KTH ROYAL INSTITUTE
OF TECHNOLOGY

Coordination

Vladimir Vlassov and Johan Montelius

Coordination

Why is coordination important?

Why is it a problem to implement?

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.

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.

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

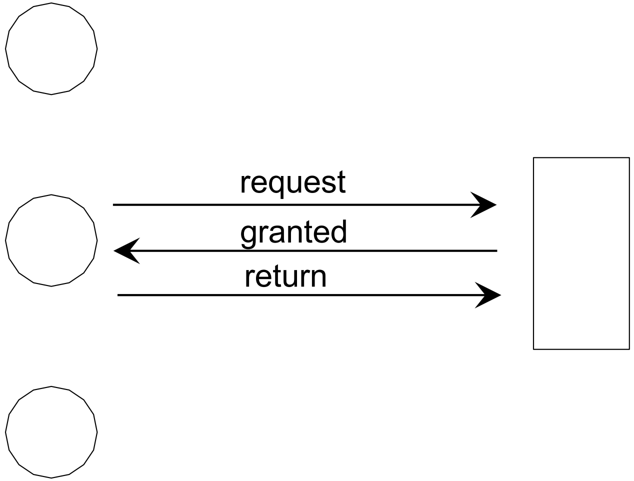
Evaluation of algorithms

- ***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?

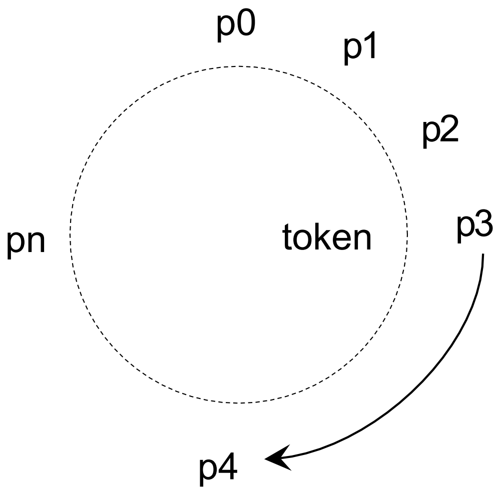
- *request* a token from the server
- *wait* for a token that grants access
- *enter* the critical section and execute in it
- *exit* the critical section and *return the token*



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

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

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?

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;

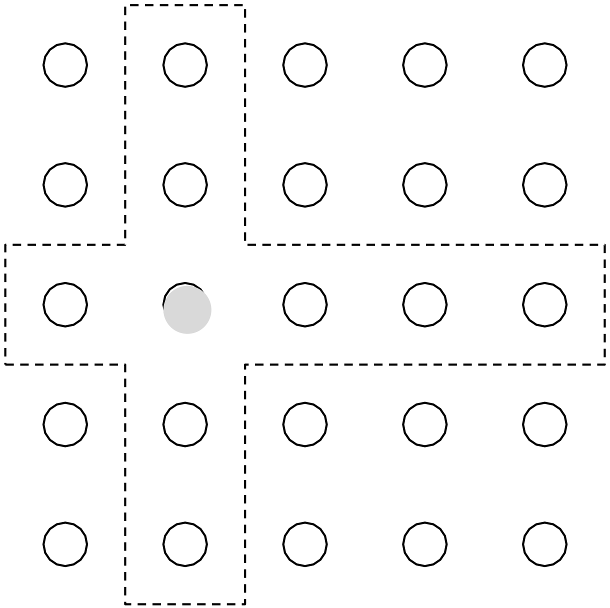
end if

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



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.

