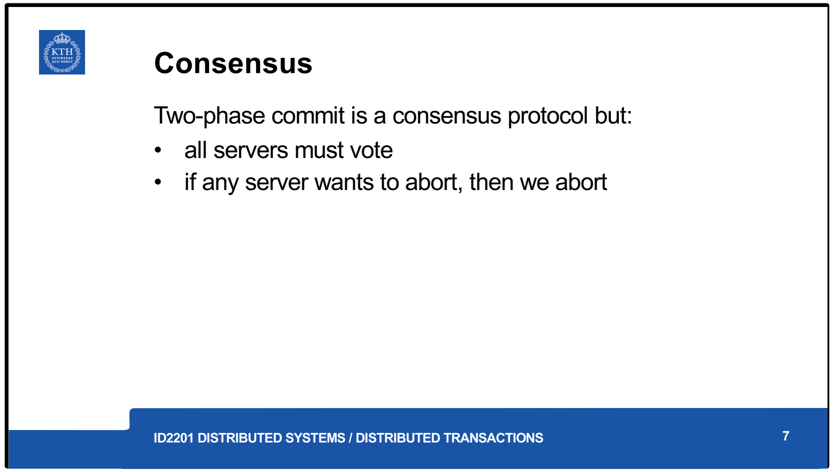
The atomicity property of transactions requires that when a distributed transaction comes to an end, either all of its operations are carried out or none of them. In the case of a distributed transaction, the client has requested operations at more than one server. A transaction comes to an end when the client requests that it be committed or aborted. A simple way to complete the transaction in an atomic manner is for the coordinator to communicate the commit or abort request to all of the transaction participants and to keep repeating the request until all of them have acknowledged that they have carried it out. This is an example of ***a one-phase atomic commit protocol***. **However, this simple one-phase atomic commit protocol is inadequate because it does not allow a server to make a unilateral decision to abort a transaction when the client requests a commit.** Reasons that prevent a server from being able to commit its part of a transaction generally relate to issues of concurrency control. For example, if locking is in use, the resolution of a deadlock can lead to the aborting of a transaction without the client being aware unless it makes another request to the server. Also, if optimistic concurrency control is in use, the failure of validation at a server would cause it to decide to abort the transaction. Finally, the coordinator may not know if a server has crashed and been replaced during the progress of a distributed transaction – such a server will need to abort the transaction.



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

The two-phase commit protocol allows any participant to abort its part of a transaction. Due to the requirement for atomicity, if one part of a transaction is aborted, then the whole transaction must be aborted. **In the first phase of the protocol, each participant votes** for the transaction to be committed or aborted.



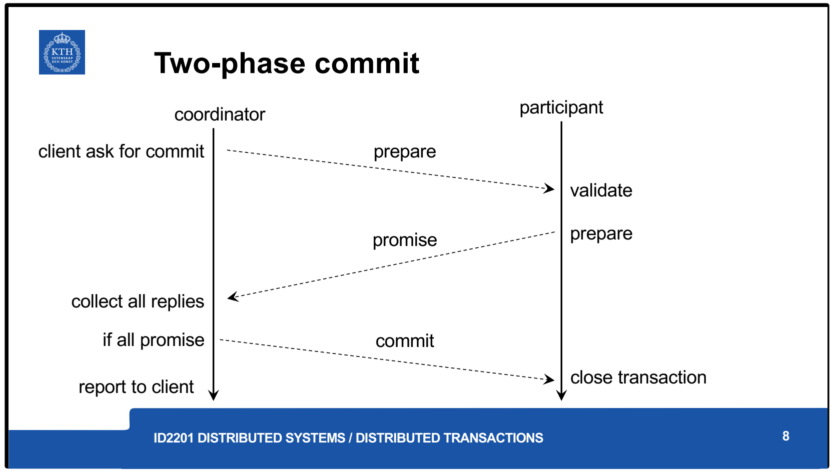
Consensus

Two-phase commit is a consensus protocol but:

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

Once a participant has voted to commit a transaction, it is not allowed to abort. Therefore, before a participant votes to commit a transaction, it must ensure that it will eventually be able to carry out its part of the commit protocol, even if it fails and is replaced in the interim. A participant in a transaction is said to be in a ***prepared*** state for a transaction if it will eventually be able to commit it. To ensure this, each participant saves in permanent storage all the objects ***it has altered in the transaction***, together with its status – ***prepared***.

