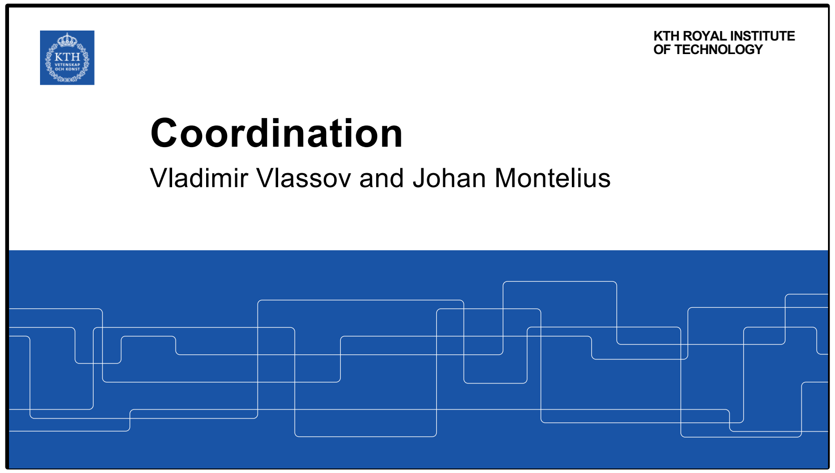
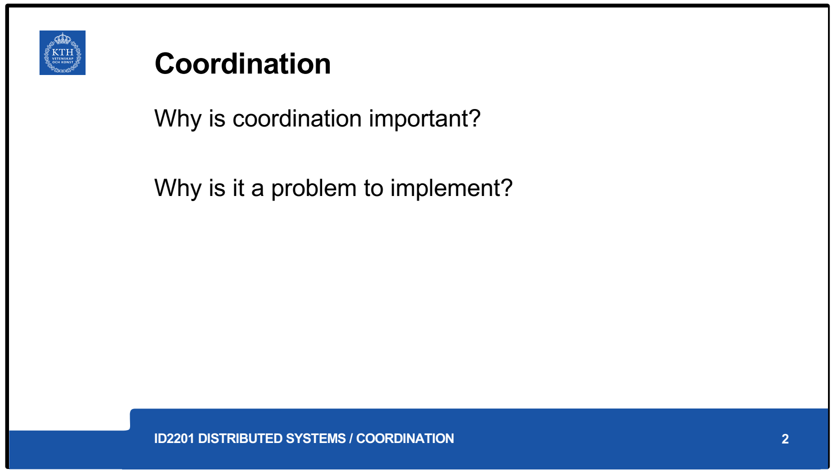


KTH ROYAL INSTITUTE
OF TECHNOLOGY

Coordination

Vladimir Vlassov and Johan Montelius



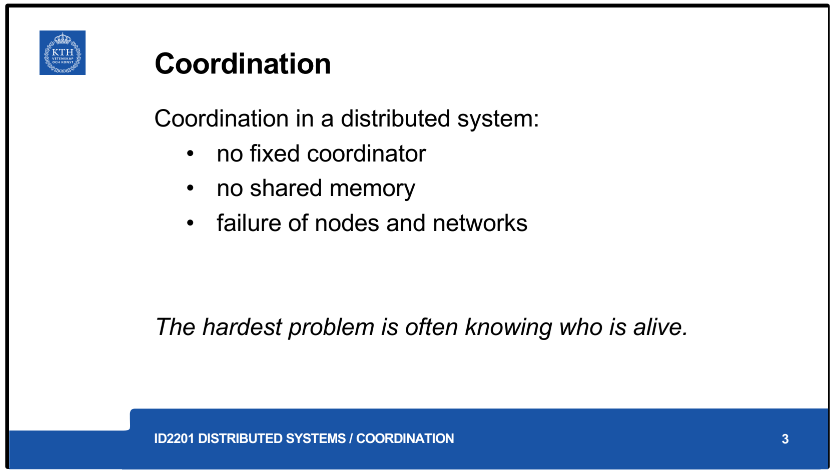
Coordination

Why is coordination important?

Why is it a problem to implement?

ID2201 DISTRIBUTED SYSTEMS / COORDINATION 2

Chapter 15 in the textbook introduces a collection of algorithms whose goals vary, but they share a fundamental aim in distributed systems: ***for a set of processes to coordinate their actions or to agree on one or more values***. The computers must coordinate their actions correctly with respect to shared resources. The computers must be able to do so ***even where there is no fixed master-slave relationship between the components*** (which would make coordination particularly simple). Avoid fixed master-slave relationships because we often require our systems to keep working correctly even if failures occur. Hence, we must avoid single points of failure, such as fixed masters. **An important distinction** will be whether the distributed system under study ***is asynchronous or synchronous***. In an asynchronous system, we can make no timing assumptions. In a synchronous system, we shall assume that there are bounds on the maximum message transmission delay, the time to execute each process step, and clock drift rates. ***The synchronous assumptions allow us to use timeouts to detect process crashes.***



Coordination

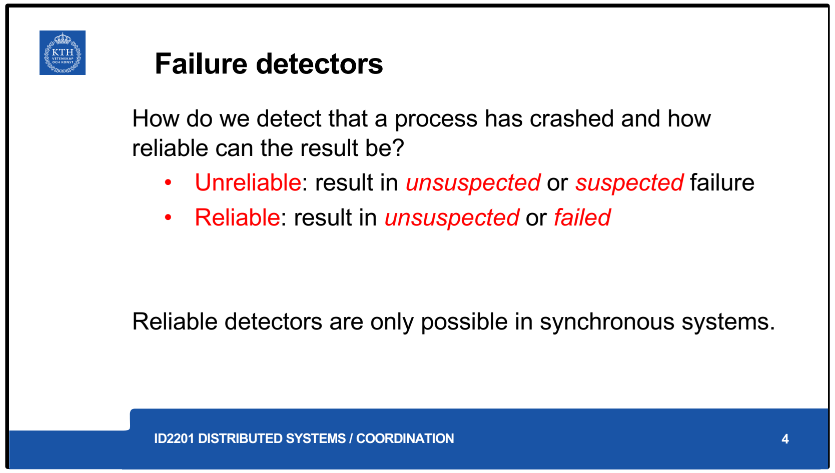
Coordination in a distributed system:

- no fixed coordinator
- no shared memory
- failure of nodes and networks

The hardest problem is often knowing who is alive.

ID2201 DISTRIBUTED SYSTEMS / COORDINATION 3

An important distinction will be whether the distributed system under study ***is asynchronous or synchronous***. In an asynchronous system, we can make no timing assumptions. In a synchronous system, we shall assume that there are bounds on the maximum message transmission delay, the time to execute each process step, and clock drift rates. ***The synchronous assumptions allow us to use timeouts to detect process crashes.***



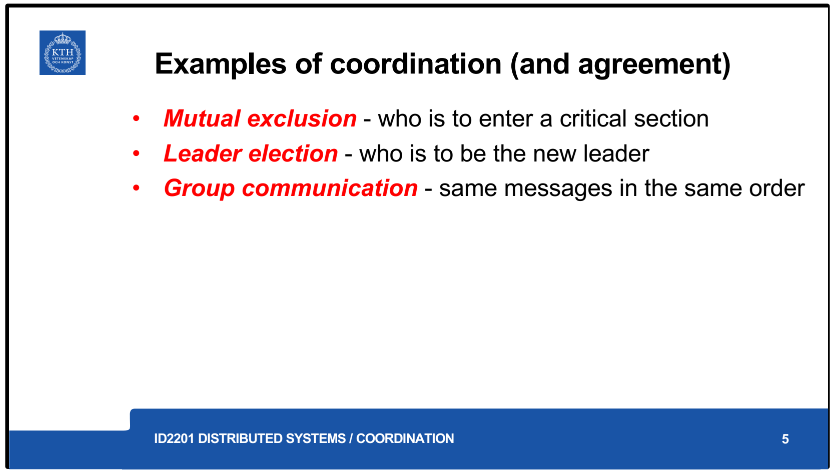
Failure detectors

How do we detect that a process has crashed and how reliable can the result be?

- Unreliable: result in *unsuspected* or *suspected* failure
- Reliable: result in *unsuspected* or *failed*

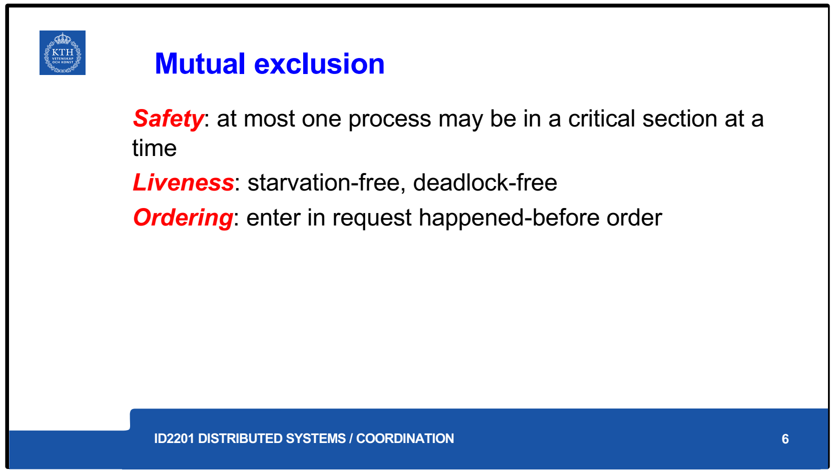
Reliable detectors are only possible in synchronous systems.