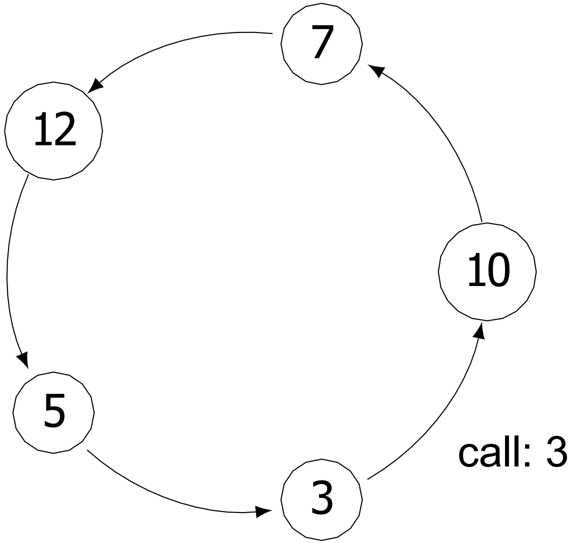
Each node can *decide* which node to trust to be the *leader*.

Requirements:

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

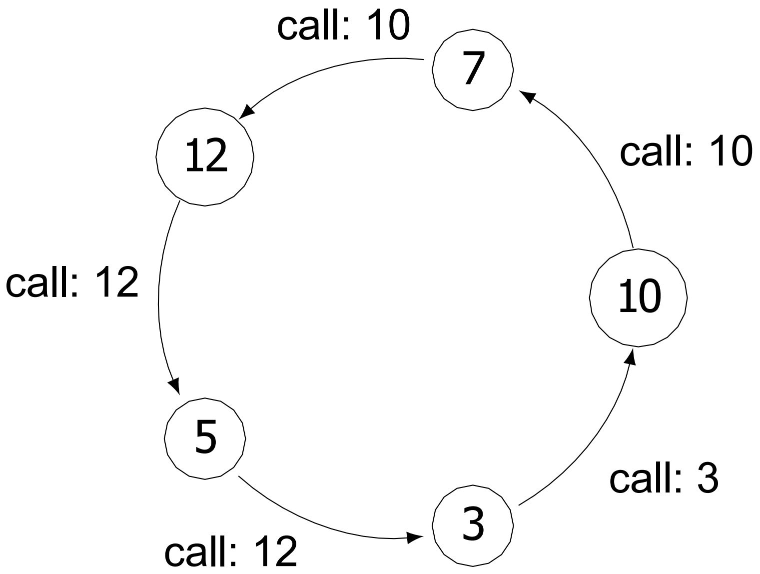
Algorithms are evaluated on the number of messages and *turnaround time*.