The two-phase commit protocol is an example of a protocol for reaching a consensus. Chapter 15 asserts that consensus cannot be achieved in an asynchronous system if processes sometimes fail. However, the two-phase commit protocol does reach consensus under those conditions. This is because **crash failures of processes are masked by replacing a crashed process with a new process whose state is set from information saved in permanent storage and information held by other processes.**



Two-phase commit

coordinator

participant

client ask for commit

prepare

validate

promise

prepare

collect all replies

if all promise

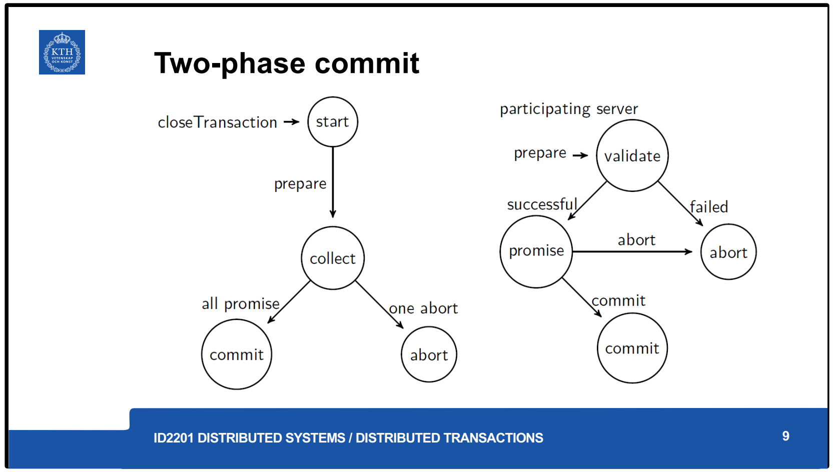
commit

close transaction

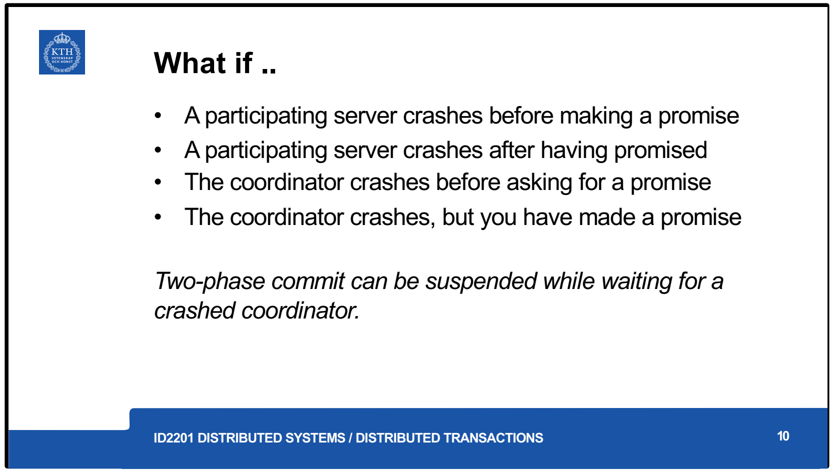
report to client

ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED TRANSACTIONS 8

canCommit?(trans) - > Yes / No (on the picture -- *Prepare*) Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.
doCommit(trans) Call from coordinator to participant to tell the participant to commit its part of a transaction.
doAbort(trans) Call from coordinator to participant to tell the participant to abort its part of a transaction.
haveCommitted(trans, participant)
Call from participant to coordinator to confirm that it has committed the transaction.
getDecision(trans) Yes / No Call from participant to the coordinator to ask for the decision on a transaction when it has voted Yes but has still had no reply after some delay. Used to recover from a server crash or delayed messages.



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 while waiting for a crashed coordinator.

ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED TRANSACTIONS 10

We assume that when a server runs, it keeps all of its objects in its volatile memory and **records its committed objects in a recovery file or files**. Therefore recovery consists of restoring the server with the latest committed versions of its objects from permanent storage. ***The recovery manager*** is responsible for restoring the server's objects after a crash.