Unreliable detector vs. reliable detector



Examples of coordination (and agreement)

- ***Mutual exclusion*** - who is to enter a critical section
- ***Leader election*** - who is to be the new leader
- ***Group communication*** - same messages in the same order

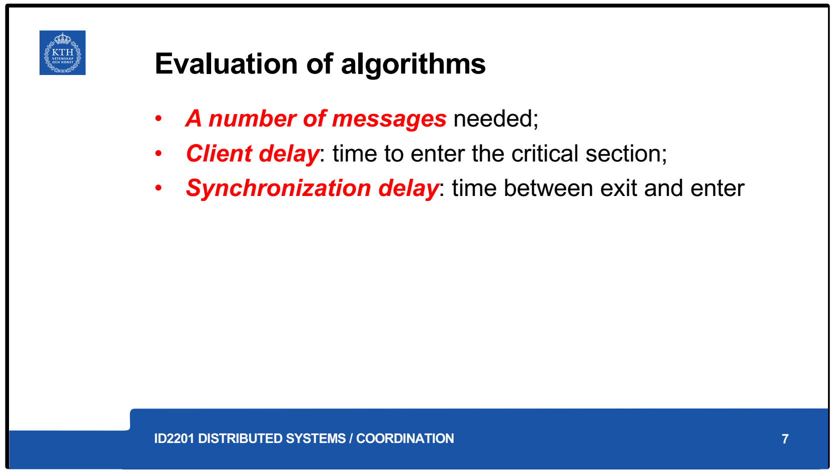


Mutual exclusion

Safety: at most one process may be in a critical section at a time

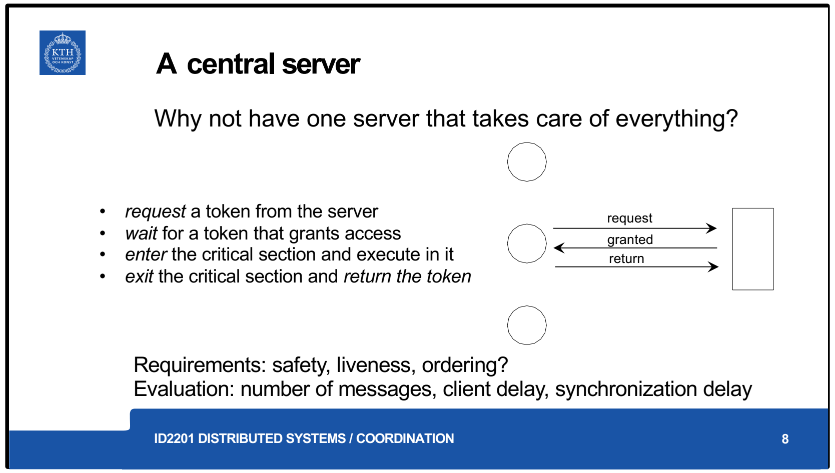
Liveness: starvation-free, deadlock-free

Ordering: enter in request happened-before order



Evaluation of algorithms

- ***A number of messages*** needed;
- ***Client delay***: time to enter the critical section;
- ***Synchronization delay***: time between exit and enter



A central server

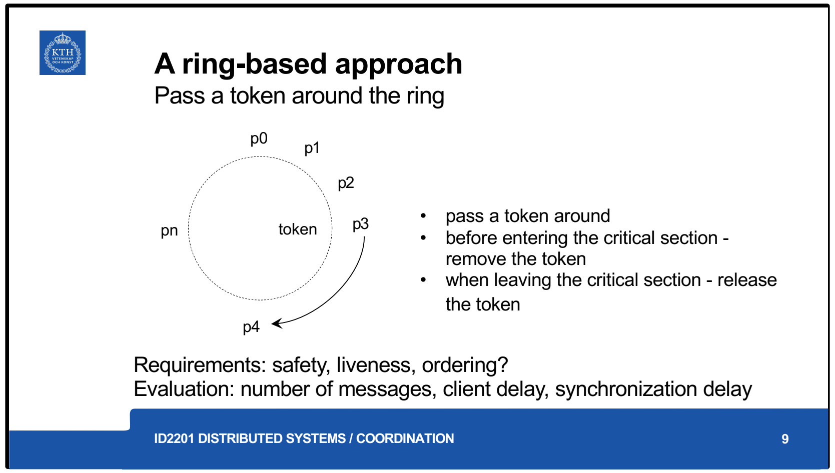
Why not have one server that takes care of everything?

- *request* a token from the server
- *wait* for a token that grants access
- *enter* the critical section and execute in it
- *exit* the critical section and *return the token*

request
granted
return

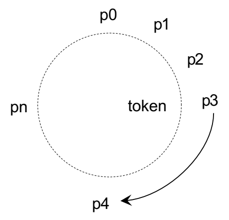
Requirements: safety, liveness, ordering?
Evaluation: number of messages, client delay, synchronization delay

Enter: 2 messages (request – grated),
Exit: 1 release message; no delay if asynchronous send
Client delay = round trip time
Synchronization delay: round trip: release - granted



A ring-based approach

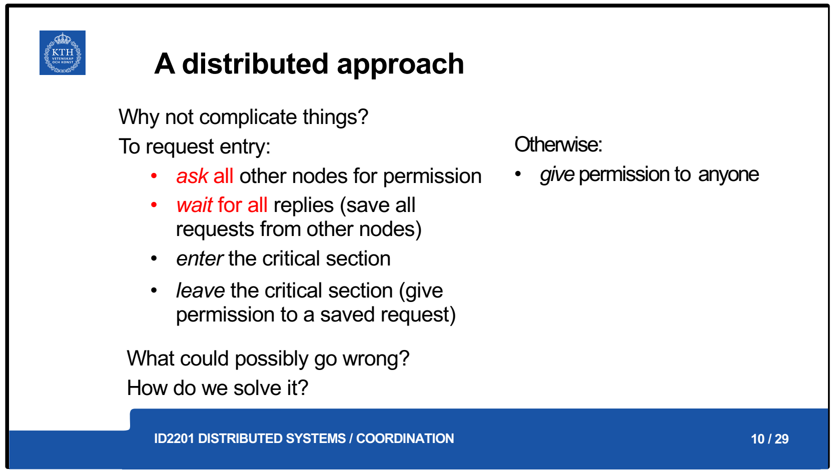
Pass a token around the ring



- pass a token around
- before entering the critical section - remove the token
- when leaving the critical section - release the token

Requirements: safety, liveness, ordering?
Evaluation: number of messages, client delay, synchronization delay

number of messages: enter: =O(N) = [0,N]; exit: 1 message
synchronization delay = one exits another enters = anywhere from 1 to N message transmissions


A distributed approach

Why not complicate things?

To request entry:

- *ask* all other nodes for permission
- *wait* for all replies (save all requests from other nodes)
- *enter* the critical section
- *leave* the critical section (give permission to a saved request)

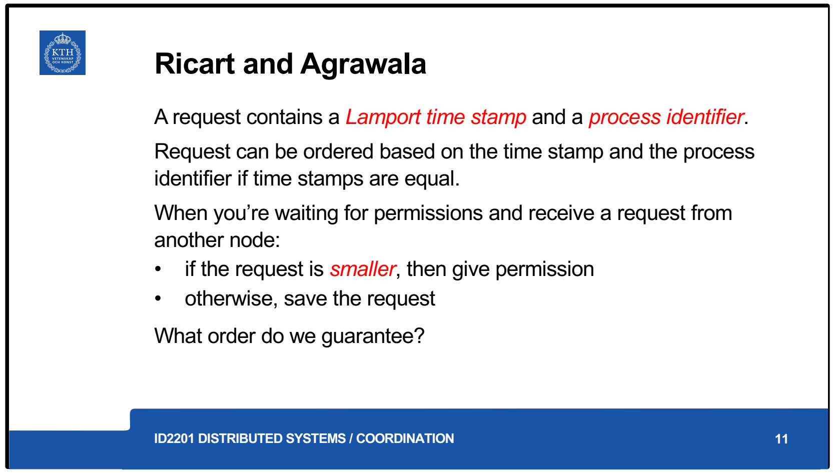
Otherwise:

- *give* permission to anyone

What could possibly go wrong?

How do we solve it?

None of the algorithms we described would tolerate the loss of messages if the channels were unreliable.



Ricart and Agrawala

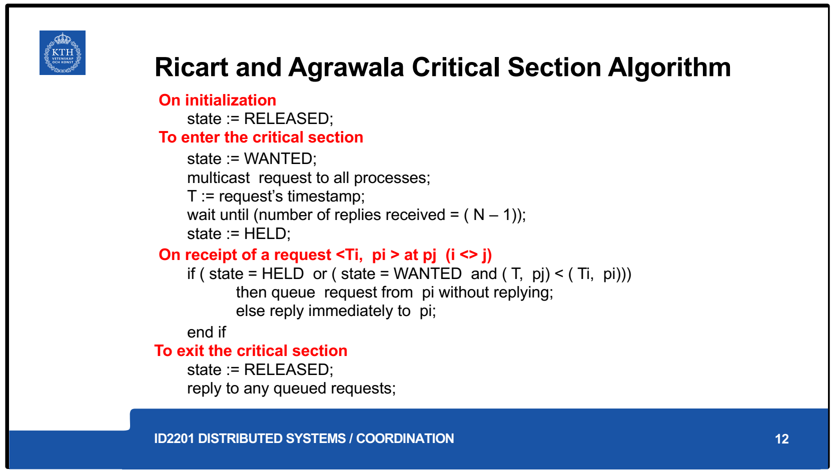
A request contains a *Lamport time stamp* and a *process identifier*.

Request can be ordered based on the time stamp and the process identifier if time stamps are equal.