A ring-based approach



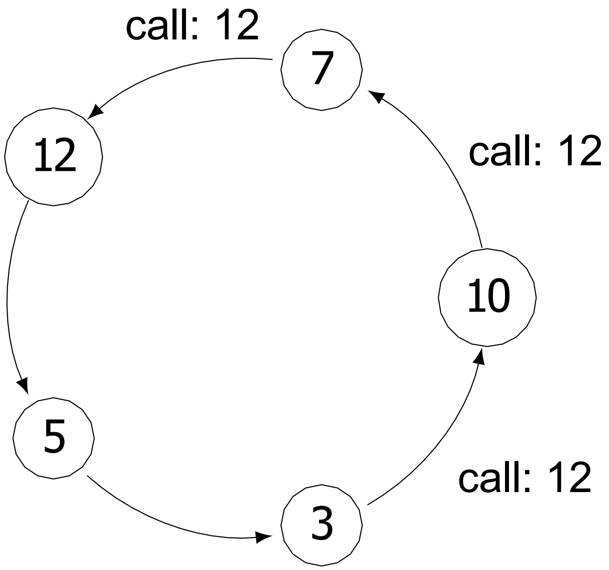
- a node starts an election

A ring-based approach



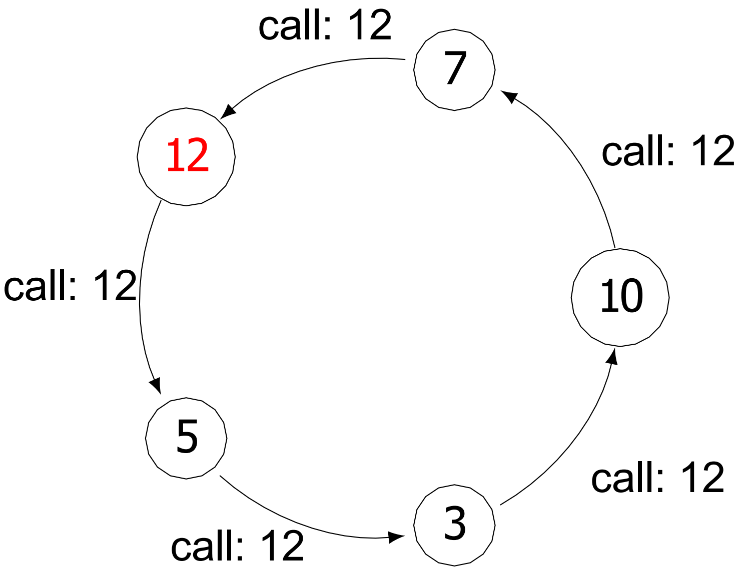
- a node starts an election
- the call is updated

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?

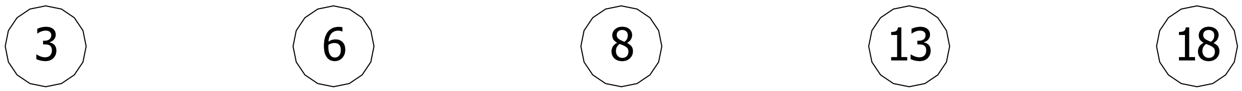
The bully algorithm

Electing a new leader when the current leader has died.

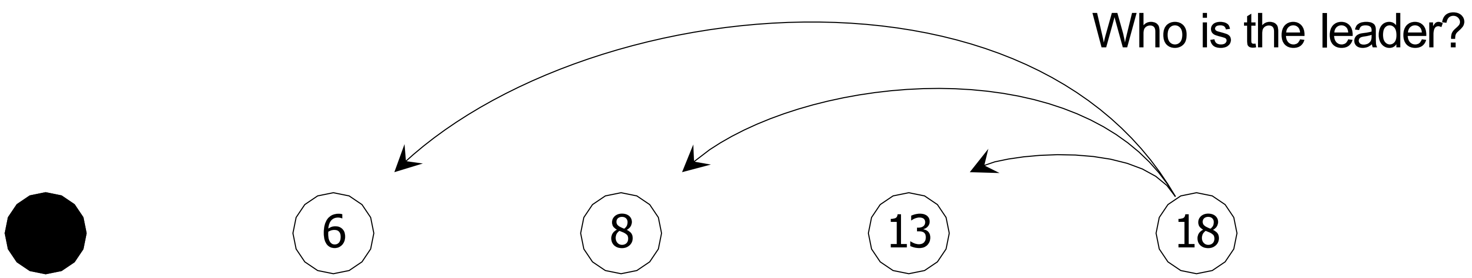
- assumes we have *reliable failure detectors*
- all nodes know the nodes with higher priority