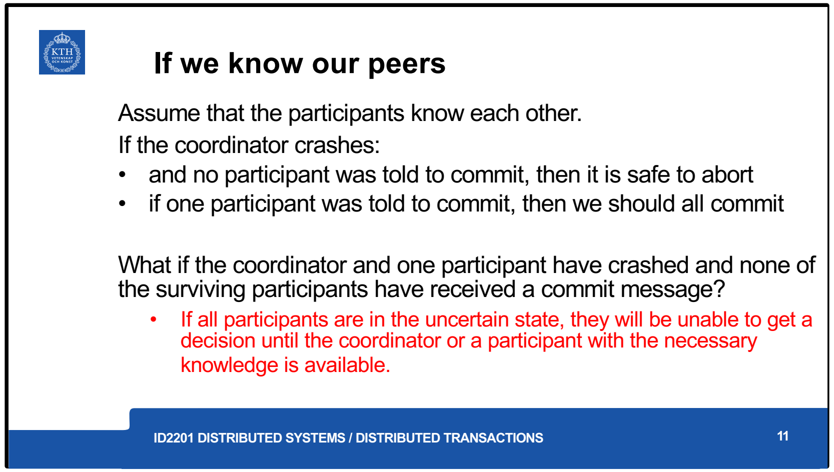
Client crashed**: Before promise:** Should recover or abort. **After promise**: Must recover. At the point when a participant says it is prepared to commit a transaction, its recovery manager must have saved both its intentions list for that transaction and the objects in that intentions list in its recovery file so that it will be able to carry out the commitment later, even if it crashes in the interim.

If the coordinator has failed, the participant will not be able to get the decision until the coordinator is replaced, which can result in extensive delays for participants in **an *uncertain* state.**

Another point at which a participant may be delayed is when it has carried out all its client requests in

the transaction but has not yet received a canCommit? (prepare) call from the coordinator. As the client sends the closeTransaction to the coordinator, a participant can only detect such a situation if it notices that it has not had a request in a particular transaction for a long time – for example, by **the time a timeout period on a lock expires. As no decision has been made at this stage, the participant can decide to abort unilaterally.**

The coordinator may be delayed when waiting for votes from the participants. As it has not yet decided the fate of the transaction, it may choose to abort it after some time. It must then announce doAbort to the participants who have already sent their votes. Some tardy participants may try to vote Yes after this, but their votes will be ignored, and



If we know our peers

Assume that the participants know each other.

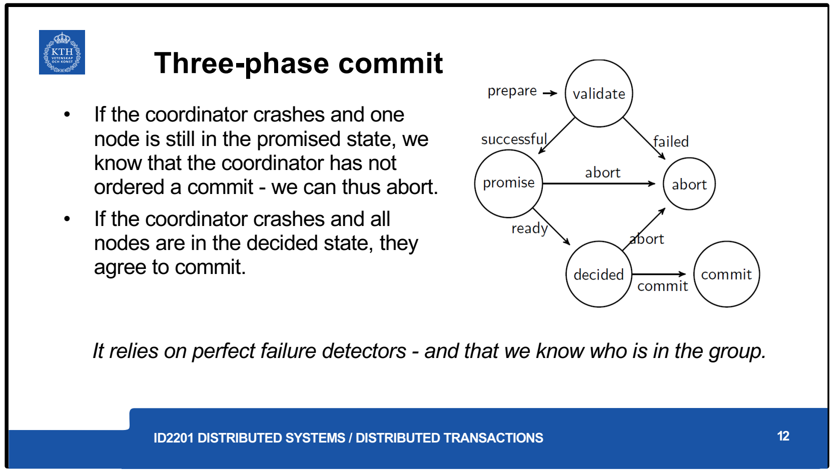
If the coordinator crashes:

- 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 have crashed and none of the surviving participants have received a commit message?

- If all participants are in the uncertain state, they will be unable to get a decision until the coordinator or a participant with the necessary knowledge is available.

Alternative strategies are available for the participants to **obtain a decision cooperatively instead of contacting the coordinator.** These strategies have the advantage that they may be used when the coordinator has failed. However, even with a cooperative protocol, **if all the participants are in the uncertain state**, they will be unable to get a decision until the coordinator or a participant with the necessary knowledge is available.