When you're waiting for permissions and receive a request from another node:

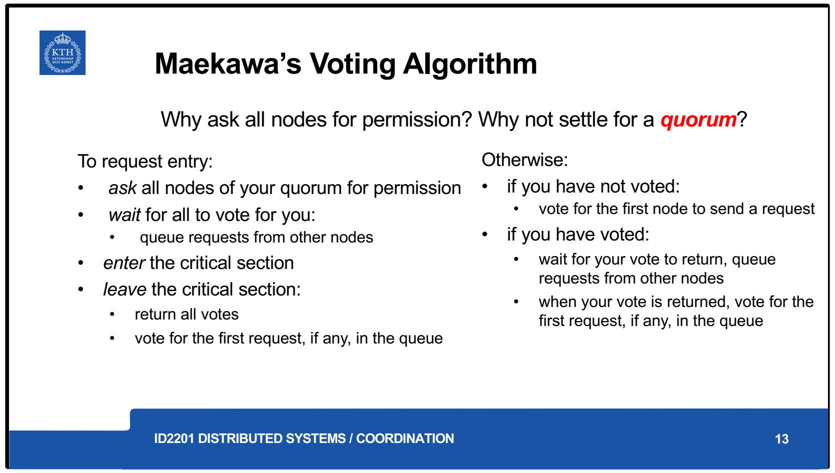
- if the request is *smaller*, then give permission
- otherwise, save the request

What order do we guarantee?



Ricart and Agrawala Critical Section Algorithm

```
On initialization
    state := RELEASED;
To enter the critical section
    state := WANTED;
    multicast  request to all processes;
    T := request's timestamp;
    wait until (number of replies received = ( N – 1));
    state := HELD;
On receipt of a request <Ti,  pi > at pj  (i <> j)
    if ( state = HELD  or ( state = WANTED  and ( T,  pj) < ( Ti,  pi)))
            then queue  request from  pi without replying;
            else reply immediately to  pi;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```



Maekawa's Voting Algorithm

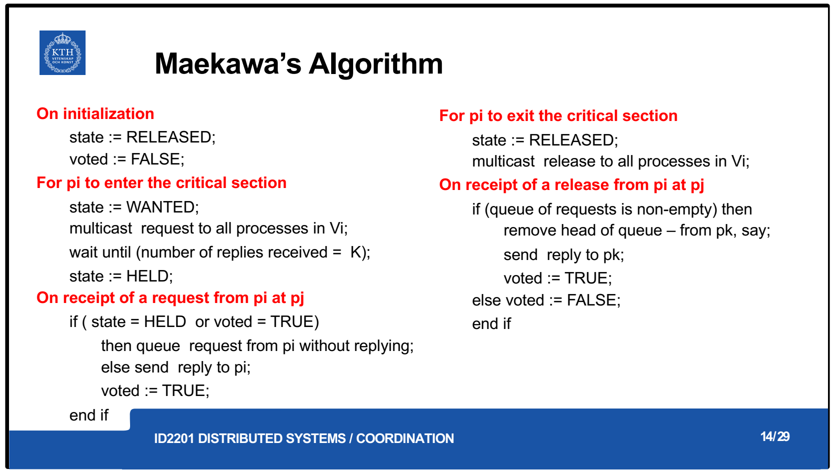
Why ask all nodes for permission? Why not settle for a ***quorum***?

To request entry:

- *ask* all nodes of your quorum for permission
- *wait* for all to vote for you:
  - queue requests from other nodes
- *enter* the critical section
- *leave* the critical section:
  - return all votes
  - vote for the first request, if any, in the queue

Otherwise:

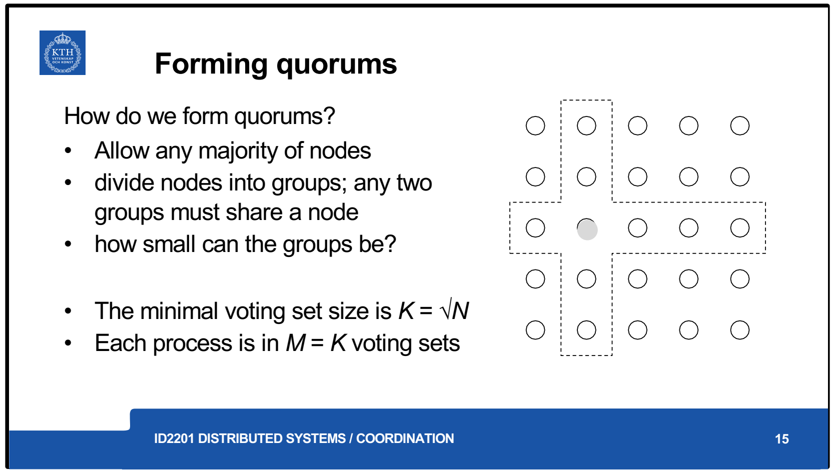
- if you have not voted:
  - vote for the first node to send a request
- if you have voted:
  - wait for your vote to return, queue requests from other nodes
  - when your vote is returned, vote for the first request, if any, in the queue



Maekawa's Algorithm

```
On initialization
    state := RELEASED;
    voted := FALSE;
For pi to enter the critical section
    state := WANTED;
    multicast  request to all processes in Vi;
    wait until (number of replies received =  K);
    state := HELD;
On receipt of a request from pi at pj
    if ( state = HELD  or voted = TRUE)
        then queue  request from pi without replying;
        else send  reply to pi;
        voted := TRUE;
    end if
```

