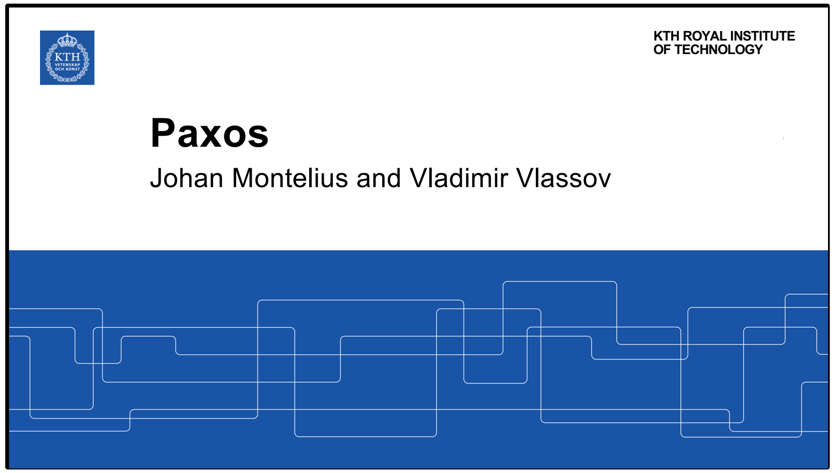
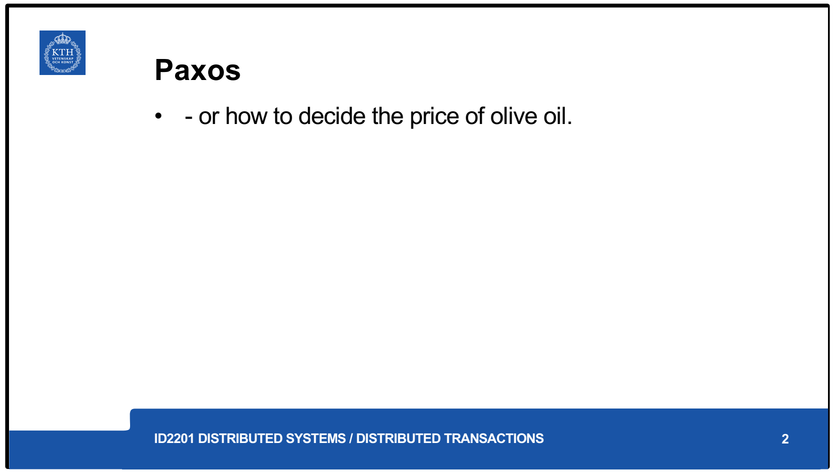


KTH ROYAL INSTITUTE OF TECHNOLOGY

Paxos

Johan Montelius and Vladimir Vlassov

Consensus: For example, in Chapter 2, we described a situation in which two armies should decide consistently to attack the common enemy or retreat. Similarly, we may require that all the correct processes controlling a spaceship's engines decide to either 'proceed' or 'abort' after each has proposed one action. In a transaction to transfer funds from one account to another, the processes involved must consistently agree to perform the respective debit and credit. In mutual exclusion, the processes agree on which process can enter the critical section. In an election, the processes agree on which is the elected process. In totally ordered multicast, the processes agree on the order of message delivery.

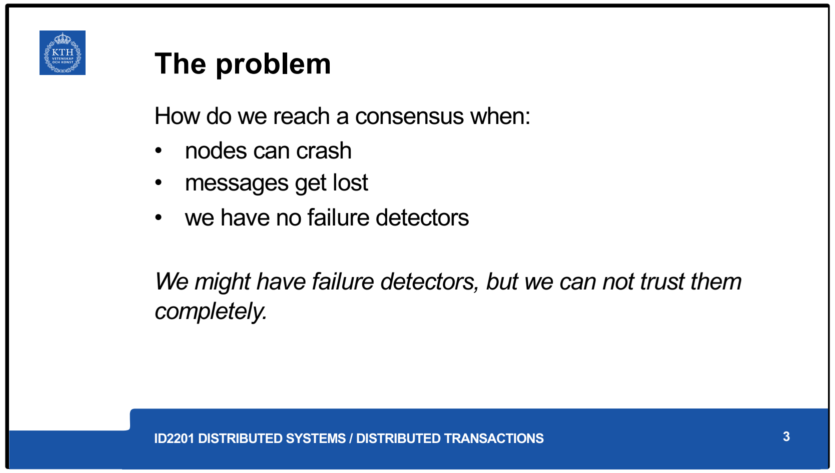


Paxos

- - or how to decide the price of olive oil.

ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED TRANSACTIONS 2

Lamport's 'Paxos' or 'Part-time Parliament' algorithm [1998] provides a way of reaching distributed agreement despite volatility – the participating processes are assumed to disappear and reappear regularly and independently. However, the algorithm depends on each process having access to its persistent store. **Paxos is a family of protocols providing distributed consensus. Consensus protocols operate over a set of replicas to reach an agreement between the servers managing the replicas to update to a common value. Paxos is, therefore, fundamentally a distributed consensus protocol for asynchronous systems. Leslie Lamport introduced the algorithm in 1989 in a paper** called "The Part-Time Parliament" [Lamport 1989, Lamport 1998]. Inspired by his description of Byzantine Generals (as discussed in Section 15.5.1), he again presented the algorithm with an analogy, referring to the behavior of a mythical parliament on the Greek island of Paxos.



The problem

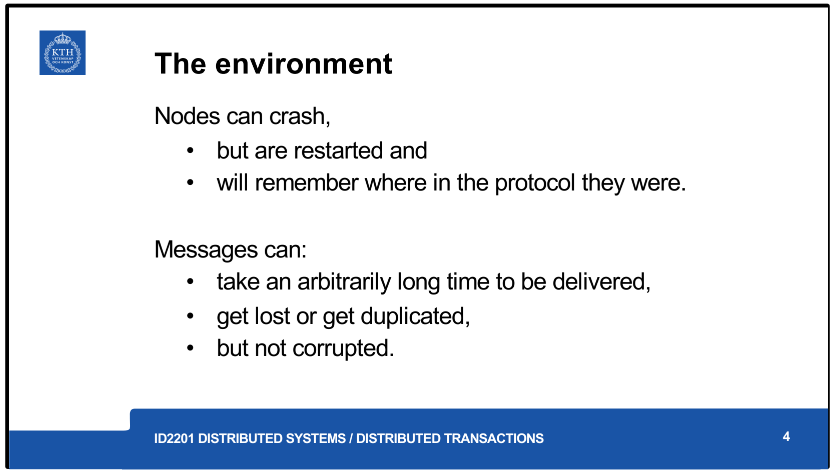
How do we reach a consensus when:

- nodes can crash
- messages get lost
- we have no failure detectors

We might have failure detectors, but we can not trust them completely.

ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED TRANSACTIONS 3

Recall that it is impossible to guarantee consistency in asynchronous systems but that various techniques have been proposed to work around this result.
Termination: Eventually, each correct process sets its decision variable. **Agreement**: The decision value of all correct processes is the same: if pi and pj are correct and have entered the decided state, then the decision variables di = dj (for all i, j). **Integrity**: If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value. **Paxos works by ensuring correctness but not liveness** – that is, Paxos is not guaranteed to terminate (we return to this issue below once we have looked at the details of the algorithm).



The environment

Nodes can crash,

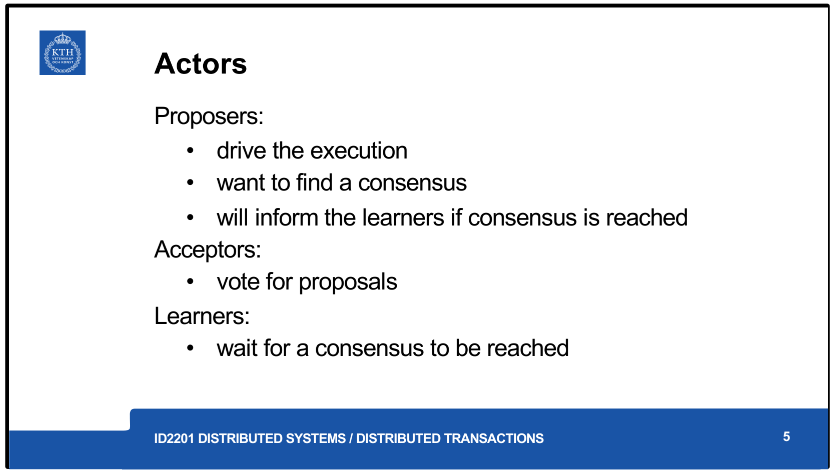
- but are restarted and
- will remember where in the protocol they were.

Messages can:

- take an arbitrarily long time to be delivered,
- get lost or get duplicated,
- but not corrupted.

ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED TRANSACTIONS 4

Will remember where in the protocol they were, what they have promised and voted for



Actors

Proposers:

- drive the execution
- want to find a consensus
- will inform the learners if consensus is reached

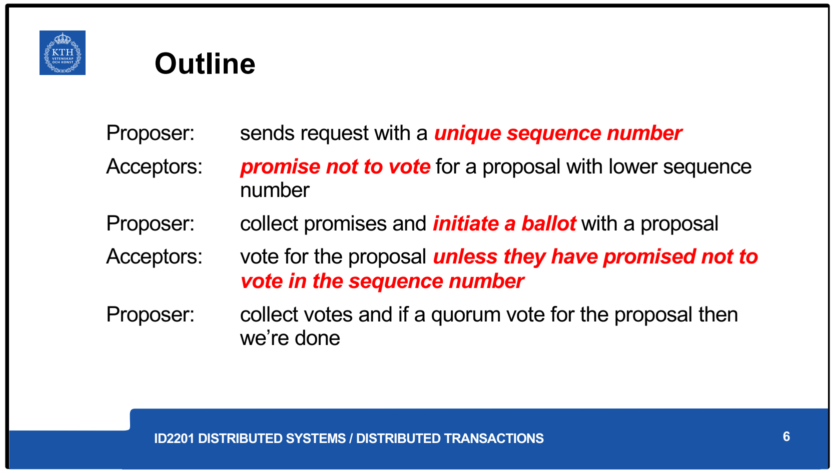
Acceptors:

- vote for proposals

Learners:

- wait for a consensus to be reached

In the original description, there is one role of a parliament member, but to distinguish its actions, we define three roles: proposer, acceptor, and learner. Assume we have five proposers – proposers want to find out the price of oil – they drive the execution. A proposer does not have its favorite price that it wants to push but **instead wants to find a price**. If no one has an idea, then it proposes a value. When it finds out the price, it informs learners. Assume we have three or five acceptors – if we have a majority of acceptors (say 3) voting and a majority of them (say 2) vote for a value, then we are done. Acceptors do not know if the consensus has been reached; they vote and promise. Eventually, one proposer will win and get the price that the majority of acceptors have voted for. The other proposers will also learn that value. Learners are external and want to find out the price. A learner asks one of the P about the price and waits for the algorithm to terminate and get the answer.

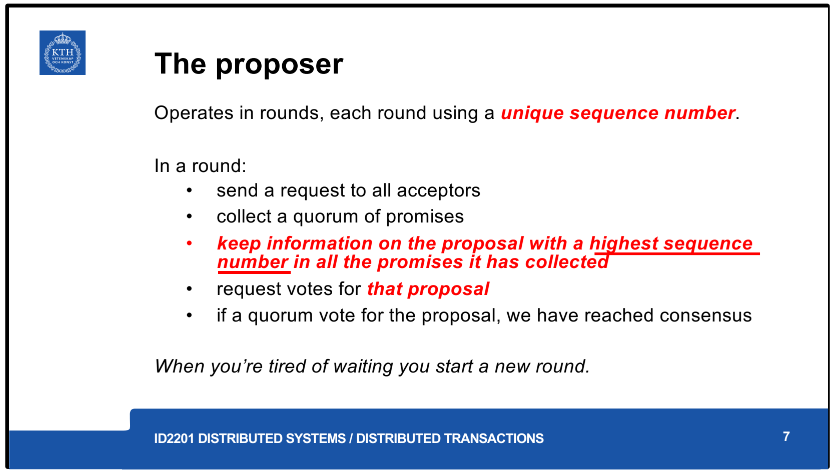


Outline

Proposer:	sends request with a ***unique sequence number***
Acceptors:	***promise not to vote*** for a proposal with lower sequence number
Proposer:	collect promises and ***initiate a ballot*** with a proposal
Acceptors:	vote for the proposal ***unless they have promised not to vote in the sequence number***
Proposer:	collect votes and if a quorum vote for the proposal then we're done

Phase 1a: prepare – request to vote. Phase 1b: promise with previous ballot or no promise; Phase 2a: Vote for ballot (the previous value with hight seq. Number); Phase 2b: Accepted to proposer and to the learners. Each proposer needs to generate a unique sequence number for a new round of the algorithm. For example, in order to guarantee uniqueness a proposer will tie the number with its PID. They can choose a number that is higher than the highest value it knows such than it module N is equal to its PID. Assume , a proposer send a request to vote in the sequence number 17, and an acceptor replies with a promise including an information about what they have voted already, if any. **Actually, the promise is NOT to vote for anything lower than** 17. If the proposer collects the majority of promises ,it can ask to vote in the sequence number it proposed, i.e. 17 – it's similar to the 2-phase commit protocol. Assume that a new proposal to vote comes with the sequence number 20, then an acceptor should promise not to vote for anything less than 20. When it receives a ballot with the sequence number 17, it refuses to vote - it can

either send a negative response or just ignore the ballot. The proposer with 17 can quickly come with another proposal number 22 and get some promises while another proposer can come with the number 42, etc. There is a battle of proposers. ***Very important note: An acceptor must include in its promise information on what he has already voted in last round if any.***