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.

prepare
validate
successful
failed
promise
abort
abort
ready
abort
decided
commit
commit

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

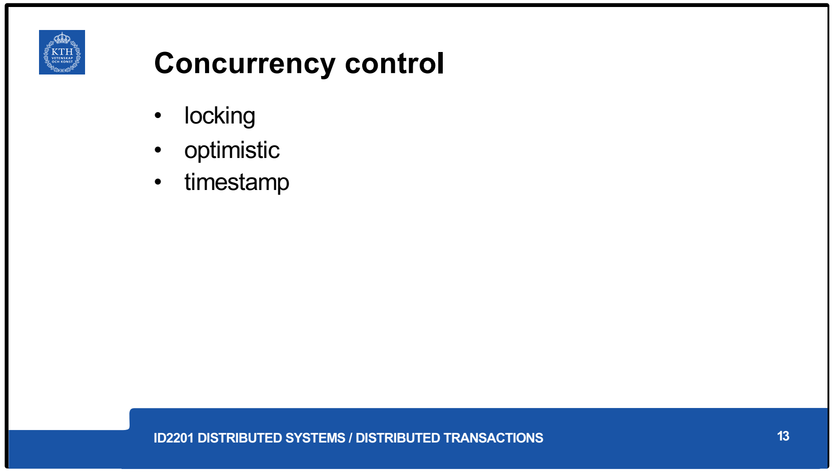
Even if a cooperative protocol allows participants to make getDecision requests to other participants, **delays will occur if all the active participants are uncertain.**

Three-phase commit protocols have been designed to alleviate such delays, but they are more expensive regarding the number of messages and rounds required for the normal (failure-free) case.

Phase 1: This is the same as for the two-phase commit.

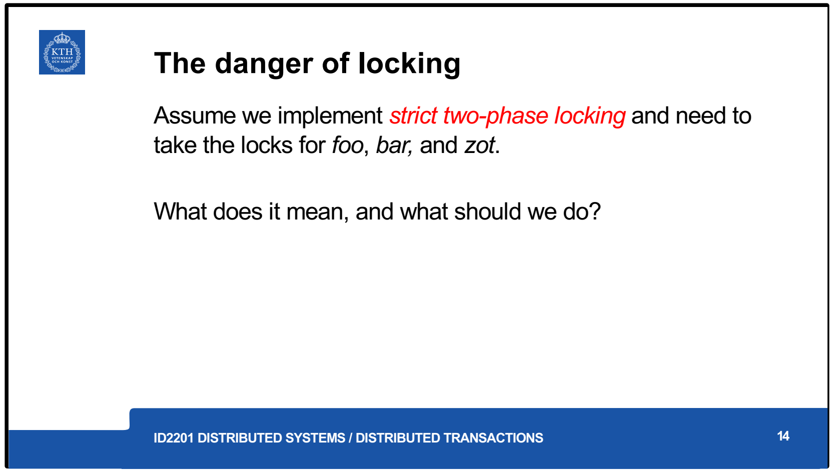
Phase 2: The coordinator collects the votes and decides. If it is No, it aborts and informs participants that voted Yes; **if the decision is Yes, it sends a preCommit (or ready) request to all the participants**. Participants that voted Yes wait for a preCommit or doAbort request. They acknowledge preCommit requests and carry out doAbort requests.

Phase 3: The coordinator collects acknowledgments. When all are received, it commits and sends doCommit to the participants. Participants wait for doCommit. When it arrives, they commit.



Concurrency control

- locking
- optimistic
- timestamp



The danger of locking

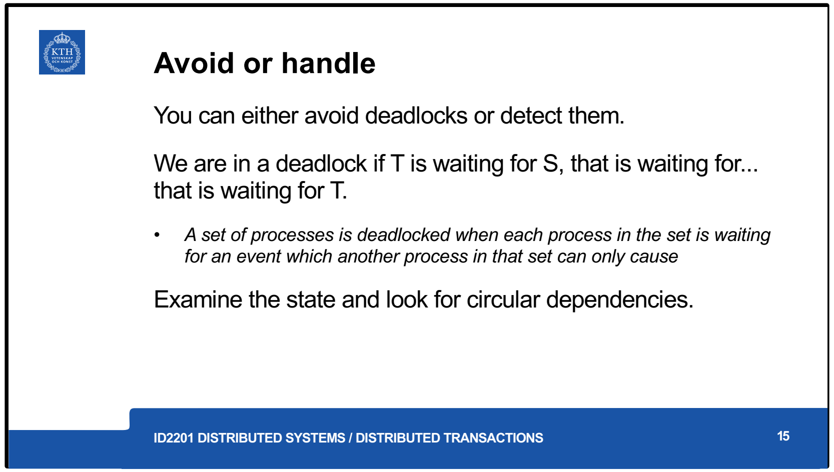
Assume we implement *strict two-phase locking* and need to take the locks for *foo*, *bar,* and *zot*.

What does it mean, and what should we do?

Two-phase locking (2PL)

By the 2PL protocol, locks are applied and removed in two phases:

1. ***Expanding phase***: locks are acquired, and no locks are released.
2. ***Shrinking phase***: locks are released, and no locks are acquired.



Avoid or handle

You can either avoid deadlocks or detect them.

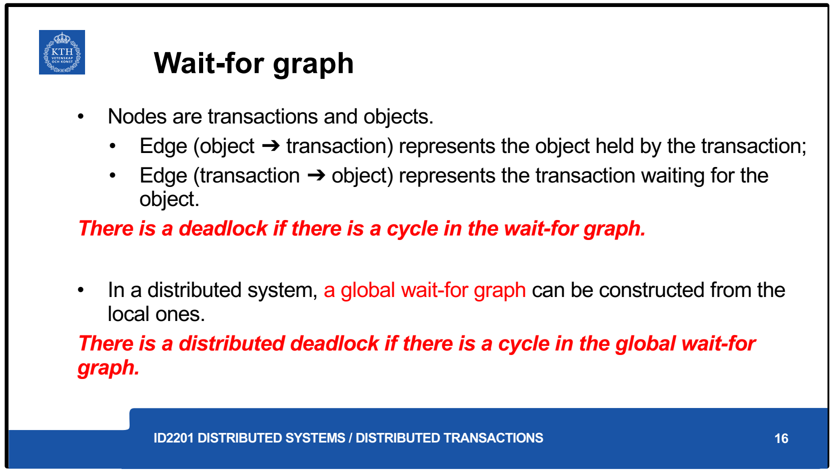
We are in a deadlock if T is waiting for S, that is waiting for... that is waiting for T.

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

Examine the state and look for circular dependencies.

ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED TRANSACTIONS 15

Most deadlock detection schemes operate by finding **cycles in the transaction wait-for graph**.
The wait-for graph is a directed graph in which nodes represent **transactions and objects**, and edges represent either an object ***held by*** a transaction (an edge object -> transaction) or a transaction waiting for an object (an edge transaction -> object). **There is a deadlock if there is a cycle in the wait-for graph.**
In a distributed system involving multiple servers accessed by multiple transactions, **a global wait-for graph can, in theory, be constructed from the local ones**. There can be a cycle in the global wait-for graph that is not in any single local one – that is, there can be a distributed deadlock.



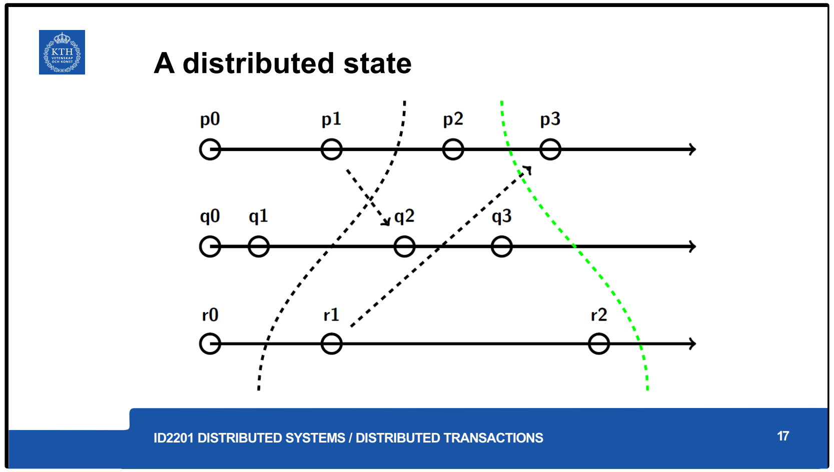
Wait-for graph

- Nodes are transactions and objects.
  - Edge (object ➔ transaction) represents the object held by the transaction;
  - Edge (transaction ➔ object) represents the transaction waiting for the object.

There is a deadlock if there is a cycle in the wait-for graph.

- In a distributed system, a global wait-for graph can be constructed from the local ones.

There is a distributed deadlock if there is a cycle in the global wait-for graph.

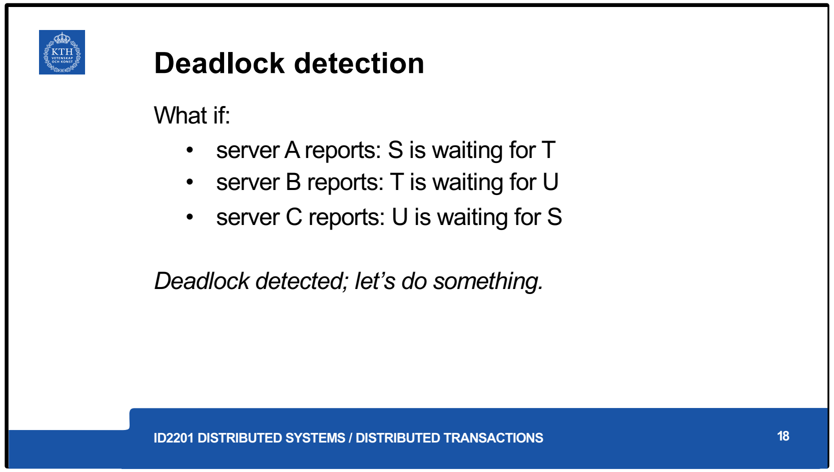


A distributed state

ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED TRANSACTIONS

17

Detection of a **distributed deadlock** requires **a cycle to be found in the global transaction wait-for graph** distributed among the servers involved in the transactions. The lock manager can build local wait-for graphs at each server.



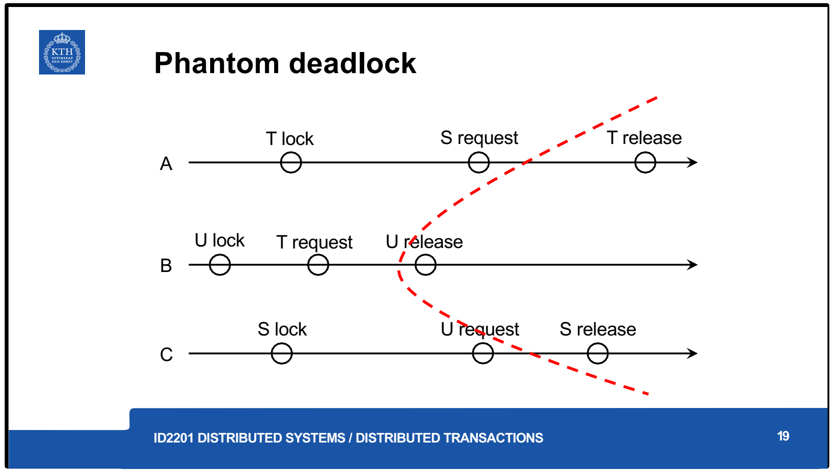
Deadlock detection

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.

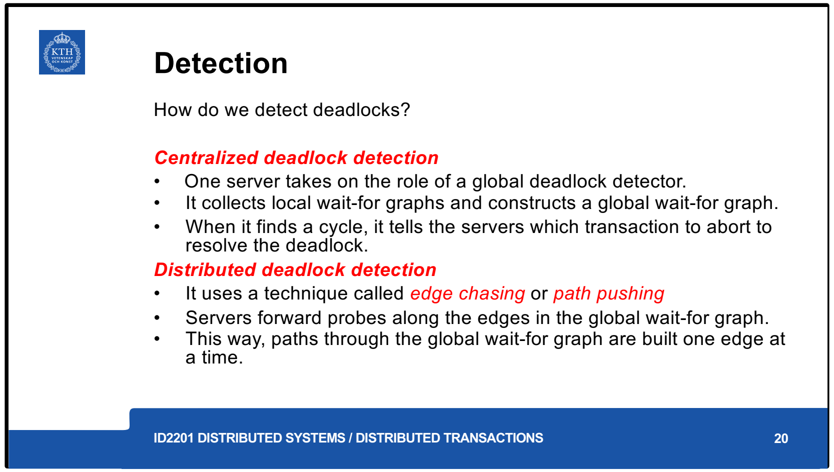
A simple solution is to use ***centralized deadlock detection***, in which one server takes on the role of a ***global deadlock detector***. From time to time, each server sends the latest copy of its local wait-for graph to the global deadlock detector, which amalgamates the information in the local graphs to construct a global wait-for graph. The global deadlock detector checks for cycles in the global wait-for graph. **When it finds a cycle, it decides how to resolve the deadlock and tells the servers which transaction to abort.**



Phantom deadlock

ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED TRANSACTIONS 19

A deadlock that is detected but is not really a deadlock is called a **phantom deadlock**. In ***distributed deadlock detection***, information about wait-for relationships between transactions is transmitted from one server to another. If there is a deadlock, the necessary information will eventually be collected in one place, and a cycle will be detected. ***As this procedure will take some time, there is a chance that one of the transactions that holds a lock (see U above) will have released it, in which case the deadlock will no longer exist.***



Detection

How do we detect deadlocks?

Centralized deadlock detection

- One server takes on the role of a global deadlock detector.
- It collects local wait-for graphs and constructs a global wait-for graph.
- When it finds a cycle, it tells the servers which transaction to abort to resolve the deadlock.

Distributed deadlock detection

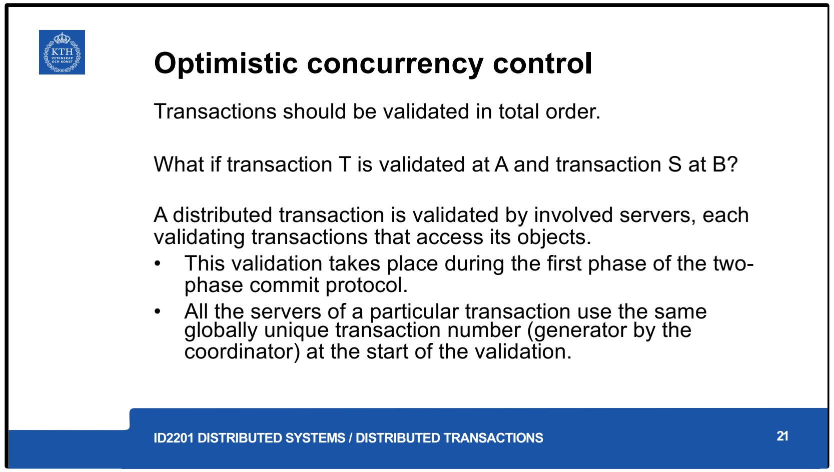
- It uses a technique called *edge chasing* or *path pushing*
- Servers forward probes along the edges in the global wait-for graph.
- This way, paths through the global wait-for graph are built one edge at a time.

ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED TRANSACTIONS 20

A simple solution is to use ***centralized deadlock detection***, in which one server takes on the role of a ***global deadlock detector***. From time to time, each server sends the latest copy of its local wait-for graph to the global deadlock detector, which amalgamates the information in the local graphs to construct a global wait-for graph. The global deadlock detector checks for cycles in the global wait-for graph. **When it finds a cycle, it decides how to resolve the deadlock and tells the servers which transaction to abort.**

A *distributed approach to deadlock detection* uses a technique called *edge chasing or path pushing*. In this approach, ***the global wait-for graph is not constructed***, but each server involved knows some of its edges. The servers attempt to find cycles by forwarding messages called ***probes***, which follow the graph's edges throughout the distributed system. ***A probe message consists of transaction wait-for relationships representing a path in the global wait-for graph***. ***This way, paths through the global wait-for graph are built one edge at a time.***

Before forwarding a probe, the server checks to see whether the transaction has just added caused the probe to contain a cycle. If this is the case, it has found a cycle in the graph, and a deadlock has been detected. When a cycle is detected, a transaction in the cycle is aborted to break the deadlock.



Optimistic concurrency control

Transactions should be validated in total order.

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

A distributed transaction is validated by involved servers, each validating transactions that access its objects.

- This validation takes place during the first phase of the two-phase commit protocol.
- All the servers of a particular transaction use the same globally unique transaction number (generator by the coordinator) at the start of the validation.

Recall that ***with optimistic concurrency control, each transaction is validated before it can commit.***
For a transaction Tv to be serializable with respect to an overlapping transaction Ti, their operations must conform to the following rules:

1. Tv writes; Ti reads: Ti must not read objects written by Tv.
2. Tv reads; Ti writes: Tv must not read objects written by Ti.
3. Tv writes; Ti writes: Ti must not write objects written by Tv and visa versa.

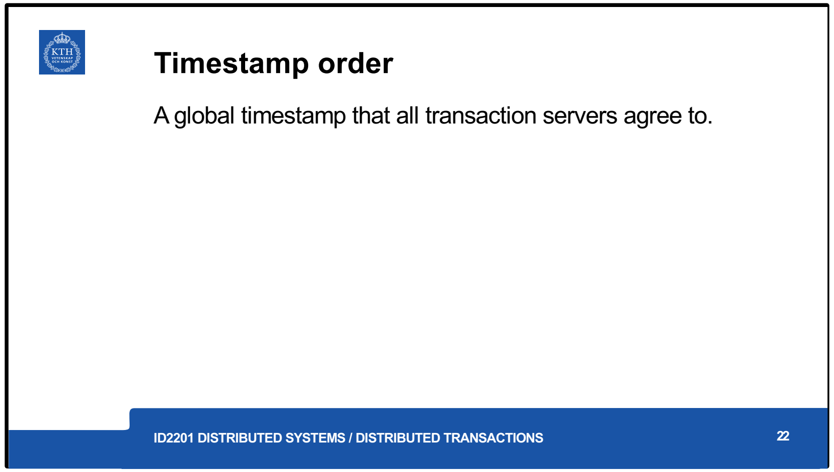
Backward validation: rule 1 is achieved as Ti reads before Tv starts validation (reads are not affected by writes); check rule 2 -- requires to keep write sets. The read set of the transaction being validated is compared with the write sets of other transactions that have already been committed - > the only way to resolve any conflicts is to abort the transaction that is undergoing validation.
Forward validation: rule 2 is achieved automatically; check rule 1 – abort the validating transaction or

conflicting future transaction or delay the validation check.
Read set is usually larger than write sets. Backward validation compares a possibly large read set against the old write sets, whereas forward validation checks a small write set against the read sets of active transactions.

Transaction numbers are assigned at the start of validation, and transactions are serialized according to the order of the transaction numbers. A distributed transaction is validated by a collection of independent servers, each validating transactions that access its objects. This validation takes place during the first phase of the two-phase commit protocol.
In a centralized validation: one validation and update at a time

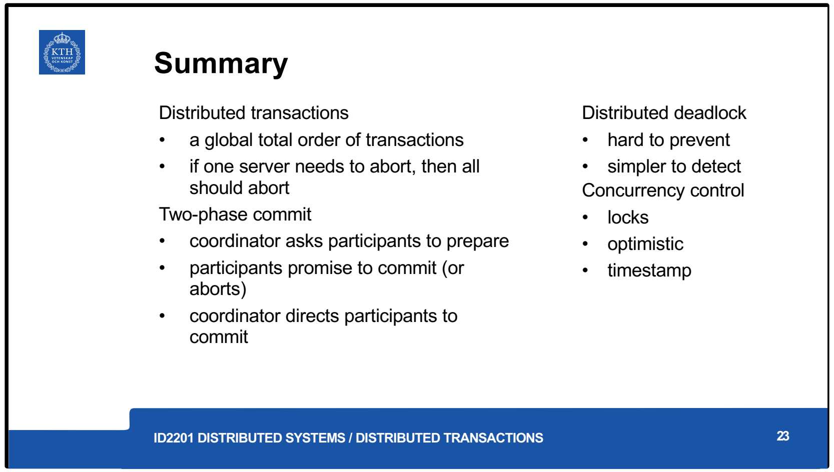
unique transaction number at the start of the validation. The coordinator of the two-phase commit protocol is responsible for generating the globally unique transaction number and passing it to the participants in the canCommit? messages.

As different servers may coordinate different transactions, the servers must (as in the distributed timestamp ordering protocol) have an agreed order for the transaction numbers they generate.



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