```
For pi to exit the critical section
    state := RELEASED;
    multicast  release to all processes in Vi;
On receipt of a release from pi at pj
    if (queue of requests is non-empty) then
        remove head of queue – from pk, say;
        send  reply to pk;
        voted := TRUE;
    else voted := FALSE;
    end if
```

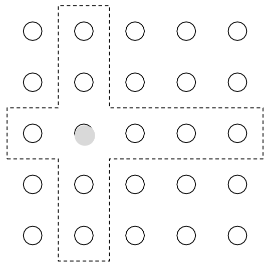


Forming quorums

How do we form quorums?

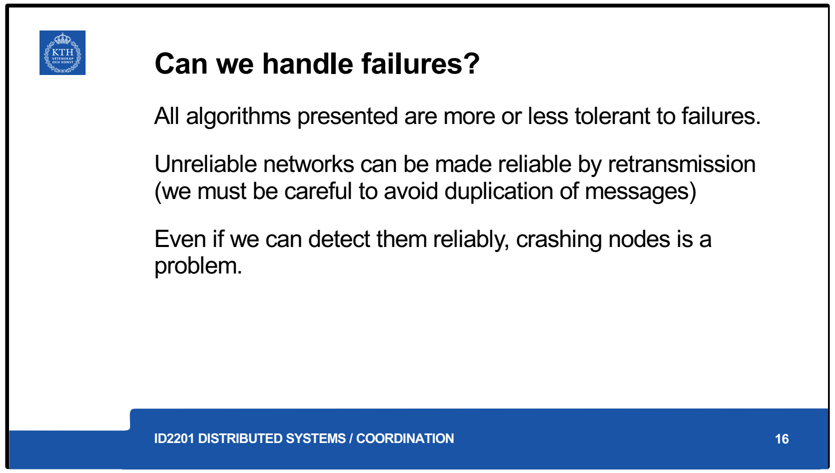
- Allow any majority of nodes
- divide nodes into groups; any two groups must share a node
- how small can the groups be?

- The minimal voting set size is $K = \sqrt{N}$
- Each process is in $M = K$ voting sets



Maekawa showed that the optimal solution, which minimizes K (size of the voting set) and allows the processes to achieve mutual exclusion, has K ≈√N and M = K (so that each process is in as many of the voting sets as there are elements in each one of those sets). It is nontrivial to calculate the optimal sets Ri. As an approximation, a simple way of deriving sets Ri such that Ri ≈ 2 √N is to place the processes in an √N by √N matrix and let Vi be the union of the row and column containing pi

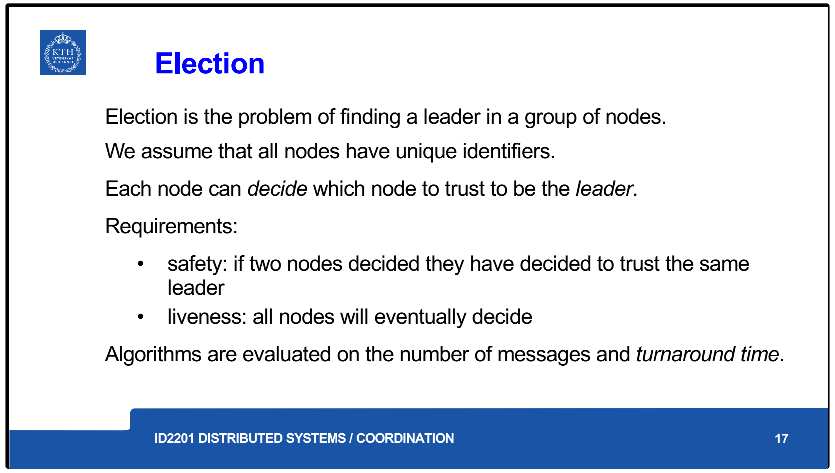
Unfortunately, the algorithm is ***deadlock-prone***. See page 640


Can we handle failures?

All algorithms presented are more or less tolerant to failures.

Unreliable networks can be made reliable by retransmission (we must be careful to avoid duplication of messages)

Even if we can detect them reliably, crashing nodes is a problem.



Election

Election is the problem of finding a leader in a group of nodes.

We assume that all nodes have unique identifiers.

Each node can *decide* which node to trust to be the *leader*.

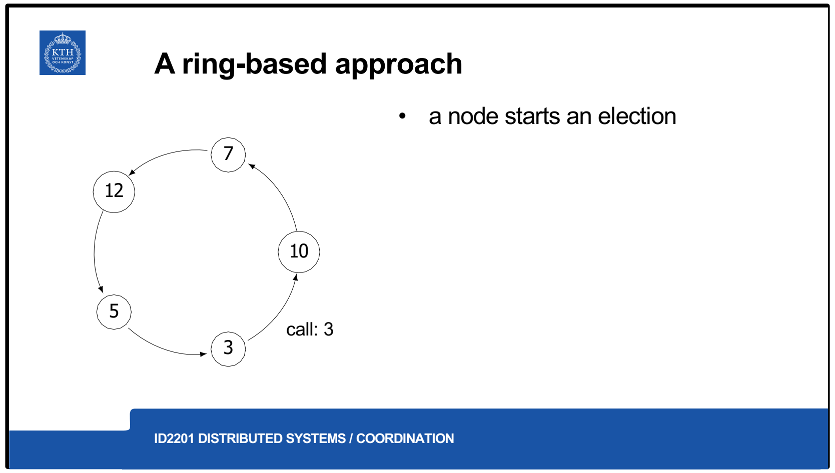
Requirements:

- safety: if two nodes decided they have decided to trust the same leader
- liveness: all nodes will eventually decide

Algorithms are evaluated on the number of messages and *turnaround time*.

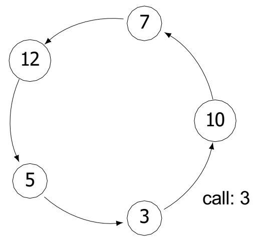
ID2201 DISTRIBUTED SYSTEMS / COORDINATION 17

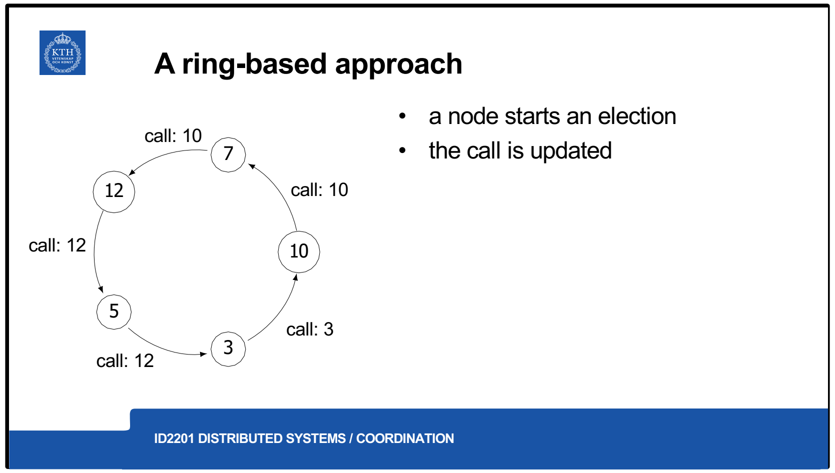
We measure the performance of an election algorithm by its **total network bandwidth utilization** (which is proportional to the total number of messages sent) and by the **turnaround time** for the algorithm: the number of serialized message transmission times between the initiation and termination of a single run.



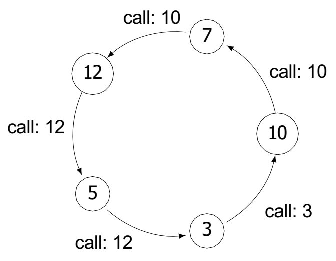
A ring-based approach

- a node starts an election

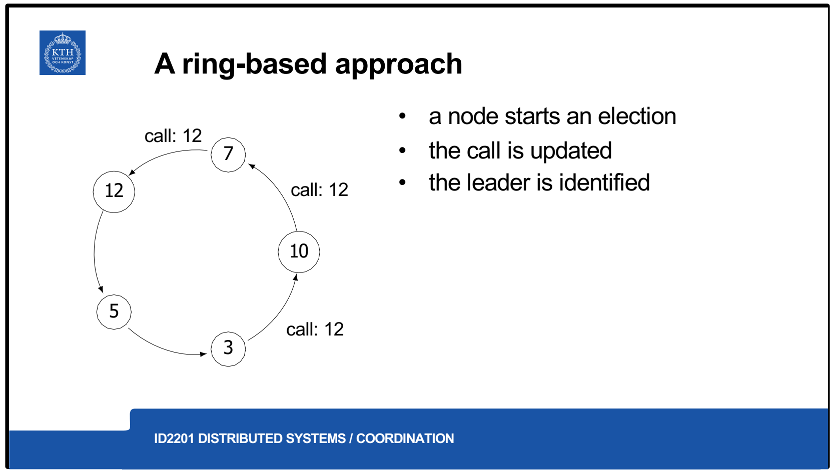




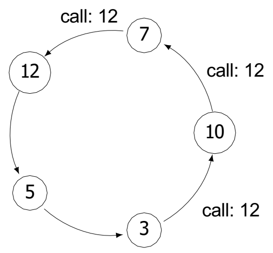
A ring-based approach



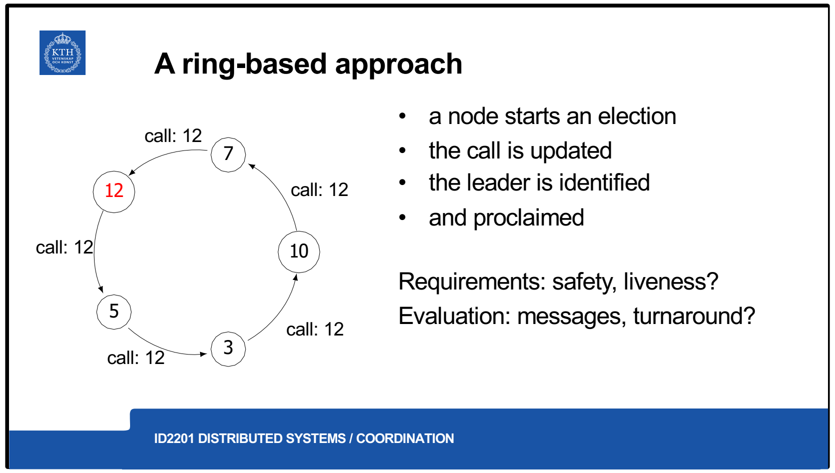
- a node starts an election
- the call is updated



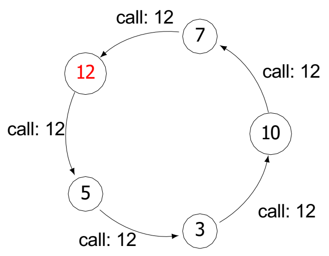
A ring-based approach



- a node starts an election
- the call is updated
- the leader is identified



A ring-based approach



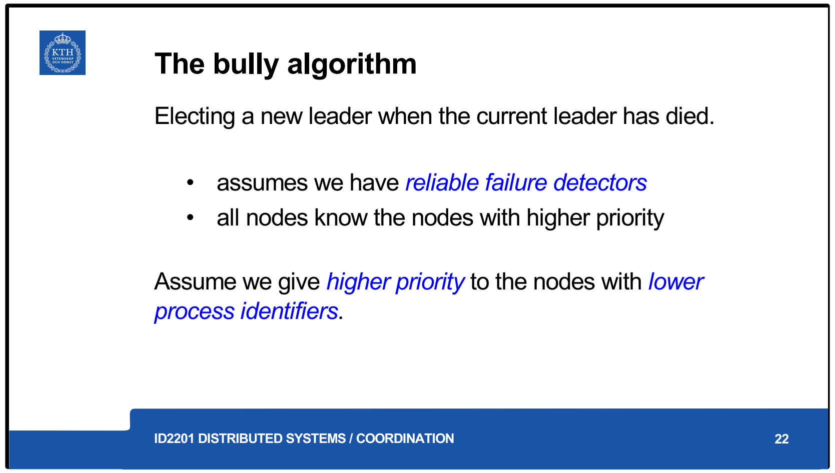
- a node starts an election
- the call is updated
- the leader is identified
- and proclaimed

Requirements: safety, liveness?

Evaluation: messages, turnaround?

ID2201 DISTRIBUTED SYSTEMS / COORDINATION