The proposer

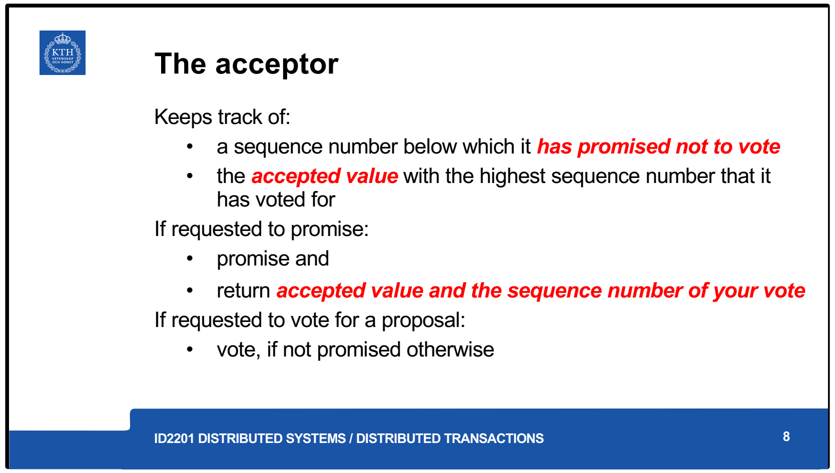
Operates in rounds, each round using a ***unique sequence number***.

In a round:

- send a request to all acceptors
- collect a quorum of promises
- ***keep information on the proposal with a highest sequence number in all the promises it has collected***
- request votes for ***that proposal***
- if a quorum vote for the proposal, we have reached consensus

When you're tired of waiting you start a new round.

Assume the proposer has collected promises in sequence number 17: one of promises reports that an acceptor has voted for 5 EUR in sequence number 10 and another promises says that its acceptor has voted for 6 EUR in the sequence number 9. In this case, the proposer will keep the value of 5 EUR in the proposal with the highest sequence number that is 10. This is very crucial for the algorithm because this means that there was already a majority for the ballot of 5 EUR in the sequence number 10 (as someone has already voted) then the proposer of the sequence number of 17 keeps it – it promotes the ballot further -- the majority will not go away but it will grow. If a proposer can not reach a consensus – no promises or no acknowledgments on voting, then at some point of time it starts a new round with a new unique sequence number.



The acceptor

Keeps track of:

- a sequence number below which it ***has promised not to vote***
- the ***accepted value*** with the highest sequence number that it has voted for

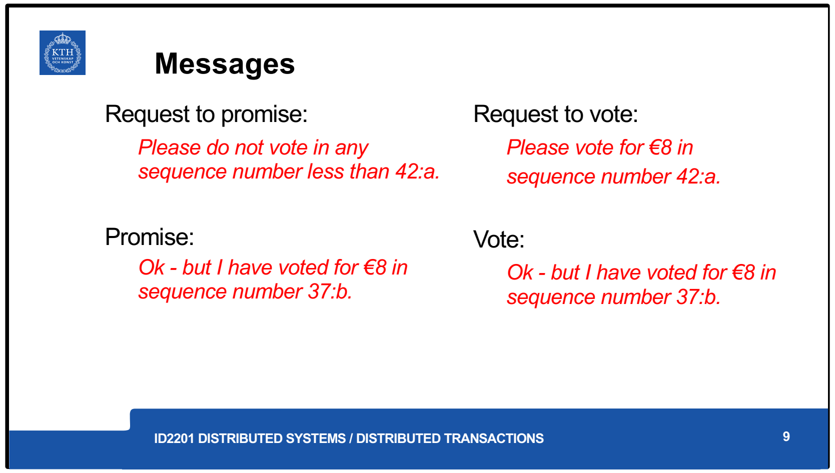
If requested to promise:

- promise and
- return ***accepted value and the sequence number of your vote***

If requested to vote for a proposal:

- vote, if not promised otherwise

An acceptor should write down what it has promised, e.g. a promise not to vote less than 17. If a new request comes that asks not to vote less than 10, the acceptor can reply positively as it does not contradict to the first promise, but it will NOT vote in the sequence number 10. In this case the acceptor gives a false expectation. If the accepter has already vote it should keep (write down) this vote, e.g. it has voted for a ballot of 12 EUR in a sequence number 22. The acceptor must always keep a ballot with the highest sequence number it has already voted for. The consensus must be reached in the same sequence number. The algorithm does not guarantee that it terminates.