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

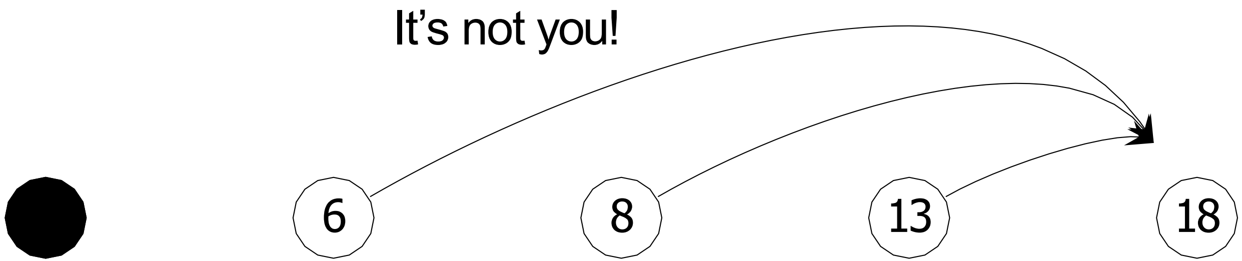
The bully algorithm



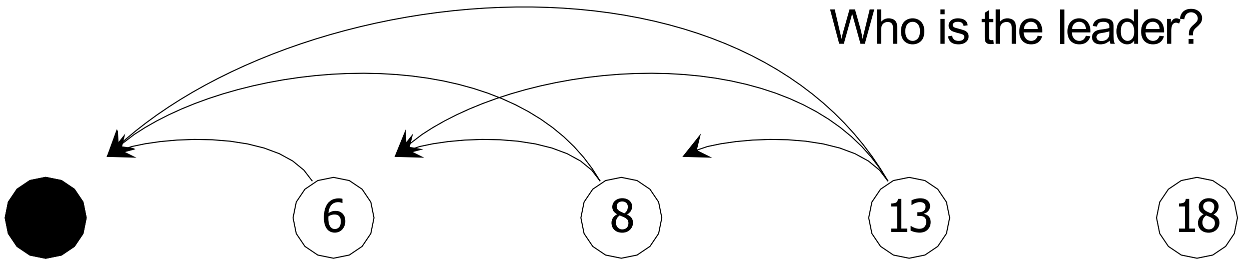
The bully algorithm



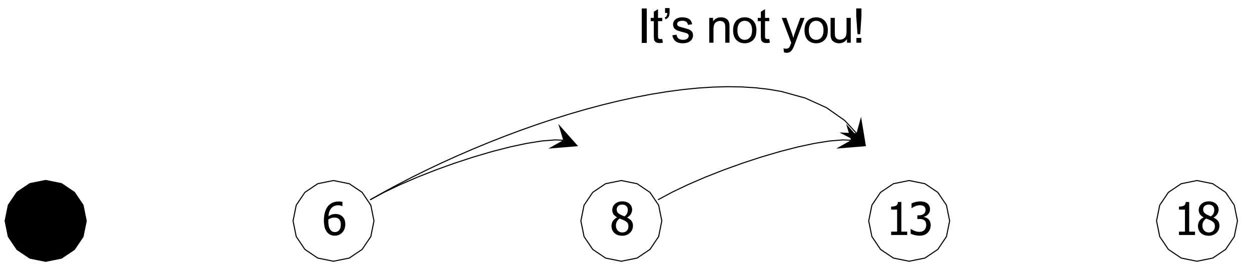
The bully algorithm



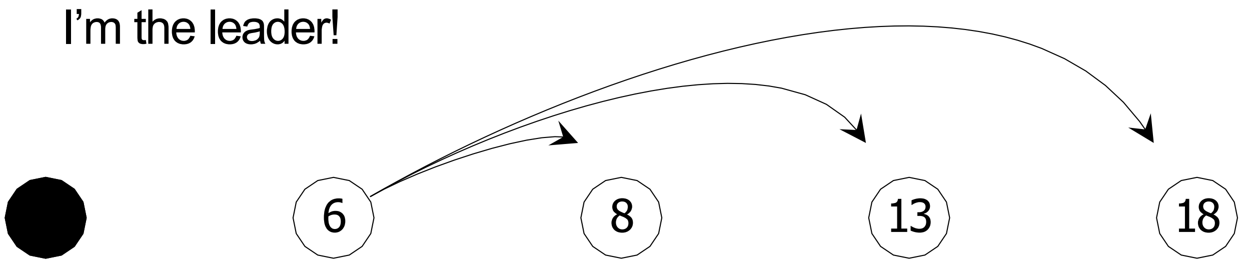
The bully algorithm



The bully algorithm



The bully algorithm

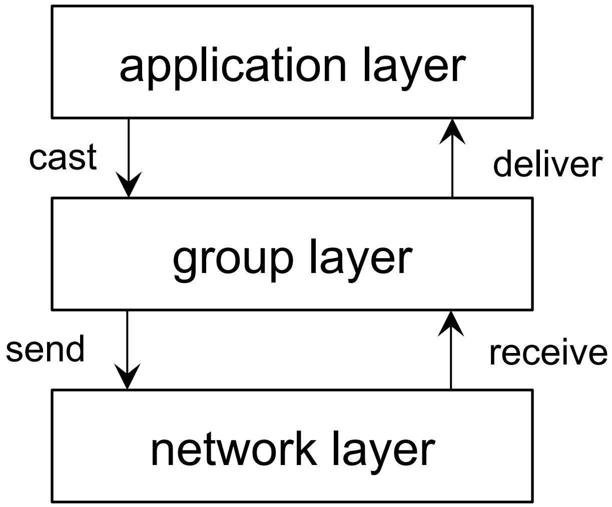


Requirements: safety, liveness?

Evaluation: messages, turnaround?

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

Basic multicast

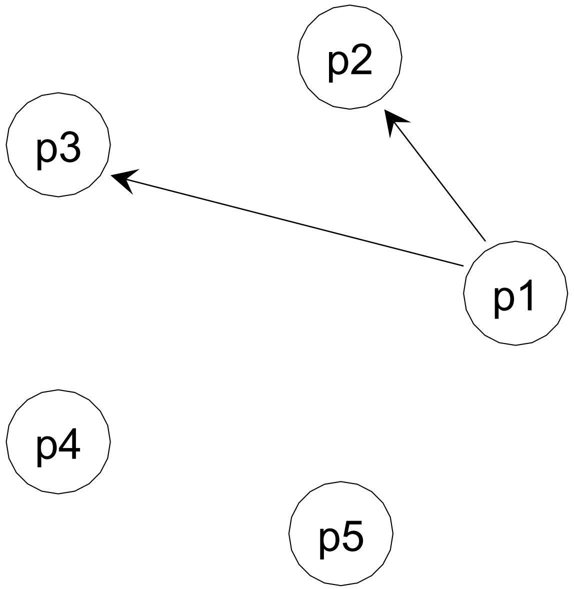
Assuming we have a reliable network layer, this is simple.

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

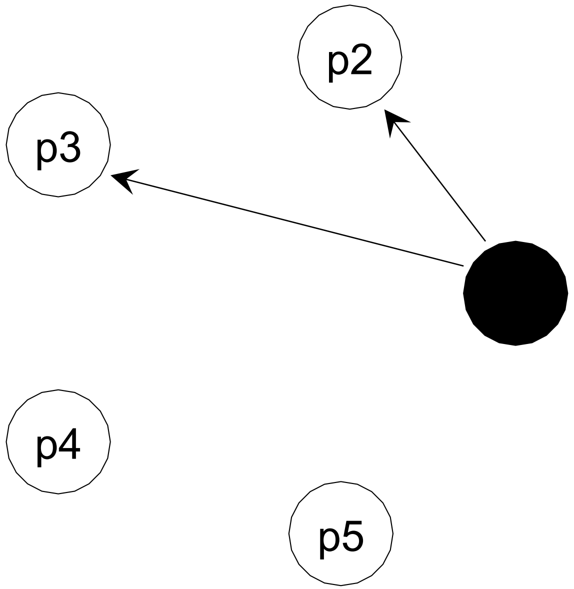
A received message is delivered.

What if nodes fail?

Worst possible scenario



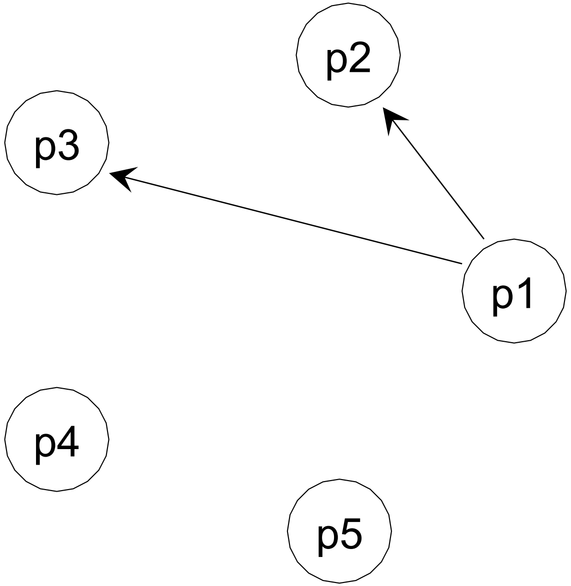
Worst possible scenario



We have violated the *agreement requirement*.

How do we fix it?

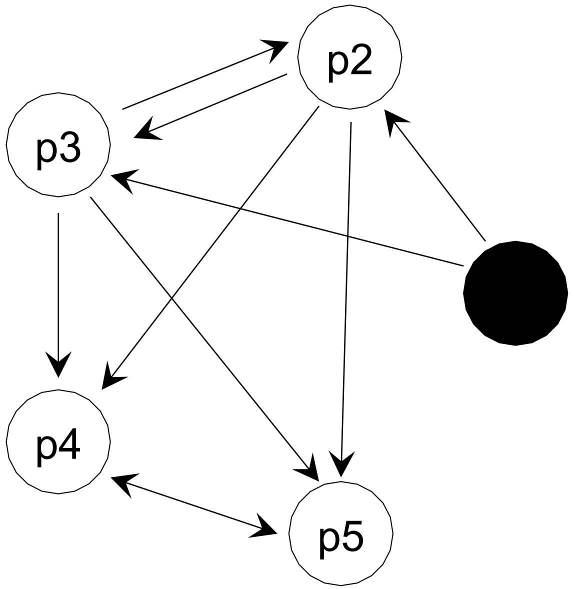
Reliable multicast



When receiving a message, forward it to all nodes.

Watch out for duplicates.

Reliable multicast



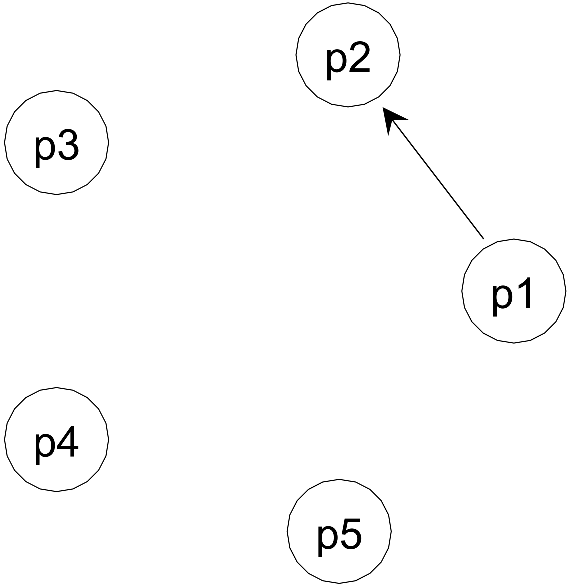
When receiving a message, forward it to all nodes.

Watch out for duplicates.

A lot of messages!

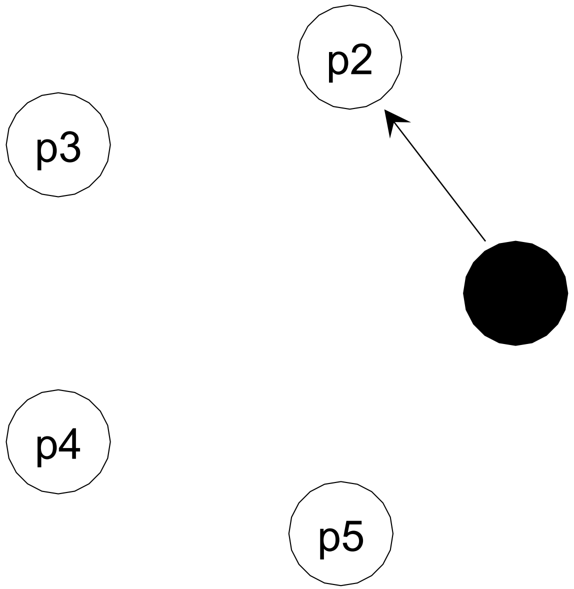
Reliable multicast is often implemented by detecting failed nodes and fixing the problem.

Uniform agreement



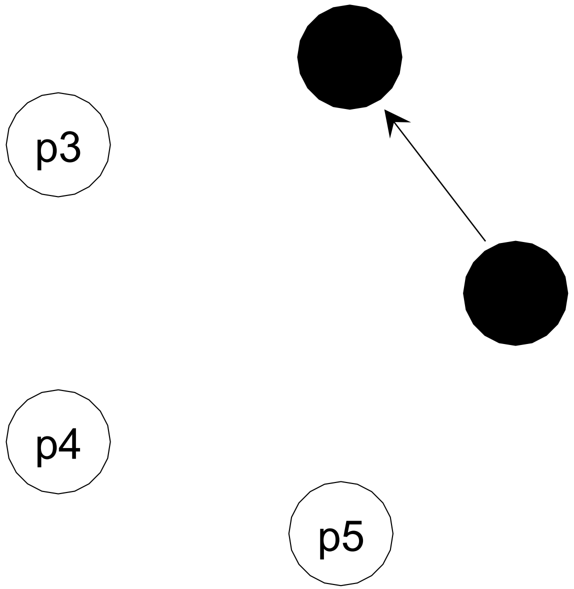
Assume we first deliver a received message before we forward it.

Uniform agreement



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

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.

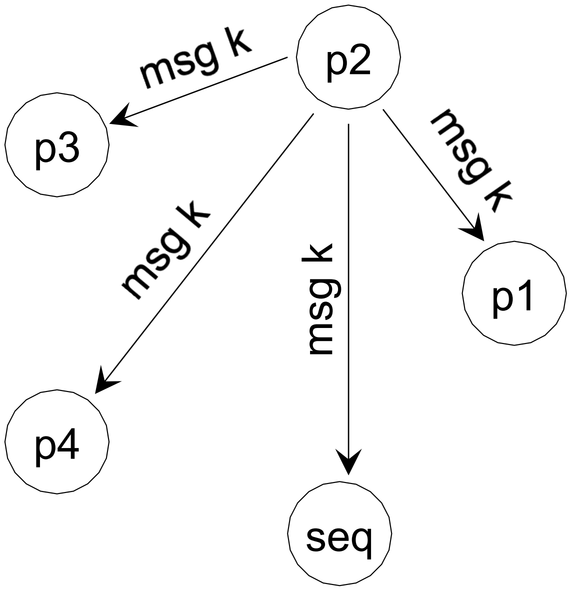
Ordered multicast

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

Sequencer

The simple way to implement ordered multicast.

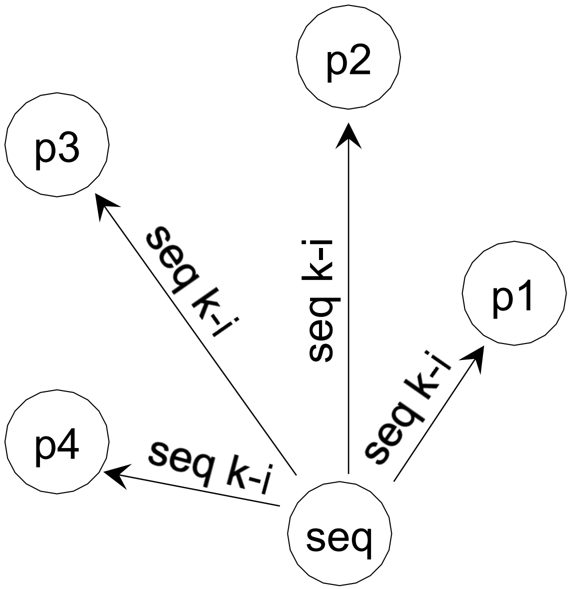
- multicast the message to all nodes
- place in a hold-back queue



Sequencer

The simple way to implement ordered multicast.

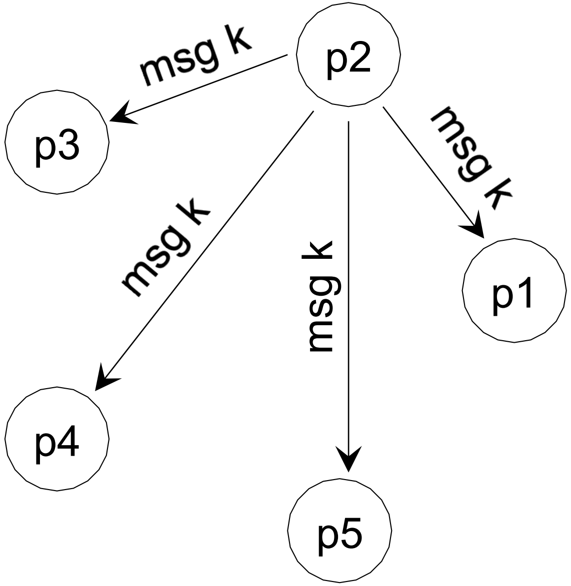
- multicast the message to all nodes
- place in a hold-back queue
- multicast a *sequence number* to all nodes
- deliver in total order



The ISIS algorithm

Similar to Ricart and Agrawala.

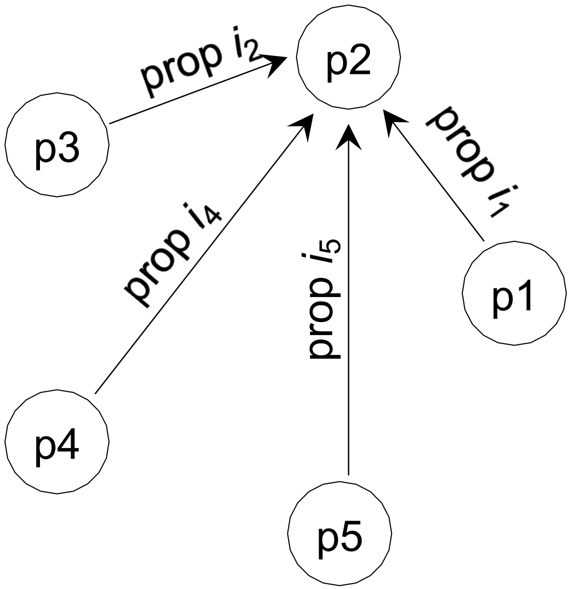
- Multicast the message to all nodes
- place in a hold-back queue



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*

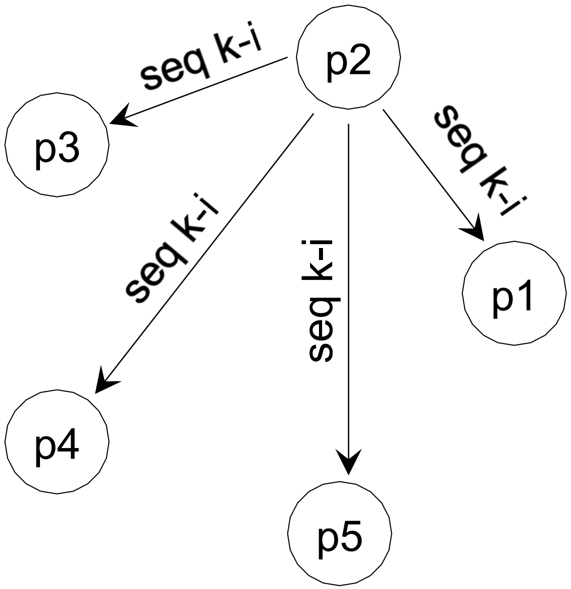


The ISIS algorithm

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.

Summary

Coordination:

- mutual exclusion
- leader election
- group communication

Biggest problem is dealing with failing nodes.