(3N – 1) messages; The **turnaround** time is also (3N – 1) since these messages are sent sequentially. **It is easy to see that condition E1 (safety)** is met. All identifiers are compared since a process must receive its identifier back before sending an elected message. For any two processes, the one with the larger identifier will not pass on the other’s identifier. It is, therefore, impossible that both should receive their identifier back. **Condition E2 (liveness) follows immediately from the guaranteed traversals of the ring (there are no failures).** Note how the non-participant and participant states are used so that duplicate messages arising when two processes start an election at the same time are extinguished as soon as possible and always before the ‘winning’ election result has been announced.



The bully algorithm

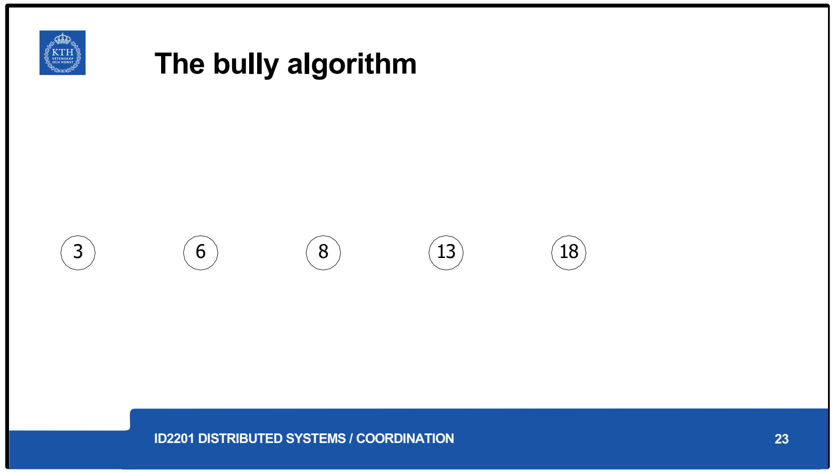
Electing a new leader when the current leader has died.

- assumes we have *reliable failure detectors*
- all nodes know the nodes with higher priority

Assume we give *higher priority* to the nodes with *lower process identifiers*.

ID2201 DISTRIBUTED SYSTEMS / COORDINATION 22

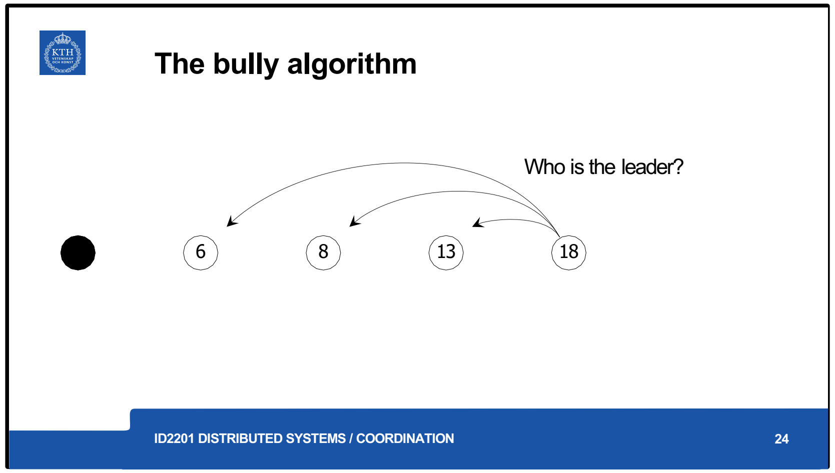
The bully algorithm allows processes to crash during an election, although it assumes that message delivery between processes is reliable. Unlike the ring-based algorithm, this algorithm assumes the system is **synchronous: it uses timeouts to detect a process failure**. Another difference is that the ring-based algorithm assumed that processes have minimal a priori knowledge of one another: each knows only how to communicate with its neighbor, and none knows the identifiers of the other processes. On the other hand, the bully algorithm assumes that each process knows which processes have higher identifiers and can communicate with all such processes.



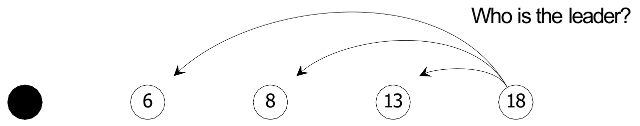
The bully algorithm

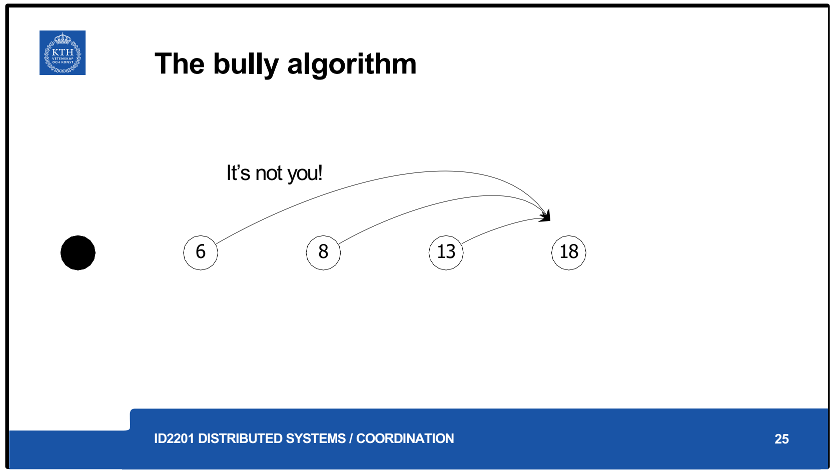
ID2201 DISTRIBUTED SYSTEMS / COORDINATION 23

This algorithm has three types of messages: an ***election*** message is sent to announce an election, an ***answer*** message is sent in response to an election message, and a ***coordinator*** message is sent to announce the identity of the elected process – the new ‘coordinator.’ A process begins an election when it notices, through timeouts, that the coordinator has failed. Several processes may discover this concurrently.

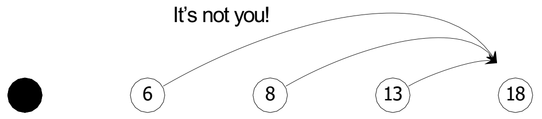


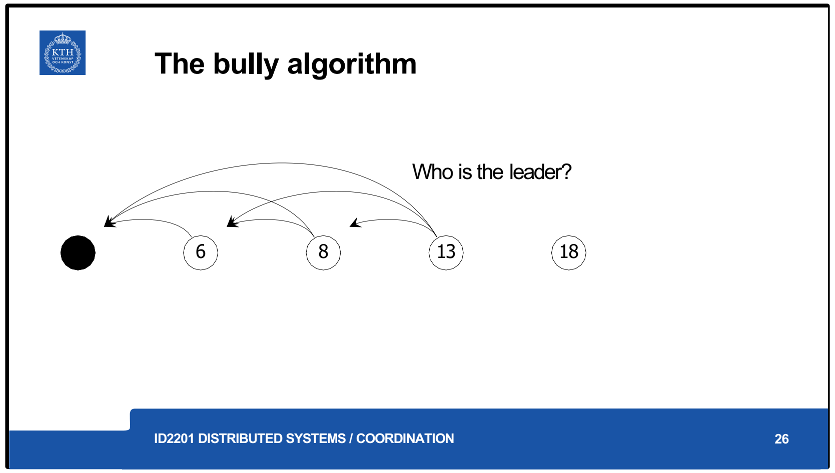
The bully algorithm



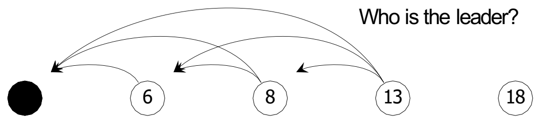


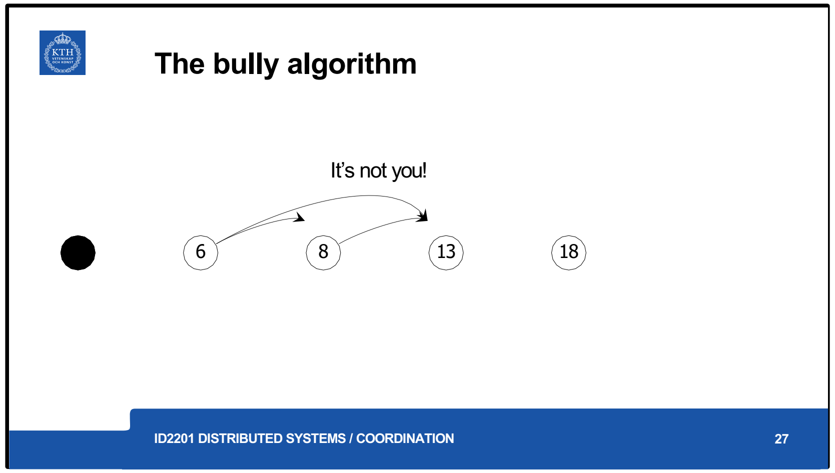
The bully algorithm



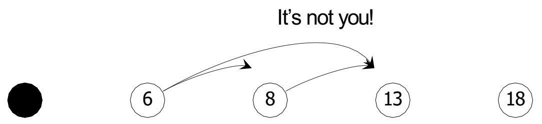


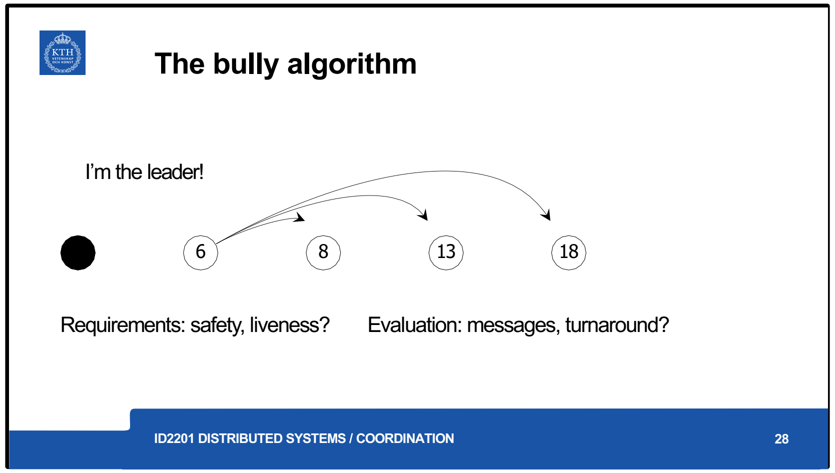
The bully algorithm





The bully algorithm



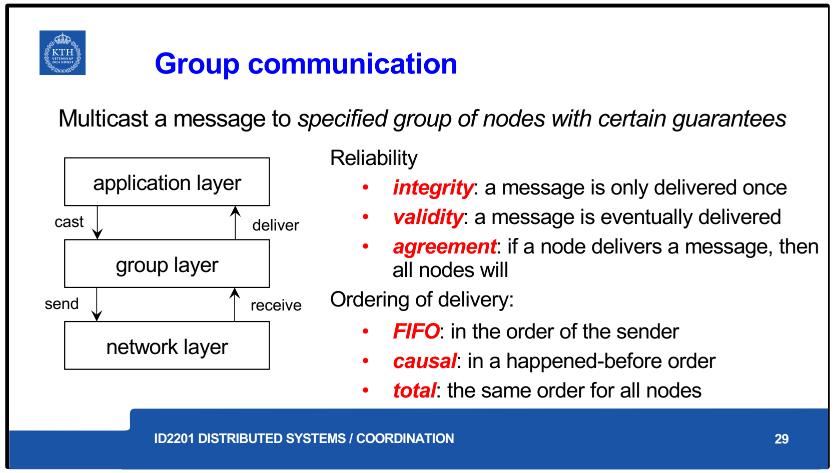


The bully algorithm

I'm the leader!
6
8
13
18

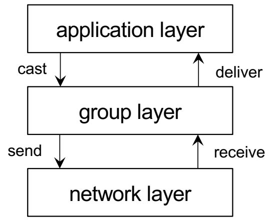
Requirements: safety, liveness? Evaluation: messages, turnaround?

But the algorithm is not guaranteed to meet the safety condition E1 if processes that have crashed are ***replaced*** by processes with the same identifiers.
Furthermore, condition E1 may be broken if the assumed timeout values are inaccurate – that is, if the processes' failure detector is unreliable.
Performance: In the base case, the next node discovers failure, immediately elects itself, and sends N – 2 coordinator messages -- O(N). The turnaround time is one message. The worst case: the last id process discovers failure – O(N^2) messages (all processes start election and receive replies)



Group communication

Multicast a message to *specified group of nodes with certain guarantees*



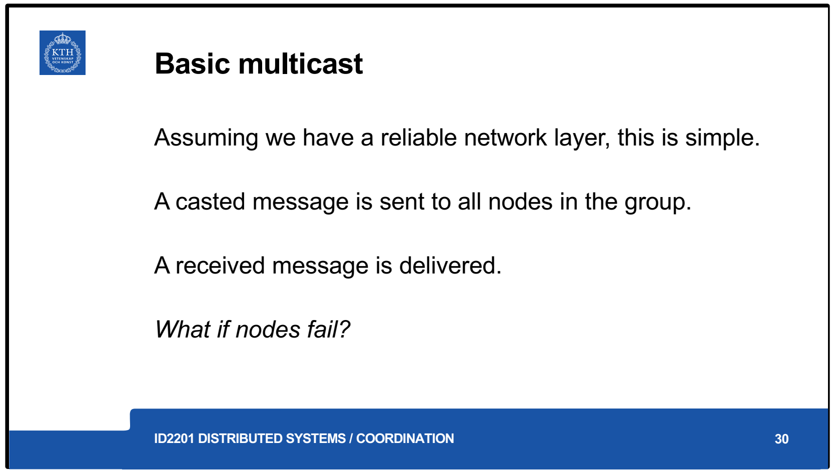
Reliability

- ***integrity***: a message is only delivered once
- ***validity***: a message is eventually delivered
- ***agreement***: if a node delivers a message, then all nodes will

Ordering of delivery:

- ***FIFO***: in the order of the sender
- ***causal***: in a happened-before order
- ***total***: the same order for all nodes

This chapter examines the key coordination and agreement problems related to group communication –how to achieve the desired reliability and ordering properties across all group members. Group communication is an example of an indirect communication technique whereby processes ***can send messages to a group***. This message is propagated to all group members with certain guarantees regarding ***reliability and ordering***.
We use the term ***deliver*** rather than receive to make clear that a multicast message is not always handed to the application layer inside the process as soon as it is received at the process's node.



Basic multicast

Assuming we have a reliable network layer, this is simple.

A casted message is sent to all nodes in the group.

A received message is delivered.

What if nodes fail?

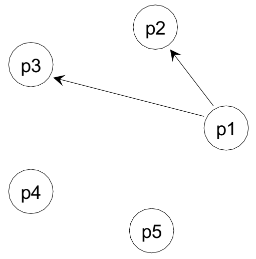
The operation ***multicast (g, m***) sends the message m to all members of the group g of processes. Correspondingly, there is an operation ***deliver (m)*** that delivers a message sent by multicast to the calling process. We use the term ***deliver*** rather than receive to make clear that a multicast message is not always handed to the application layer inside the process as soon as it is received at the process' s node.

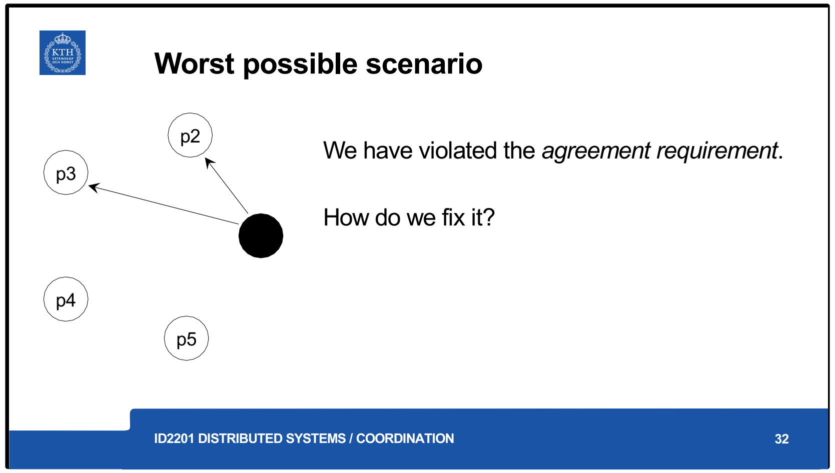
A multicast is a for loop of send.

Acknowledgments? The problem of ***ack-implosion*** if the number of processes is large.

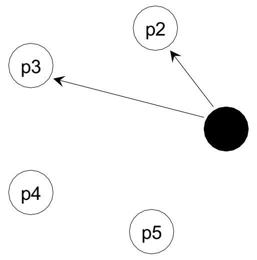
A ***basic multicast*** primitive guarantees, unlike IP multicast, that ***a correct process will eventually deliver the message as long as the multicaster does not crash***. We call the primitive ***B-multicast*** and its corresponding basic delivery primitive ***B-deliver***.

Worst possible scenario



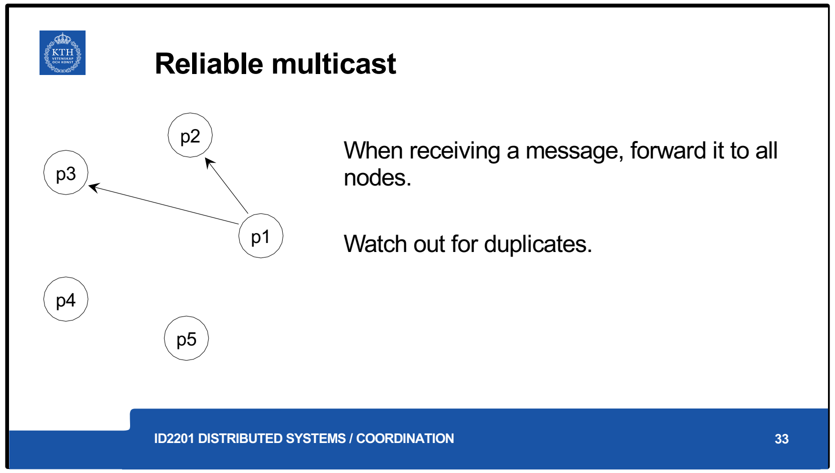


Worst possible scenario



We have violated the *agreement requirement*.

How do we fix it?



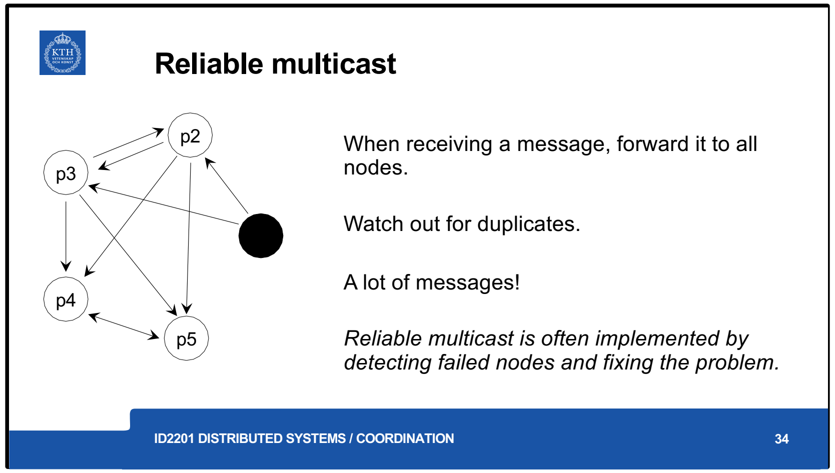
Reliable multicast

p2
p3
p1
p4
p5

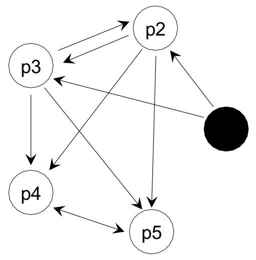
When receiving a message, forward it to all nodes.

Watch out for duplicates.

In Reliable multicast, all correct processes in the group must receive a message if any of them does.
To R-multicast a message, a process B-multicasts the message to the processes in the destination group (including itself). When the message is B-delivered, the recipient, in turn, B-multicasts the message to the group (if it is not the original sender) and then R-delivers the message. Since a message may arrive more than once, duplicates of the message are detected and not delivered:
If I have not seen the message (receive it – put in a received set; if I am not the sender– B-multicast it; R-deliver the message)
Else do nothing



Reliable multicast



When receiving a message, forward it to all nodes.

Watch out for duplicates.

A lot of messages!

Reliable multicast is often implemented by detecting failed nodes and fixing the problem.

Reliable multicast algorithm

On initialization

Received := {};

For process p to R-multicast message m to group g

B-multicast(g, m); // (p in g) is included as a destination

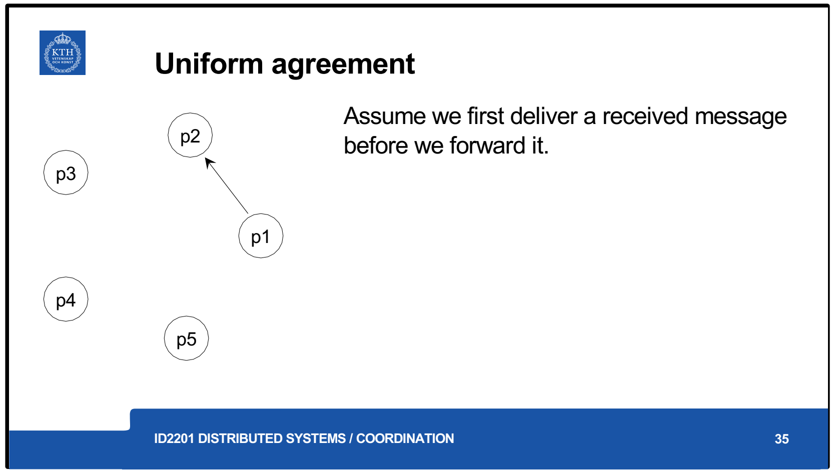
On B-deliver(m) at process q with g = group(m)

if (m does not belong to Recevied) then Received := Received and m;

if (q <> p) then B-multicast(g, m); end if

R-deliver m;

end if

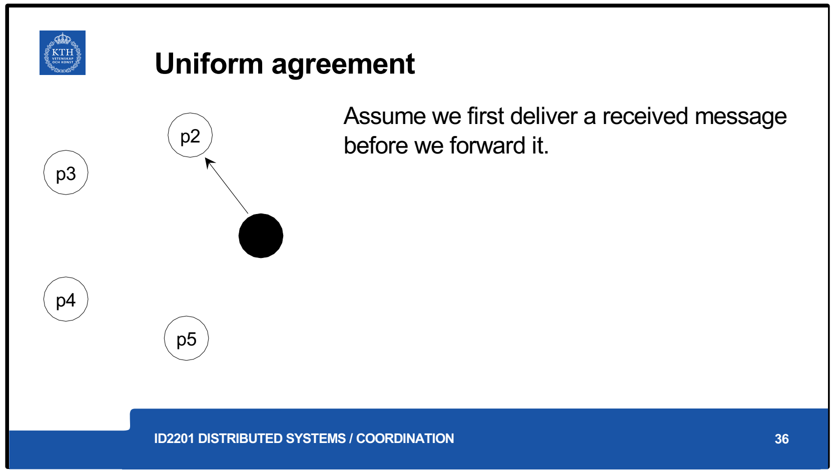


Uniform agreement

Assume we first deliver a received message before we forward it.

Correct processes – processes that never fail

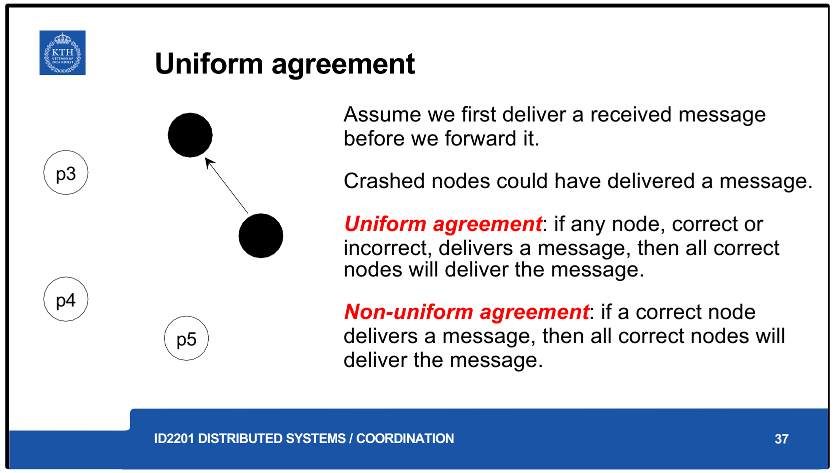
To R-multicast a message, a process B-multicasts the message to the processes in the destination group (including itself). ***When the message is B-delivered, the recipient, in turn, B-multicasts the message to the group (if it is not the original sender) and <u>then</u> R-delivers the message.*** Since a message may arrive more than once, duplicates of the message are detected and not delivered.



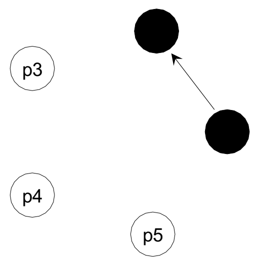
Uniform agreement

Assume we first deliver a received message before we forward it.

Correct processes – processes that never fail



Uniform agreement



Assume we first deliver a received message before we forward it.

Crashed nodes could have delivered a message.

Uniform agreement: if any node, correct or incorrect, delivers a message, then all correct nodes will deliver the message.

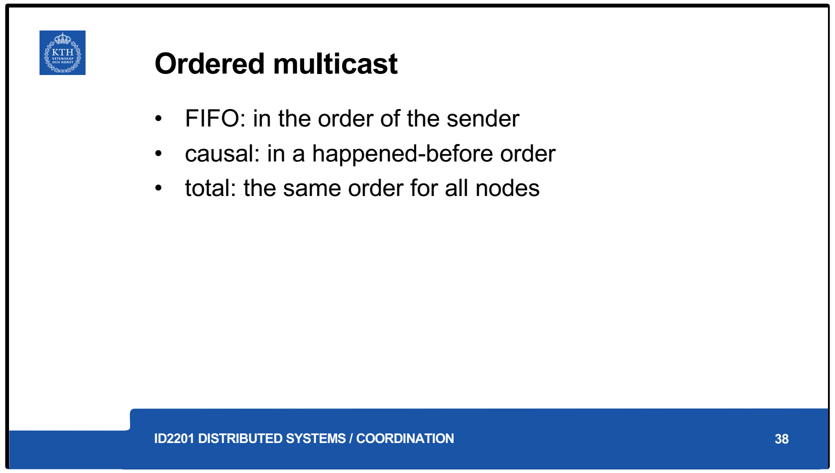
Non-uniform agreement: if a correct node delivers a message, then all correct nodes will deliver the message.

The R-multicast on top of the B-multicast satisfies the uniform agreement.

Any property that holds whether or not processes are correct is called a **uniform property**.

Correct processes – processes that never fail

The uniform agreement is useful in applications where a process may take an action that produces an observable inconsistency before it crashes. For example, an update to a bank account sent to a group of servers – the multicast should have a uniform agreement. If the multicast does not satisfy uniform agreement, then a client that accesses a server just before it crashes may observe an update that no other server will process.

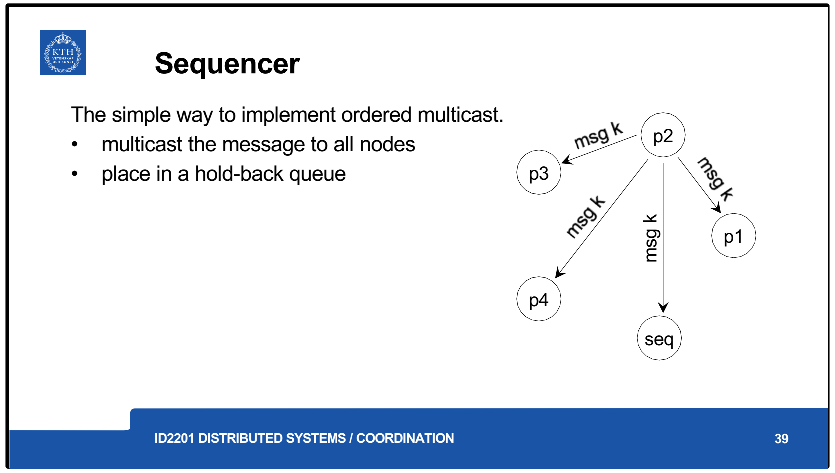


Ordered multicast

- FIFO: in the order of the sender
- causal: in a happened-before order
- total: the same order for all nodes

FIFO ordering: If a correct process issues multicast(g, m) and then multicast(g, m'), then every correct process that delivers m' will deliver m before m'.
Causal ordering: If multicast(g, m) -> multicast(g, m'), where -> is the happened-before relation induced only by messages sent between the members of g,
then any correct process that delivers m' will deliver m before m'.
Total ordering: If a correct process delivers message m before it delivers m', then any other correct process that delivers m' will deliver m before m'.
The definitions of ordered multicast do not assume or imply reliability. For example, the reader should check that, under total ordering, if the correct process p delivers message m and then delivers m', then a correct process q can deliver m without also delivering m' or any other message ordered after m.
We can also form hybrids of ordered and reliable protocols. ***A reliable, totally ordered multicast*** is often

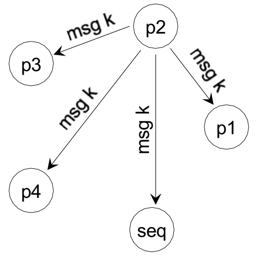
referred to in the literature as ***an atomic multicast***.



Sequencer

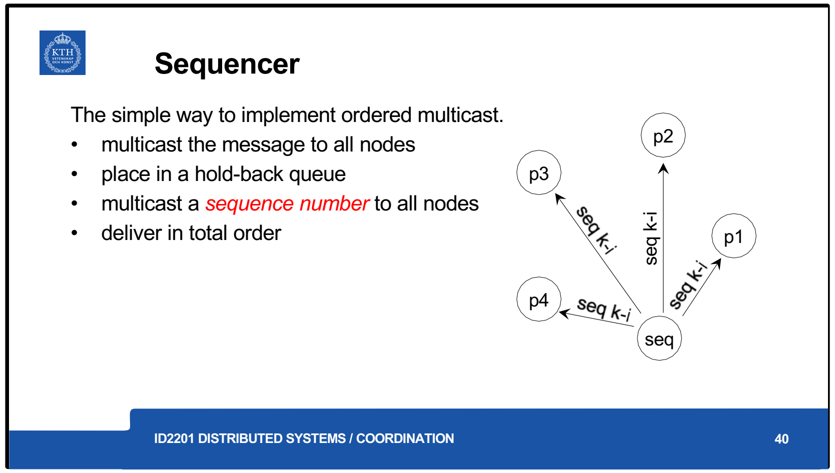
The simple way to implement ordered multicast.

- multicast the message to all nodes
- place in a hold-back queue



The basic approach to implementing total ordering is to assign ***totally ordered identifiers*** to multicast messages so that each process makes the same ordering decision based upon these identifiers.

We discuss two main methods for assigning identifiers to messages. The first of these is for a process called a ***sequencer*** to assign them.



Sequencer

The simple way to implement ordered multicast.

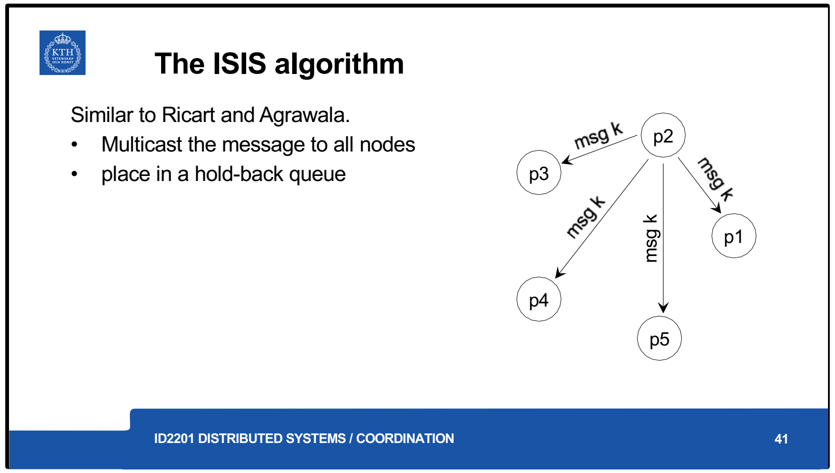
- multicast the message to all nodes
- place in a hold-back queue
- multicast a *sequence number* to all nodes
- deliver in total order

p2
p3
p1
p4
seq
seq k-i
seq k-i
seq k-i
seq k-i

The basic approach to implementing total ordering is to assign ***totally ordered identifiers*** to multicast messages so that each process makes the same ordering decision based on these identifiers.

We discuss two main methods for assigning identifiers to messages. The first of these is for a process called a ***sequencer*** to assign the.

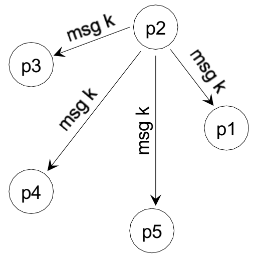
The obvious problem with a sequencer-based scheme is that the sequencer may become a bottleneck and is a critical point of failure.



The ISIS algorithm

Similar to Ricart and Agrawala.

- Multicast the message to all nodes
- place in a hold-back queue

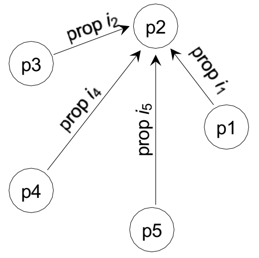


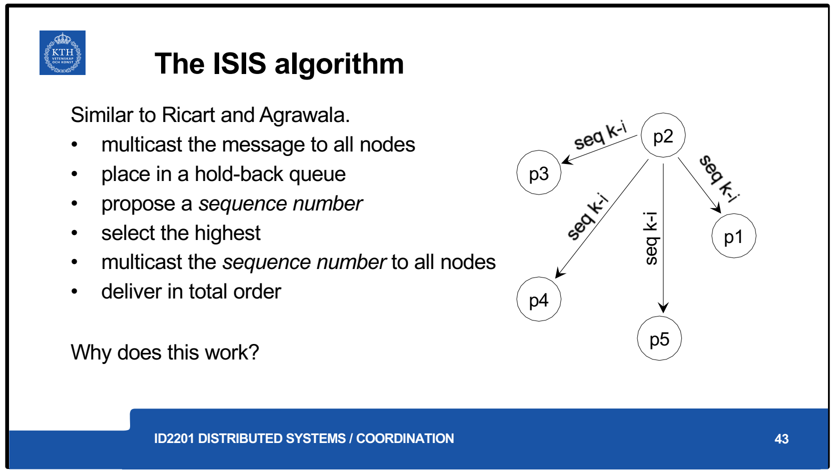
The second method we examine for achieving a totally ordered multicast is one in which the processes collectively agree on assigning **sequence numbers** to messages in a distributed fashion.

The ISIS algorithm

Similar to Ricart and Agrawala.

- multicast the message to all nodes
- place in a hold-back queue
- propose a *sequence number*
- *select the highest*



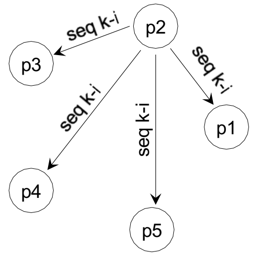


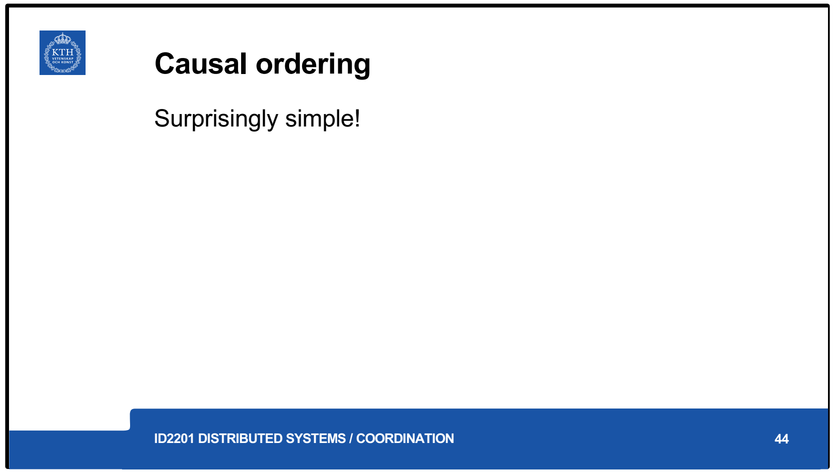
The ISIS algorithm

Similar to Ricart and Agrawala.

- multicast the message to all nodes
- place in a hold-back queue
- propose a *sequence number*
- select the highest
- multicast the *sequence number* to all nodes
- deliver in total order

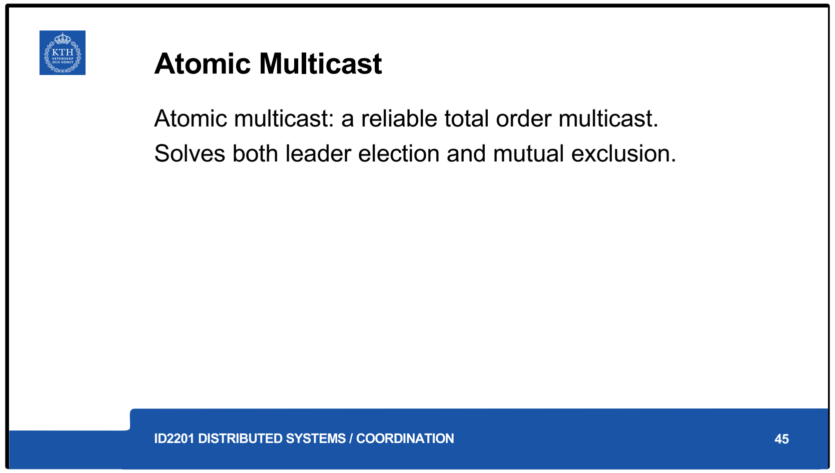
Why does this work?





Causal ordering

Surprisingly simple!

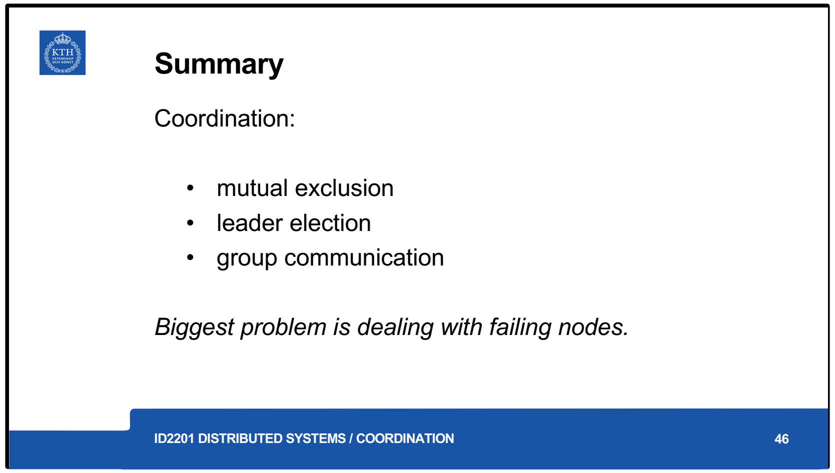


Atomic Multicast

Atomic multicast: a reliable total order multicast.
Solves both leader election and mutual exclusion.

ID2201 DISTRIBUTED SYSTEMS / COORDINATION 45

The definitions of ordered multicast do not assume or imply reliability. For example, the reader should check that, under total ordering, if the correct process p delivers message m and then delivers m’, then a correct process q can deliver m without also delivering m’ or any other message ordered after m.
We can also form hybrids of ordered and reliable protocols. ***A reliable, totally ordered multicast*** is often referred to in the literature as ***an atomic multicast***.



Summary

Coordination:

- mutual exclusion
- leader election
- group communication

Biggest problem is dealing with failing nodes.