Messages

Request to promise:

Please do not vote in any sequence number less than 42:a.

Promise:

Ok - but I have voted for €8 in sequence number 37:b.

Request to vote:

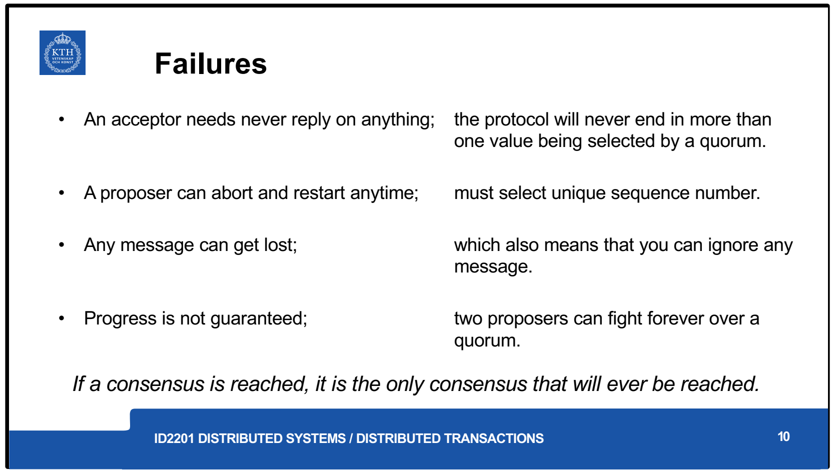
Please vote for €8 in sequence number 42:a.

Vote:

Ok - but I have voted for €8 in sequence number 37:b.

ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED TRANSACTIONS 9

The request to vote is sent out when there is a majority of promisses.



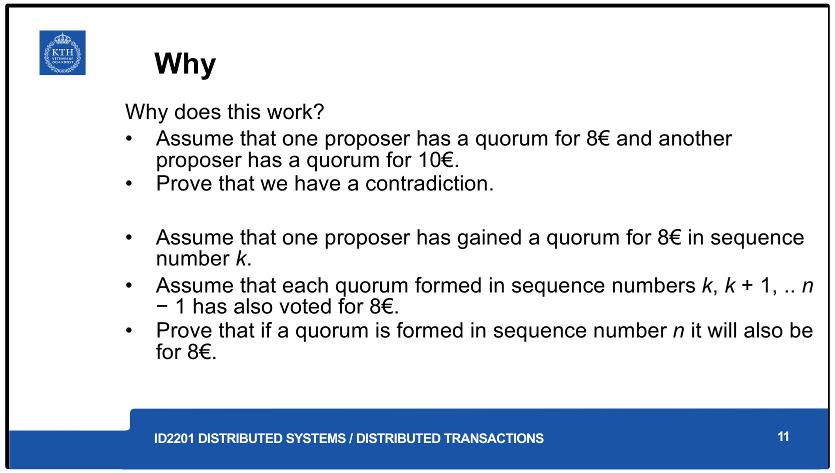
Failures

• An acceptor needs never reply on anything;	the protocol will never end in more than one value being selected by a quorum.
• A proposer can abort and restart anytime;	must select unique sequence number.
• Any message can get lost;	which also means that you can ignore any message.
• Progress is not guaranteed;	two proposers can fight forever over a quorum.

If a consensus is reached, it is the only consensus that will ever be reached.

ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED TRANSACTIONS 10

Let’s play the game: we have 3 acceptors and 5 proposers.
An acceptor does not have to reply to anything – nothing wrong with this.



Why

Why does this work?

- Assume that one proposer has a quorum for 8€ and another proposer has a quorum for 10€.
- Prove that we have a contradiction.

- Assume that one proposer has gained a quorum for 8€ in sequence number *k*.
- Assume that each quorum formed in sequence numbers *k*, *k* + 1, .. *n* − 1 has also voted for 8€.
- Prove that if a quorum is formed in sequence number *n* it will also be for 8€.

ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED TRANSACTIONS 11

Prove by contraction: assume that something that should not happen, happens. Assume the one proposer has gain a quorum for 8 EUR and another in 10 EUR. Assume 3 acceptors and the first one has 2 out of 3 votes for 8 EUR in the sequence number k while the second one has 2 votes for 10 EUR in the sequence number m. … **Should consider and end up with contradiction**.

The second way to prove is an induction: