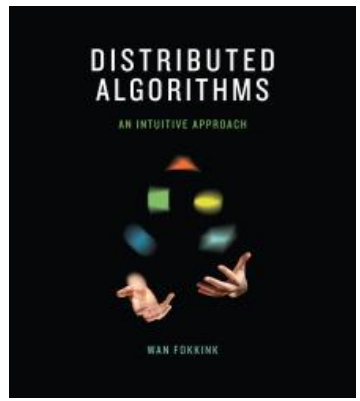
Distributed Algorithms



Wan Fokkink

Distributed Algorithms: An Intuitive Approach

MIT Press, 2013 (revised 1st edition, 2015)

Algorithms

A skilled programmer must have good insight into algorithms.

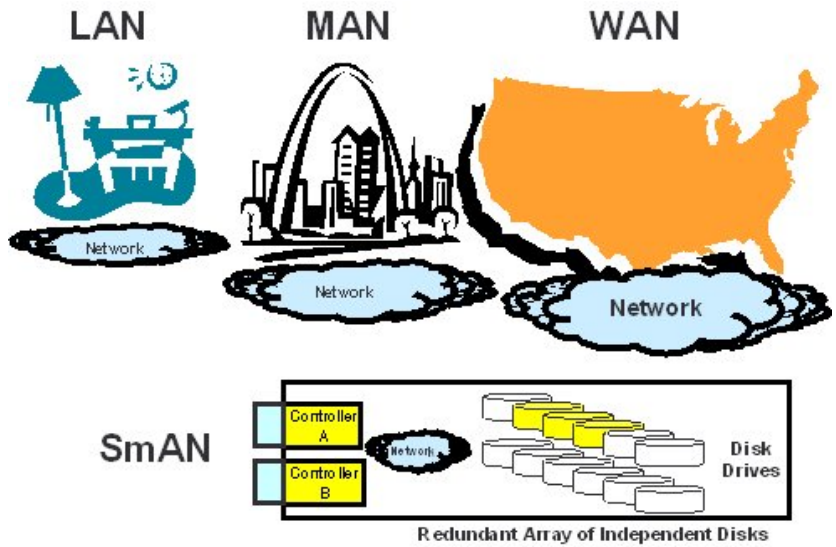
At bachelor level you were offered courses on basic algorithms: searching, sorting, pattern recognition, graph problems, ...

You learned how to detect such subproblems within your programs, and solve them effectively.

You're trained in algorithmic thought for uniprocessor programs (e.g. divide-and-conquer, greedy, memoization).

Distributed systems

A distributed system is an interconnected collection of autonomous processes.



Motivation:

- information exchange
- resource sharing
- parallelization to increase performance
- replication to increase reliability
- multicore programming

Distributed versus uniprocessor

Distributed systems differ from uniprocessor systems in three aspects.

- *Lack of knowledge on the global state*: A process has no up-to-date knowledge on the local states of other processes.

  Example: termination and deadlock detection become an issue.

- *Lack of a global time frame*: No total order on events by their temporal occurrence.

  Example: mutual exclusion becomes an issue.

- *Nondeterminism*: Execution by processes is nondeterministic, so running a system twice can give different results.

  Example: race conditions.

Aim of this course

This course offers a bird's-eye view on a wide range of algorithms for basic and important challenges in distributed systems.

It aims to provide you with an algorithmic frame of mind for solving fundamental problems in distributed computing.

- Handwaving correctness arguments.
- Back-of-the-envelope complexity calculations.
- Carefully developed exercises to acquaint you with intricacies of distributed algorithms.

Message passing

The two main paradigms to capture communication in a distributed system are message passing and shared memory.

We'll only consider message passing.

(The course *Concurrency & Multithreading* is dedicated to shared memory.)

Asynchronous communication means that sending and receiving of a message are *independent events*.

In case of synchronous communication, sending and receiving of a message are coordinated to form a *single event*; a message is only allowed to be sent if its destination is ready to receive it.

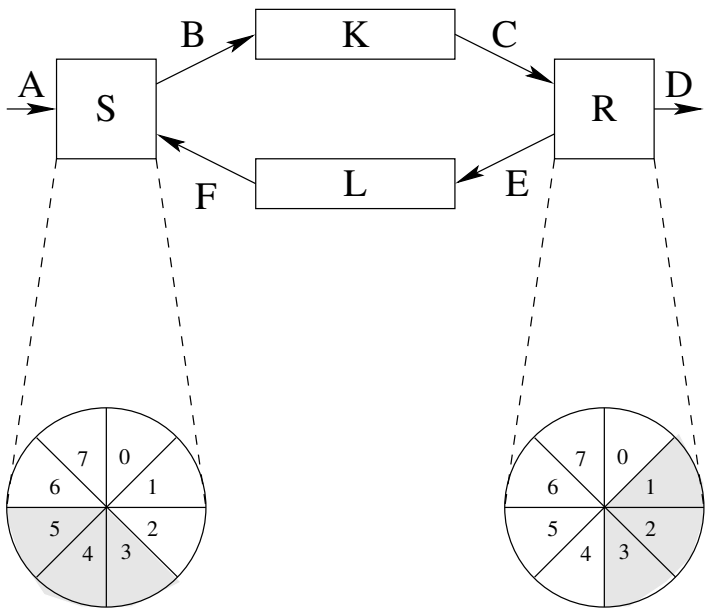
We'll mainly consider asynchronous communication.

Communication protocols

In a computer network, messages are transported through a medium, which may lose, duplicate or garble these messages.

A communication protocol detects and corrects such flaws during message passing.

Example: Sliding window protocols.



Assumptions

Unless stated otherwise, we assume:

- a strongly connected network
- each process knows only its neighbors
- message passing communication
- asynchronous communication
- channels are non-FIFO
- the delay of messages in channels is arbitrary but finite
- channels don't lose, duplicate or garble messages
- processes don't crash
- processes have unique id's

Directed versus undirected channels

Channels can be *directed* or *undirected*.

Question: What is more general, an algorithm for a directed or for an undirected network ?

Remarks:

- Algorithms for undirected channels often include ack's.
- Acyclic networks must always be undirected (else the network wouldn't be strongly connected).

Complexity measures

Resource consumption of an execution of a distributed algorithm can be considered in several ways.

Message complexity: Total number of messages exchanged.

Bit complexity: Total number of bits exchanged.
(Only interesting when messages can be very long.)

Time complexity: Amount of time consumed.
(We assume: (1) event processing takes no time, and (2) a message is received at most one time unit after it is sent.)

Space complexity: Amount of memory needed for the processes.

Different executions require different consumption of resources.

We consider worst- and average-case complexity (the latter with a probability distribution over all executions).

Big O notation

Complexity measures state how resource consumption (messages, time, space) grows in relation to input size.

For example, if an algorithm has a worst-case message complexity of $O(n^2)$, then for an input of size n, the algorithm in the worst case takes *in the order of* n^2 messages.

Let $f, g : \mathbb{N} \to \mathbb{R}_{>0}$.

$f = O(g)$ if, for some $C > 0$, $f(n) \leq C \cdot g(n)$ for all $n \in \mathbb{N}$.

$f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$.

Formal framework

Now follows a formal framework for describing distributed systems, mainly to fix terminology.

In this course, correctness proofs and complexity estimations of distributed algorithms are presented in an *informal* fashion.

(The course *Protocol Validation* treats algorithms and tools to prove correctness of distributed algorithms and network protocols.)

Transition systems

The (global) state of a distributed system is called a configuration.

The configuration evolves in discrete steps, called transitions.

A transition system consists of:

- a set $\mathcal{C}$ of configurations;
- a binary transition relation $\rightarrow$ on $\mathcal{C}$; and
- a set $\mathcal{I} \subseteq \mathcal{C}$ of initial configurations.

$\gamma \in \mathcal{C}$ is terminal if $\gamma \rightarrow \delta$ for no $\delta \in \mathcal{C}$.

Executions

An execution is a sequence $\gamma_0 \gamma_1 \gamma_2 \cdots$ of configurations that either is infinite or ends in a terminal configuration, such that:

- $\gamma_0 \in \mathcal{I}$, and
- $\gamma_i \to \gamma_{i+1}$ for all $i \geq 0$

  (excluding, for finite executions, the terminal γ_i at the end).

A configuration δ is reachable if there is a $\gamma_0 \in \mathcal{I}$ and a sequence $\gamma_0 \gamma_1 \gamma_2 \cdots \gamma_k = \delta$ with $\gamma_i \to \gamma_{i+1}$ for all $0 \leq i < k$.

States and events

A configuration of a distributed system is composed from the states at its processes, and the messages in its channels.

A transition is associated to an event (or, in case of synchronous communication, two events) at one (or two) of its processes.

A process can perform internal, send and receive events.

A process is an initiator if its first event is an internal or send event.

An algorithm is centralized if there is exactly one initiator.

A decentralized algorithm can have multiple initiators.

Assertions

An assertion is a predicate on the configurations of an algorithm.

An assertion is a safety property if it is true in each configuration of each execution of the algorithm.

"*something bad will never happen*"

An assertion is a liveness property if it is true in some configuration of each execution of the algorithm.

"*something good will eventually happen*"

Invariants

Assertion P on configurations is an invariant if:

- $P(\gamma)$ for all $\gamma \in \mathcal{I}$, and
- if $\gamma \to \delta$ and $P(\gamma)$, then $P(\delta)$.

Each invariant is a safety property.

Question: Give a transition system S and an assertion P such that P is a safety property but not an invariant for S.

Causal order

In each configuration of an asynchronous system, applicable events at different processes are independent.

The causal order $\prec$ on occurrences of events in an execution is the smallest *transitive* relation such that:

- if a and b are events at the same process and a occurs before b, then $a \prec b$; and
- if a is a send and b the corresponding receive event, then $a \prec b$.

This relation is *irreflexive*.

$a \preceq b$ denotes $a \prec b \vee a = b$.

Computations

If neither $a \preceq b$ nor $b \preceq a$, then a and b are called concurrent.

A permutation of concurrent events in an execution doesn't affect the result of the execution.

These permutations together form a computation.

All executions of a computation start in the same initial configuration, and if they are finite, they all end in the same terminal configuration.

Question

Consider the finite execution abc.

Let $a \prec b$ be the only causal relationship.

Which executions are in the same computation ?

Lamport's clock

A logical clock C maps occurrences of events in a *computation* to a *partially ordered* set such that $a \prec b \Rightarrow C(a) < C(b)$.

Lamport's clock LC assigns to each event a the length k of a longest causality chain $a_1 \prec \cdots \prec a_k = a$.

LC can be computed at run-time:

Let a be an event, and k the clock value of the previous event at the same process ($k = 0$ if there is no such previous event).

* If a is an internal or send event, then $LC(a) = k + 1$.
* If a is a receive event, and b the send event corresponding to a, then $LC(a) = \max\{k, LC(b)\} + 1$.

Question

Consider the following sequences of events at processes p_0, p_1, p_2:

$$\begin{array}{lllll} p_0: & a & s_1 & r_3 & b \\ p_1: & c & r_2 & s_3 & \\ p_2: & r_1 & d & s_2 & e \end{array}$$

s_i and r_i are corresponding send and receive events, for $i = 1, 2, 3$.

Provide all events with Lamport's clock values.

Answer: 1 2 8 9

1 6 7

3 4 5 6

Vector clock

Given processes $p_0, \ldots, p_{N-1}$.

We define a *partial order* on $\mathbb{N}^N$ by:

$$(k_0, \ldots, k_{N-1}) \leq (\ell_0, \ldots, \ell_{N-1}) \Leftrightarrow k_i \leq \ell_i \text{ for all } i = 0, \ldots, N-1.$$

Vector clock VC maps each event in a computation to a unique value in $\mathbb{N}^N$ such that $a \prec b \Leftrightarrow VC(a) < VC(b)$.

$VC(a) = (k_0, \ldots, k_{N-1})$ where each k_i is the length of a longest causality chain $a_1^i \prec \cdots \prec a_{k_i}^i$ of events at process p_i with $a_{k_i}^i \preceq a$.

VC can also be computed at run-time.

Question

Consider the same sequences of events at processes p_0, p_1, p_2:

$$
\begin{array}{lllll}
p_0: & a & s_1 & r_3 & b \\
p_1: & c & r_2 & s_3 & \\
p_2: & r_1 & d & s_2 & e
\end{array}
$$

Provide all events with vector clock values.

Answer:

(1 0 0)	(2 0 0)	(3 3 3)	(4 3 3)
(0 1 0)	(2 2 3)	(2 3 3)	
(2 0 1)	(2 0 2)	(2 0 3)	(2 0 4)

Vector clock - Correctness

Let $a \prec b$.

Any causality chain for a is also one for b. So $VC(a) \leq VC(b)$.

At the process where b occurs, there is a longer causality chain for b than for a. So $VC(a) < VC(b)$.

Let $VC(a) < VC(b)$.

Consider the longest causality chain $a_1^i \prec \cdots \prec a_k^i = a$ of events at the process p_i where a occurs.

$VC(a) < VC(b)$ implies that the i-th coefficient of $VC(b)$ is $\geq k$.

So $a \preceq b$.

Since a and b are distinct, $a \prec b$.

Snapshots

A snapshot of an execution of a distributed algorithm should return a configuration of an execution *in the same computation*.

Snapshots can be used for:

- Restarting after a failure.
- Off-line determination of stable properties, which remain true as soon as they have become true.

  Examples: deadlock, garbage.
- Debugging.

Challenge: Take a snapshot without freezing the execution.

Snapshots

We distinguish basic messages of the underlying distributed algorithm and control messages of the snapshot algorithm.

A snapshot of a (basic) execution consists of:

- a local snapshot of the (basic) state of each process, and
- the channel state of (basic) messages in transit for each channel.

A snapshot is meaningful if it is a configuration of an execution in the same computation as the actual execution.

Snapshots

We need to avoid the following situations.

1. Process p takes a local snapshot, and then sends a message m to process q, where:
   - q takes a local snapshot after the receipt of m,
   - or m is included in the channel state of pq.

2. p sends m to q, and then takes a local snapshot, where:
   - q takes a local snapshot before the receipt of m,
   - and m is not included in the channel state of pq.

Chandy-Lamport algorithm

Consider a directed network with *FIFO* channels.

Initiators take a local snapshot of their state, and send a control message $\langle$**marker**$\rangle$ to their neighbors.

When a process that hasn't yet taken a snapshot receives $\langle$**marker**$\rangle$, it

- takes a local snapshot of its state, and
- sends $\langle$**marker**$\rangle$ to all its neighbors.

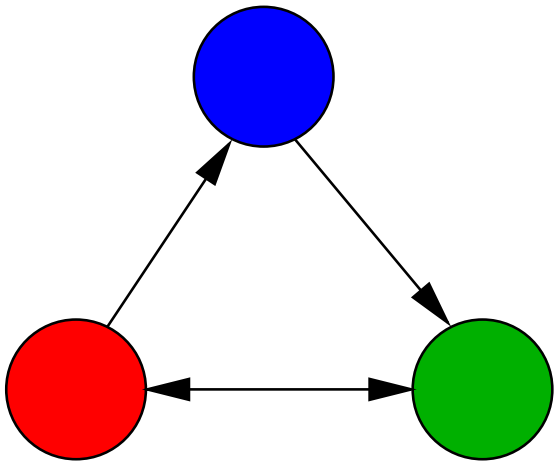
Process q computes as channel state of pq the messages it receives via pq after taking its local snapshot and before receiving $\langle$**marker**$\rangle$ from p.

If channels are FIFO, this produces a meaningful snapshot.

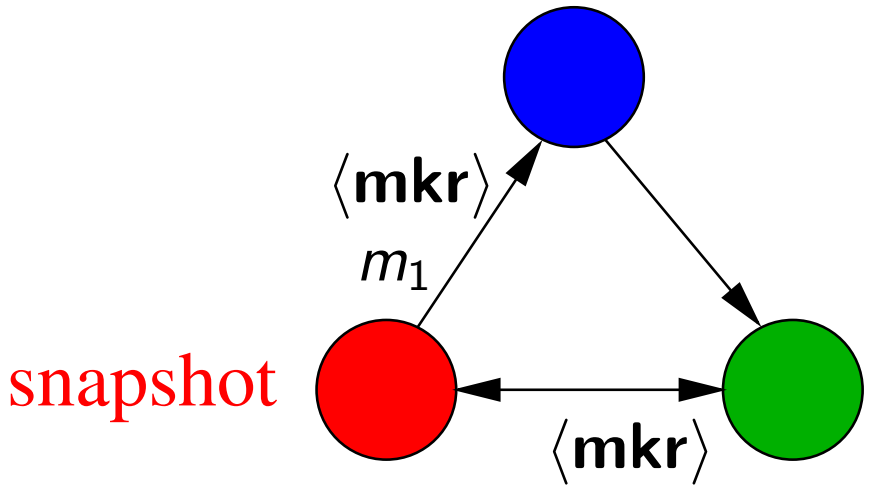
Message complexity: $\Theta(E)$

Worst-case time complexity: $O(D)$

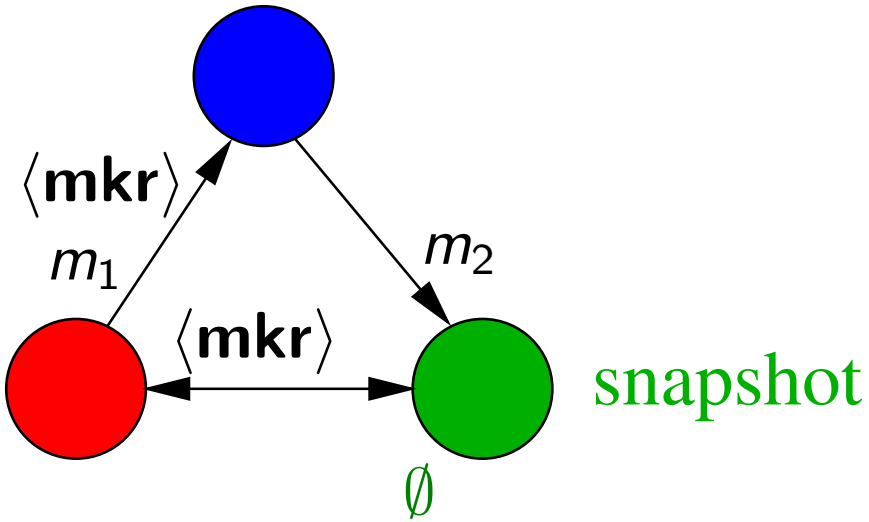
Chandy-Lamport algorithm - Example



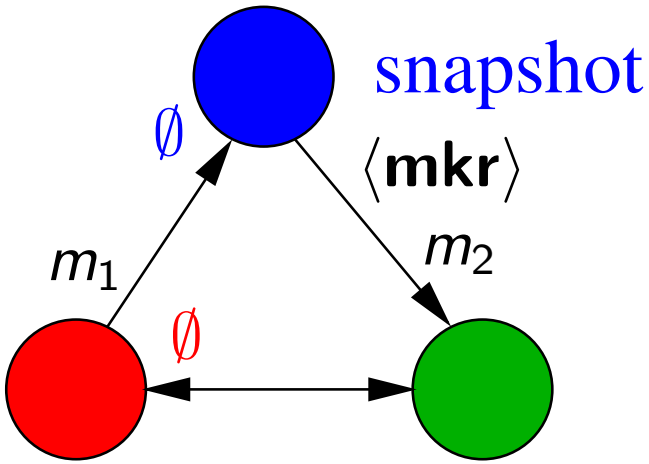
Chandy-Lamport algorithm - Example



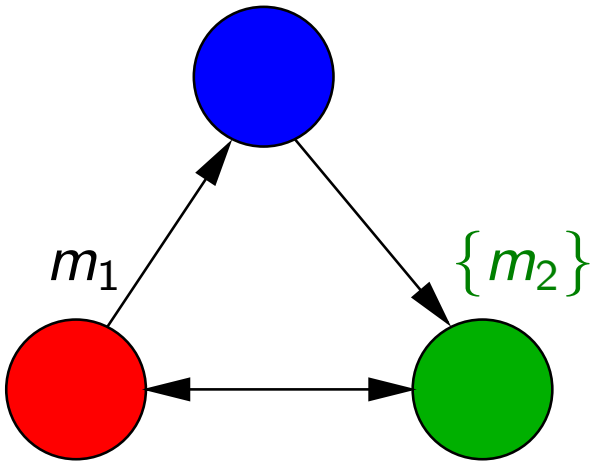
Chandy-Lamport algorithm - Example



Chandy-Lamport algorithm - Example



Chandy-Lamport algorithm - Example



The snapshot (processes red/blue/green, channels $\emptyset, \emptyset, \emptyset, \{m_2\}$) isn't a configuration in the actual execution.

The send of m_1 isn't causally before the send of m_2.

So the snapshot is a configuration of an execution that is in the same computation as the actual execution.

Chandy-Lamport algorithm - Correctness

Claim: If a post-snapshot event e is causally before an event f, then f is also post-snapshot.

This implies that the snapshot is a configuration of an execution that is in the same *computation* as the actual execution.

Proof: The case that e and f occur at the same process is trivial.

Let e be a send and f the corresponding receive event.

Let e occur at p and f at q.

e is post-snapshot at p, so p sent $\langle \textbf{marker} \rangle$ to q before e.

Channels are FIFO, so q receives this $\langle \textbf{marker} \rangle$ before f.

Hence f post-snapshot at q.

Lai-Yang algorithm

Suppose channels are *non-FIFO*. We use piggybacking.

Initiators take a local snapshot of their state.

When a process has taken its local snapshot, it appends *true* to each outgoing basic message.

When a process that hasn't yet taken a snapshot receives a message with *true* or a *control message* (see next slide) for the first time, it takes a local snapshot of its state *before reception of this message*.

Process q computes as channel state of pq the basic messages without the tag *true* that it receives via pq after its local snapshot.

Lai-Yang algorithm - Control messages

Question: How does q know when it can determine the channel state of pq ?

p sends a control message to q, informing q how many basic messages without the tag *true* p sent into pq.

These control messages also ensure that all processes eventually take a local snapshot.

Lai-Yang algorithm - Multiple snapshots

Question: How can multiple subsequent snapshots be supported ?

Answer: Each snapshot is provided with a sequence number.

Basic message carry the sequence number of the last snapshot at the sender (instead of *true* or *false*).

Control messages carry the sequence number of their snapshot.

What we need from last lecture

fully asynchronous message passing framework

channels are non-FIFO, and can be directed or undirected

configurations and transitions at the global level

states and events (*internal/send/receive*) at local level

(non)initiator

(de)centralized algorithm;

causal order $\prec$ on events in an execution

computation of executions, by reordering *concurrent* events

snapshot algorithm to compute a configuration of a *computation*

basic/control algorithm

Wave algorithms

A decide event is a special internal event.

In a wave algorithm, each computation (also called wave) satisfies the following properties:

- *termination*: it is finite;
- *decision*: it contains one or more decide events; and
- *dependence*: for each decide event e and process p, $f \prec e$ for an event f at p.

Wave algorithms - Example

In the *ring algorithm*, the initiator sends a token, which is passed on by all other processes.

The initiator decides after the token has returned.

Question: For each process, which event is causally before the decide event ?

The ring algorithm is an example of a traversal algorithm.

Traversal algorithms

A traversal algorithm is a *centralized* wave algorithm;
i.e., there is one initiator, which sends around a token.

- In each computation, the token first visits all processes.
- Finally, the token returns to the initiator, who performs a decide event.

Traversal algorithms build a spanning tree:

- the initiator is the root; and
- each noninitiator has as parent the neighbor from which it received the token first.

Tarry's algorithm (from 1895)

Consider an undirected network.

R1 A process never forwards the token through the same channel twice.

R2 A process only forwards the token to its parent when there is no other option.

The token travels through each channel both ways, and finally ends up at the initiator.

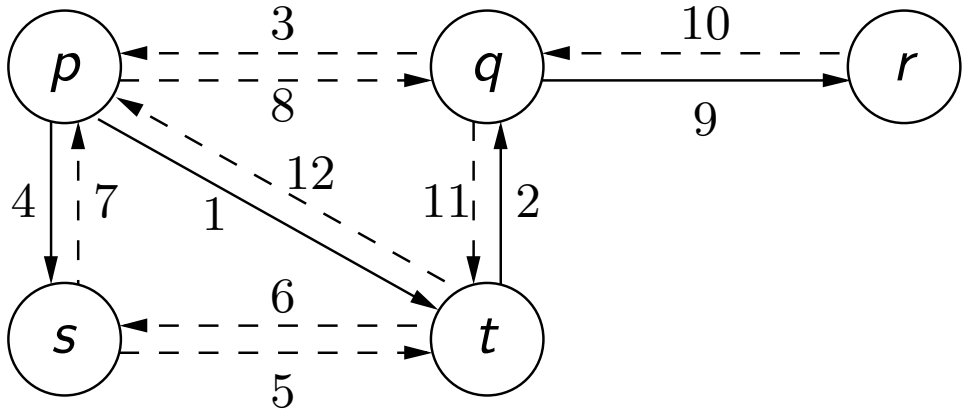
Message complexity: $2E$ messages

Time complexity: $\leq 2E$ time units

Gaston Tarry

Tarry's algorithm - Example

p is the initiator.



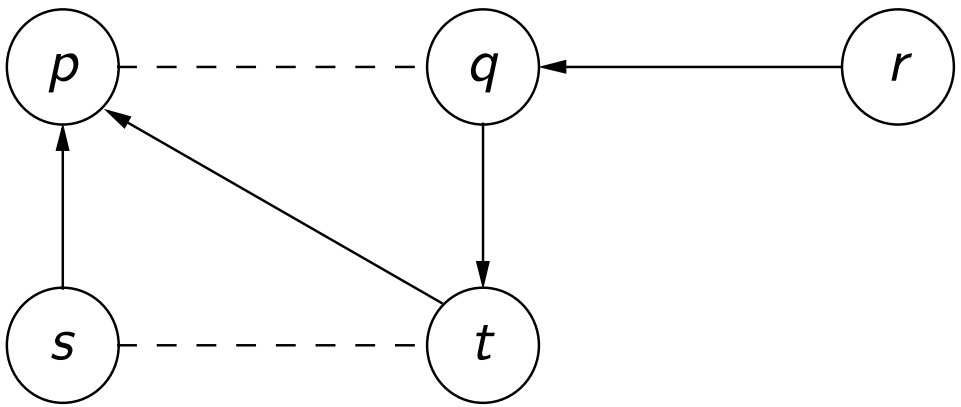
The network is undirected and unweighted.

Arrows and numbers mark the path of the token.

Solid arrows establish a parent-child relation (in the opposite direction).

Tarry's algorithm - Spanning tree

The parent-child relation is the reversal of the solid arrows.



Tree edges, which are part of the spanning tree, are solid.

Frond edges, which aren't part of the spanning tree, are dashed.

Tarry's algorithm - Correctness

Claim: The token θ travels through each channel in either direction, and ends up at the initiator.

Proof : A noninitiator holding θ, received θ once more than it sent θ.

So by R1, this process can send θ into a channel.

Hence θ ends at the initiator, after traversing all its channels both ways.

Assume some channel isn't traversed by θ both ways.

Let noninitiator q be the earliest visited process with such a channel.

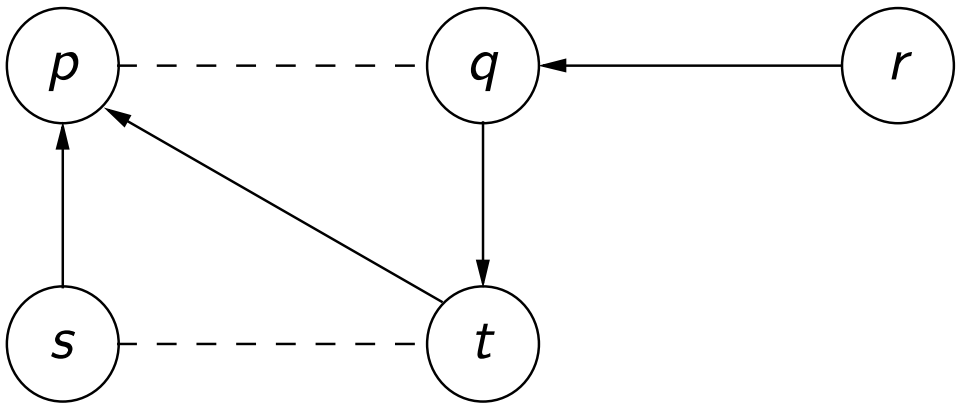
q sends θ to its parent p. Namely, since θ visits p before q, it traverses the channel pq both ways.

So by R2, q sends θ into all its channels.

Since q sends and receives θ an equal number of times, it also receives θ through all its channels.

So θ travels through all channels of q both ways; contradiction.

Question



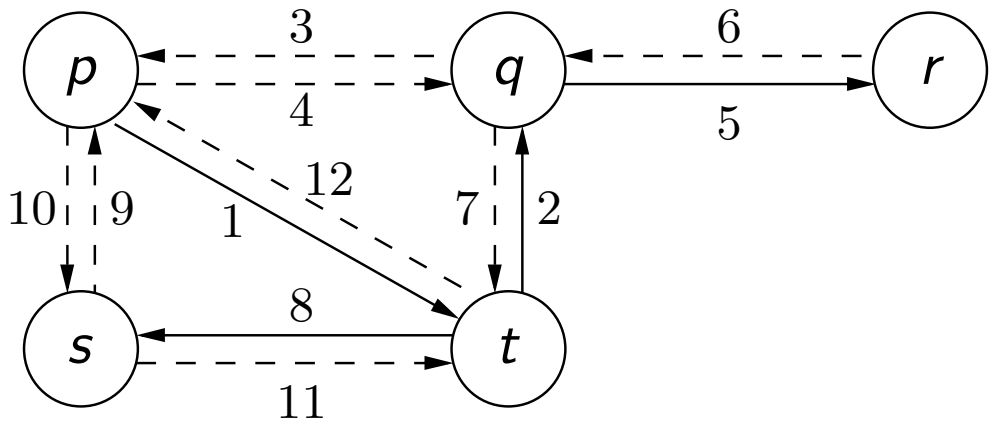
Could this spanning tree have been produced by a depth-first search starting at p?

Depth-first search

Depth-first search is obtained by adding to Tarry's algorithm:

R3 When a process receives the token, it immediately sends it back through the same channel if this is allowed by R1,2.

Example:



In the spanning tree of a depth-first search, all frond edges connect an ancestor with one of its descendants in the spanning tree.

Depth-first search with neighbor knowledge

To prevent transmission of the token through a frond edge, visited processes are included in the token.

The token isn't forwarded to processes in this list (except when a process sends the token back to its parent).

Message complexity: $2N - 2$ messages

Each tree edge carries 2 tokens.

Time complexity: $\leq 2N - 2$ time units

Bit complexity: Up to kN bits per message (where k bits are needed to represent one process).

Awerbuch's algorithm

A process holding the token for the first time informs all neighbors except its parent and the process to which it forwards the token.

The token is only forwarded when these neighbors have all acknowledged reception.

The token is only forwarded to processes that weren't yet visited by the token (except when a process sends the token to its parent).

Awerbuch's algorithm - Complexity

Message complexity: $\leq 4E$ messages

Each frond edge carries 2 info and 2 ack messages.

Each tree edges carries 2 tokens, and possibly 1 info/ack pair.

Time complexity: $\leq 4N - 2$ time units

Each tree edge carries 2 tokens.

Each process waits at most 2 time units for ack's to return.

Cidon's algorithm

Abolish ack's from Awerbuch's algorithm.

The token is forwarded without delay.

Each process p records to which process fw_p it forwarded the token last.

Suppose process p receives the token from a process $q \neq fw_p$.

Then p marks pq as frond edge and *dismisses* the token.

Suppose process q receives an info message from fw_q.

Then q marks pq as frond edge and continues forwarding the token.

Cidon's algorithm - Complexity

Message complexity: $\leq 4E$ messages

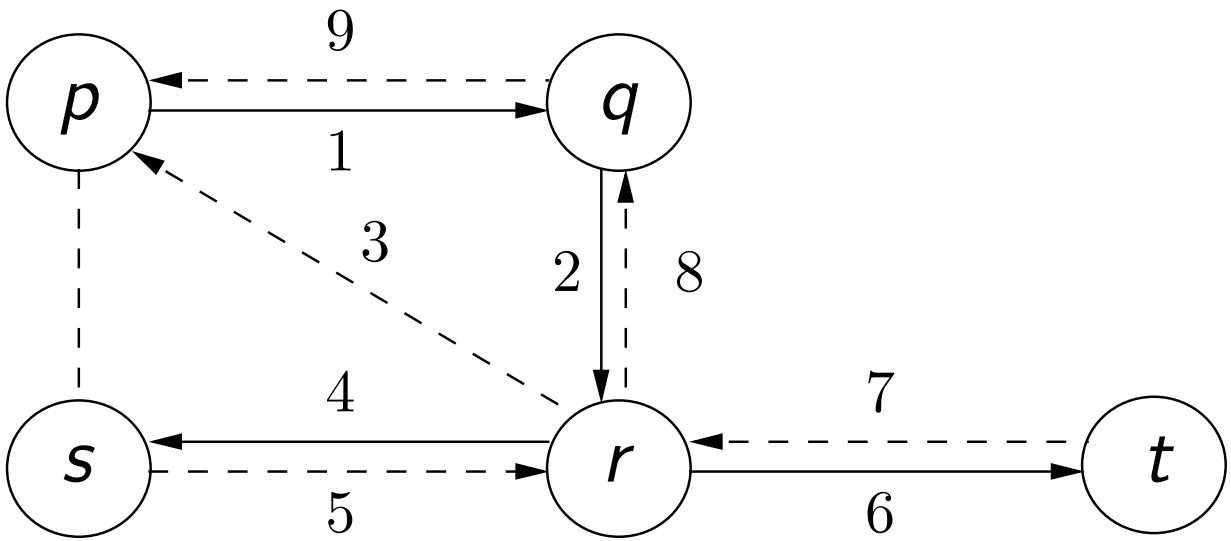
Each channel carries at most 2 info messages and 2 tokens.

Time complexity: $\leq 2N - 2$ time units

Each tree edge carries 2 tokens.

At least once per time unit, a token is forwarded through a tree edge.

Cidon's algorithm - Example



Tree algorithm

The tree algorithm is a decentralized wave algorithm for undirected, acyclic networks.

The local algorithm at a process p:

- p waits until it received messages from all neighbors except one, which becomes its *parent*.

  Then it sends a message to its parent.

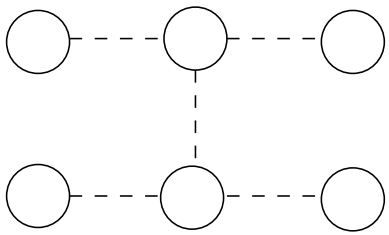
- If p receives a *message* from its parent, it decides.

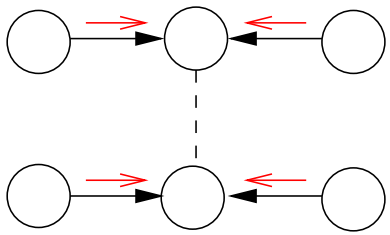
  It sends the decision to all neighbors except its parent.

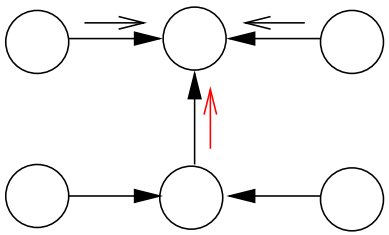
- If p receives a *decision* from its parent, it passes it on to all other neighbors.

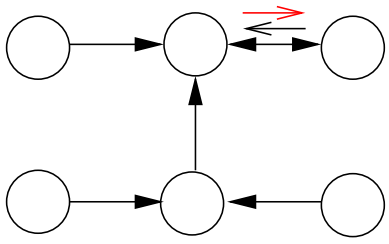
Always *two* (neighboring) processes decide.

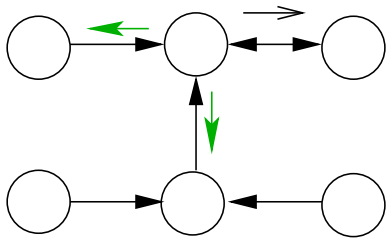
Tree algorithm - Example

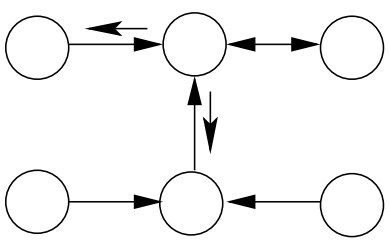


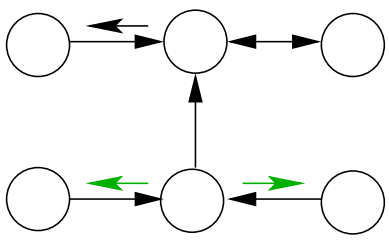


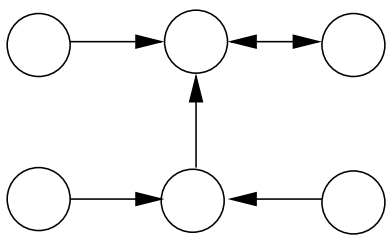












Questions

What happens if the tree algorithm is applied to a network containing a cycle ?

Apply the tree algorithm to compute the size of an undirected, acyclic network.

Tree algorithm - Correctness

Claim: If the tree algorithm is run on an acyclic network with $N > 1$, then exactly two processes decide.

Proof: Suppose some process p never sends a message.

p doesn't receive a message through two of its channels, qp and rp.

q doesn't receive a message through two of its channels, pq and sq.

Continuing this argument, we get a cycle of processes that don't receive a message through two of their channels.

Since the network topology is a tree, there is no cycle; contradiction.

So each process eventually sends a message.

Clearly each channel carries at least one message.

There are $N - 1$ channels, so one channel carries two messages.

Only the two processes connected by this channel decide.

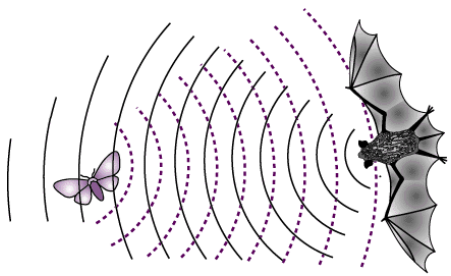
Echo algorithm

The echo algorithm is a centralized wave algorithm for undirected networks.

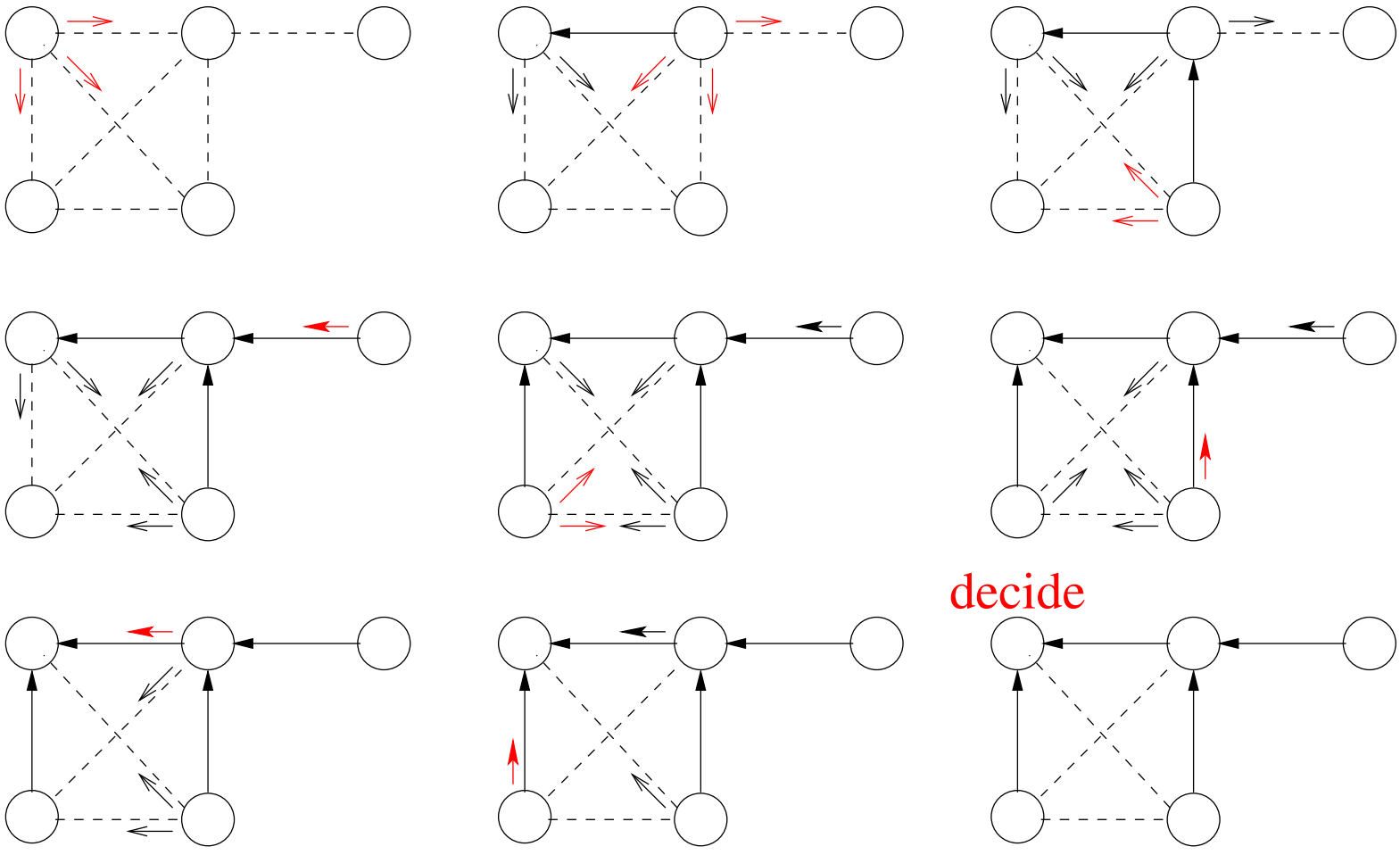
- The initiator sends a message to all neighbors.
- When a noninitiator receives a message for the first time, it makes the sender its *parent*.

  Then it sends a message to all neighbors except its parent.
- When a noninitiator has received a message from all neighbors, it sends a message to its parent.
- When the initiator has received a message from all neighbors, it *decides*.

Message complexity: $2E$ messages



Echo algorithm - Example



Questions

Use the echo algorithm to determine the largest process id.

Let each process initiate a run of the echo algorithm, tagged by its id.

Processes only participate in the "largest" wave they have seen so far.

Which of these concurrent waves complete ?

Communication and resource deadlock

A deadlock occurs if there is a cycle of processes waiting until:

- another process on the cycle sends some input (communication deadlock)
- or resources held by other processes on the cycle are released (resource deadlock)

Both types of deadlock are captured by the N-out-of-M model:

A process can wait for N grants out of M requests.

Examples:

- A process is waiting for one message from a group of processes: $N = 1$
- A database transaction first needs to lock several files: $N = M$.

Wait-for graph

A (non-blocked) process can issue a request to M other processes, and becomes blocked until N of these requests have been granted.

Then it informs the remaining $M - N$ processes that the request can be dismissed.

Only non-blocked processes can grant a request.

A (directed) wait-for graph captures dependencies between processes.

There is an edge from node p to node q if p sent a request to q that wasn't yet dismissed by p or granted by q.

Wait-for graph - Example 1

Suppose process p must wait for a message from process q.

In the wait-for graph, node p sends a request to node q.

Then edge pq is created in the wait-for graph, and p becomes blocked.

When q sends a message to p, the request of p is granted.

Then edge pq is removed from the wait-for graph, and p becomes unblocked.

Wait-for graph - Example 2

Suppose two processes p and q want to claim a resource.

In the wait-for graph, nodes u, v representing p, q send a request to node w representing the resource. Edges uw and vw are created.

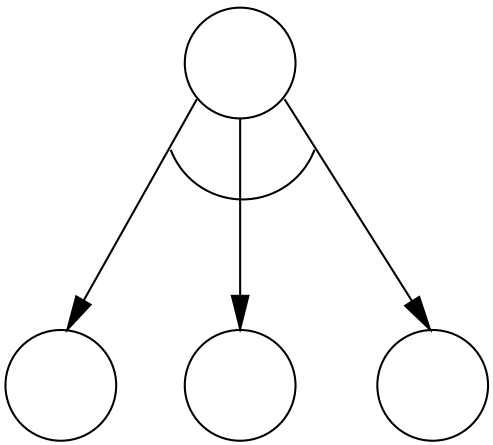
Since the resource is free, the resource is given to say p.
So w sends a grant to u. Edge uw is removed.

The basic (mutual exclusion) algorithm requires that the resource must be released by p before q can claim it.
So w sends a request to u, creating edge wu in the wait-for graph.

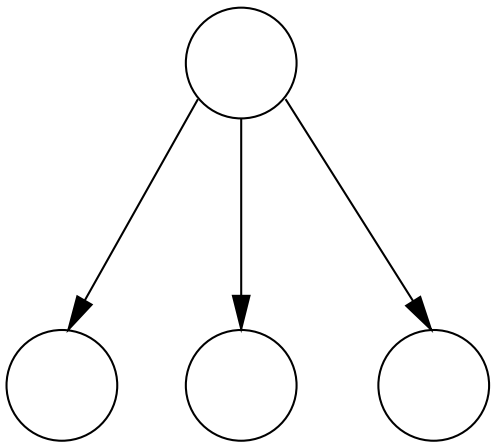
After p releases the resource, u grants the request of w.
Edge wu is removed.

The resource is given to q. Hence w grants the request from v.
Edge vw is removed and edge wv is created.

Drawing wait-for graphs



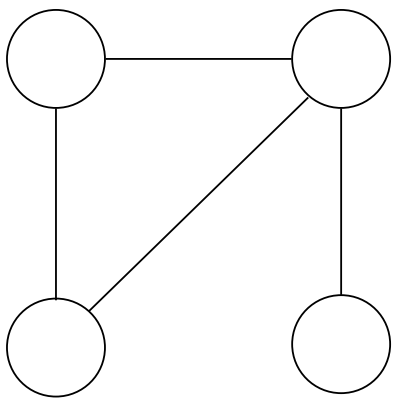
AND (3–out–of–3) request



OR (1–out–of–3) request

Questions

Draw the wait-for graph for the initial configuration of the tree algorithm, applied to the following network.



Static analysis on a wait-for graph

A snapshot is taken of the wait-for graph.

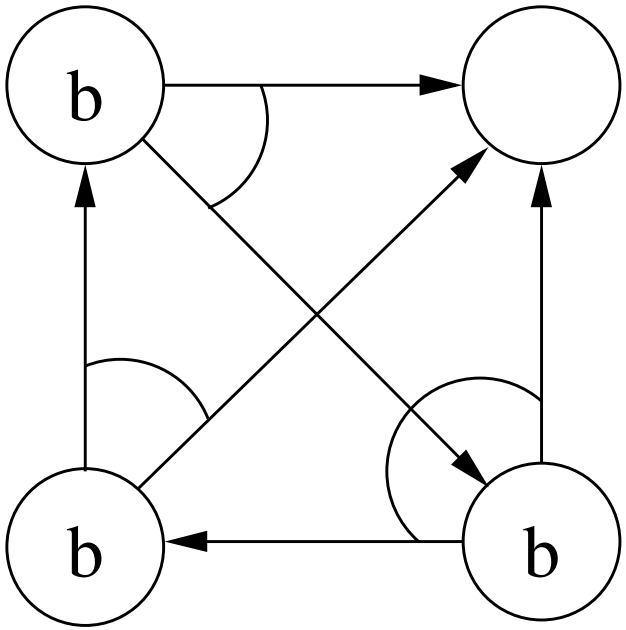
A *static analysis* on the wait-for graph may reveal deadlocks:

- Non-blocked nodes can grant requests.
- When a request is granted, the corresponding edge is removed.
- When an N-out-of-M request has received N grants, the requester becomes unblocked.

  (The remaining $M - N$ outgoing edges are dismissed.)

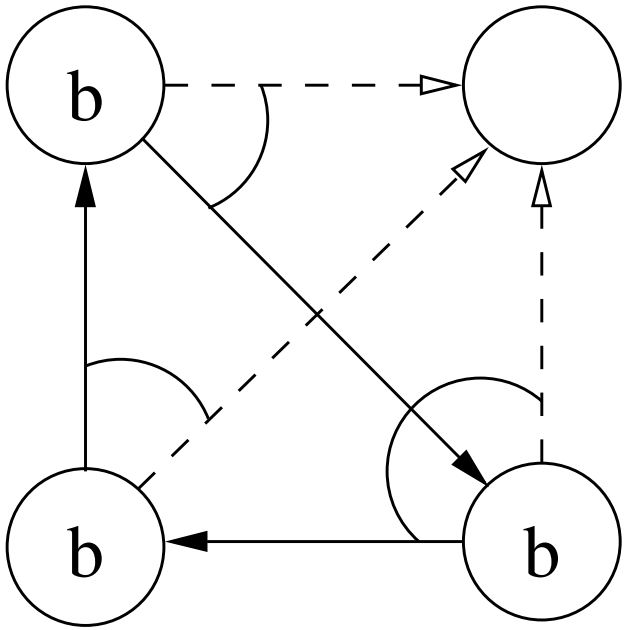
When no more grants are possible, nodes that remain blocked in the wait-for graph are deadlocked in the snapshot of the basic algorithm.

Static analysis - Example 1



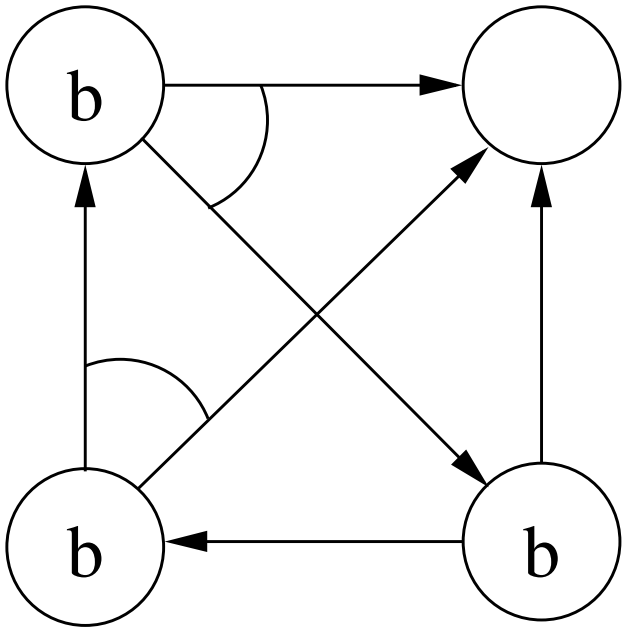
Is there a deadlock ?

Static analysis - Example 1

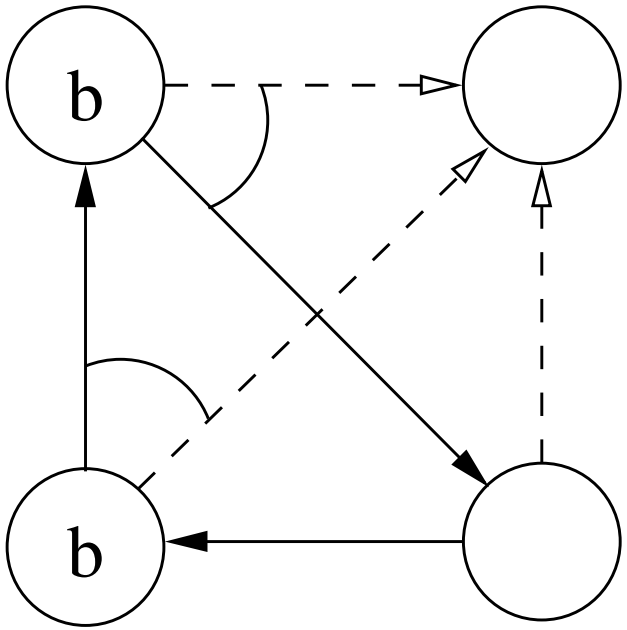


Deadlock

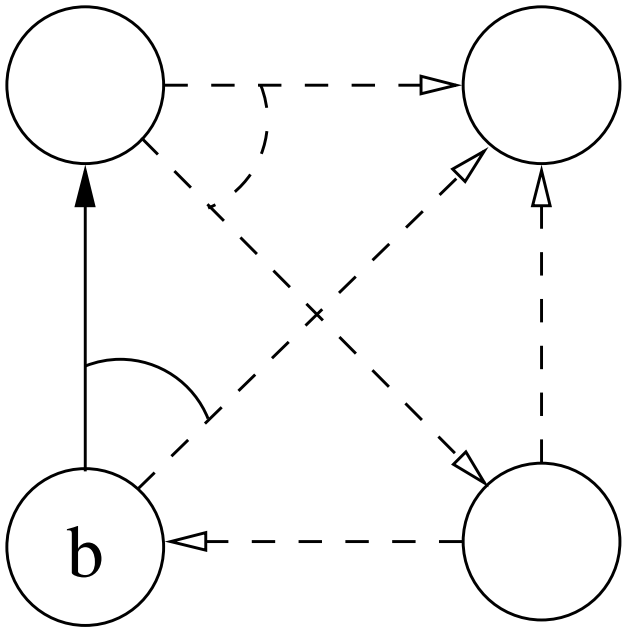
Static analysis - Example 2



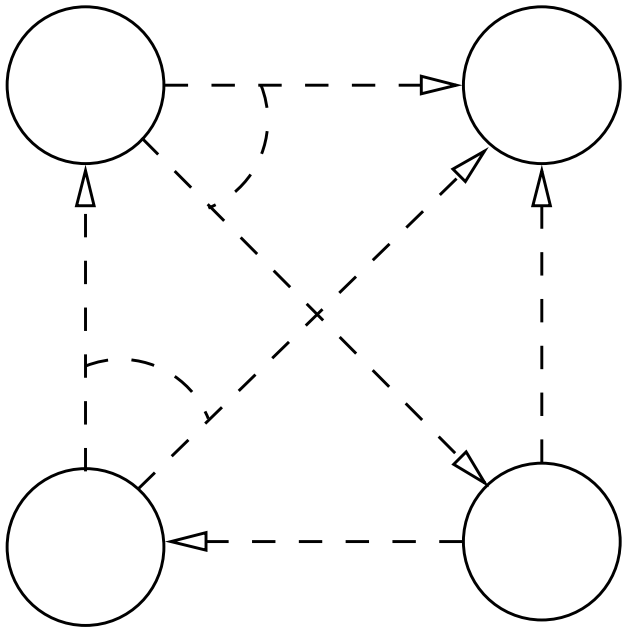
Static analysis - Example 2



Static analysis - Example 2



Static analysis - Example 2



No deadlock

Bracha-Toueg deadlock detection algorithm - Snapshot

Given an undirected network, and a basic algorithm.

A process that suspects it is deadlocked, initiates
a (Lai-Yang) *snapshot* to compute the wait-for graph.

Each node u takes a local snapshot of:

- requests it sent or received that weren't yet granted or dismissed;
- grant and dismiss messages in edges.

Then it computes:

Out_u: the nodes it sent a request to (not granted)

In_u: the nodes it received a request from (not dismissed)

Bracha-Toueg deadlock detection algorithm

$requests_u$ is the number of grants u requires to become unblocked.

When u receives a grant message, $requests_u \leftarrow requests_u - 1$.

If $requests_u$ becomes 0, u sends grant messages to all nodes in In_u.

If after termination of the deadlock detection run, $requests > 0$ at the initiator, then it is deadlocked (in the basic algorithm).

Challenge: The initiator must detect termination of deadlock detection.

Bracha-Toueg deadlock detection algorithm

Initially $notified_u = false$ and $free_u = false$ at all nodes u.

The initiator starts a deadlock detection run by executing *Notify*.

$Notify_u$: $notified_u \leftarrow true$
for all $w \in Out_u$ send NOTIFY to w
if $requests_u = 0$ then $Grant_u$
for all $w \in Out_u$ await DONE from w

$Grant_u$: $free_u \leftarrow true$
for all $w \in In_u$ send GRANT to w
for all $w \in In_u$ await ACK from w

While a node is awaiting DONE or ACK messages,
it can process incoming NOTIFY and GRANT messages.

Bracha-Toueg deadlock detection algorithm

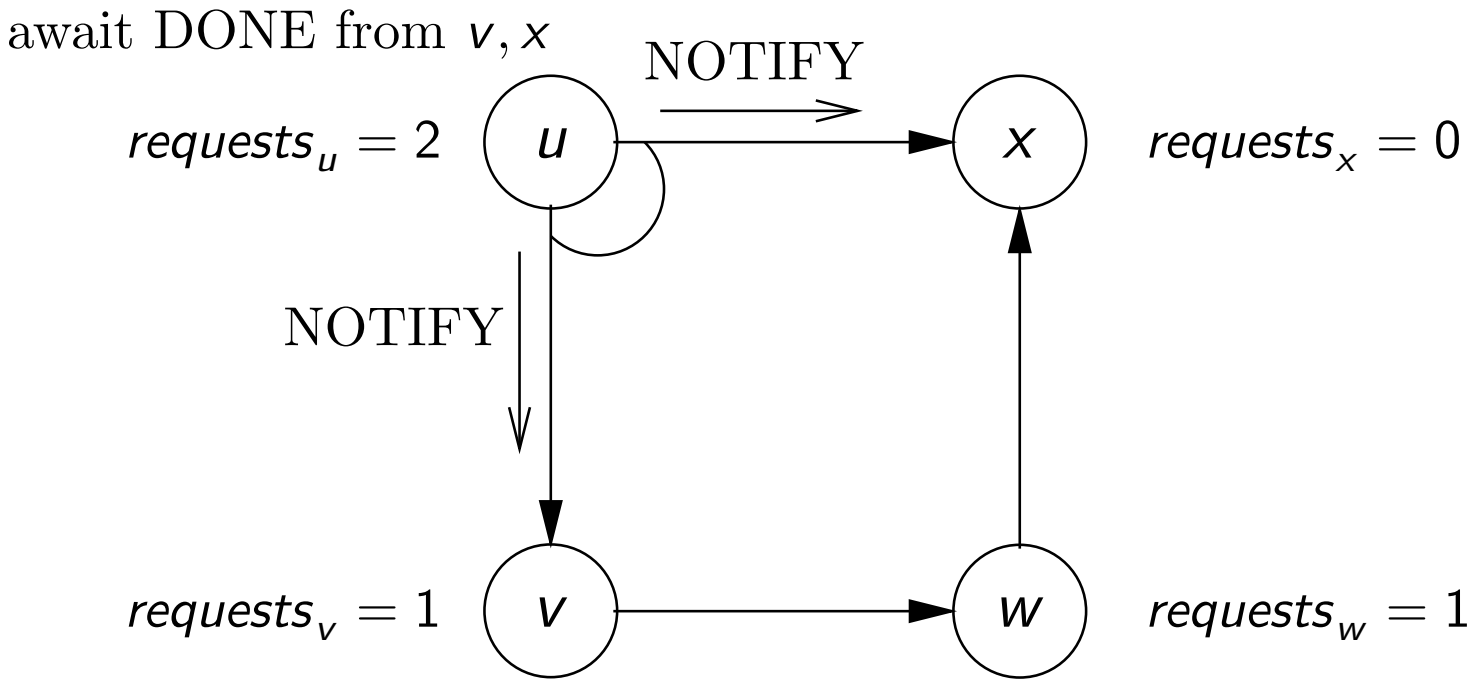
Let u receive NOTIFY.
If $notified_u = false$, then u executes $Notify_u$.
u sends back DONE.

Let u receive GRANT.
If $requests_u > 0$, then $requests_u \leftarrow requests_u - 1$;
if $requests_u$ becomes 0, then u executes $Grant_u$.
u sends back ACK.

When the initiator has received DONE from all nodes in its *Out* set, it checks the value of its *free* field.

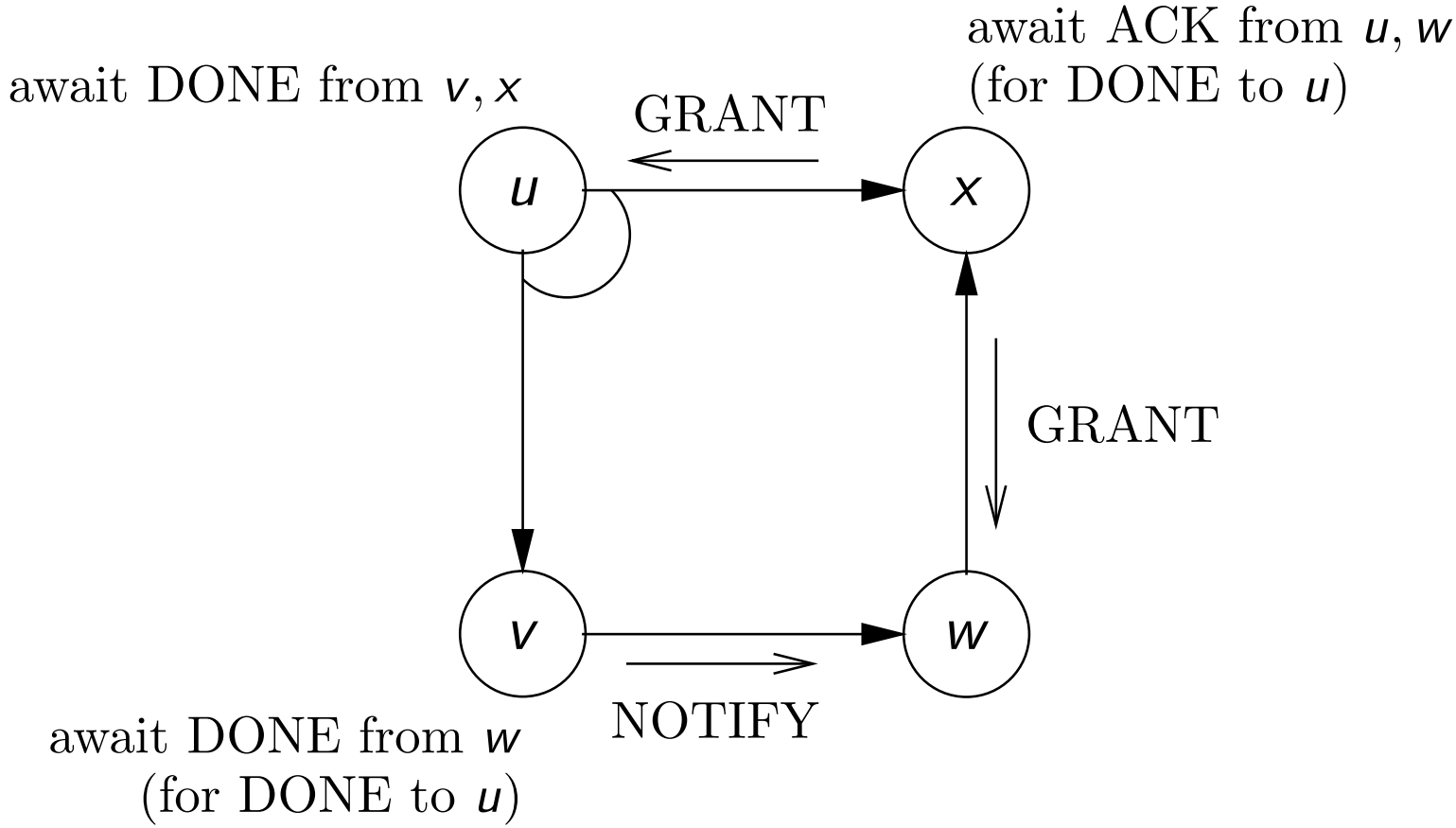
If it is still *false*, the initiator concludes it is deadlocked.

Bracha-Toueg deadlock detection algorithm - Example

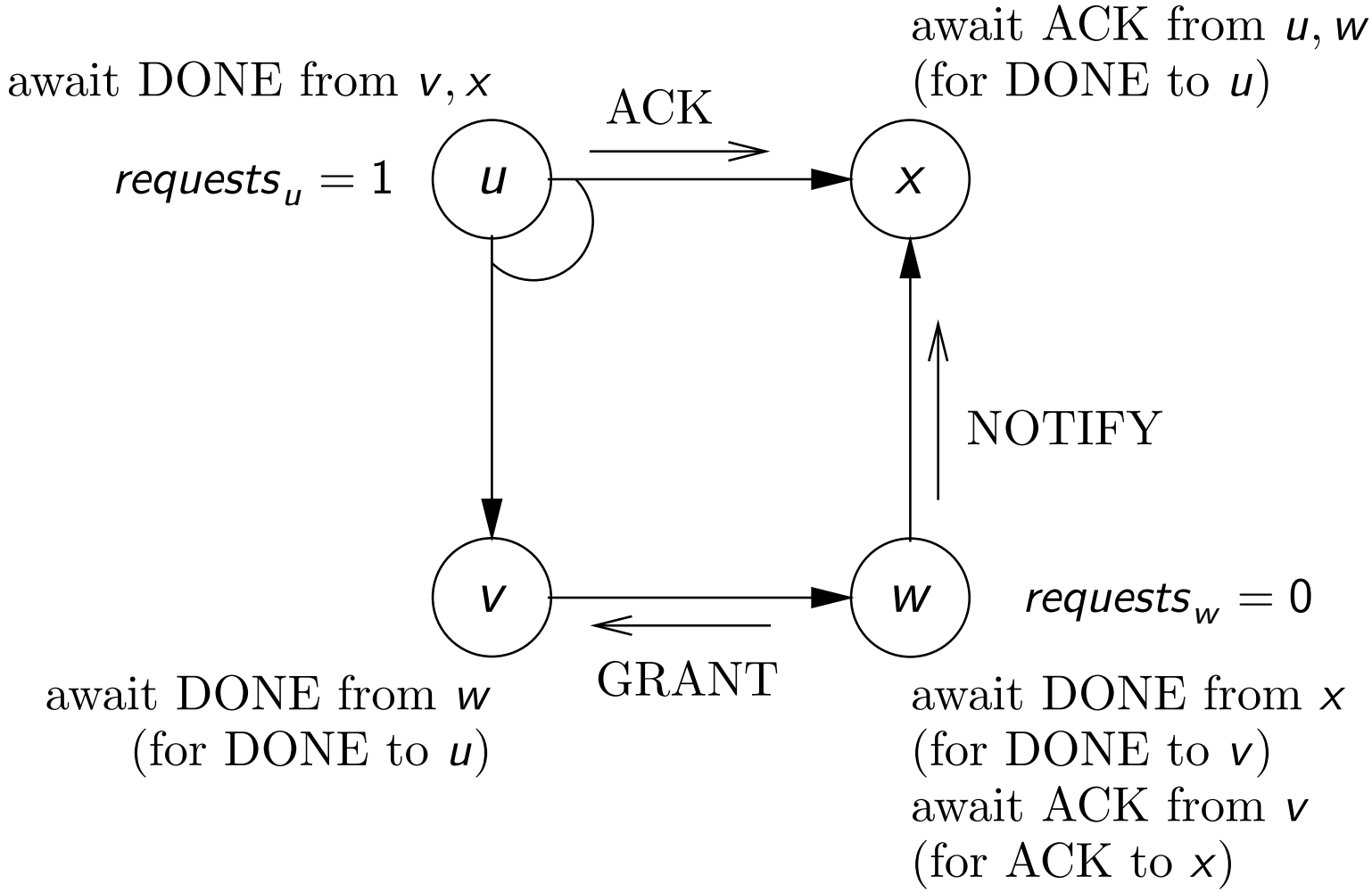


u is the initiator

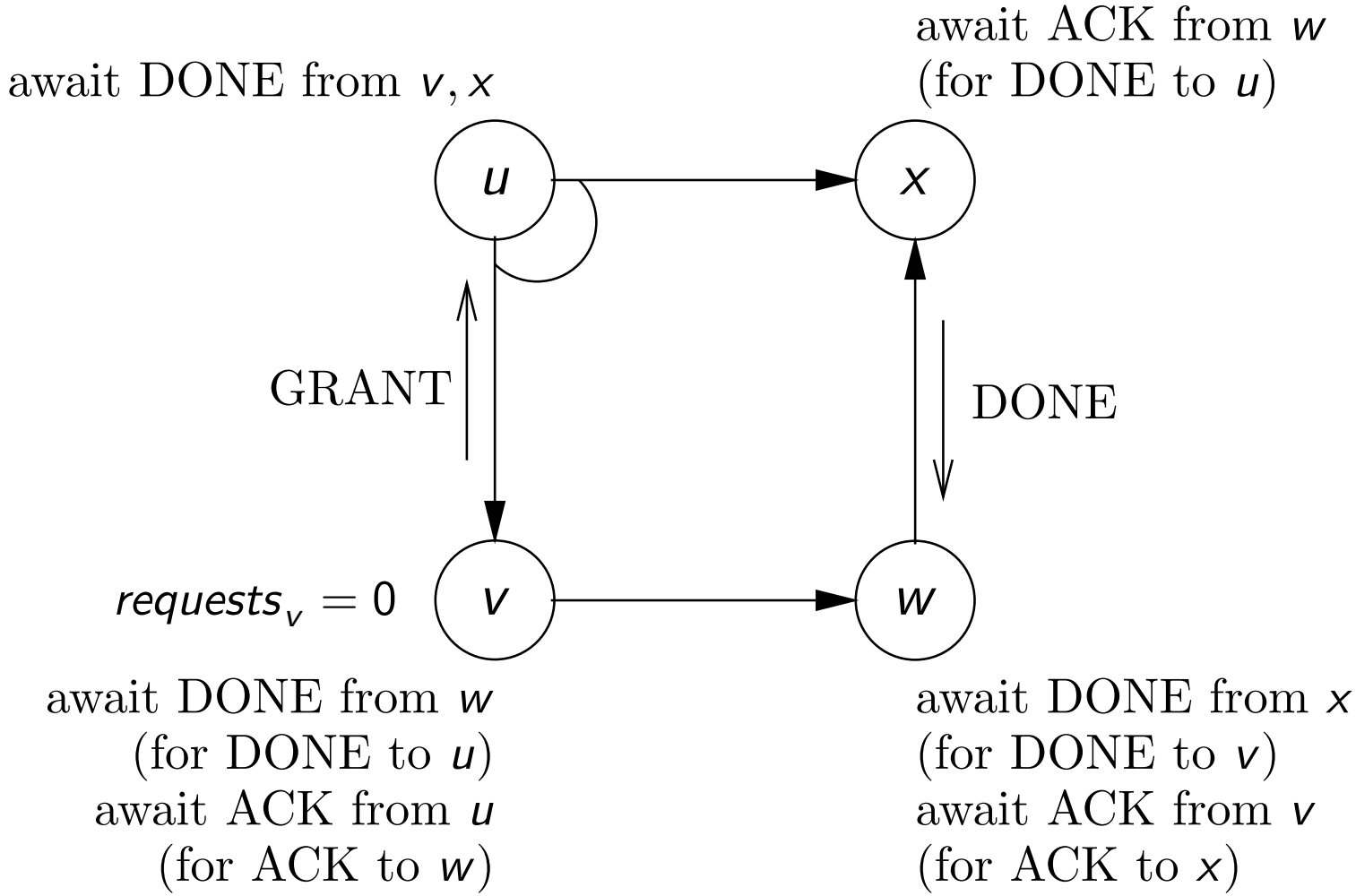
Bracha-Toueg deadlock detection algorithm - Example



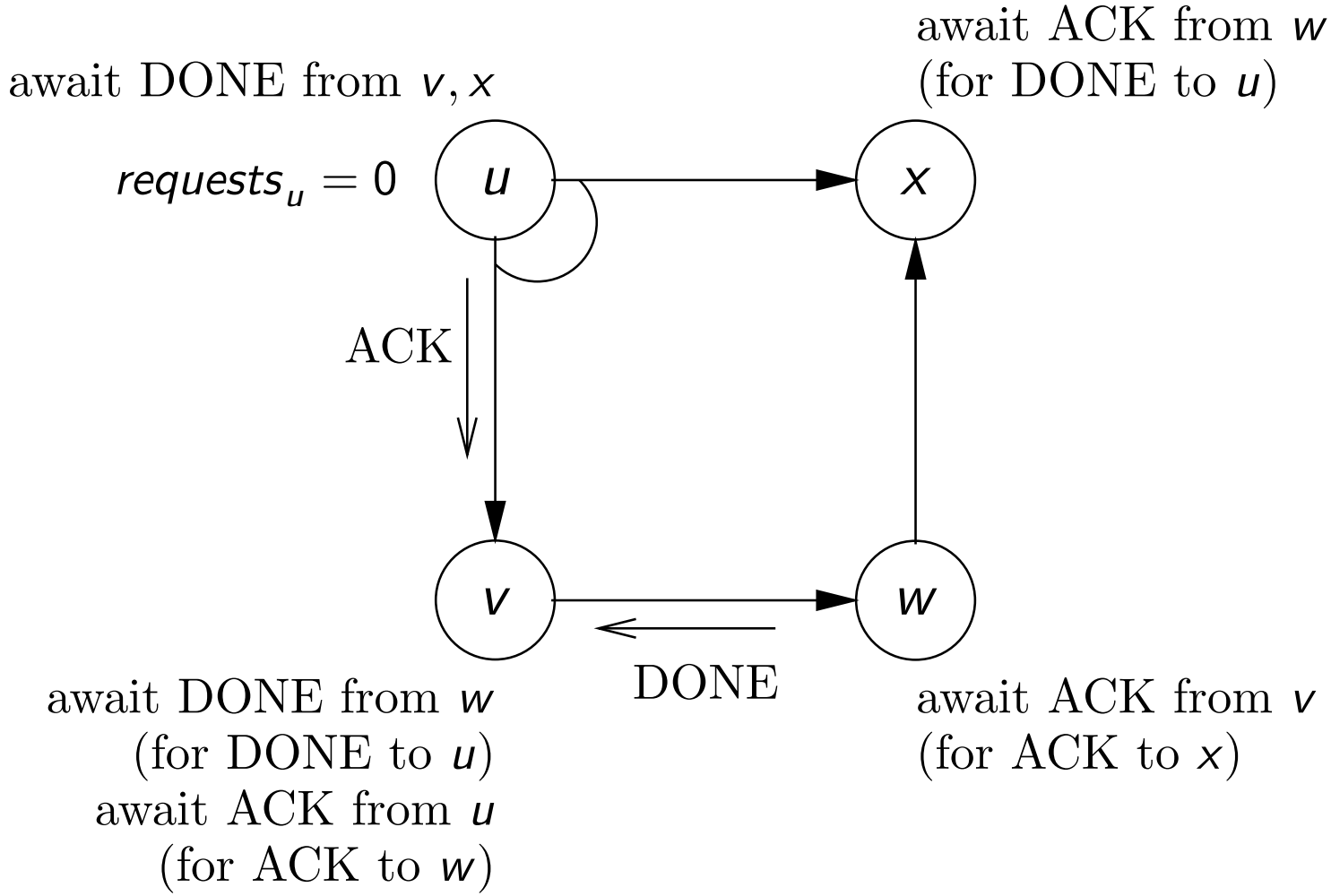
Bracha-Toueg deadlock detection algorithm - Example



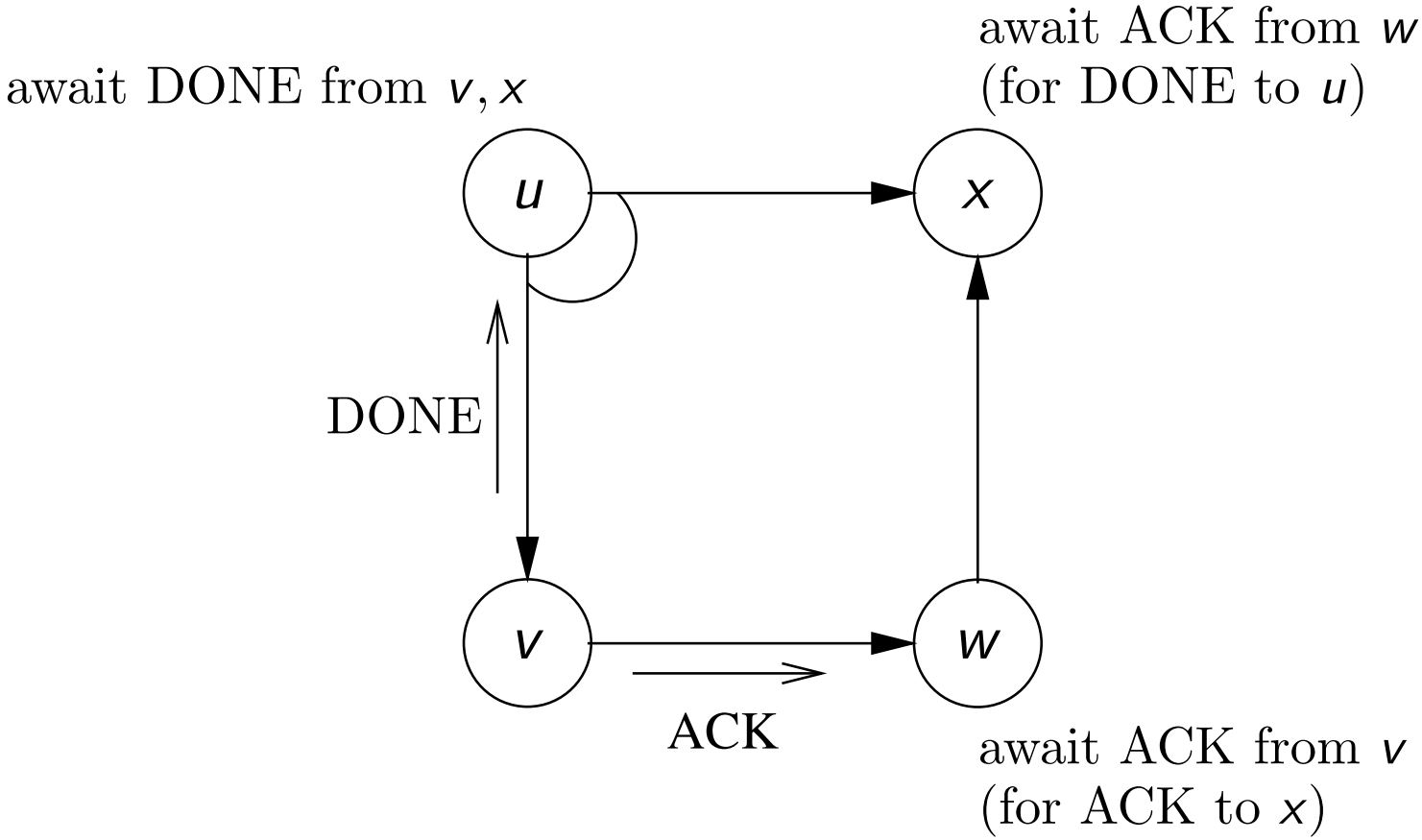
Bracha-Toueg deadlock detection algorithm - Example



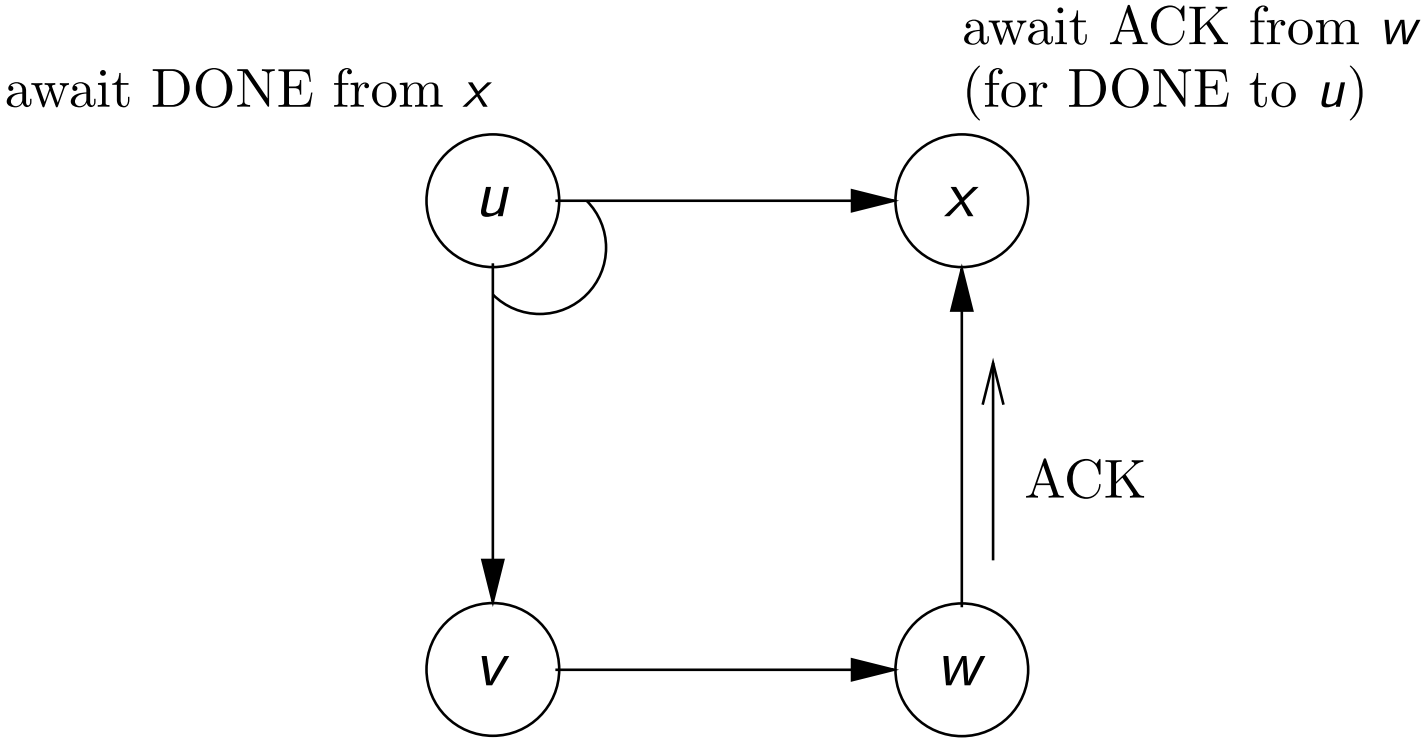
Bracha-Toueg deadlock detection algorithm - Example



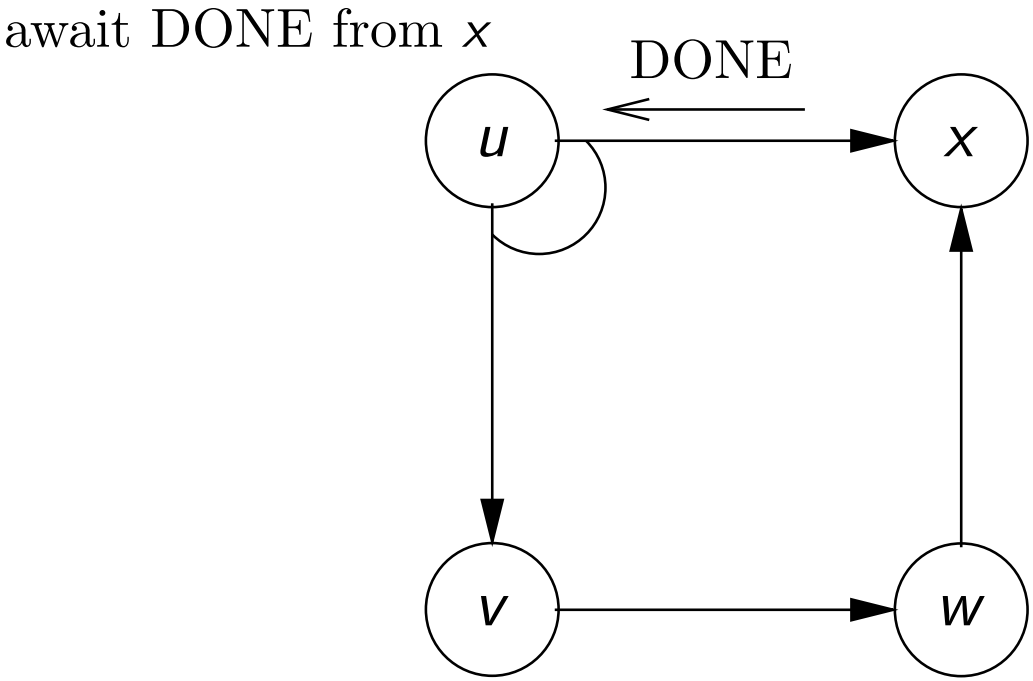
Bracha-Toueg deadlock detection algorithm - Example



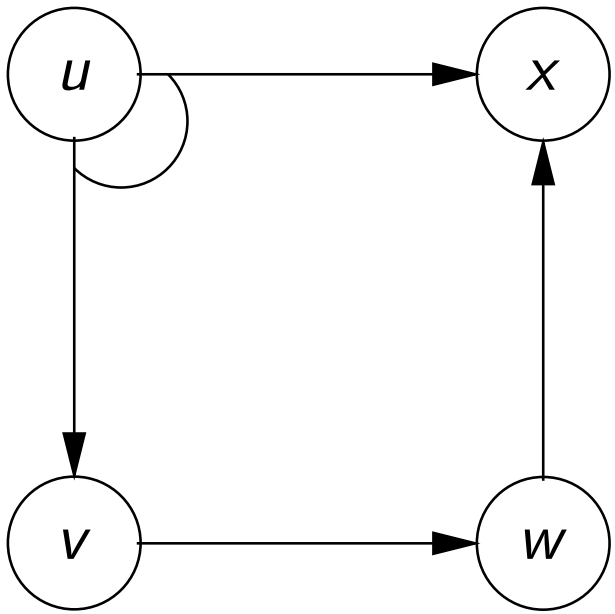
Bracha-Toueg deadlock detection algorithm - Example



Bracha-Toueg deadlock detection algorithm - Example



Bracha-Toueg deadlock detection algorithm - Example



$free_u = true$, so u concludes that it isn't deadlocked.

Bracha-Toueg deadlock detection algorithm - Correctness

The Bracha-Toueg algorithm is deadlock-free:

The initiator eventually receives DONE's from all nodes in its *Out* set.

At that moment the Bracha-Toueg algorithm has terminated.

Two types of trees are constructed, similar to the echo algorithm:

1. NOTIFY/DONE's construct a tree T rooted in the initiator.
2. GRANT/ACK's construct disjoint trees T_v,
   rooted in a node v where from the start $requests_v = 0$.

The NOTIFY/DONE's only complete when all GRANT/ACK's have completed.

Bracha-Toueg deadlock detection algorithm - Correctness

In a deadlock detection run, requests are granted as much as possible.

Therefore, if the initiator has received DONE's from all nodes in its *Out* set and its *free* field is still *false*, it is deadlocked.

Vice versa, if its *free* field is *true*, there is no deadlock (yet), *(if resource requests are granted nondeterministically)*.

Question

Could we apply the Bracha-Toueg algorithm to itself, to establish that it is a deadlock-free algorithm ?

Answer: No.

The Bracha-Toueg algorithm can only establish whether a deadlock is present in a snapshot of one computation of the basic algorithm.

Lecture in a nutshell

wave algorithm

traversal algorithm

- ring algorithm
- Tarry's algorithm
- depth-first search

tree algorithm

echo algorithm

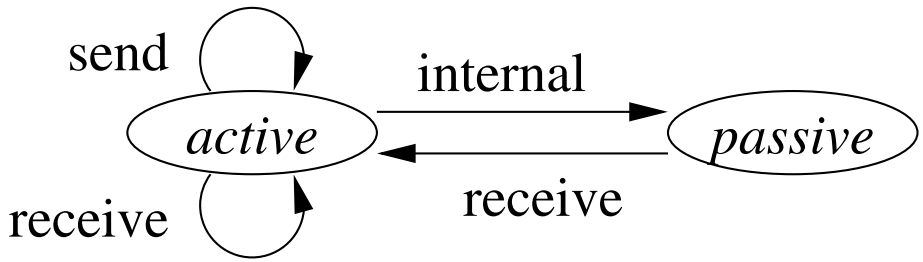
communication and resource deadlock

wait-for graph

Bracha-Toueg deadlock detection algorithm

Termination detection

The *basic* algorithm is terminated if (1) each process is passive, and (2) no basic messages are in transit.



The *control* algorithm concerns termination detection and announcement.

Announcement is simple; we focus on detection.

Termination detection shouldn't influence basic computations.

Dijkstra-Scholten algorithm

Requires a centralized basic algorithm, and an undirected network.

A *tree* T is maintained, which has the initiator p_0 as the root, and includes all active processes. *Initially*, T consists of p_0.

cc_p estimates (from above) the number of children of process p in T.

- When p sends a basic message, $cc_p \leftarrow cc_p + 1$.
- Let this message be received by q.
  - If q isn't yet in T, it joins T with parent p and $cc_q \leftarrow 0$.
  - If q is already in T, it sends a control message to p that it isn't a new child of p. Upon receipt of this message, $cc_p \leftarrow cc_p - 1$.
- When a noninitiator p is passive and $cc_p = 0$, it quits T and informs its parent that it is no longer a child.
- When the initiator p_0 is passive and $cc_{p_0} = 0$, it calls *Announce*.

Question

Let the initiator send a basic message and then become passive.

Why doesn't it immediately detect termination ?

Shavit-Francez algorithm

Allows a decentralized basic algorithm; requires an undirected network.

A *forest* F of (disjoint) trees is maintained, rooted in initiators.

Initially, each initiator of the basic algorithm constitutes a tree in F.

- When a process p sends a basic message, $cc_p \leftarrow cc_p + 1$.
- Let this message be received by q.
  - If q isn't yet in *a tree in* F, it joins F with parent p and $cc_q \leftarrow 0$.
  - If q is already in *a tree in* F, it sends a control message to p that it isn't a new child of p. Upon receipt, $cc_p \leftarrow cc_p - 1$.
- When a noninitiator p is passive and $cc_p = 0$, it informs its parent that it is no longer a child.

A passive initiator p with $cc_p = 0$ starts a wave, tagged with its id.

Processes in a tree refuse to participate; *decide* calls *Announce*.

Rana's algorithm

Allows a decentralized basic algorithm; requires an undirected network.

Each basic message is acknowledged.

A logical clock provides *(basic and control)* events with a time stamp.

The time stamp of a process is the highest time stamp of its events so far (initially it is 0).

If at time t a process becomes quiet, i.e. (1) it has become passive, and (2) all basic messages it sent have been acknowledged, it starts a wave (of control messages), *tagged with* t (and its id).

Only processes *that have been quiet from a time* $\leq t$ *on* take part in the wave.

If a wave completes, its initiator calls *Announce*.

Rana's algorithm - Correctness

Suppose a wave, tagged with some t, doesn't complete.

Then some process p doesn't take part in this wave.

Due to this wave, p's logical time becomes greater than t.

When p becomes quiet, it starts a new wave, tagged with a $t' > t$.

Rana's algorithm - Correctness

Suppose a quiet process q takes part in a wave,
and is later on made active by a basic message from a process p
that wasn't yet visited by this wave.

Then this wave won't complete.

Namely, let the wave be tagged with t.

When q takes part in the wave, its logical clock becomes $> t$.

By the ack from q to p, in response to the basic message from p,
the logical clock of p becomes $> t$.

So p won't take part in the wave (because it is tagged with t).

Question

What is a drawback of the Dijkstra-Scholten as well as Rana's algorithm ?

Answer: Requires one control message for every basic message.

Weight-throwing termination detection

Requires a centralized basic algorithm; allows a directed network.

The initiator has weight 1, all noninitiators have weight 0.

When a process *sends* a basic message, it transfers part of its weight to this message.

When a process *receives* a basic message, it adds the weight of this message to its own weight.

When a noninitiator becomes passive, it returns its weight to the initiator.

When the initiator becomes passive, and has regained weight 1, it calls *Announce*.

Weight-throwing termination detection - Underflow

Underflow: The weight of a process can become too small to be divided further.

Solution 1: The process gives itself extra weight, and informs the initiator that there is additional weight in the system.

An ack from the initiator is needed before the extra weight can be used, to avoid race conditions.

Solution 2: The process initiates a weight-throwing termination detection sub-call, and only returns its weight to the initiator when it has become passive and this sub-call has terminated.

Question

Why is the following termination detection algorithm not correct ?

- Each basic message is acknowledged.
- If a process becomes quiet, i.e. (1) it has become passive, and (2) all basic messages it sent have been acknowledged, then it starts a wave (tagged with its id).
- Only quiet processes take part in the wave.
- If the wave completes, its initiator calls *Announce*.

Answer: Let a process p that wasn't yet visited by the wave make a quiet process q that was already visited active again.

Next p becomes quiet before the wave arrives.

Now the wave can complete while q is active.

Token-based termination detection

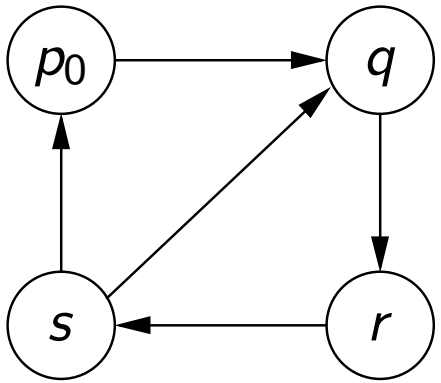
The following centralized termination detection algorithm allows a decentralized basic algorithm and a directed network.

A process p_0 is initiator of a traversal algorithm to check whether all processes are passive.

Complication 1: Due to the directed channels, reception of basic messages can't be acknowledged.

Complication 2: A traversal of only passive processes doesn't guarantee termination (even if there are no basic messages in the channels).

Complication 2 - Example



The token is at p_0; only s is active.

The token travels to r.

s sends a basic message to q, making q active.

s becomes passive.

The token travels on to p_0, which falsely calls *Announce*.

Safra's algorithm

Allows a decentralized basic algorithm and a directed network.

Each process maintains a counter of type $\mathbb{Z}$; initially it is 0.

At each outgoing/incoming basic message, the counter is increased/decreased.

At any time, the sum of all counters in the network is ≥ 0, and it is 0 if and only if no basic messages are in transit.

At each round trip, the token carries the sum of the counters of the processes it has traversed.

Complication: The token may end a round trip with a negative sum, when a visited passive process becomes active by a basic message, and sends basic messages that are received by an unvisited process.

Safra's algorithm

Processes are colored white or black. Initially they are white, and a process that receives a basic message becomes black.

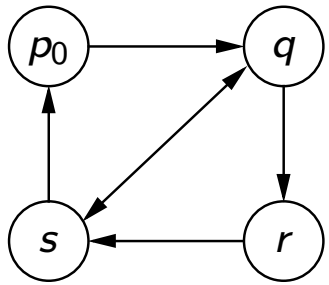
- When p_0 is passive, it sends a white token with counter 0.
- A noninitiator only forwards the token when it is passive.
- When a *black* process receives the token, the process becomes white and the token black.

  The token will stay black for the rest of the round trip.
- Eventually the token returns to p_0, who waits until it is passive:

- If the token is white and the sum of all counters is zero, p_0 calls *Announce*.
- Else, p_0 sends a white token again.

Safra's algorithm - Example

The token is at p_0; only s is active; no messages are in transit; all processes are white with counter 0.



s sends a basic message **m** to q, setting the counter of s to 1.

s becomes passive.

The token travels around the network, white with sum 1.

The token travels on to r, white with sum 0.

m travels to q and back to s, making them active, black, with counter 0.

s becomes passive.

The token travels from r to p_0, black with sum 0.

q becomes passive.

After two more round trips of the token, p_0 calls *Announce*.

Safra's algorithm - Correctness

When the system has terminated,

- the token will color all processes white, and
- the counters of the processes sum up to zero.

So the token eventually returns to the initiator white with counter 0.

Suppose a token returns to the initiator white with counter 0.

Since the token is white: if reception of a message is included in the counter, then sending this message is included in the counter too.

So, since the counter is 0:

- no process was made active after the token's visit, and
- no messages are in transit.

Question

Any suggestions for an optimization of Safra's algorithm ?

(Hint: Can we do away with black tokens ?)

Answer: When a *black* process gets the token, it dismisses the token (and becomes white).

When the process becomes passive, it sends a fresh token, tagged with its id.

Garbage collection

Processes are provided with memory.

Objects carry *pointers* to local objects and *references* to remote objects.

A root object can be created in memory; objects are always accessed by navigating from a root object.

Aim of garbage collection: To reclaim inaccessible objects.

Three operations by processes to build or delete a reference:

- *Creation*: The object owner sends a pointer to another process.
- *Duplication*: A process that isn't object owner sends a reference to another process.
- *Deletion*: The reference is deleted at its process.

Reference counting

Reference counting tracks the number of references to an object.

If it drops to zero, and there are no pointers, the object is garbage.

Advantage: Can be performed at run-time.

Drawback: Can't reclaim *cyclic* garbage.

Indirect reference counting

A tree is maintained for each object, with the object at the root, and the references to this object as the other nodes in the tree.

Each object maintains a counter how many references to it have been *created*.

Each reference is supplied with a counter how many times it has been *duplicated*.

References keep track of their parent in the tree, where they were duplicated or created from.

Indirect reference counting

If a process receives a reference, but already holds a reference to or owns this object, it sends back a decrement.

When a duplicated (or created) reference has been deleted, and its counter is zero, a decrement is sent to the process it was duplicated from (or to the object owner).

When the counter of the object becomes zero, and there are no pointers to it, the object can be reclaimed.

Weighted reference counting

Each object carries a total weight (equal to the weights of all references to the object), and a partial weight.

When a reference is created, the partial weight of the object is divided over the object and the reference.

When a reference is duplicated, the weight of the reference is divided over itself and the copy.

When a reference is deleted, the object owner is notified, and the weight of the deleted reference is subtracted from the total weight of the object.

If the total weight of the object becomes equal to its partial weight, and there are no pointers to the object, it can be reclaimed.

Weighted reference counting - Underflow

When the weight of a reference (or object) becomes too small
to be divided further, no more duplication (or creation) is possible.

Solution 1: The reference increases its weight,
and tells the object owner to increase its total weight.

An ack from the object owner to the reference is needed before
the additional weight can be used, to avoid race conditions.

Solution 2: The process at which the underflow occurs
creates an artificial object with a new total weight,
and with a reference to the original object.

Duplicated references are then to the artificial object,
so that references to the original object become *indirect*.

Question

Why is it much more important to address underflow of weight than overflow of a reference counter ?

Answer: At each reference creation and duplication, weight decreases *exponentially* fast, while the reference counter increases *linearly*.

Garbage collection $\Rightarrow$ termination detection

Garbage collection algorithms can be *transformed* into (existing and new) termination detection algorithms.

Given a basic algorithm.

Let each process p host one artificial root object O_p.

There is also a special non-root object Z.

Initially, only initiators p hold a reference from O_p to Z.

Each basic message carries a duplication of the Z-reference.

When a process becomes passive, it deletes its Z-reference.

The basic algorithm is terminated if and only if Z is garbage.

Garbage collection $\Rightarrow$ termination detection - Examples

Indirect *reference counting* $\Rightarrow$ Dijkstra-Scholten *termination detection*.

Weighted *reference counting* $\Rightarrow$ weight-throwing *termination detection*.

Mark-scan

Mark-scan garbage collection consists of two phases:

- A traversal of all accessible objects, which are marked.
- All unmarked objects are reclaimed.

Drawback: In a distributed setting, mark-scan usually requires *freezing the basic computation*.

In mark-copy, the second phase consists of copying all marked objects to contiguous empty memory space.

In mark-compact, the second phase compacts all marked objects without requiring empty space.

Copying is significantly faster than compaction, but leads to fragmentation of the memory space (and uses more memory).

Generational garbage collection

In practice, most objects either can be reclaimed shortly after their creation, or stay accessible for a very long time.

Garbage collection in Java, which is based on mark-scan, therefore divides objects into multiple generations.

- Garbage in the youngest generation is collected *frequently* using mark-*copy*.
- Garbage in the older generations is collected *sporadically* using mark-*compact*.

This lecture in a nutshell

termination detection

- Dijkstra-Scholten algorithm
- Shavit-Francez algorithm
- Rana's algorithm
- weight throwing
- Safra's algorithm

garbage collection $\Rightarrow$ termination detection

- indirect reference counting
- weighted reference counting
- mark-scan
- generational garbage collection

Routing

Routing means guiding a packet in a network to its destination.

A routing table at node u stores for each $v \neq u$ a neighbor w of u:
Each packet with destination v that arrives at u is passed on to w.

Criteria for good routing algorithms:

- use of optimal paths
- robust with respect to topology changes in the network
- cope with very large, dynamic networks
- table adaptation to avoid busy edges

Chandy-Misra algorithm

Consider an undirected, weighted network, with weights $\omega_{vw} > 0$.

A *centralized* algorithm to compute all shortest paths to initiator u_0.

Initially, $dist_{u_0}(u_0) = 0$, $dist_v(u_0) = \infty$ if $v \neq u_0$, and $parent_v(u_0) = \perp$.

u_0 sends the message $\langle 0 \rangle$ to its neighbors.

Let node v receives $\langle d \rangle$ from neighbor w. If $d + \omega_{vw} < dist_v(u_0)$, then:

- $dist_v(u_0) \leftarrow d + \omega_{vw}$ and $parent_v(u_0) \leftarrow w$
- v sends $\langle dist_v(u_0) \rangle$ to its neighbors (except w)

Termination detection by e.g. the Dijkstra-Scholten algorithm.

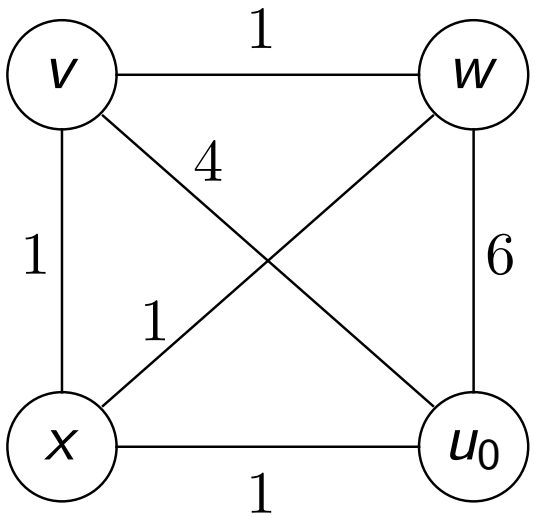
Question

Why is Rana's algorithm not a good choice for detecting termination ?

Answer: Nodes tend to become quiet, and start a wave, often.

Chandy-Misra algorithm - Example

$dist_{u_0}$	$\leftarrow$	0	$parent_{u_0}$	$\leftarrow$	$\perp$
$dist_w$	$\leftarrow$	6	$parent_w$	$\leftarrow$	u_0
$dist_v$	$\leftarrow$	7	$parent_v$	$\leftarrow$	w
$dist_x$	$\leftarrow$	8	$parent_x$	$\leftarrow$	v
$dist_x$	$\leftarrow$	7	$parent_x$	$\leftarrow$	w
$dist_v$	$\leftarrow$	4	$parent_v$	$\leftarrow$	u_0
$dist_w$	$\leftarrow$	5	$parent_w$	$\leftarrow$	v
$dist_x$	$\leftarrow$	6	$parent_x$	$\leftarrow$	w
$dist_x$	$\leftarrow$	5	$parent_x$	$\leftarrow$	v
$dist_x$	$\leftarrow$	1	$parent_x$	$\leftarrow$	u_0
$dist_w$	$\leftarrow$	2	$parent_w$	$\leftarrow$	x
$dist_v$	$\leftarrow$	3	$parent_v$	$\leftarrow$	w
$dist_v$	$\leftarrow$	2	$parent_v$	$\leftarrow$	x



Chandy-Misra algorithm - Complexity

Worst-case message complexity: Exponential

Worst-case message complexity for minimum-hop: $O(N^2 \cdot E)$

For each root, the algorithm requires at most $O(N \cdot E)$ messages.

Merlin-Segall algorithm

A *centralized* algorithm to compute all shortest paths to initiator u_0.

Initially, $dist_{u_0}(u_0) = 0$, $dist_v(u_0) = \infty$ if $v \neq u_0$,
and the $parent_v(u_0)$ values form a sink tree with root u_0.

Each round, u_0 sends $\langle 0 \rangle$ to its neighbors.

1. Let node v get $\langle d \rangle$ from neighbor w.

   If $d + \omega_{vw} < dist_v(u_0)$, then $dist_v(u_0) \leftarrow d + \omega_{vw}$
   (and v stores w as future value for $parent_v(u_0)$).

   If $w = parent_v(u_0)$, then v sends $\langle dist_v(u_0) \rangle$ to its neighbors
   except $parent_v(u_0)$.

2. When a $v \neq u_0$ has received a message from all neighbors,
   it sends $\langle dist_v(u_0) \rangle$ to $parent_v(u_0)$, and updates $parent_v(u_0)$.

u_0 starts a new round after receiving a message from all neighbors.

Merlin-Segall algorithm - Termination + complexity

After i rounds, all shortest paths of $\leq i$ hops have been computed.

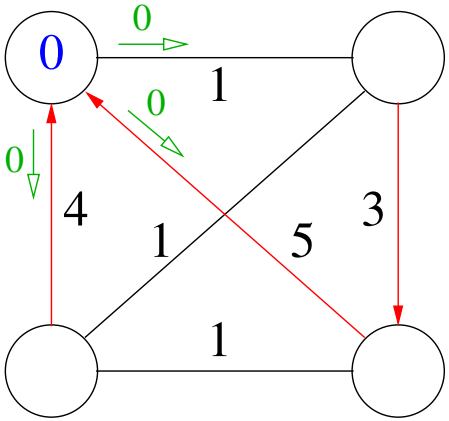
So the algorithm can terminate after $N - 1$ rounds.

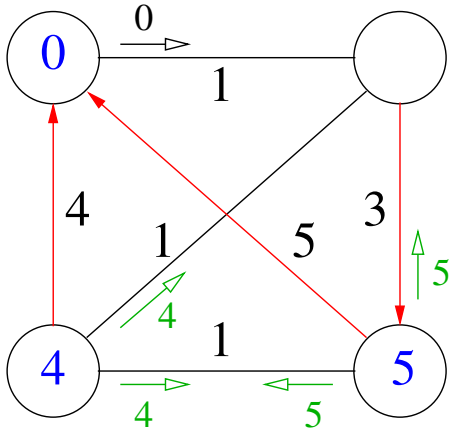
Message complexity: $\Theta(N^2 \cdot E)$

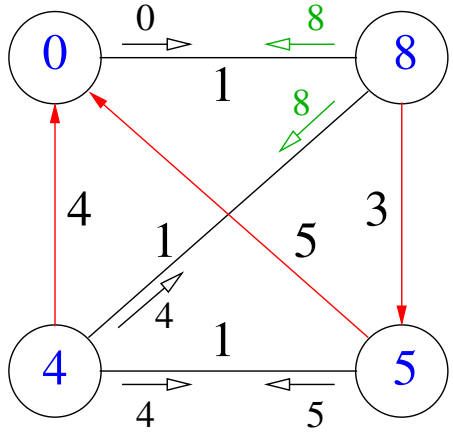
For each root, the algorithm requires $\Theta(N \cdot E)$ messages.

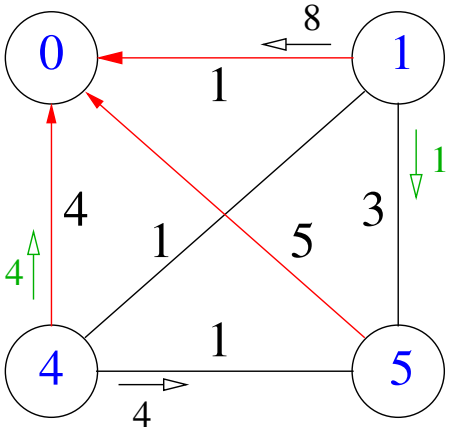
No separate termination detection is needed.

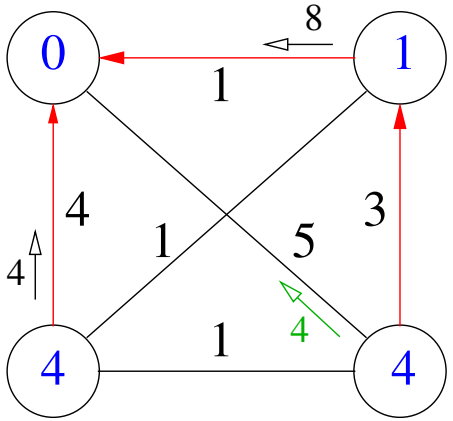
Merlin-Segall algorithm - Example (round 1)



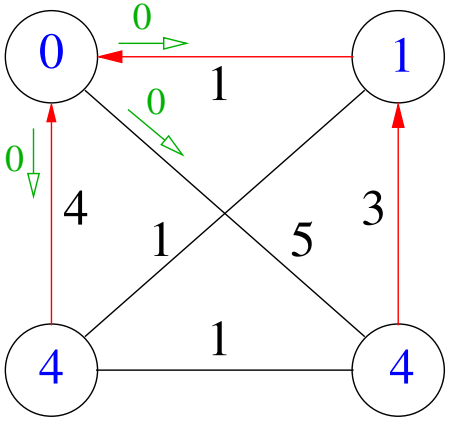


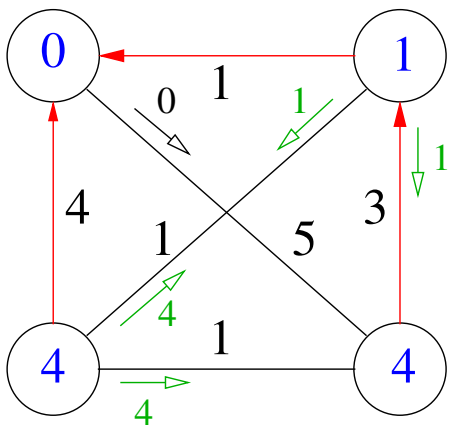


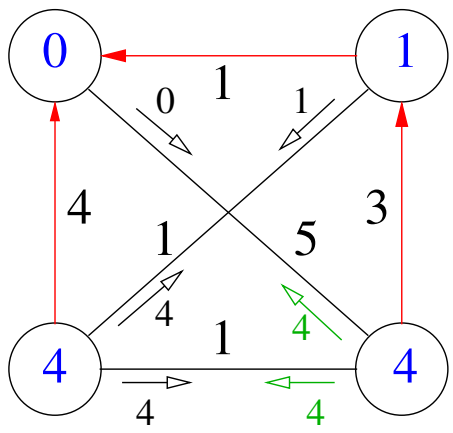


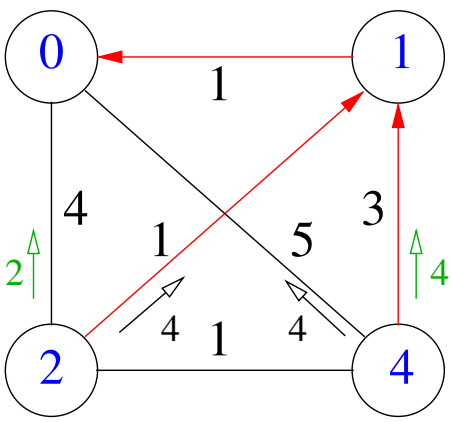


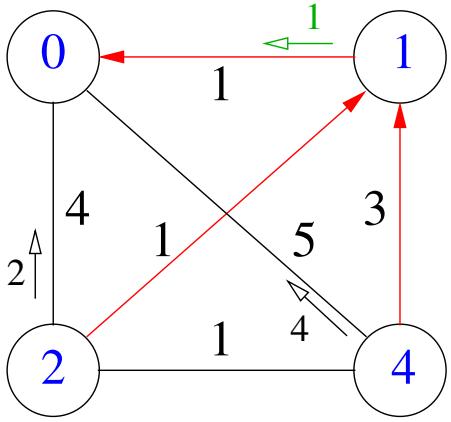
Merlin-Segall algorithm - Example (round 2)



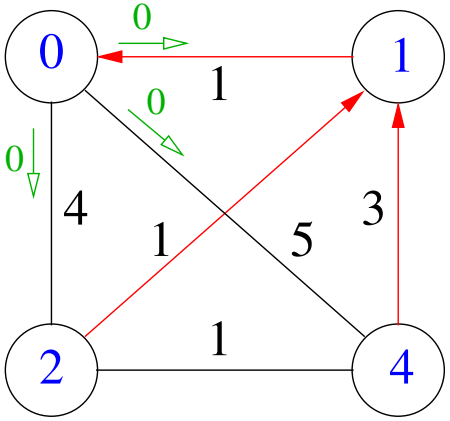


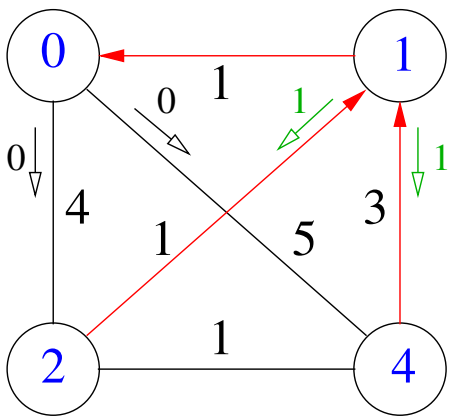


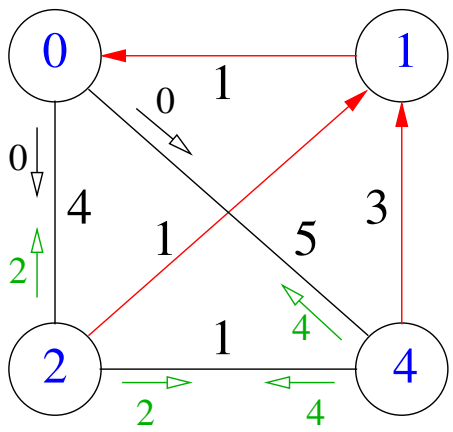


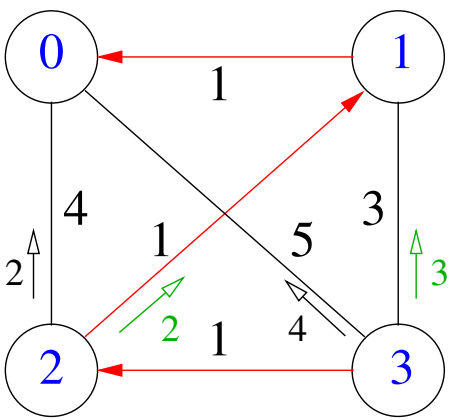


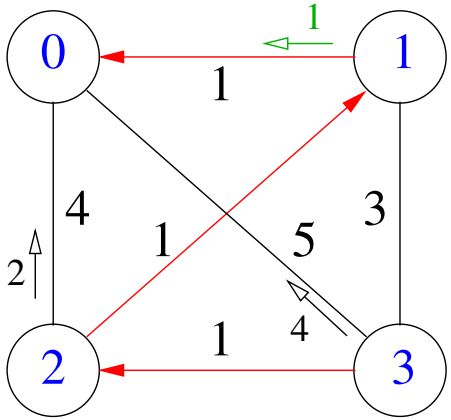
Merlin-Segall algorithm - Example (round 3)

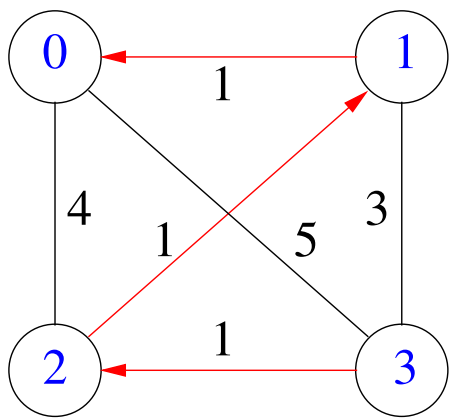












Merlin-Segall algorithm - Topology changes

A number is attached to distance messages.

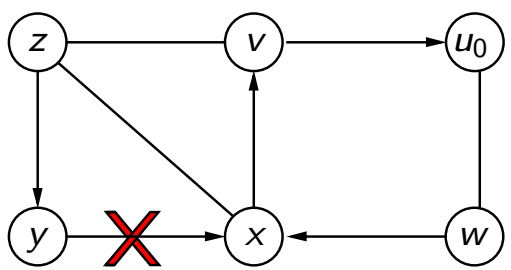
When an edge fails or becomes operational, adjacent nodes send a message to u_0 via the sink tree.

(If the message meets a failed tree link, it is discarded.)

When u_0 receives such a message, it starts a new set of $N - 1$ rounds, *with a higher number*.

If the failed edge is part of the sink tree, the sink tree is rebuilt.

Example:



x informs u_0 (via v) that an edge of the sink tree has failed.

Toueg's algorithm

Computes for each pair u, v a shortest path from u to v.

$d^S(u, v)$, with S a set of nodes, denotes the length of a shortest path from u to v with all *intermediate* nodes in S.

$$
\begin{aligned}
d^S(u,u) &= 0 \\
d^\emptyset(u,v) &= \omega_{uv} \quad \text{if } u \neq v \text{ and } uv \in E \\
d^\emptyset(u,v) &= \infty \quad \text{if } u \neq v \text{ and } uv \notin E \\
d^{S\cup\{w\}}(u,v) &= \min\{\, d^S(u,v),\ d^S(u,w) + d^S(w,v) \,\} \quad \text{if } w \notin S
\end{aligned}
$$

When S contains all nodes, d^S is the standard distance function.

Floyd-Warshall algorithm

We first discuss a *uniprocessor* algorithm.

Initially, $S = \emptyset$; the first three equations define $d^{\emptyset}$.

While S doesn't contain all nodes, a pivot $w \notin S$ is selected:

- $d^{S \cup \{w\}}$ is computed from d^S using the fourth equation.
- w is added to S.

When S contains all nodes, d^S is the standard distance function.

Time complexity: $\Theta(N^3)$

Question

Which complications arise when the Floyd-Warshall algorithm is turned into a distributed algorithm ?

- All nodes must pick the pivots in the same order.
- Each round, nodes need the distance values of the pivot to compute their own routing table.

Toueg's algorithm

Assumption: Each node knows the id's of all nodes.

(Because pivots must be picked uniformly at all nodes.)

Initially, at each node u:

- $S_u = \emptyset$;
- $dist_u(u) = 0$ and $parent_u(u) = \perp$;
- for each $v \neq u$, either
  $dist_u(v) = \omega_{uv}$ and $parent_u(v) = v$ if there is an edge uv, or
  $dist_u(v) = \infty$ and $parent_u(v) = \perp$ otherwise.

Toueg's algorithm

At the w-pivot round, w broadcasts its values $dist_w(v)$, for all nodes v.

If $parent_u(w) = \perp$ for a node $u \neq w$ at the w-pivot round,
then $dist_u(w) = \infty$, so $dist_u(w) + dist_w(v) \geq dist_u(v)$ for all nodes v.

Hence the sink tree toward w can be used to broadcast $dist_w$.

If u is in the sink tree toward w, it sends $\langle \textbf{request}, w \rangle$ to $parent_u(w)$,
to let it pass on $dist_w$.

If u isn't in the sink tree toward w, it proceeds to the next pivot round,
with $S_u \leftarrow S_u \cup \{w\}$.

Toueg's algorithm

Consider any node u in the sink tree toward w.

If $u \neq w$, it waits for the values $dist_w(v)$ from $parent_u(w)$.

u forwards the values $dist_w(v)$ to neighbors that send $\langle \textbf{request}, w \rangle$ to u.

If $u \neq w$, it checks for each node v whether

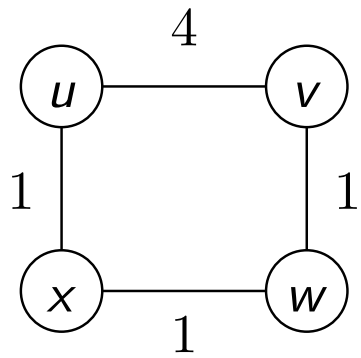
$$dist_u(w) + dist_w(v) < dist_u(v).$$

If so, $dist_u(v) \leftarrow dist_u(w) + dist_w(u)$ and $parent_u(v) \leftarrow parent_u(w)$.

Finally, u proceeds to the next pivot round, with $S_u \leftarrow S_u \cup \{w\}$.

Toueg's algorithm - Example

pivot u	$dist_x(v)$	$\leftarrow$	5	$dist_v(x)$	$\leftarrow$	5
	$parent_x(v)$	$\leftarrow$	u	$parent_v(x)$	$\leftarrow$	u
pivot v	$dist_u(w)$	$\leftarrow$	5	$dist_w(u)$	$\leftarrow$	5
	$parent_u(w)$	$\leftarrow$	v	$parent_w(u)$	$\leftarrow$	v
pivot w	$dist_x(v)$	$\leftarrow$	2	$dist_v(x)$	$\leftarrow$	2
	$parent_x(v)$	$\leftarrow$	w	$parent_v(x)$	$\leftarrow$	w
pivot x	$dist_u(w)$	$\leftarrow$	2	$dist_w(u)$	$\leftarrow$	2
	$parent_u(w)$	$\leftarrow$	x	$parent_w(u)$	$\leftarrow$	x
	$dist_u(v)$	$\leftarrow$	3	$dist_v(u)$	$\leftarrow$	3
	$parent_u(v)$	$\leftarrow$	x	$parent_v(u)$	$\leftarrow$	w



Toueg's algorithm - Complexity + drawbacks

Message complexity: $O(N^2)$

There are N pivot rounds, each taking at most $O(N)$ messages.

Drawbacks:

- Uniform selection of pivots requires that all nodes know the nodes in the network in advance.
- Global broadcast of $dist_w$ at the w-pivot round causes a high bit complexity.
- Not robust with respect to topology changes.

Question

Which addition needs to be made to the algorithm to allow that a node u can discard the routing table of the pivot w at some point ?

Answer: Next to $\langle \textbf{request}, w \rangle$, u informs all other neighbors that they do not need to forward w's routing table to u.

Then the message complexity increases to $O(N \cdot E)$.

Toueg's algorithm - Optimization

Let $parent_u(w) = x$ with $x \neq w$ at the start of the w-pivot round.

If $dist_x(v)$ doesn't change in this round, then neither does $dist_u(v)$ (for any v).

Upon reception of $dist_w$, x first updates $dist_x$, and only forwards values $dist_w(v)$ for which $dist_x(v)$ has changed.

Distance-vector routing

Consider a network in which nodes or links may fail or are added.

Such a change is eventually detected by its neighbors.

In distance-vector routing, at a change in the local topology or routing table, a node sends its entire *routing table* to its neighbors.

Each node locally computes shortest paths
(e.g. with the Bellman-Ford algorithm, if links can have negative weights).

Link-state routing

In link-state routing, nodes periodically sends a link-state packet, with

- the node's edges and their weights (based on latency, bandwidth)
- a sequence number (which increases with each broadcast)

Link-state packets are flooded through the network.

Nodes store link-state packets, to obtain a view of the entire network.

Sequence numbers avoid that new info is overwritten by old info.

Each node locally computes shortest paths (e.g. with Dijkstra's alg.).

Question

Flooding entire routing tables (instead of only edges and weights) tends to produce a less efficient algorithm.

Why is that ?

Answer: A routing table may be based on remote edges that have recently crashed.

And, of course, the bit complexity increases dramatically.

Link-state routing - Time-to-live

When a node recovers from a crash, its sequence number starts at 0.

So its link-state packets may be ignored for a long time.

Therefore link-state packets carry a time-to-live (TTL) field.

After this time the information from the packet may be discarded.

To reduce flooding, each time a link-state packet is forwarded, its TTL field decreases.

When it becomes 0, the packet is discarded.

Autonomous systems

The OSPF protocol for routing on the Internet uses *link-state routing*.

The RIP protocol employs *distance-vector routing*.

Link-state / distance-vector routing doesn't scale to the Internet, because it uses flooding / sends entire routing tables.

Therefore the Internet is divided into autonomous systems.

Each autonomous system uses the OSPF or RIP protocol.

Border gateway protocol

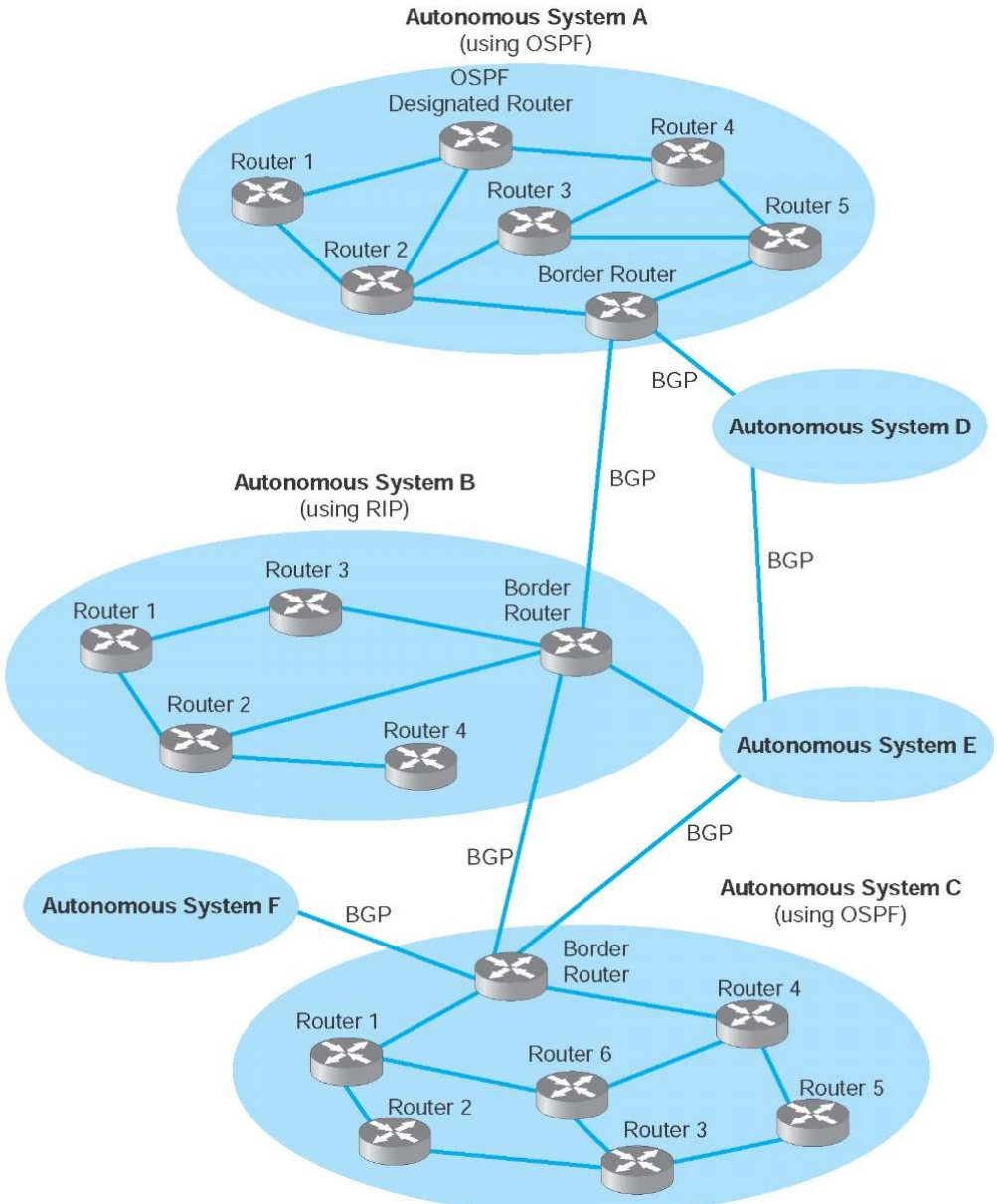
The Border Gateway Protocol routes between autonomous systems.

Routers broadcast updates of their routing tables (a la Chandy-Misra).

A router may update its routing table

- because it detects a topology change, or
- because of an update in the routing table of a neighbor.

Routing on the Internet



Lecture in a nutshell

routing tables to guide a packet to its destination

Chandy-Misra algorithm has *exponential* worst-case message complexity but only $O(N^2 \cdot E)$ for minimum-hop paths

Merlin-Segall algorithm has message complexity $\Theta(N^2 \cdot E)$

Toueg's algorithm has message complexity $O(N^2)$
(but has a high bit complexity, and requires uniform selection of pivots)

link-state / distance-vector routing and the border gateway protocol employ classical routing algorithms on the Internet

Breadth-first search

Consider an undirected, unweighted network.

A breadth-first search tree is a sink tree in which each tree path to the root is minimum-hop.

The Chandy-Misra algorithm for minimum-hop paths computed a breadth-first search tree using $O(N \cdot E)$ messages (for each root).

The following centralized algorithm requires $O(N \cdot \sqrt{E})$ messages (for each root).

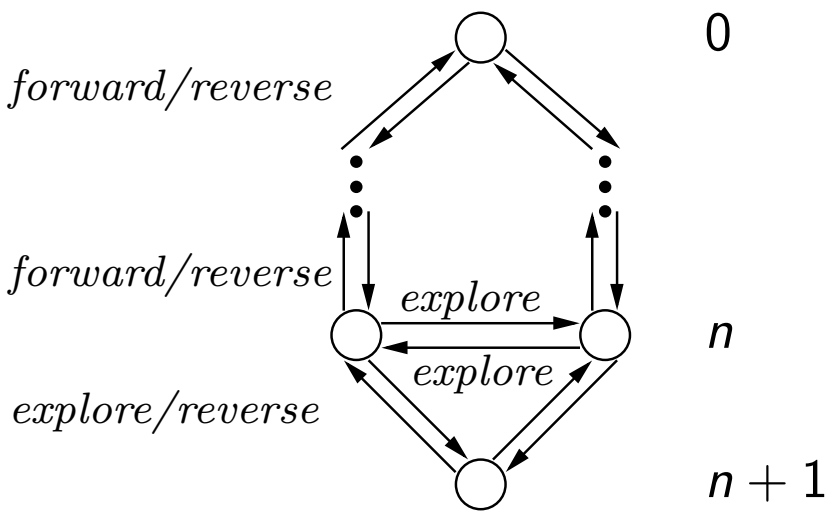
Breadth-first search - A "simple" algorithm

Initially (after round 0), the initiator is at *distance* 0, noninitiators are at *distance* ∞, and *parents* are undefined.

After round $n \geq 0$, the tree has been constructed up to depth n.

Nodes at distance n know which neighbors are at distance $n-1$.

In round $n+1$:



Breadth-first search - A "simple" algorithm

- Messages $\langle$**forward**, $n\rangle$ travel down the tree, from the initiator to nodes at distance n.
- When a node at distance n gets $\langle$**forward**, $n\rangle$, it sends $\langle$**explore**, $n+1\rangle$ to neighbors that aren't at distance $n-1$.

Let a node v receive $\langle$**explore**, $n+1\rangle$.

- If $dist_v = \infty$, then $dist_v \leftarrow n+1$, the sender becomes v's parent, and v sends back $\langle$**reverse**, *true*$\rangle$.
- If $dist_v = n+1$, then v stores that the sender is at distance n, and v sends back $\langle$**reverse**, *false*$\rangle$.
- If $dist_v = n$, then this is a negative ack for the $\langle$**explore**, $n+1\rangle$ that v sends into this edge.

Breadth-first search - A "simple" algorithm

- A *noninitiator* at distance n (or $< n$) waits until all messages $\langle$**explore**, $n+1\rangle$ (resp. $\langle$**forward**, $n\rangle$) have been answered.

  Then it sends $\langle$**reverse**, $b\rangle$ to its parent, where $b = \mathit{true}$ if and only if new nodes were added to its subtree.

- The *initiator* waits until all messages $\langle$**forward**, $n\rangle$ (or, in round 1, $\langle$**explore**, $1\rangle$) have been answered.

  If no new nodes were added in round $n+1$, it terminates.

  Else, it continues with round $n+2$.

In round $n+2$, nodes only send a **forward** to children that reported newly discovered nodes in round $n+1$.

Breadth-first search - Complexity

Worst-case message complexity: $O(N^2 + E) = O(N^2)$

There are at most N rounds.

Each round, tree edges carry at most 1 **forward** and 1 replying **reverse**.

In total, edges carry 1 **explore** and 1 replying **reverse** or **explore**.

Worst-case time complexity: $O(N^2)$

Round n is completed in at most $2n$ time units, for $n = 1, \ldots, N$.

Frederickson's algorithm

Computes ℓ levels per round, with $1 \leq \ell < N$.

Initially, the initiator is at distance 0, noninitiators are at distance ∞, and parents are undefined.

After round n, the tree has been constructed up to depth ℓn.

In round $n+1$:

- $\langle$**forward**$, \ell n\rangle$ travels down the tree, from the initiator to nodes at distance ℓn.
- When a node at distance ℓn gets $\langle$**forward**$, \ell n\rangle$, it sends $\langle$**explore**$, \ell n + 1\rangle$ to neighbors that aren't at distance $\ell n - 1$.

Frederickson's algorithm - Complications

Complication 1: In round $n+1$, a node at a distance $> \ell n$ may send multiple **explore**'s into an edge.

How can this happen ?

Which (small) complication may arise as a result of this ?

How can this be resolved ?

Solution: **reverse**'s in reply to **explore**'s are supplied with a distance.

Complication 2: A node w may receive a **forward** from a non-parent v.

How can this happen ?

Solution: w can dismiss this **forward**.

In the previous round, w informed v that it is no longer a child.

Frederickson's algorithm

Let a node v receive $\langle \textbf{explore}, k \rangle$. We distinguish two cases.

- $k < dist_v$:

  $dist_v \leftarrow k$, and the sender becomes v's parent.

  If ℓ doesn't divide k, then v sends $\langle \textbf{explore}, k+1 \rangle$ to its other neighbors.

  If ℓ divides k, then v sends back $\langle \textbf{reverse}, k, \mathit{true} \rangle$.

- $k \geq dist_v$:

  If $k = dist_v$ and ℓ divides k, then v sends back $\langle \textbf{reverse}, k, \mathit{false} \rangle$.

  Else v doesn't send a reply (because it sent $\langle \textbf{explore}, dist_v + 1 \rangle$ into this edge).

Frederickson's algorithm

- Nodes at a distance $\ell n < k < \ell(n+1)$ wait until a message $\langle \textbf{reverse}, k+1, \_ \rangle$ or $\langle \textbf{explore}, j \rangle$ with $j \in \{k,\ k+1,\ k+2\}$ has been received from all neighbors.

  Then they send $\langle \textbf{reverse}, k, \mathit{true} \rangle$ to their parent.

- *Noninitiators* at a distance ℓn (or $< \ell n$) wait until all messages $\langle \textbf{explore}, \ell n + 1 \rangle$ (resp. $\langle \textbf{forward}, \ell n \rangle$) have been answered with a **reverse** or **explore** (resp. **reverse**).

  Then they send $\langle \textbf{reverse}, b \rangle$ to their parent, where $b = \mathit{true}$ if and only if they received $\langle \textbf{reverse}, \_, \mathit{true} \rangle$ from a child.

Frederickson's algorithm

- The *initiator* waits until all messages $\langle \textbf{forward}, \ell n \rangle$ (or, in round 1, $\langle \textbf{explore}, 1 \rangle$) have been answered.

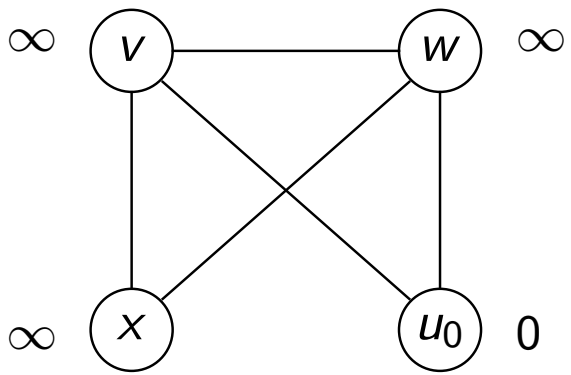
  If it is certain that no unexplored nodes remain, it terminates.

  Else, it continues with round $n + 2$.

In round $n + 2$, nodes only send a **forward** to children that reported newly discovered nodes in round $n + 1$.

Question

Apply Frederickson's algorithm to the network below, with initiator u_0 and $\ell = 2$:



Give a computation in which:

- w becomes the parent of x, and
- in round 2, v sends a spurious **forward** to w.

Frederickson's algorithm - Complexity

Worst-case message complexity: $O(\frac{N^2}{\ell} + \ell \cdot E)$

There are at most $\lceil \frac{N-1}{\ell} \rceil + 1$ rounds.

Each round, tree edges carry at most 1 **forward** and 1 replying **reverse**.

In total, edges carry at most 2ℓ **explore**'s and 2ℓ replying **reverse**'s.

(In total, frond edges carry at most 1 spurious **forward**.)

Worst-case time complexity: $O(\frac{N^2}{\ell})$

Round n is completed in at most $2\ell n$ time units, for $n = 1, \ldots, \lceil \frac{N-1}{\ell} \rceil + 1$.

If $\ell = \lceil \frac{N}{\sqrt{E}} \rceil$, both message and time complexity are $O(N \cdot \sqrt{E})$.

Questions

What is the optimal value of ℓ in case the network is:

- a complete graph
- acyclic

Question

Even with cycle-free routes, we can still get a deadlock. Why ?

Hint: Consider the (bounded) memory at the processes.

Store-and-forward deadlocks

A store-and-forward deadlock occurs when a group of packets are all waiting for the use of a buffer slot occupied by a packet in the group.

A controller avoids such deadlocks.

It prescribes whether a packet can be generated or forwarded, and in which buffer slot it is put next.

Deadlock-free packet switching

Consider an undirected network, supplied with routing tables.

Processes store data packets traveling to their destination in buffers.

Possible events:

- *Generation*: A new packet is placed in an empty buffer slot.
- *Forwarding*: A packet is forwarded to an empty buffer slot of the next node on its route.
- *Consumption*: A packet at its destination node is removed from the buffer.

At a node with an empty buffer, packet generation must be allowed.

Synchronous versus asynchronous networks

For simplicity we assume synchronous communication.

In an asynchronous setting, a node can only eliminate a packet when it is sure that the packet will be accepted at the next node.

Question: How can this be achieved in an undirected network ?

Answer: A packet can only be eliminated by the sender when its reception has been acknowledged.

Destination controller

The network consists of nodes $u_0, \ldots, u_{N-1}$.

T_i denotes the sink tree (with respect to the routing tables) with root u_i for $i = 0, \ldots, N-1$.

In the destination controller, each node carries N buffer slots.

- When a packet with destination u_i is generated at v, it is placed in the i^{th} buffer slot of v.
- If vw is an edge in T_i, then the i^{th} buffer slot of v is linked to the i^{th} buffer slot of w.

Destination controller - Correctness

Theorem: The destination controller is deadlock-free.

Proof: Consider a reachable configuration γ.

Make forwarding and consumption transitions to a configuration δ where no forwarding or consumption is possible.

For each i, since T_i is acyclic, packets in an i^{th} buffer slot can travel to their destination, where they are consumed.

So in δ, all buffers are empty.

Hops-so-far controller

The network consists of nodes $u_0, \ldots, u_{N-1}$.

T_i is the sink tree (with regard to the routing tables) with root u_i for $0 = 1, \ldots, N-1$.

K is the length of a longest path in any T_i.

In the hops-so-far controller, each node carries $K+1$ buffer slots, numbered from 0 to K.

- A generated packet is placed in the 0^{th} buffer slot.
- For each edge vw and any $j < K$, the j^{th} buffer slot of v is linked to the $(j+1)^{\text{th}}$ buffer slot of w, and vice versa.

Hops-so-far controller - Correctness

Theorem: The hops-so-far controller is deadlock-free.

Proof: Consider a reachable configuration γ.

Make forwarding and consumption transitions to a configuration δ where no forwarding or consumption is possible.

Packets in a K^{th} buffer slot are at their destination.

So in δ, K^{th} buffer slots are all empty.

Suppose all i^{th} buffer slots are empty in δ, for some $1 \leq i \leq K$.

Then all $(i-1)^{\text{th}}$ buffer slots must also be empty in δ.

For else some packet in an $(i-1)^{\text{th}}$ buffer slot could be forwarded or consumed.

Concluding, in δ all buffers are empty.

Acyclic orientation cover

Consider an undirected network.

An acyclic orientation is a directed, acyclic network obtained by directing all edges.

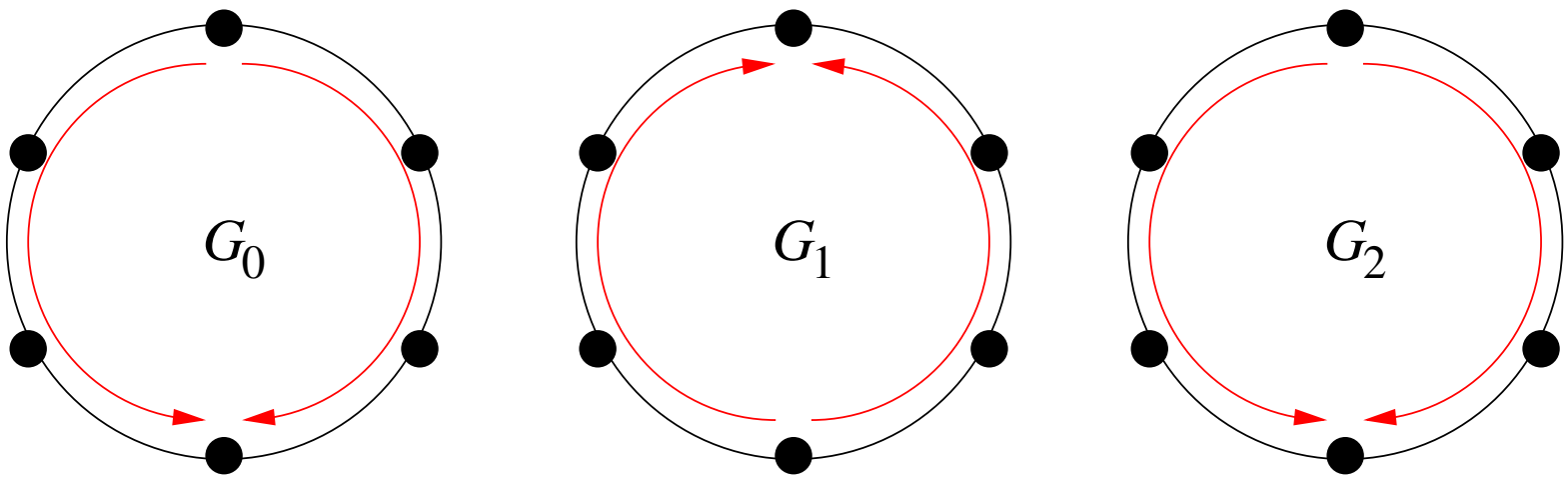
Let $\mathcal{P}$ be a set of paths in the (undirected) network.

An acyclic orientation cover of $\mathcal{P}$ consists of acyclic orientations $G_0, \ldots, G_{n-1}$ such that each path in $\mathcal{P}$ is the concatenation of paths $P_0, \ldots, P_{n-1}$ in $G_0, \ldots, G_{n-1}$.

Acyclic orientation cover - Example

For each undirected ring there exists a cover, consisting of three acyclic orientations, of the collection of minimum-hop paths.

For instance, in case of a ring of size six:



Acyclic orientation cover controller

Let $\mathcal{P}$ be the set of paths in the network induced by the sink trees.

Let $G_0, \ldots, G_{n-1}$ be an acyclic orientation cover of $\mathcal{P}$.

In the acyclic orientation cover controller, nodes have n buffer slots, numbered from 0 to $n-1$.

- A generated packet is placed in the 0^{th} buffer slot.
- Let vw be an edge in G_i.

  The i^{th} buffer slot of v is linked to the i^{th} buffer slot of w.

  Moreover, if $i < n-1$, then the i^{th} buffer slot of w is linked to the $(i+1)^{\text{th}}$ buffer slot of v.

Acyclic orientation cover controller - Intuition

Consider a packet; it is routed via the sink tree of its destination.

Its path is a concatenation of paths $P_0, \ldots, P_{n-1}$ in $G_0, \ldots, G_{n-1}$.

While the packet is in an i^{th} slot with $i < n-1$, it can be forwarded.

If the packet ends up in the $(n-1)^{\text{th}}$ buffer slot at a node, then it is being routed via the last part P_{n-1} of the path.

In that case the packet can be routed to its destination via $(n-1)^{\text{th}}$ buffer slots.

Acyclic orientation cover controller - Example

For each undirected ring there exists a deadlock-free controller that:

- uses three buffer slots per node, and
- allows packets to travel via minimum-hop paths.

Acyclic orientation cover controller - Correctness

Theorem: Let all packets be routed via paths in $\mathcal{P}$.

Then the acyclic orientation cover controller is deadlock-free.

Proof: Consider a reachable configuration γ.

Make forwarding and consumption transitions to a configuration δ where no forwarding or consumption is possible.

Since G_{n-1} is acyclic, packets in an $(n-1)^{\text{th}}$ buffer slot can travel to their destination. So in δ, all $(n-1)^{\text{th}}$ buffer slots are empty.

Suppose all i^{th} buffer slots are empty in δ, for some $i = 1, \ldots, n-1$.

Then all $(i-1)^{\text{th}}$ buffer slots must also be empty in δ.

For else, since G_{i-1} is acyclic, some packet in an $(i-1)^{\text{th}}$ buffer slot could be forwarded or consumed.

Concluding, in δ all buffers are empty.

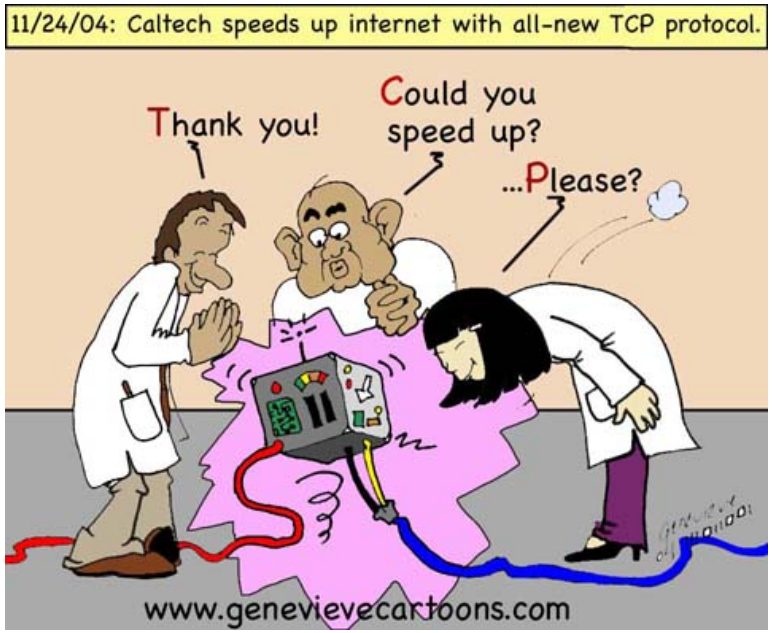
Question

Consider an acyclic orientation cover for the minimum-hop paths in a ring of four nodes.

Show how the resulting acyclic orientation cover controller links buffer slots.

Slow-start algorithm in TCP

Back to the asynchronous, pragmatic world of the Internet.



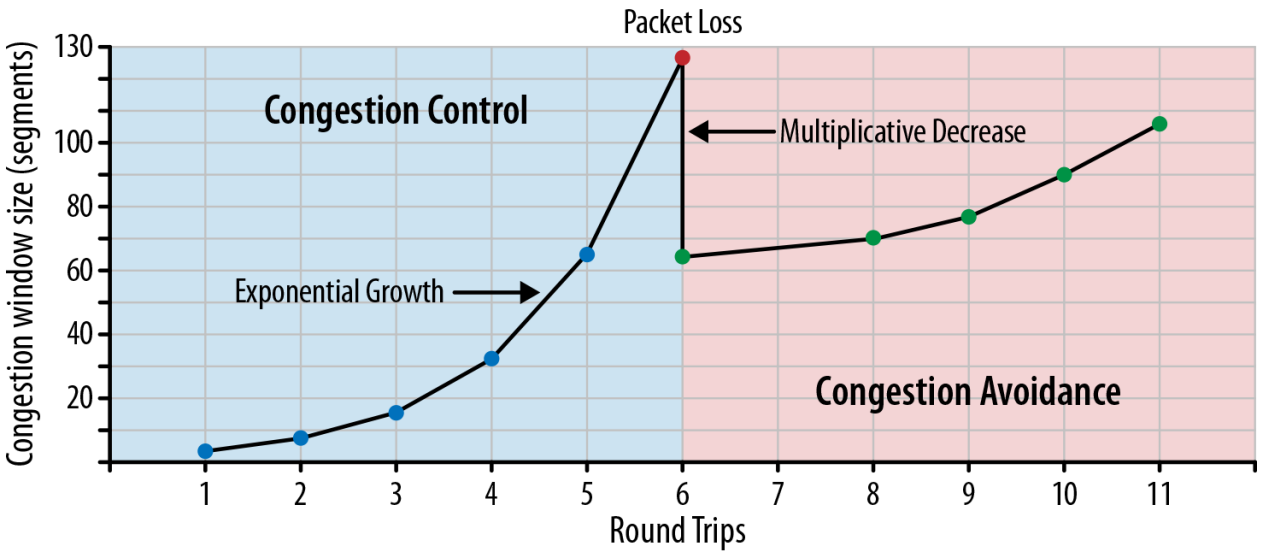
To avoid congestion, in TCP, nodes maintain a congestion window for each of their edges.

It is the maximum number of *unacknowledged* packets on this edge.

Congestion window

The congestion window grows *linearly* with each received ack, up to some threshold.

Question: Explain why the congestion window may *double* with every "round trip time".



The congestion window is *reset to the initial size* (in TCP Tahoe) or *halved* (in TCP Reno) with each lost data packet.

Take-home messages of the current lecture

Frederickon's algorithm to compute a breadth-first search tree

iterative deepening (a la Frederickson's alg.)

optimization of a parameter (ℓ) based on a complexity analysis

importance of deadlock-free packet switching

acyclic orientation cover controller

congestion window in TCP

Election algorithms

Often a leader process is needed to coordinate a distributed task.

In an *election algorithm*, each computation should terminate in a configuration where *one* process is the leader.

Assumptions:

- All processes have the *same local algorithm*.
- The algorithm is *decentralized*:

  The initiators can be any non-empty set of processes.
- Process id's are *unique*, and from a *totally ordered set*.

Chang-Roberts algorithm

Consider a directed ring.

Initially only *initiators* are active, and send a message with their id.

Let an *active* process p receive a message q:

- If $q < p$, then p *dismisses* the message.
- If $q > p$, then p becomes passive, and *passes on* the message.
- If $q = p$, then p becomes the leader.

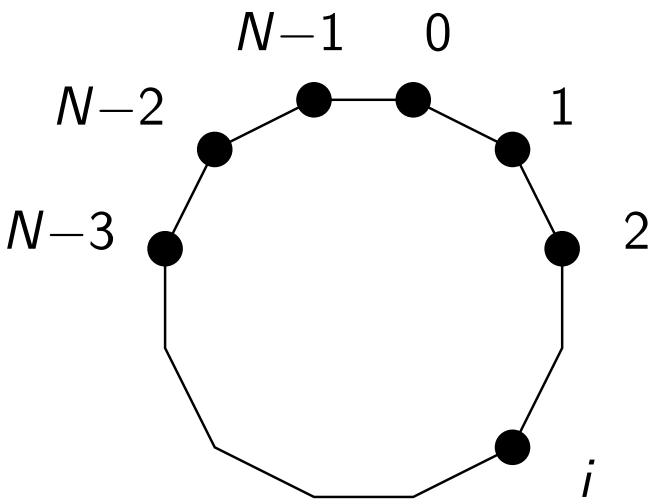
Passive processes (including all noninitiators) pass on messages.

Worst-case message complexity: $O(N^2)$

Average-case message complexity: $O(N \log N)$

Chang-Roberts algorithm - Example

All processes are initiators.



anti-clockwise: $\frac{N(N+1)}{2}$ messages

clockwise: $2N-1$ messages

Franklin's algorithm

Consider an undirected ring.

Each *active* process p repeatedly compares its own id with the id's of its nearest *active* neighbors on both sides.

If such a neighbor has a larger id, then p becomes *passive*.

Initially, initiators are active, and noninitiators are passive.

Each round, an active process p:

- sends its id to its neighbors on either side, and
- receives id's q and r:
  - if $\max\{q, r\} < p$, then p starts another round
  - if $\max\{q, r\} > p$, then p becomes passive
  - if $\max\{q, r\} = p$, then p becomes the leader

Passive processes pass on incoming messages.

Franklin's algorithm - Complexity

Worst-case message complexity: $O(N \log N)$

In each round, at least half of the active processes become passive.

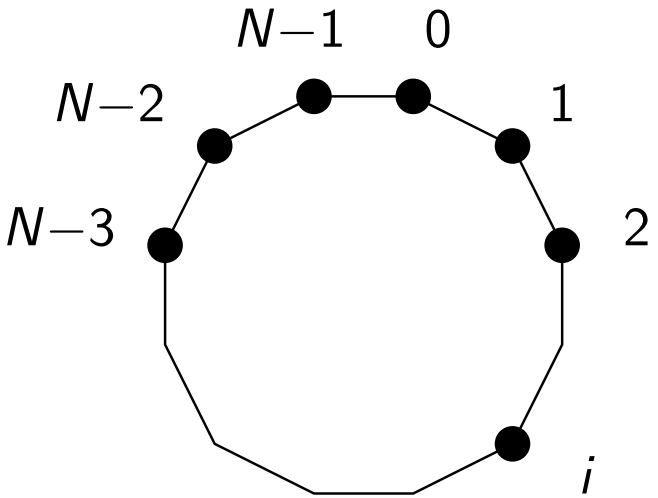
So there are at most $\lfloor \log_2 N \rfloor + 1$ rounds.

Each round takes $2N$ messages.

Question: Give an example with $N = 4$ that takes three rounds.

Question: Show that for any N there is a ring that takes two rounds.

Franklin's algorithm - Example



after 1 round only node $N-1$ is active

after 2 rounds node $N-1$ is the leader

Suppose this ring is directed with a clockwise orientation.

If a process would only compare its id with the one of its predecessor, then it would take N rounds to complete.

Dolev-Klawe-Rodeh algorithm

Consider a directed ring.

The comparison of id's of an active process p and
its nearest active neighbors q and r is performed at r.

$$\cdots \blacktriangleright s \cdots \blacktriangleright q \cdots \blacktriangleright p \cdots \blacktriangleright r \cdots \blacktriangleright t \cdots \blacktriangleright$$

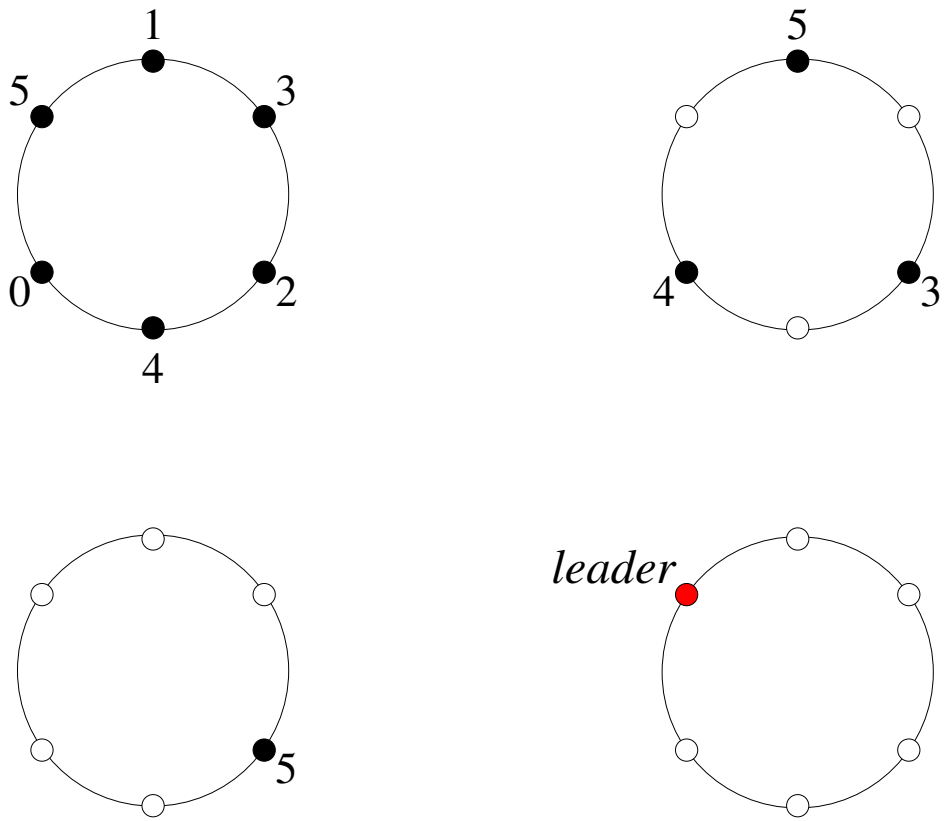
- If $\max\{q, r\} < p$, then r changes its id to p, and sends out p.
- If $\max\{q, r\} > p$, then r becomes passive.
- If $\max\{q, r\} = p$, then r announces this id to all processes.

  The process that originally had the id p becomes the leader.

Worst-case message complexity: $O(N \log N)$

Dolev-Klawe-Rodeh algorithm - Example

Consider the following clockwise oriented ring.



Tree election algorithm for acyclic networks

Question: How can the *tree algorithm* be used to make the process with the largest id in an undirected, acyclic network the leader ?

(Be careful that a leaf may be a noninitiator.)

Start with a wake-up phase, driven by the initiators.

- Initially, initiators send a wake-up message to all neighbors.
- When a noninitiator receives a first wake-up message, it sends a wake-up message to all neighbors.
- A process wakes up when it has received wake-up messages from all neighbors.

Tree election algorithm

The local algorithm at an awake process p:

- p waits until it has received id's from all neighbors except one, which becomes its parent.
- p computes the largest id $\max_p$ among the received id's and its own id.
- p sends a *parent request* to its parent, tagged with $\max_p$.
- If p receives a parent request from its parent, tagged with q, it computes $\max'_p$, being the maximum of $\max_p$ and q.
- Next p sends an *information message* to all neighbors except its parent, tagged with $\max'_p$.
- This information message is forwarded through the network.
- The process with id $\max'_p$ becomes the leader.

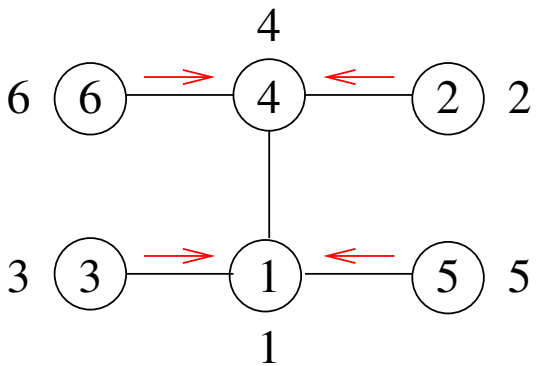
Message complexity: $2N - 2$ messages

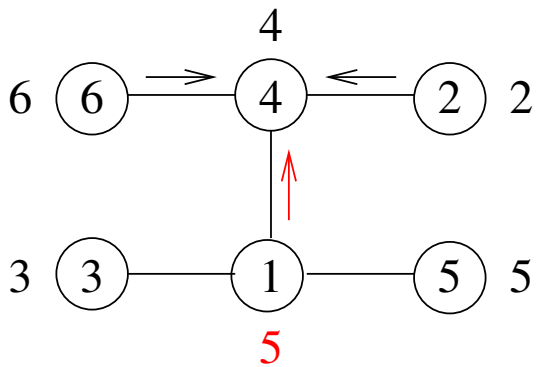
Question

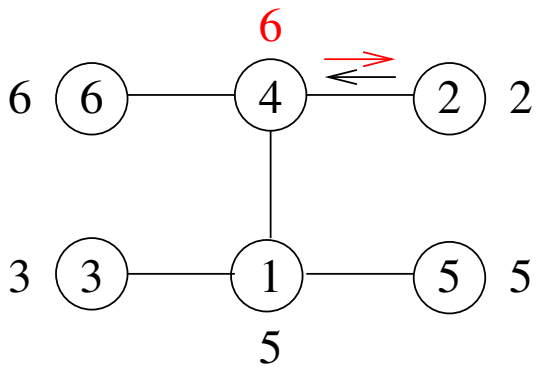
In case a process p receives a parent request from its parent, why does it need to recompute $\max_p$?

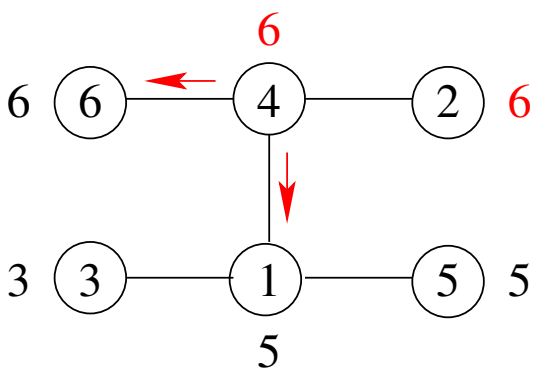
Tree election algorithm - Example

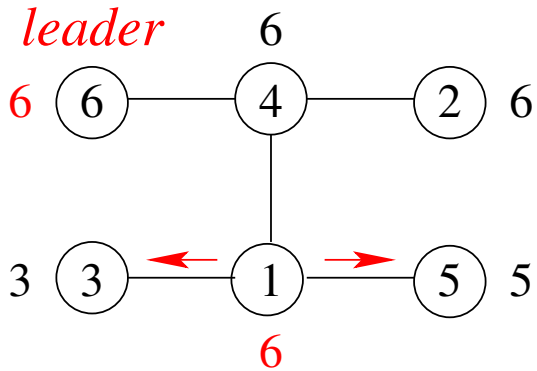
The wake-up phase is omitted.

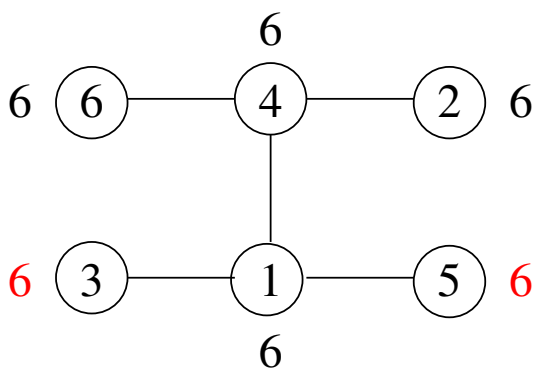












Echo algorithm with extinction

Each *initiator* starts a wave, tagged with its id.

Non-initiators join the first wave that hits them.

At any time, each process takes part in at most one wave.

Suppose a process p in wave q is hit by a wave r:

- if $q < r$, then p changes to wave r
  (it abandons all earlier messages);
- if $q > r$, then p continues with wave q
  (it dismisses the incoming message);
- if $q = r$, then the incoming message is treated according to the echo algorithm of wave q.

If wave p executes a decide event (at p), p becomes the leader.

Worst-case message complexity: $O(N \cdot E)$

Minimum spanning trees

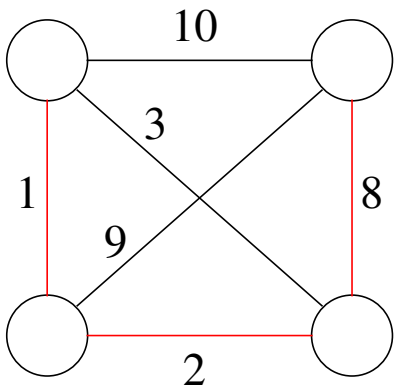
Consider an undirected, weighted network.

We assume that different edges have different weights.

(Or weighted edges can be totally ordered by also taking into account the id's of endpoints of an edge, and using a lexicographical order.)

In a minimum spanning tree, the sum of the weights of the edges in the spanning tree is minimal.

Example:



Fragments

Lemma: Let F be a fragment
(i.e., a connected subgraph of the minimum spanning tree M).

Let e be the *lowest-weight* outgoing edge of F
(i.e., e has exactly one endpoint in F).

Then e is in M.

Proof: Suppose not.

Then $M \cup \{e\}$ has a cycle,
containing e and another outgoing edge f of F.

Replacing f by e in M gives a spanning tree
with a smaller sum of weights of edges.

Kruskal's algorithm

A *uniprocessor* algorithm for computing minimum spanning trees.

- Initially, each node forms a separate fragment.
- In each step, a lowest-weight outgoing edge of a fragment is added to the spanning tree, joining two fragments.

This algorithm also works when edges have the same weight.

Then the minimum spanning tree may not be unique.

Gallager-Humblet-Spira algorithm

Consider an undirected, weighted network,
in which different edges have different weights.

Distributed computation of a minimum spanning tree:

- Initially, each process is a fragment.
- The processes in a fragment F together search for the lowest-weight outgoing edge e_F.
- When e_F has been found, the fragment at the other end is asked to collaborate in a merge.

Complications: Is an edge outgoing? Is it lowest-weight?

Level, name and core edge

Each fragment carries a (unique) name $fn : \mathbb{R}$ and a level $\ell : \mathbb{N}$.

Its level is the maximum number of joins any process in the fragment has experienced.

Neighboring fragments $F = (fn, \ell)$ and $F' = (fn', \ell')$ can be joined as follows:

$$\ell < \ell' \ \wedge \ F \xrightarrow{e_F} F' : \quad F \cup F' = (fn', \ell')$$

$$\ell = \ell' \ \wedge \ e_F = e_{F'} : \quad F \cup F' = (\text{weight } e_F, \ell + 1)$$

The core edge of a fragment is the last edge that connected two sub-fragments at the same level. Its end points are the core nodes.

Parameters of a process

Its *state*:

- sleep (for noninitiators)
- find (looking for a lowest-weight outgoing edge)
- found (reported a lowest-weight outgoing edge to the core edge)

The *status* of its *channels*:

- basic edge (undecided)
- branch edge (in the spanning tree)
- rejected (not in the spanning tree)

The *name* and *level* of its fragment.

Its *parent* (toward the core edge).

Initialization

Non-initiators wake up when they receive a (**connect** or **test**) message.

Each initiator, and noninitiator after it has woken up:

- sets its level to 0
- sets its *lowest-weight* edge to branch
- sends $\langle \textbf{connect}, 0 \rangle$ into this channel
- sets its other channels to basic
- sets its state to found

Joining two fragments

Let fragments $F = (fn, \ell)$ and $F' = (fn', \ell')$ be joined via channel pq.

- If $\ell < \ell'$, then p sent $\langle \textbf{connect}, \ell \rangle$ to q.

  q sends $\langle \textbf{initiate}, fn', \ell', \frac{find}{found} \rangle$ to p.

  $F \cup F'$ inherits the core edge of F'.

- If $\ell = \ell'$, then p and q sent $\langle \textbf{connect}, \ell \rangle$ to each other.

  They send $\langle \textbf{initiate}, \text{weight } pq, \ell + 1, find \rangle$ to each other.

  $F \cup F'$ gets core edge pq.

At reception of $\langle \textbf{initiate}, fn, \ell, \frac{find}{found} \rangle$, a process stores fn and ℓ, sets its state to *find* or *found*, and adopts the sender as its parent.

It passes on the message through its other branch edges.

Computing the lowest-weight outgoing edge

In case of $\langle$**initiate**, *fn*, ℓ, *find*$\rangle$, p checks in increasing order of weight if one of its *basic* edges pq is *outgoing*, by sending $\langle$**test**, *fn*, $\ell\rangle$ to q.

While $\ell > \mathit{level}_q$, q *postpones* processing the incoming **test** message.

Let $\ell \leq \mathit{level}_q$.

- If q is in fragment *fn*, then q replies **reject**.

  In this case p and q will set pq to rejected.

- Else, q replies **accept**.

When a basic edge is accepted, or there are no basic edges left, p *stops* the search.

Questions

Why does q postpone processing the incoming $\langle \textbf{test}, \_, \ell \rangle$ message from p while $\ell > \mathit{level}_q$?

Answer: p and q might be in the same fragment, in which case $\langle \textbf{initiate}, \mathit{fn}, \ell, \mathit{find} \rangle$ is on its way to q.

Why does this postponement not lead to a deadlock ?

Answer: There is always a fragment with a smallest level.

Reporting to the core nodes

- p waits for all branch edges, *except its parent*, to report.
- p sets its state to *found*.
- p computes the minimum λ of (1) these reports, and (2) the weight of its lowest-weight outgoing basic edge (or ∞, if no such channel was found).
- If $\lambda < \infty$, p stores either the branch edge that sent λ, or its basic edge of weight λ.
- p sends $\langle \textbf{report}, \lambda \rangle$ to its parent.

Termination or **changeroot** at the core nodes

A core node receives reports through *all* its branch edges, including the core edge.

- If the minimum reported value $\mu = \infty$, the core nodes terminate.
- If $\mu < \infty$, the core node that received μ first sends **changeroot** toward the lowest-weight outgoing basic edge.

  (The core edge becomes a regular tree edge.)

Ultimately **changeroot** reaches the process p that reported the lowest-weight outgoing basic edge.

p sets this channel to branch, and sends $\langle \textbf{connect}, \mathit{level}_p \rangle$ into it.

Starting the join of two fragments

When q receives $\langle \textbf{connect}, level_p \rangle$ from p, $level_q \geq level_p$.

Namely, either $level_p = 0$, or q earlier sent **accept** to p.

- If $level_q > level_p$, then q sets qp to *branch* and sends $\langle \textbf{initiate}, name_q, level_q, \frac{find}{found} \rangle$ to p.
- As long as $level_q = level_p$ and qp isn't a branch edge, q *postpones* processing the **connect** message.
- If $level_q = level_p$ and qp is a branch edge (meaning that q sent $\langle \textbf{connect}, level_q \rangle$ to p), then q sends $\langle \textbf{initiate}, \text{weight } qp, level_q + 1, find \rangle$ to p (and vice versa).

  In this case pq becomes the core edge.

Questions

If $level_q = level_p$ and qp isn't a branch edge, why does q postpone processing the incoming **connect** message from p?

Answer: The fragment of q might be in the process of joining another fragment at a level $\geq level_q$.

Then the fragment of p should subsume the name and level of that joint fragment, instead of joining q's fragment at an equal level.

Why does this postponement not give rise to a deadlock?

(I.e., why can't there be a cycle of fragments waiting for a reply to a postponed **connect** message?)

Answer: Because different channels have different weights.

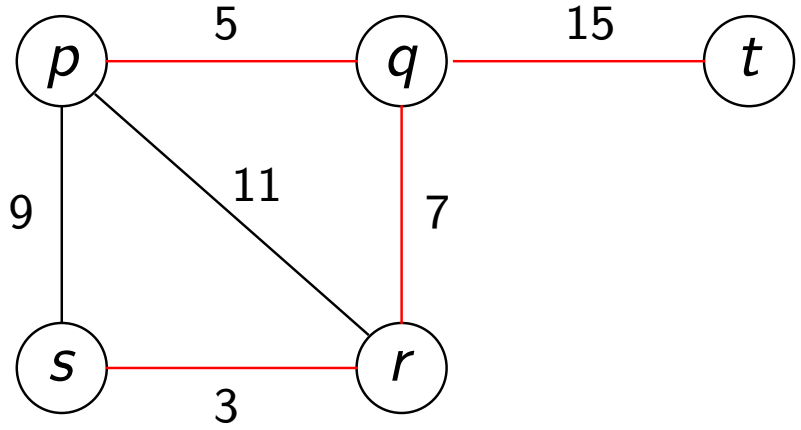
Question

Suppose a process reported a lowest-weight outgoing basic edge, and in return receives $\langle$**initiate**, *fn*, ℓ, *find*$\rangle$.

Why must it test again whether this basic edge is outgoing ?

Answer: Its fragment may in the meantime have joined the fragment at the other side of this basic edge.

Gallager-Humblet-Spira algorithm - Example



pq qp	$\langle$**connect**, 0$\rangle$
pq qp	$\langle$**initiate**, 5, 1, *find*$\rangle$
ps qr	$\langle$**test**, 5, 1$\rangle$
tq	$\langle$**connect**, 0$\rangle$
qt	$\langle$**initiate**, 5, 1, *find*$\rangle$
tq	$\langle$**report**, $\infty\rangle$
rs sr	$\langle$**connect**, 0$\rangle$
rs sr	$\langle$**initiate**, 3, 1, *find*$\rangle$
sp rq	**accept**
pq	$\langle$**report**, 9$\rangle$
qp	$\langle$**report**, 7$\rangle$
qr	$\langle$**connect**, 1$\rangle$
sp rq	$\langle$**test**, 3, 1$\rangle$
ps qr	**accept**
rs	$\langle$**report**, 7$\rangle$
sr	$\langle$**report**, 9$\rangle$
rq	$\langle$**connect**, 1$\rangle$
rq qp qr qt rs	$\langle$**initiate**, 7, 2, *find*$\rangle$
ps sp	$\langle$**test**, 7, 2$\rangle$
pr	$\langle$**test**, 7, 2$\rangle$
rp	**reject**
pq tq qr sr rq	$\langle$**report**, $\infty\rangle$

Gallager-Humblet-Spira algorithm - Complexity

Worst-case message complexity: $O(E + N \log N)$

- A rejected channel requires a **test-reject** or **test-test** pair.

Between two subsequent joins, a process:

- receives one **initiate**
- sends at most one **test** that triggers an **accept**
- sends one **report**
- sends at most one **changeroot** or **connect**

A fragment at level ℓ contains $\geq 2^{\ell}$ processes.

So each process experiences at most $\lfloor \log_2 N \rfloor$ joins.

Back to election

By two extra messages at the very end,
the core node with the largest id becomes the leader.

So Gallager-Humblet-Spira induces an election algorithm for general undirected networks.

(We must impose an order on channels of equal weight.)

Lower bounds for the average-case message complexity
of election algorithms based on comparison of id's:

Rings: $\Omega(N \log N)$

General networks: $\Omega(E + N \log N)$

Lecture in a nutshell

leader election

decentralized, uniform local algorithm, unique process id's

Chang-Roberts and Dolev-Klawe-Rodeh algorithm on directed rings

tree election algorithm

echo algorithm with extinction

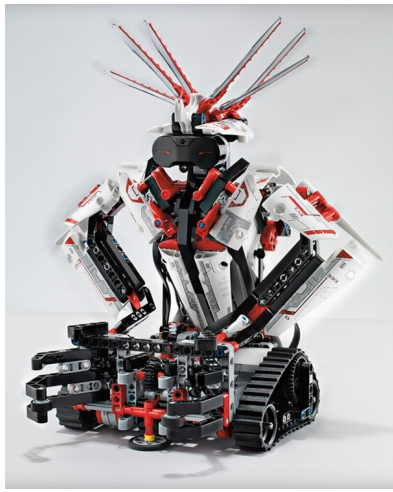
Gallager-Humblet-Spira minimum spanning tree algorithm

Election in anonymous networks

In an anonymous network, processes (and channels) have no unique id.

Processes may be anonymous for several reasons:

- Transmitting/storing id's is too expensive (IEEE 1394 bus).
- Processes don't want to reveal their id (security protocols).
- Absence of unique hardware id's (LEGO Mindstorms).



Question: Suppose there is one *leader*.

How can each process be provided with a unique id ?

Impossibility of election in anonymous rings

Theorem: There is no election algorithm for anonymous rings that always terminates.

Proof: Consider an anonymous ring of size N.

In a symmetric configuration, all processes are in the same state and all channels carry the same messages.

- There is a symmetric initial configuration.
- If γ_0 is symmetric and $\gamma_0 \to \gamma_1$, then there are transitions $\gamma_1 \to \gamma_2 \to \cdots \to \gamma_N$ with γ_N symmetric.

In a symmetric configuration there isn't one leader.

So there is an infinite computation in which no leader is elected.

Fairness

An execution is fair if each event that is applicable in infinitely many configurations, occurs infinitely often in the computation.

Each election algorithm for anonymous rings has a *fair* infinite execution.

(Basically because in the proof, $\gamma_0 \to \gamma_1$ can be chosen freely.)

Probabilistic algorithms

In a probabilistic algorithm, a process may flip a coin, and perform an event based on the outcome of this coin flip.

Probabilistic algorithms where *all* computations terminate in a *correct* configuration aren't interesting.

Because letting the coin e.g. always flip heads yields a correct non-probabilistic algorithm.

Las Vegas and Monte Carlo algorithms

A probabilistic algorithm is Las Vegas if:

- the probability that it terminates is greater than zero, and
- all terminal configurations are correct.

It is Monte Carlo if:

- it always terminates, and
- the probability that a terminal configuration is correct is greater than zero.

Questions

Even if the probability that a Las Vegas algorithm terminates is 1, this doesn't always imply termination. Why is that ?

Assume a Monte Carlo algorithm, and a (deterministic) algorithm to check whether a run of the Monte Carlo algorithm terminated correctly.

Give a Las Vegas algorithm that terminates with probability 1.

Itai-Rodeh election algorithm

Given an anonymous, directed ring; *all processes know the ring size N*.

We adapt the Chang-Roberts algorithm: each initiator sends out an id, and the largest id is the only one making a round trip.

Each initiator selects a random id from $\{1, \ldots, N\}$.

Complication: Different processes may select the same id.

Solution: Each message is supplied with a hop count.

A message that arrives at its source has hop count N.

If several processes select the same largest id, then they start a new election round, *with a higher round number*.

Itai-Rodeh election algorithm

Initially, *initiators* are active in round 0, and *noninitiators* are passive.

Let p be *active*. At the start of election round n, p randomly selects id_p, sends $(n, id_p, 1, false)$, and waits for a message (n', i, h, b).

The 3^{rd} value is the hop count. The 4^{th} value signals if another process *with the same id* was encountered during the round trip.

- p gets (n', i, h, b) with $n' > n$, or $n' = n$ and $i > id_p$: it becomes *passive* and sends $(n', i, h + 1, b)$.
- p gets (n', i, h, b) with $n' < n$, or $n' = n$ and $i < id_p$: it *dismisses* the message.
- p gets (n, id_p, h, b) with $h < N$: it sends $(n, id_p, h + 1, true)$.
- p gets $(n, id_p, N, true)$: it proceeds to round $n + 1$.
- p gets $(n, id_p, N, false)$: it becomes the leader.

Passive processes pass on messages, increasing their hop count by one.

Itai-Rodeh election algorithm - Correctness

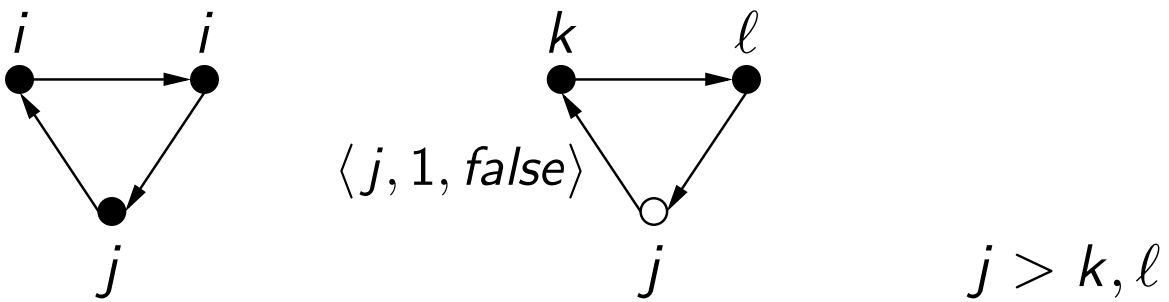
Question: How can an infinite computation occur ?

Correctness: The Itai-Rodeh election algorithm is Las Vegas.

Eventually one leader is elected, with probability 1.

Without rounds, the algorithm would be flawed.

Example:



Average-case message complexity: $O(N \log N)$

Election in arbitrary anonymous networks

The echo algorithm with extinction, with random selection of id's, can be used for election in anonymous undirected networks in which *all processes know the network size*.

Initially, initiators are active in round 0, and noninitiators are passive.

Each active process selects a random id, and starts a wave, tagged with its id and round number 0.

Let process p in wave i of round n be hit by wave j of round n':

- If $n' > n$, or $n' = n$ and $j > i$, then p adopts wave j of round n', and treats the message according to the echo algorithm.
- If $n' < n$, or $n' = n$ and $j < i$, then p dismisses the message.
- If $n' = n$ and $j = i$, then p treats the message according to the echo algorithm.

Election in arbitrary anonymous networks

Each message sent upwards in the constructed tree reports the size of its subtree.

All other messages report 0.

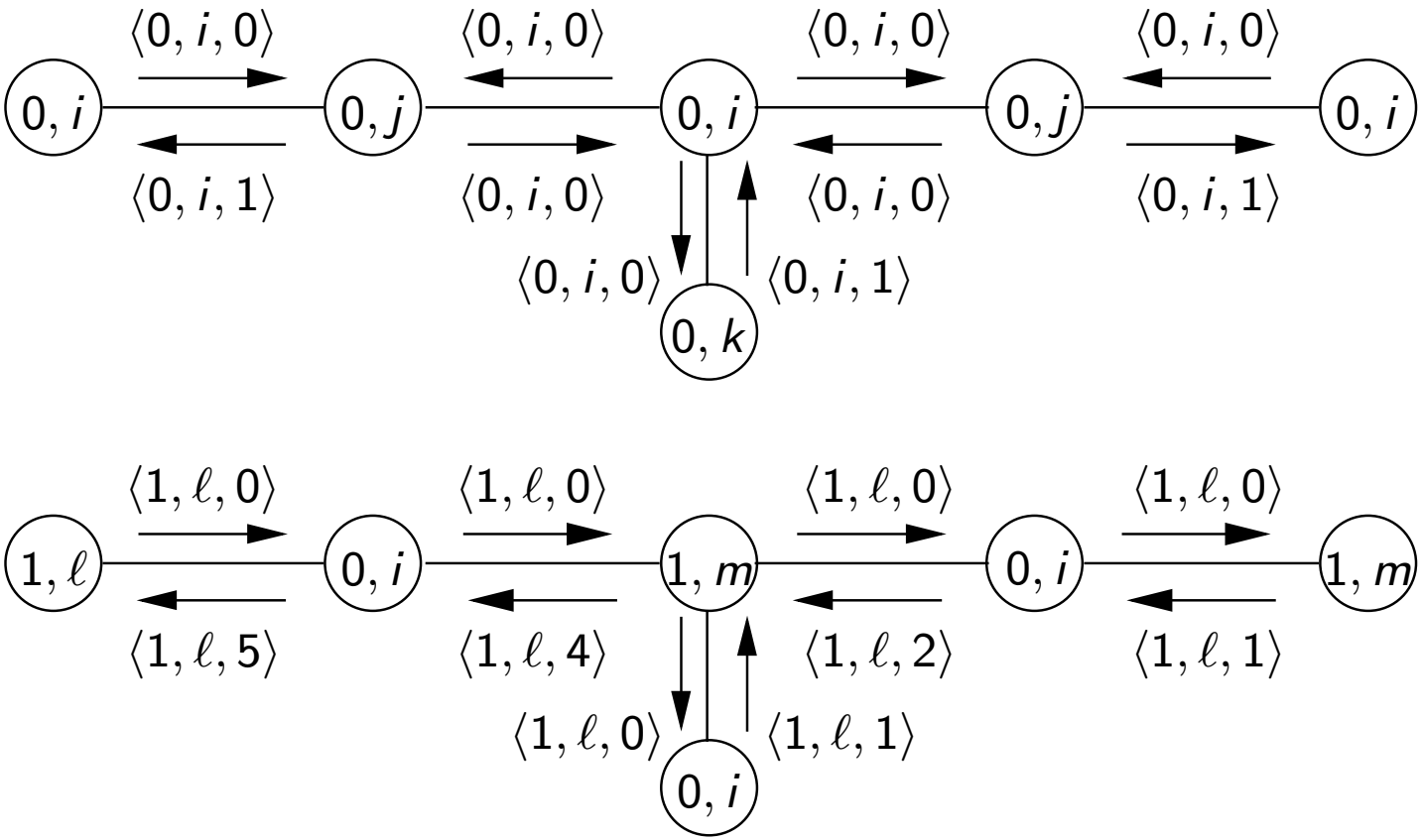
When a process *decides*, it computes the size of the constructed tree.

If the constructed tree covers the network, it becomes the leader.

Else, it selects a new id, and initiates a new wave, in the next round.

Election in arbitrary anonymous networks - Example

$i > j > k > \ell > m$. Only waves that complete are shown.



The process at the left computes size 6, and becomes the leader.

Question

Is there another scenario in which the right-hand side node progresses to round 2 ?

Computing the size of a network

Theorem: There is no Las Vegas algorithm to compute the size of an anonymous ring.

This implies that there is no Las Vegas algorithm for election in an anonymous ring *if processes don't know the ring size*.

Because when a leader is known, the network size can be computed using a centralized wave algorithm with the leader as initiator.

Impossibility of computing anonymous network size

Theorem: There is no Las Vegas algorithm to compute the size of an anonymous ring.

Proof: Consider an anonymous, directed ring $p_0, \ldots, p_{N-1}$.

Suppose a computation C of a (probabilistic) ring size algorithm terminates with the correct outcome N.

Consider the ring $p_0, \ldots, p_{2N-1}$.

Let each event at a p_i in C be executed concurrently at p_i and p_{i+N}.

This computation terminates with the incorrect outcome N.

Itai-Rodeh ring size algorithm

Each process p maintains an *estimate* est_p of the ring size.

Initially $est_p = 2$. (Always $est_p \leq N$.)

p initiates an estimate round (1) at the start of the algorithm, and (2) at each update of est_p.

Each round, p selects a random id_p in $\{1, \ldots, R\}$, sends $(est_p, id_p, 1)$, and waits for a message (est, id, h). (Always $h \leq est$.)

- $est < est_p$. Then p *dismisses* the message.
- $est > est_p$.
  - If $h < est$, then p sends $(est, id, h+1)$, and $est_p \leftarrow est$.
  - If $h = est$, then $est_p \leftarrow est + 1$.
- $est = est_p$.
  - If $h < est$, then p sends $(est, id, h+1)$.
  - If $h = est$ and $id \neq id_p$, then $est_p \leftarrow est + 1$.
  - If $h = est$ and $id = id_p$, then p *dismisses* the message (possibly its own message returned).

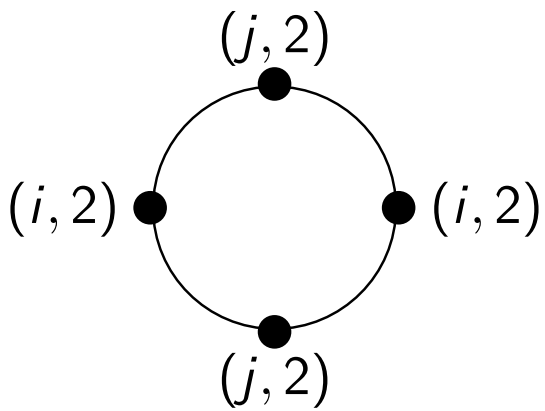
Itai-Rodeh ring size algorithm - Correctness

When the algorithm terminates, $est_p \leq N$ for all p.

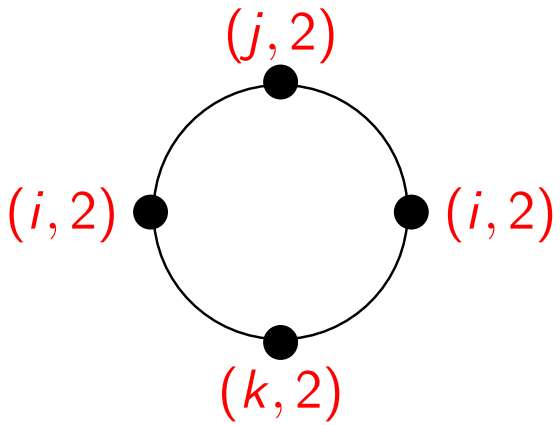
The Itai-Rodeh ring size algorithm is a Monte Carlo algorithm.

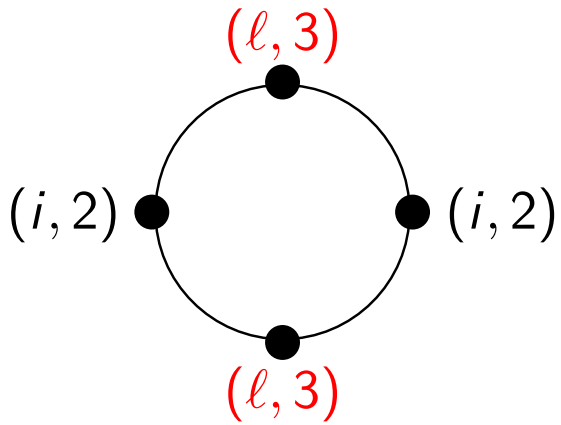
Possibly, in the end $est_p < N$.

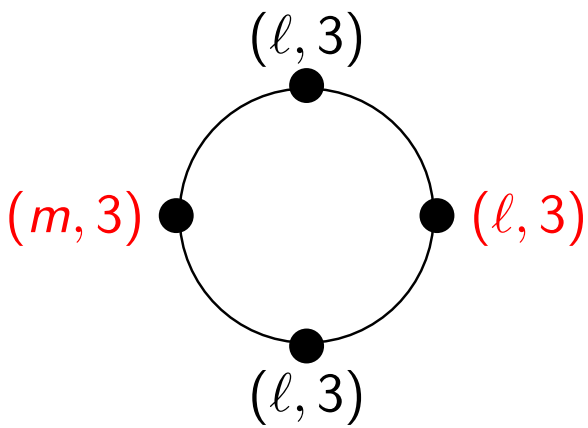
Example:

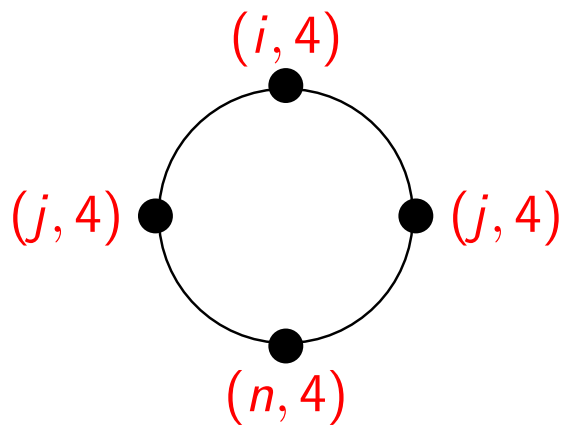


Itai-Rodeh ring size algorithm - Example









Itai-Rodeh ring size algorithm - Termination

Question: Upon message-termination, is est_p always the same at all p ?

There is no Las Vegas algorithm for general termination detection in anonymous rings.

Itai-Rodeh ring size algorithm - Complexity

The probability of computing an incorrect ring size tends to zero when R tends to infinity.

Worst-case message complexity: $O(N^3)$

The N processes start at most $N-1$ estimate rounds.

Each round they send a message, which takes at most N steps.

Question

Give an (always correctly terminating) algorithm for computing the network size of anonymous, acyclic networks.

Answer: Use the tree algorithm, whereby each process reports the size of its subtree to its parent.

IEEE 1394 election algorithm

The IEEE 1394 standard is a serial multimedia bus.

It connects digital devices,
which can be added/removed dynamically.

Transmitting/storing id's is too expensive,
so the network is anonymous.

The network size is unknown to the processes.

The tree algorithm for undirected, acyclic networks is used.

Networks that contain a cycle give a time-out.

IEEE 1394 election algorithm

When a process has one possible parent, it sends a parent request to this neighbor. If the request is accepted, an ack is sent back.

The last two parentless processes can send parent requests to each other simultaneously. This is called root contention.

Each of the two processes in root contention randomly decides to either *immediately send* a parent request again, or to *wait some time* for a parent request from the other process.

Question: Is it optimal for performance to give probability 0.5 to both sending immediately and waiting for some time?

Lecture in a nutshell

anonymous network

impossibility of election in anonymous networks

Las Vegas / Monte Carlo algorithms

Itai-Rodeh election algorithm for directed rings (Las Vegas)

echo election algorithm for anonymous networks (Las Vegas)

no Las Vegas algorithm for computing anonymous network size

Itai-Rodeh ring size algorithm (Monte Carlo)

IEEE 1394 election algorithm

Fault tolerance

A process may (1) *crash*, i.e., execute no further events, or even (2) be *Byzantine*, meaning that it can perform arbitrary events.

Assumption: The network is *complete*, i.e., there is an undirected channel between each pair of different processes.

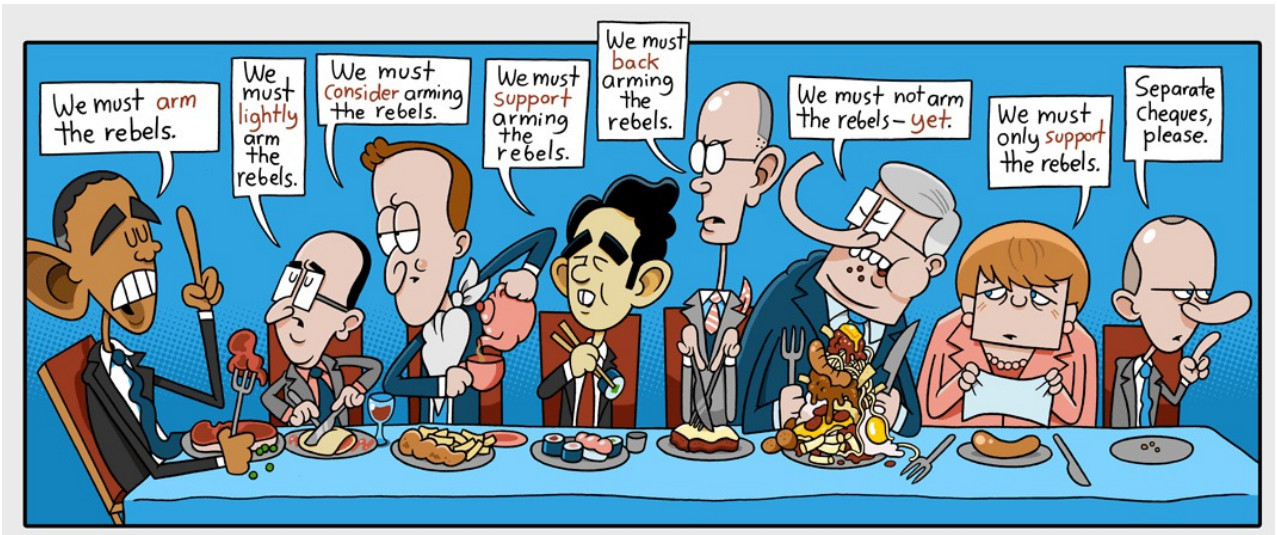
So failing processes never make the remaining network disconnected.

Assumption: Crashing of processes can't be observed.

Consensus

Binary consensus: Initially, all processes randomly select 0 or 1.

Eventually, all correct processes must uniformly decide 0 or 1.



Consensus underlies many important problems in distributed computing: termination detection, mutual exclusion, leader election, ...

Consensus - Assumptions

k-crash consensus: At most k processes may crash.

Validity: If all processes randomly select the same initial value b, then all correct processes decide b.

This excludes trivial solutions where e.g. processes always decide 0.

By validity, each k-crash consensus algorithm with $k \geq 1$ has a bivalent initial configuration that can reach terminal configurations with a decision 0 as well as with a decision 1.

Impossibility of 1-crash consensus

Theorem: No algorithm for 1-crash consensus always terminates.

Idea: A decision is determined by an event e at a process p.

Since p may crash, after e the other processes must be able to decide without input from p.

b-potent set of processes

A set S of processes is called b-potent, in a configuration, if by only executing events at processes in S, some process in S can decide b.

Question: Consider any k-crash consensus algorithm.

Why should each set of $N - k$ processes be b-potent for some b?

Impossibility of 1-crash consensus

Theorem: No algorithm for 1-crash consensus always terminates.

Proof: Consider a 1-crash consensus algorithm.

Let γ be a bivalent configuration: $\gamma \to \gamma_0$ and $\gamma \to \gamma_1$, where γ_0 can lead to decision 0 and γ_1 to decision 1.

- Let the transitions correspond to events at *different* processes.
  Then $\gamma_0 \to \delta \leftarrow \gamma_1$ for some δ. So γ_0 or γ_1 is bivalent.
- Let the transitions correspond to events at *one* process p.
  In γ, p can crash, so the other processes are b-potent for some b.
  Likewise for γ_0 and γ_1. It follows that γ_{1-b} is bivalent.

So each bivalent configuration has a transition to a bivalent configuration.

Hence each bivalent initial configuration yields an infinite computation.

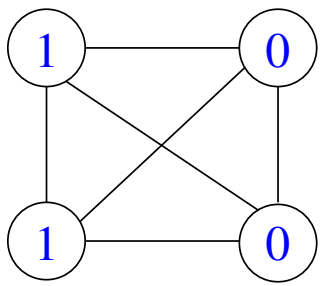
There exist *fair* infinite computations.

Impossibility of 1-crash consensus - Example

Let $N = 4$. At most one process can crash.

There are voting rounds, in which each process broadcasts it value.

Since one process may crash, in a round, processes can only wait for three votes.



The left (resp. right) processes might in every round receive two 1-votes and one 0-vote (resp. two 0-votes and one 1-vote).

(Admittedly, this scheduling of messages is unfair.)

Impossibility of $\lceil \frac{N}{2} \rceil$-crash consensus

Theorem: Let $k \geq \frac{N}{2}$. There is no Las Vegas algorithm for k-crash consensus.

Proof: Suppose, toward a contradiction, there is such an algorithm.

Divide the set of processes in S and T, with $|S| = \lfloor \frac{N}{2} \rfloor$ and $|T| = \lceil \frac{N}{2} \rceil$.

Suppose all processes in S select 0 and all processes in T select 1.

Suppose that messages between processes in S and in T are very slow.

Since $k \geq \frac{N}{2}$, at some point the processes in S must assume the processes in T all crashed, and decide 0.

Likewise, at some point the processes in T must assume the processes in S all crashed, and decide 1.

Question

Give a Monte Carlo algorithm for k-crash consensus for any k.

Answer: Let any process decide for its initial (random) value.

With a (very small) positive probability all correct processes decide for the same value.

Bracha-Toueg crash consensus algorithm

Let $k < \frac{N}{2}$. Initially, each correct process randomly selects 0 or 1, with weight 1. In round n, at each correct, undecided p:

- p sends $\langle n, value_p, weight_p \rangle$ to all processes (including itself).
- p waits until $N - k$ messages $\langle n, b, w \rangle$ have arrived.

  (p dismisses/stores messages from earlier/future rounds.)

  If $w > \frac{N}{2}$ for an $\langle n, b, w \rangle$, then $value_p \leftarrow b$. *(This b is unique.)*

  Else, $value_p \leftarrow 0$ if most messages voted 0, $value_p \leftarrow 1$ otherwise.

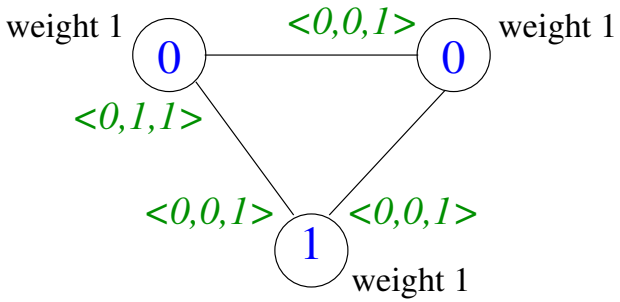
  $weight_p \leftarrow$ the number of incoming votes for $value_p$ in round n.
- If $w > \frac{N}{2}$ for $> k$ incoming messages $\langle n, b, w \rangle$, then p decides b.

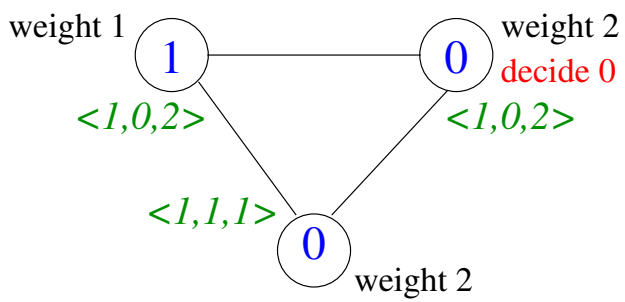
  (Note that $k < N - k$.)

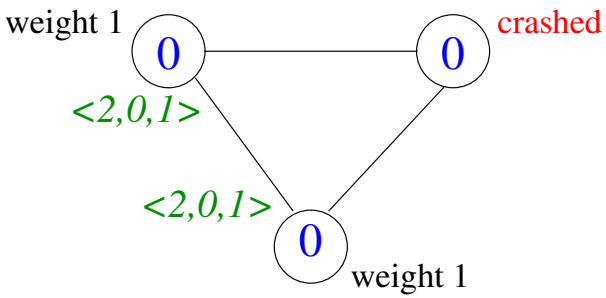
If p decides b, it broadcasts $\langle n + 1, b, N - k \rangle$ and $\langle n + 2, b, N - k \rangle$, and terminates.

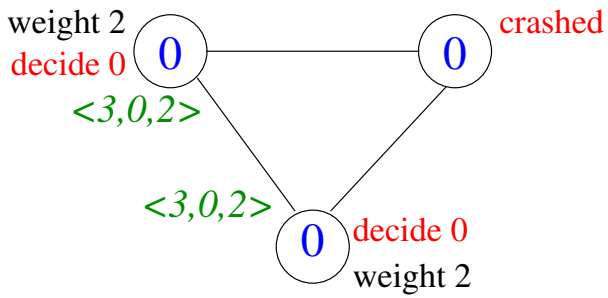
Bracha-Toueg crash consensus algorithm - Example

$N = 3$ and $k = 1$. Each round a correct process requires two incoming messages, and two b-votes with weight 2 to decide b.









(Messages of a process to itself aren't depicted.)

Bracha-Toueg crash consensus algorithm - Correctness

Theorem: Let $k < \frac{N}{2}$. The Bracha-Toueg k-crash consensus algorithm is a Las Vegas algorithm that terminates with probability 1.

Proof (part I): Suppose a process decides b in round n.

Then in round n, $value_q = b$ and $weight_q > \frac{N}{2}$ for $> k$ processes q.

So in round n, each correct process receives a $\langle q, b, w \rangle$ with $w > \frac{N}{2}$.

So in round $n+1$, all correct processes vote b.

So in round $n+2$, all correct processes vote b with weight $N - k$.

Hence, after round $n+2$, all correct processes have decided b.

Concluding, all correct processes decide for the same value.

Bracha-Toueg crash consensus algorithm - Correctness

Proof (part II): *Assumption*: Scheduling of messages is fair.

Due to fair scheduling, there is a chance $\rho > 0$ that in a round n all processes receive the first $N - k$ messages from the same processes.

After round n, all correct processes have the same value b.

After round $n + 1$, all correct processes have value b with weight $N - k$.

After round $n + 2$, all correct processes have decided b.

Concluding, the algorithm terminates with probability 1.

Impossibility of $\lceil \frac{N}{3} \rceil$-Byzantine consensus

Theorem: Let $k \geq \frac{N}{3}$. There is no Las Vegas algorithm for k-Byzantine consensus.

Proof: Suppose, toward a contradiction, there is such an algorithm.

Since $k \geq \frac{N}{3}$, we can choose sets S and T of processes with $|S| = |T| = N - k$ and $|S \cap T| \leq k$.

Suppose all processes in S select 0 and all processes in T select 1.

Suppose that messages between processes in S and in T are very slow.

Suppose all processes that aren't in $S \cup T$ are Byzantine.

The processes in S can then, with the help of the Byzantine processes, decide 0.

Likewise the processes in T can decide 1.

Bracha-Toueg Byzantine consensus algorithm

Let $k < \frac{N}{3}$.

Again, in every round, each correct process:

- broadcasts its value,
- waits for $N - k$ incoming messages, and
- changes its value to the majority of votes in the round.

(No weights are needed.)

A correct process decides b if it receives $> \frac{N+k}{2}$ b-votes in one round.

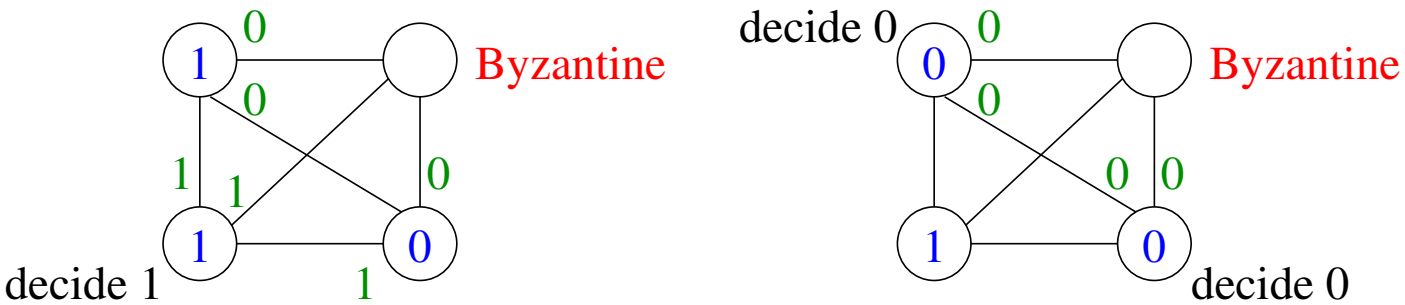
Then more than half of the correct processes voted b in this round.

(Note that $\frac{N+k}{2} < N - k$.)

Echo mechanism

Complication: A Byzantine process may send different votes to different processes.

Example: Let $N = 4$ and $k = 1$. Each round, a correct process waits for three votes, and needs three b-votes to decide b.



Solution: Each incoming vote is verified using an echo mechanism.

A vote is *accepted* after $> \frac{N+k}{2}$ confirming echos.

Bracha-Toueg Byzantine consensus algorithm

Initially, each correct process randomly selects 0 or 1.

In round n, at each correct, undecided p:

- p sends $\langle \textbf{vote}, n, value_p \rangle$ to all processes (including itself).
- If p receives $\langle \textbf{vote}, m, b \rangle$ from q, it sends $\langle \textbf{echo}, q, m, b \rangle$ to all processes (including itself).
- p counts incoming $\langle \textbf{echo}, q, n, b \rangle$ messages for each q, b.

  When $> \frac{N+k}{2}$ such messages arrived, p *accepts* q's b-vote.
- The round is completed when p has accepted $N - k$ votes.

  If most votes are for 0, then $value_p \leftarrow 0$. Else, $value_p \leftarrow 1$.

Bracha-Toueg Byzantine consensus algorithm

Processes dismiss/store messages from earlier/future rounds.

If multiple messages $\langle \textbf{vote}, m, \_ \rangle$ or $\langle \textbf{echo}, q, m, \_ \rangle$ arrive via the same channel, only the first one is taken into account.

If $> \frac{N+k}{2}$ of the *accepted* votes are for b, then p decides b.

When p decides b, it broadcasts $\langle \textbf{decide}, b \rangle$ and terminates.

The other processes interpret $\langle \textbf{decide}, b \rangle$ as a b-vote by p, and a b-echo by p for each q, for all rounds to come.

Questions

If an undecided process receives $\langle \textbf{decide}, b \rangle$, why can it in general not immediately decide b?

Answer: The message may originate from a Byzantine process.

What happens if all k Byzantine processes keep silent?

Answer: The $N - k$ correct processes reach consensus in two rounds.

Bracha-Toueg Byzantine consensus alg. - Example

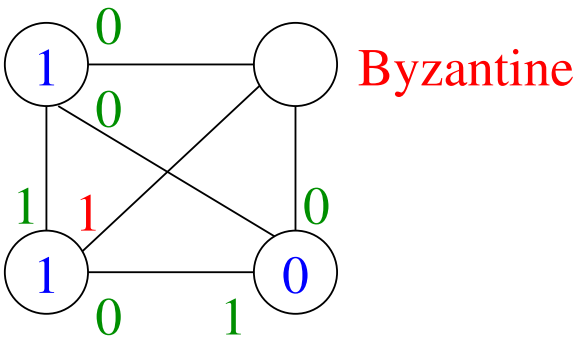
We study the previous example again, now with verification of votes.

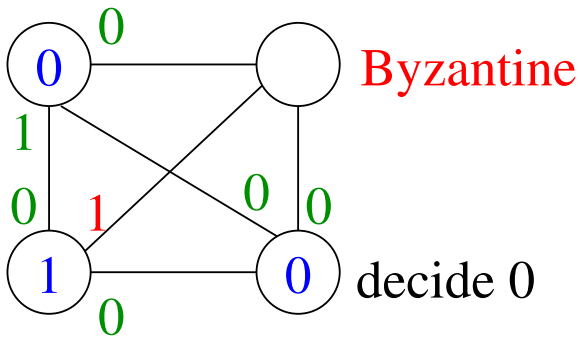
$N = 4$ and $k = 1$, so each round a correct process needs:

- $> \frac{N+k}{2}$, i.e. three, confirmations to accept a vote;
- $N - k$, i.e. three, accepted votes to determine a value; and
- $> \frac{N+k}{2}$, i.e. three, accepted b-votes to decide b.

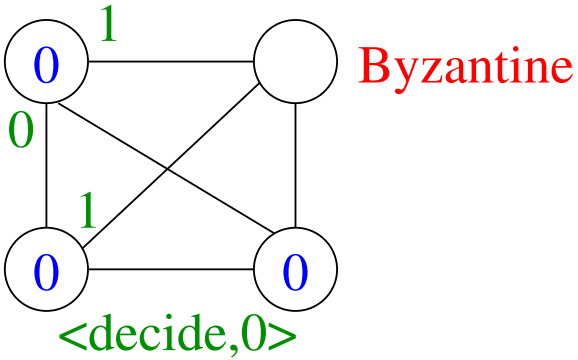
Only relevant **vote** messages are depicted (without their round number).

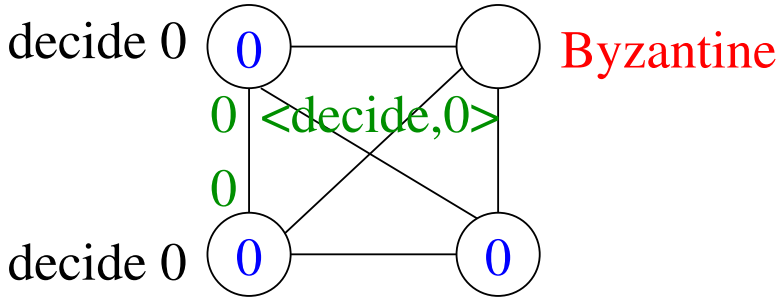
Bracha-Toueg Byzantine consensus alg. - Example





In round zero, the left bottom process doesn't accept vote 1 by the Byzantine process, since none of the other two correct processes confirm this vote. So it waits for (and accepts) vote 0 by the right bottom process, and thus doesn't decide 1 in round zero.





Bracha-Toueg Byzantine consensus alg. - Correctness

Theorem: Let $k < \frac{N}{3}$. The Bracha-Toueg k-Byzantine consensus algorithm is a Las Vegas algorithm that terminates with probability 1.

Proof: Each round, the correct processes eventually accept $N - k$ votes, since there are $\geq N - k$ correct processes. *(Note that* $N - k > \frac{N+k}{2}$*.)*

In round n, let correct processes p and q accept votes for b and b', respectively, from a process r.

Then they received $> \frac{N+k}{2}$ messages $\langle \textbf{echo}, r, n, b \rangle$ resp. $\langle \textbf{echo}, r, n, b' \rangle$.

$> k$ processes, so at least one correct process, sent such messages to both p and q.

So $b = b'$.

Bracha-Toueg Byzantine consensus alg. - Correctness

Suppose a correct process decides b in round n.

In this round it accepts $> \frac{N+k}{2}$ b-votes.

So in round n, correct processes accept $> \frac{N+k}{2} - k = \frac{N-k}{2}$ b-votes.

Hence, after round n, $value_q = b$ for each correct q.

So correct processes will vote b in all rounds $m > n$.

Because they will accept $\geq N - 2k > \frac{N-k}{2}$ b-votes.

Bracha-Toueg Byzantine consensus alg. - Correctness

Let S be a set of $N - k$ correct processes.

Assuming fair scheduling, there is a chance $\rho > 0$ that in a round each process in S accepts $N - k$ votes from the processes in S.

With chance ρ^2 this happens in consecutive rounds $n, n + 1$.

After round n, all processes in S have the same value b.

After round $n + 1$, all processes in S have decided b.

Lecture in a nutshell

crashed / Byzantine processes

complete network / crashes can't be observed

(binary) consensus

no algorithm for 1-crash consensus always terminates

if $k \geq \frac{N}{2}$, there is no Las Vegas algorithm for k-crash consensus

Bracha-Toueg k-crash consensus algorithm for $k < \frac{N}{2}$

if $k \geq \frac{N}{3}$, there is no Las Vegas algorithm for k-Byzantine consensus

Bracha-Toueg k-Byzantine consensus algorithm for $k < \frac{N}{3}$

Failure detection

A failure detector at a process keeps track which processes have (or may have) crashed.

Given an upper bound on network latency, and heartbeat messages, one can implement a failure detector.

With a failure detector, the proof of impossibility of 1-crash consensus no longer applies.

For this setting, terminating crash consensus algorithms exist.

Failure detection

Aim: To detect *crashed* processes.

We assume a time domain, with a total order.

$F(\tau)$ is the set of crashed processes at time τ.

$$\tau_1 \leq \tau_2 \Rightarrow F(\tau_1) \subseteq F(\tau_2) \qquad \text{(i.e., no restart)}$$

Assumption: Processes can't observe $F(\tau)$.

$H(p, \tau)$ is the set of processes that p *suspects* to be crashed at time τ.

Each computation is decorated with :

- a failure pattern F
- a failure detector history H

Complete failure detector

We require that failure detectors are complete :

From some time onward, each crashed process is suspected by each correct process.

Strongly accurate failure detector

A failure detector is strongly accurate if only crashed processes are ever suspected.

Assumptions:

- Each correct process broadcasts **alive** every ν time units.
- $d_{\max}$ is a known upper bound on network latency.

A process from which no message is received for $\nu + d_{\max}$ time units has crashed.

This failure detector is *complete* and *strongly accurate*.

Weakly accurate failure detector

A failure detector is weakly accurate if some (correct) process is never suspected by any process.

Assume a complete and *weakly accurate* failure detector.

We give a rotating coordinator algorithm for $(N-1)$-crash consensus.

Consensus with weakly accurate failure detection

Processes are numbered: $p_0, \ldots, p_{N-1}$.

Initially, each process randomly selects 0 or 1. In round n:

- p_n (if not crashed) broadcasts its value.
- Each process waits:
  - either for an incoming message from p_n, in which case it adopts the value of p_n;
  - or until it suspects that p_n has crashed.

After round $N-1$, each correct process decides for its value.

Correctness: Let p_j never be suspected.

After round j, all correct processes have the same value b.

Hence, after round $N-1$, all correct processes decide b.

Eventually strongly accurate failure detector

A failure detector is eventually strongly accurate if
from some time onward, only crashed processes are suspected.

Assumptions:

- Each correct process broadcasts **alive** every ν time units.
- There is an *unknown* upper bound on network latency.

Each process q initially guesses as network latency $d_q = 1$.

If q receives no message from p for $\nu + d_q$ time units, then
q *suspects* that p has crashed.

If q receives a message from a suspected process p, then
p is no longer suspected and $d_q \leftarrow d_q + 1$.

This failure detector is *complete* and *eventually strongly accurate*.

Impossibility of $\lceil \frac{N}{2} \rceil$-crash consensus

Theorem: Let $k \geq \frac{N}{2}$. There is no Las Vegas algorithm for k-crash consensus based on an *eventually strongly accurate* failure detector.

Proof: Suppose, toward a contradiction, there is such an algorithm.

Divide the set of processes in S and T, with $|S| = \lfloor \frac{N}{2} \rfloor$ and $|T| = \lceil \frac{N}{2} \rceil$.

Suppose all processes in S select 0 and all processes in T select 1.

Suppose that for a long time the processes in S suspect the processes in T crashed, and the processes in T suspect the processes in S crashed.

The processes in S can then decide 0, while the process in T can decide 1.

Chandra-Toueg k-crash consensus algorithm

A failure detector is eventually weakly accurate if
from some time onward *some* (correct) process is never suspected.

Let $k < \frac{N}{2}$. A complete and *eventually weakly accurate* failure detector is used for k-crash consensus.

Each process q records the last round lu_q in which it updated $value_q$.

Initially, $value_q \in \{0, 1\}$ and $lu_q = -1$.

Processes are numbered: $p_0, \ldots, p_{N-1}$.

Round n is coordinated by p_c with $c = n \bmod N$.

Chandra-Toueg k-crash consensus algorithm

- In round n, each correct q sends $\langle \textbf{vote}, n, value_q, lu_q \rangle$ to p_c.
- p_c (if not crashed) waits until $N - k$ such messages arrived, and selects one, say $\langle \textbf{vote}, n, b, \ell \rangle$, with ℓ as large as possible.

  $value_{p_c} \leftarrow b$, $lu_{p_c} \leftarrow n$, and p_c broadcasts $\langle \textbf{value}, n, b \rangle$.
- Each correct q waits:
  - either until $\langle \textbf{value}, n, b \rangle$ arrives: then $value_q \leftarrow b$, $lu_q \leftarrow n$, and q sends $\langle \textbf{ack}, n \rangle$ to p_c;
  - or until it suspects p_c crashed: then q sends $\langle \textbf{nack}, n \rangle$ to p_c.
- If p_c receives $> k$ messages $\langle \textbf{ack}, n \rangle$, then p_c decides b, and broadcasts $\langle \textbf{decide}, b \rangle$.

An undecided process that receives $\langle \textbf{decide}, b \rangle$, decides b.

Chandra-Toueg algorithm - Correctness

Theorem: Let $k < \frac{N}{2}$. The Chandra-Toueg algorithm is an (always terminating) k-crash consensus algorithm.

Proof (part I): If the coordinator in some round n receives $> k$ **ack**'s, then (for some $b \in \{0, 1\}$):

(1) there are $> k$ processes q with $lu_q \geq n$, and

(2) $lu_q \geq n$ implies $value_q = b$.

Properties (1) and (2) are preserved in all rounds $m > n$.

This follows by induction on $m - n$.

By (1), in round m the coordinator receives a **vote** with $lu \geq n$.

Hence, by (2), the coordinator of round m sets its value to b, and broadcasts $\langle \textbf{value}, m, b \rangle$.

So from round n onward, processes can only decide b.

Chandra-Toueg algorithm - Correctness

Proof (part II):

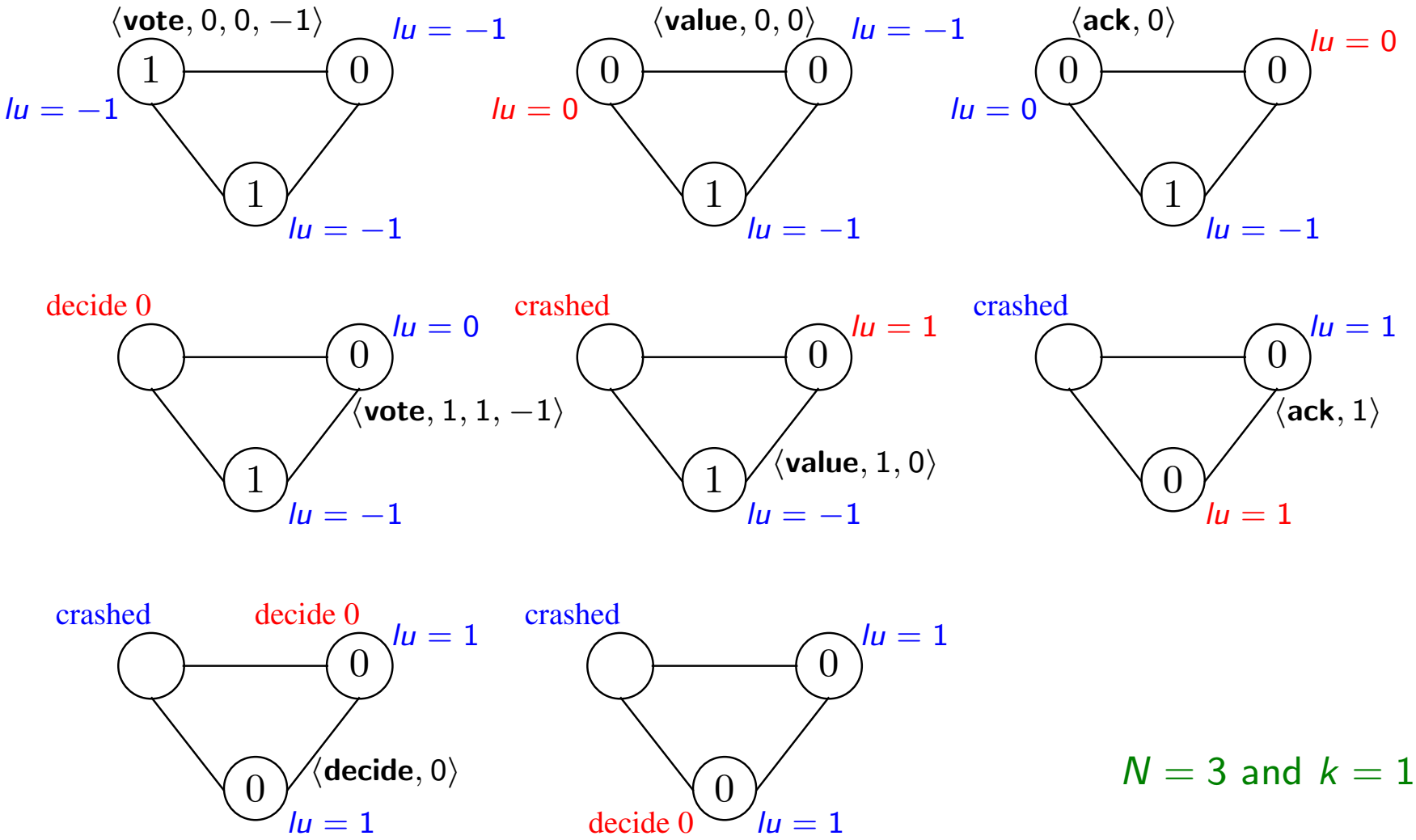
Since the failure detector is eventually weakly accurate,
from some round onward, some process p will never be suspected.

So when p becomes the coordinator, it receives $\geq N - k$ **ack**'s.

Since $N - k > k$, it decides.

All correct processes eventually receive the **decide** message of p,
and also decide.

Chandra-Toueg algorithm - Example



$N = 3$ and $k = 1$

Messages and ack's that a process sends to itself and 'irrelevant' messages are omitted.

Question

Why is it difficult to devise a failure detector for Byzantine processes ?

Answer: Failure detectors are usually based on the *absence* of events.

Local clocks with bounded drift

Let's forget about Byzantine processes for a moment.

The time domain is $\mathbb{R}_{\geq 0}$.

Each process p has a local clock $C_p(\tau)$, which returns a time value at *real* time τ.

Local clocks have bounded drift, compared to real time:

If C_p isn't adjusted between times τ_1 and τ_2, then

$$\frac{1}{\rho}(\tau_2 - \tau_1) \;\leq\; C_p(\tau_2) - C_p(\tau_1) \;\leq\; \rho(\tau_2 - \tau_1)$$

for some *known* $\rho > 1$.

Clock synchronization

At certain time intervals, the processes *synchronize* clocks :

They read each other's clock values, and adjust their local clocks.

The aim is to achieve, for some δ, and all τ,

$$|C_p(\tau) - C_q(\tau)| \leq \delta$$

Due to drift, this precision may degrade over time, necessitating repeated clock synchronizations.

We assume a *known* bound $d_{\max}$ on network latency.

For simplicity, let $d_{\max}$ be much smaller than δ, so that this latency can be ignored in the clock synchronization.

Clock synchronization

Suppose that after each synchronization, at say real time τ, for all processes p, q:

$$|C_p(\tau) - C_q(\tau)| \leq \delta_0$$

for some $\delta_0 < \delta$.

Due to ρ-bounded drift of local clocks, at real time $\tau + R$,

$$|C_p(\tau + R) - C_q(\tau + R)| \leq \delta_0 + (\rho - \frac{1}{\rho})R < \delta_0 + \rho R$$

So synchronizing every $\frac{\delta - \delta_0}{\rho}$ (real) time units suffices.

Impossibility of $\lceil \frac{N}{3} \rceil$-Byzantine synchronizers

Theorem: Let $k \geq \frac{N}{3}$. There is no k-Byzantine clock synchronizer.

Proof: Let $N = 3$, $k = 1$. Processes are p, q, r; r is Byzantine.

(The construction below easily extends to general N and $k \geq \frac{N}{3}$.)

Let the clock of p run faster than the clock of q.

Suppose a synchronization takes place at real time τ.

r sends $C_p(\tau) + \delta$ to p, and $C_q(\tau) - \delta$ to q.

p and q can't recognize that r is Byzantine.

So they have to stay within range δ of the value reported by r.

Hence p can't decrease, and q can't increase its clock value.

By repeating this scenario at each synchronization round,
the clock values of p and q get further and further apart.

Mahaney-Schneider synchronizer

Consider a complete network of N processes.

Suppose at most $k < \frac{N}{3}$ processes are Byzantine.

Each correct process in a synchronization round:

1. Collects the clock values of all processes (waiting for $2d_{\max}$).
2. Discards those reported values τ for which $< N - k$ processes report a value in the interval $[\tau - \delta, \tau + \delta]$.

   (They are from Byzantine processes.)
3. Replaces all discarded/non-received values by an accepted value.
4. Takes the average of these N values as its new clock value.

Mahaney-Schneider synchronizer - Correctness

Lemma: Let $k < \frac{N}{3}$. If in some synchronization round values a_p and a_q pass the filters of correct processes p and q, respectively, then

$$|a_p - a_q| \leq 2\delta$$

Proof: $\geq N - k$ processes reported a value in $[a_p - \delta,\ a_p + \delta]$ to p.

And $\geq N - k$ processes reported a value in $[a_q - \delta,\ a_q + \delta]$ to q.

Since $N - 2k > k$, at least one correct process r reported a value in $[a_p - \delta,\ a_p + \delta]$ to p, and in $[a_q - \delta,\ a_q + \delta]$ to q.

Since r reports the same value to p and q, it follows that

$$|a_p - a_q| \leq 2\delta$$

Mahaney-Schneider synchronizer - Correctness

Theorem: Let $k < \frac{N}{3}$. The Mahaney-Schneider synchronizer is k-Byzantine.

Proof : a_{pr} (resp. a_{qr}) denotes the value that correct process p (resp. q) accepts from or assigns to process r, in some synchronization round.

By the lemma, for all r, $|a_{pr} - a_{qr}| \leq 2\delta$.

Moreover, $a_{pr} = a_{qr}$ for all *correct* r.

Hence, for all correct p and q,

$$|\frac{1}{N}(\sum_{\text{processes } r} a_{pr}) - \frac{1}{N}(\sum_{\text{processes } r} a_{qr})| \leq \frac{1}{N} k 2\delta < \frac{2}{3}\delta$$

So we can take $\delta_0 = \frac{2}{3}\delta$.

There should be a synchronization every $\frac{\delta - \delta_0}{\rho} = \frac{\delta}{3\rho}$ time units.

Lecture in a nutshell (part I)

complete failure detector

strongly accurate failure detector

rotating coordinator crash consensus algorithm with
a weakly accurate failure detector

eventually strongly accurate failure detector

k-crash consensus for $k \geq \frac{N}{2}$ remains impossible with
an eventually strongly accurate failure detection

Chandra-Toueg crash consensus algorithm with
an eventually weakly accurate failure detector

Lecture in a nutshell (part II)

local clocks with ρ-bounded drift (where ρ is known)

synchronize clocks so that they stay within δ of each other

for $k \geq \frac{N}{3}$, there is no k-Byzantine synchronizer

Mahaney-Schneider k-Byzantine synchronizer

Synchronous networks

Let's again forget about Byzantine processes for a moment.

A synchronous network proceeds in pulses. In one pulse, each process:

1. sends messages
2. receives messages
3. performs internal events

A message is sent and received in the same pulse.

Such synchrony is called lockstep.

Building a synchronous network

Assume ρ-bounded local clocks with precision δ.

For simplicity, we ignore the network latency.

When a process reads clock value $(i-1)\rho^2\delta$, it starts pulse i.

Key question: Does a process p receive all messages for pulse i before it starts pulse $i+1$? That is, for all q,

$$C_q^{-1}((i-1)\rho^2\delta) \;\leq\; C_p^{-1}(i\rho^2\delta)$$

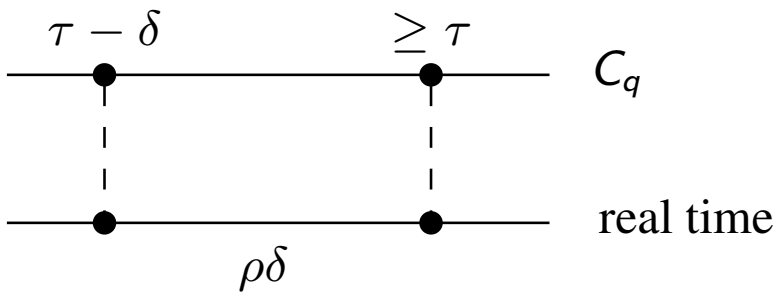
Because then q starts pulse i no later than p starts pulse $i+1$.

($C_r^{-1}(\tau)$ is the moment in time the clock of r returns τ.)

Building a synchronous network

Since the clock of q is ρ-bounded from below,

$$C_q^{-1}(\tau) \;\leq\; C_q^{-1}(\tau - \delta) + \rho\delta$$



Since local clocks have precision δ,

$$C_q^{-1}(\tau - \delta) \;\leq\; C_p^{-1}(\tau)$$

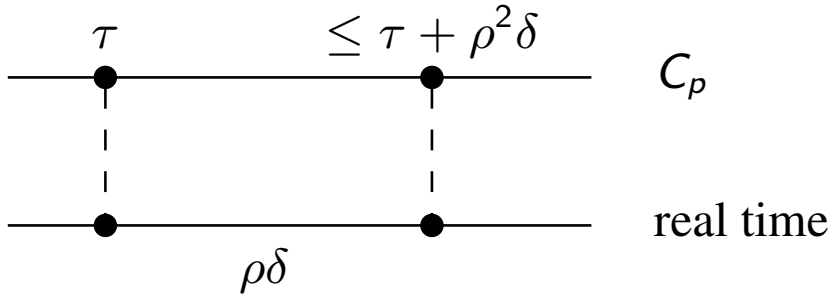
Hence, for all τ,

$$C_q^{-1}(\tau) \;\leq\; C_p^{-1}(\tau) + \rho\delta$$

Building a synchronous network

Since the clock of p is ρ-bounded from above,

$$C_p^{-1}(\tau) + \rho\delta \;\leq\; C_p^{-1}(\tau + \rho^2\delta)$$



Hence,

$$\begin{aligned} C_q^{-1}((i-1)\rho^2\delta) &\leq C_p^{-1}((i-1)\rho^2\delta) + \rho\delta \\ &\leq C_p^{-1}(i\rho^2\delta) \end{aligned}$$

Byzantine consensus for synchronous systems

In a synchronous system, the proof of impossibility of 1-crash consensus no longer applies.

Because within a pulse, a process is guaranteed to receive a message from all correct processes.

For this setting, terminating Byzantine consensus algorithms exist.

Byzantine broadcast

Consider a synchronous network with at most $k < \frac{N}{3}$ Byzantine processes.

One process g, called the general, is given an input $x_g \in \{0, 1\}$.

The other processes, called lieutenants, know who is the general.

Requirements for k-Byzantine broadcast:

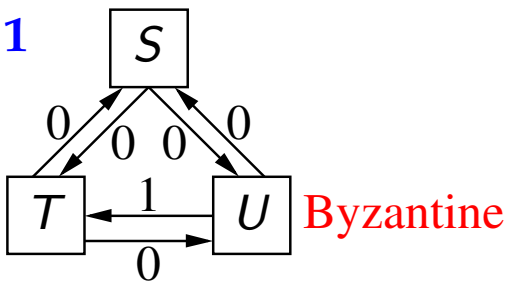
- Termination: Every correct process decides 0 or 1.
- Agreement: All correct processes decide the same value.
- Dependence: If the general is correct, it decides x_g.

Impossibility of $\lceil \frac{N}{3} \rceil$-Byzantine broadcast

Theorem: Let $k \geq \frac{N}{3}$. There is no k-Byzantine broadcast algorithm for synchronous networks.

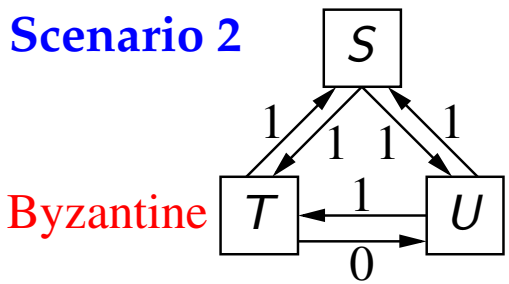
Proof: Divide the processes into three sets S, T and U with each $\leq k$ elements. Let $g \in S$.

Scenario 1



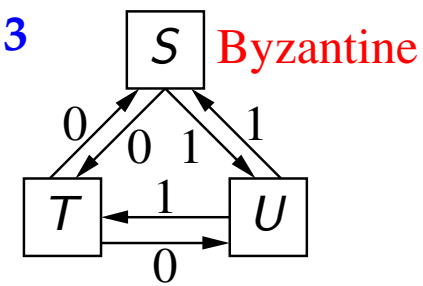
The processes in S and T decide 0

Scenario 2



The processes in S and U decide 1

Scenario 3



The processes in T decide 0 and in U decide 1

Lamport-Shostak-Pease Byzantine broadcast algorithm

$Broadcast_g(N, k)$ terminates after $k + 1$ pulses.

Pulse 1: The general g broadcasts and decides x_g.

If a lieutenant q receives b from g then $x_q \leftarrow b$, else $x_q \leftarrow 0$.

If $k = 0$: each lieutenant q decides x_q.

If $k > 0$: for each lieutenant p, each (correct) lieutenant q takes part in $Broadcast_p(N-1, k-1)$ in pulse 2 (g is excluded).

Pulse $k + 1$ ($k > 0$):

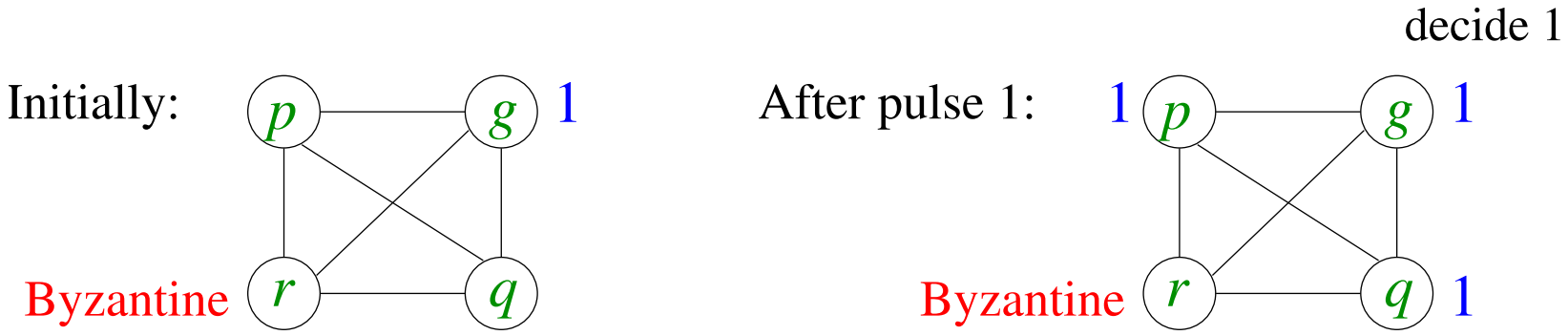
Lieutenant q has, for each lieutenant p, computed a value in $Broadcast_p(N-1, k-1)$; it stores this value in $M_q[p]$.

$x_q \leftarrow major(M_q)$; lieutenant q decides x_q.

(*major* maps each list over $\{0, 1\}$ to 0 or 1, such that if more than half of the elements in the list M are b, then $major(M) = b$.)

Lamport-Shostak-Pease broadcast alg. - Example 1

$N = 4$ and $k = 1$; general correct.



After pulse 1, g has decided 1, and the correct lieutenants p and q carry the value 1.

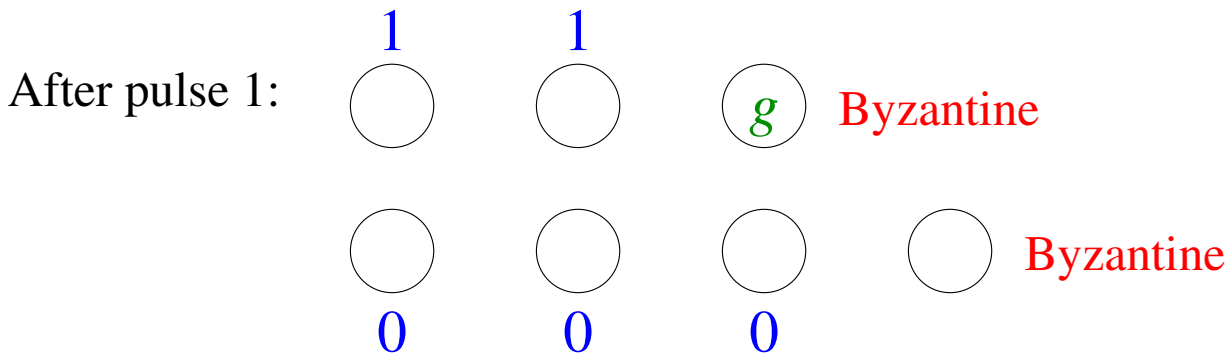
Consider the sub-network without g.

In $Broadcast_p(3, 0)$ and $Broadcast_q(3, 0)$, p and q both compute 1, while in $Broadcast_r(3, 0)$ they may compute an arbitrary value.

So p and q both build a list $[1, 1, \_]$, and decide 1.

Lamport-Shostak-Pease broadcast alg. - Example 2

$N = 7$ and $k = 2$; general Byzantine. (Channels are omitted.)

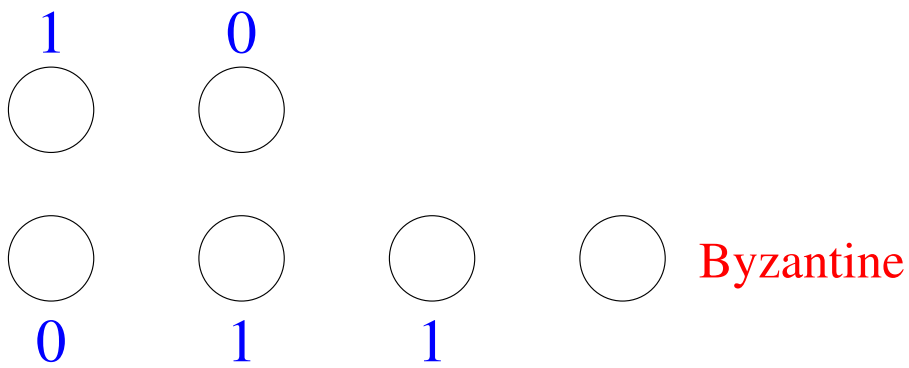


$k - 1 < \frac{N-1}{3}$, so by induction, the recursive calls $Broadcast_p(6, 1)$ lead to the same list $M = [1, 1, 0, 0, 0, b]$, for some $b \in \{0, 1\}$, at all correct lieutenants.

So in $Broadcast_g(7, 2)$, they all decide $major(M)$.

Lamport-Shostak-Pease broadcast alg. - Example 2

For instance, in $Broadcast_p(6, 1)$ with p the Byzantine lieutenant, the following values could be distributed by p.



Then the five subcalls $Broadcast_q(5, 0)$, for the correct lieutenants q, would at each correct lieutenant lead to the list $[1, 0, 0, 1, 1]$.

So in that case $b = major([1, 0, 0, 1, 1]) = 1$.

Questions

Question: Draw a tree of all recursive subcalls of $Broadcast_g(7, 2)$.

Question: Consider $Broadcast_g(N, 1)$, with a correct general g.

Let fewer than $\frac{N-1}{2}$ lieutenants be Byzantine.

Argue that all correct processes decide x_g.

Lamport-Shostak-Pease broadcast alg. - Correctness

Lemma: If the general g is correct, and $< \frac{N-k}{2}$ lieutenants are Byzantine, then in $Broadcast_g(N, k)$ all correct processes decide x_g.

Proof: By induction on k. Case $k = 0$ is trivial, because g is correct.

Let $k > 0$.

Since g is correct, in pulse 1, at all correct lieutenants p, $x_p \leftarrow x_g$.

Since $\frac{(N-1)-(k-1)}{2} = \frac{N-k}{2}$, by induction, for all correct lieutenants p, in $Broadcast_p(N-1, k-1)$ the value $x_p = x_g$ is computed.

Since a majority of the lieutenants is correct (because $k > 0$), in pulse $k+1$, at each correct lieutenant p, $x_p \leftarrow major(M_p) = x_g$.

Lamport-Shostak-Pease broadcast alg. - Correctness

Theorem: Let $k < \frac{N}{3}$. $Broadcast_g(N, k)$ is an (always terminating) k-Byzantine broadcast algorithm for synchronous networks.

Proof: By induction on k.

If g is correct, the theorem follows from the lemma and $k < \frac{N}{3}$.

Let g be Byzantine (so $k > 0$). Then $\leq k-1$ lieutenants are Byzantine.

Since $k-1 < \frac{N-1}{3}$, by induction, for every lieutenant p, all correct lieutenants compute in $Broadcast_p(N-1, k-1)$ the *same* value.

Hence, all correct lieutenants compute the *same* list M.

So in pulse $k+1$, all correct lieutenants decide $major(M)$.

Partial synchrony

A synchronous system can be obtained if local clocks have known *bounded drift*, and there is a known *upper bound on network latency*.

In a partially synchronous system,

- the bounds on the inaccuracy of local clocks and network latency are unknown, or
- these bounds are known, but only valid from some unknown point in time.

Dwork, Lynch and Stockmeyer showed that, for $k < \frac{N}{3}$, there is a k-Byzantine broadcast algorithm for partially synchronous systems.

These ideas are at the core of the *Paxos* consensus protocol.

Public-key cryptosystems

Given a large message domain $\mathcal{M}$.

A public-key cryptosystem consists of functions $S_q, P_q : \mathcal{M} \to \mathcal{M}$, for each process q, with

$$S_q(P_q(m)) \;=\; P_q(S_q(m)) \;=\; m \qquad \text{for all } m \in \mathcal{M}.$$

S_q is kept *secret*, P_q is made *public*.

Underlying assumption: Computing S_q from P_q is very expensive.

p sends a *secret* message m to q: $P_q(m)$

p sends a *signed* message m to q: $\langle m, S_p(m) \rangle$

Such signatures guarantee that Byzantine processes can't lie about the messages they have received.

Lamport-Shostak-Pease authentication algorithm

Pulse 1: The general broadcasts $\langle x_g, (S_g(x_g), g)\rangle$, and decides x_g.

Pulse i: If a lieutenant q receives a message $\langle v, (\sigma_1, p_1) : \cdots : (\sigma_i, p_i)\rangle$ that is valid, i.e.:

- $p_1 = g$
- $p_1, \ldots, p_i, q$ are distinct
- $P_{p_k}(\sigma_k) = v$ for all $k = 1, \ldots, i$

then q includes v in the set W_q.

If $i \leq k$, then in pulse $i + 1$, q sends to all other lieutenants

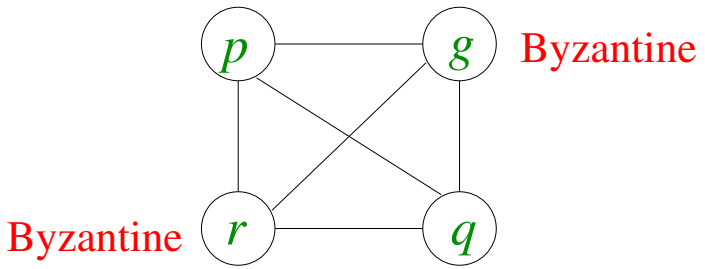
$$\langle v, (\sigma_1, p_1) : \cdots : (\sigma_i, p_i) : (S_q(v), q)\rangle$$

After pulse $k + 1$, each correct lieutenant p decides

v	if W_p is a singleton $\{v\}$, or
0	otherwise (the general is Byzantine)

Lamport-Shostak-Pease authentication alg. - Example

$N = 4$ and $k = 2$.



pulse 1: g sends $\langle 1, (S_g(1), g)\rangle$ to p and q
g sends $\langle 0, (S_g(0), g)\rangle$ to r
$W_p = W_q = \{1\}$

pulse 2: p broadcasts $\langle 1, (S_g(1), g) : (S_p(1), p)\rangle$
q broadcasts $\langle 1, (S_g(1), g) : (S_q(1), q)\rangle$
r sends $\langle 0, (S_g(0), g) : (S_r(0), r)\rangle$ to q
$W_p = \{1\}$ and $W_q = \{0, 1\}$

pulse 3: q broadcasts $\langle 0, (S_g(0), g) : (S_r(0), r) : (S_q(0), q)\rangle$
$W_p = W_q = \{0, 1\}$
p and q decide 0

Lamport-Shostak-Pease authentication alg. - Correctness

Theorem: The Lamport-Shostak-Pease authentication algorithm is an (always terminating) k-Byzantine broadcast algorithm, for any k.

Proof: If the general is correct, then owing to authentication, correct lieutenants q only add x_g to W_q. So they all decide x_g.

Let a correct lieutenant receive a valid message $\langle v, \ell \rangle$ in a pulse $\leq k$.

In the next pulse, it makes all correct lieutenants p add v to W_p.

Let a correct lieutenant receive a valid message $\langle v, \ell \rangle$ in pulse $k+1$.

Since ℓ has length $k+1$, it contains a correct q.

Then q received a valid message $\langle v, \_ \rangle$ in a pulse $\leq k$.

In the next pulse, q made all correct lieutenants p add v to W_p.

So after pulse $k+1$, W_p is the same for all correct lieutenants p.

Lamport-Shostak-Pease authentication alg. - Optimization

Dolev-Strong optimization: Each correct lieutenant broadcasts at most two messages, with different values.

Because when it has broadcast two different values, all correct lieutenants are certain to decide 0.

Lecture in a nutshell

ρ-bounded local clocks with precision δ

synchronous network

Byzantine broadcast

no k-Byzantine broadcast if $k \geq \frac{N}{3}$

Lamport-Shostak-Pease broadcast algorithm if $k < \frac{N}{3}$

Lamport-Shostak-Pease authentication algorithm

Mutual exclusion

Processes contend to enter their *critical section*.

A process (allowed to be) in its critical section is called *privileged*.

For each computation we require:

Mutual exclusion: Always at most one process is privileged.

Starvation-freeness: If a process p tries to enter its critical section, and no process stays privileged forever, then p eventually becomes privileged.

Applications: Distributed shared memory, replicated data, atomic commit.

Mutual exclusion with message passing

Mutual exclusion algorithms with *message passing* are generally based on one of the following paradigms.

- Leader election: A process that wants to become privileged sends a request to the leader.
- Token passing: The process holding the token is privileged.
- Logical clock: Requests to enter a critical section are prioritized by means of logical time stamps.
- Quorum: To become privileged, a process needs permission from a quorum of processes.

  Each pair of quorums has a non-empty intersection.

Ricart-Agrawala algorithm

When a process p_i wants to access its critical section, it sends $request(ts_i, i)$ to all other processes, with ts_i its logical time stamp.

When p_j receives this request, it sends permission to p_i as soon as :

- p_j isn't privileged, and
- p_j doesn't have a pending request with time stamp ts_j where $(ts_j, j) < (ts_i, i)$ (lexicographical order).

p_i enters its critical section when it has received permission from all other processes.

When p_i exits its critical section, it sends permission to all pending requests.

Ricart-Agrawala algorithm - Example 1

$N = 2$, and p_0 and p_1 both are at logical time 0.

p_1 sends $request(1, 1)$ to p_0.

When p_0 receives this message, it sends permission to p_1, setting the time at p_0 to 2.

p_0 sends $request(2, 0)$ to p_1.

When p_1 receives this message, it doesn't send permission to p_0, because $(1, 1) < (2, 0)$.

p_1 receives permission from p_0, and enters its critical section.

Ricart-Agrawala algorithm - Example 2

$N = 2$, and p_0 and p_1 both are at logical time 0.

p_1 sends $request(1, 1)$ to p_0, and p_0 sends $request(1, 0)$ to p_1.

When p_0 receives the request from p_1,
it doesn't send permission to p_1, because $(1, 0) < (1, 1)$.

When p_1 receives the request from p_0,
it sends permission to p_0, because $(1, 0) < (1, 1)$.

p_0 and p_1 both set their logical time to 2.

p_0 receives permission from p_1, and enters its critical section.

Ricart-Agrawala algorithm - Correctness

Mutual exclusion: When p sends permission to q:

- p isn't privileged; and
- p won't get permission from q to enter its critical section until q has entered and left its critical section.

  (Because p's pending or future request is larger than q's current request.)

Starvation-freeness: Each request will eventually become the smallest request in the network.

Ricart-Agrawala algorithm - Optimization

Drawback: High message overhead, because requests must be sent to all other processes.

Carvalho-Roucairol optimization: After a process q has exited its critical section, q only needs to send requests to the processes that q has sent permission to since this exit.

Suppose q is waiting for permissions and didn't send a request to p.

If p sends a request to q that is smaller than q's request, then q sends both permission and a request to p.

This optimization is correct since for each pair of distinct processes, at least one must ask permission from the other.

Question

Let first p_0 and then p_1 become privileged.

Next they want to become privileged again.

Which scenario's are possible, if the Carvalho-Roucairol optimization is employed ?

Answer: p_0 needs permission from p_1, but not vice versa.

If p_0's request reaches p_1 before it wants to become privileged again, then p_1 sends permission and later a request to p_0.

Else p_1 enters its critical section, and answers p_0's request only after exiting the critical section.

Raymond's algorithm

Given an undirected network, with a sink tree.

At any time, the root, holding a token, is privileged.

Each process maintains a FIFO queue, which can contain id's of its children, and its own id. Initially, this queue is empty.

Queue maintenance:

- When a non-root wants to enter its critical section, it adds its id to its own queue.
- When a non-root gets a new head at its (non-empty) queue, it asks its parent for the token.
- When a process receives a request for the token from a child, it adds this child to its queue.

Raymond's algorithm

When the root exits its critical section (and its queue is non-empty),

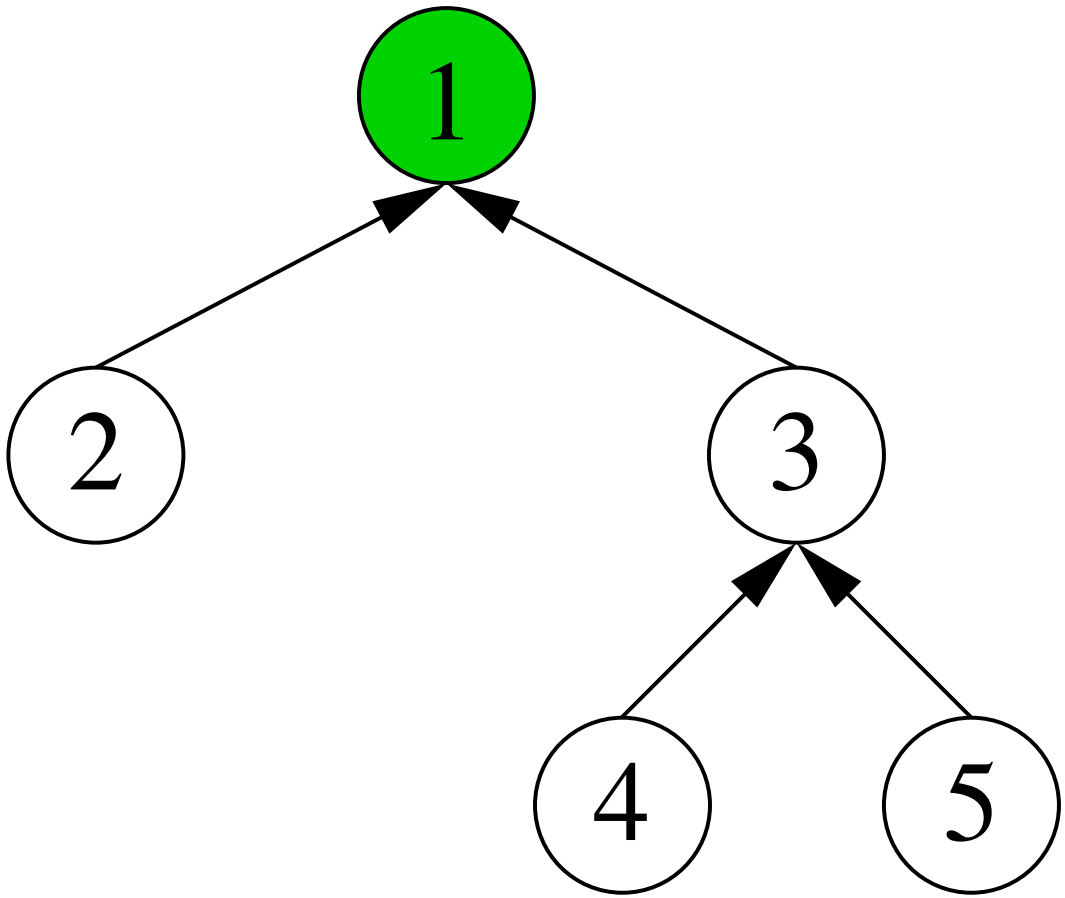
- it sends the token to the process q at the head of its queue,
- makes q its parent, and
- removes q from the head of its queue.

Let p get the token from its parent, with q at the head of its queue:

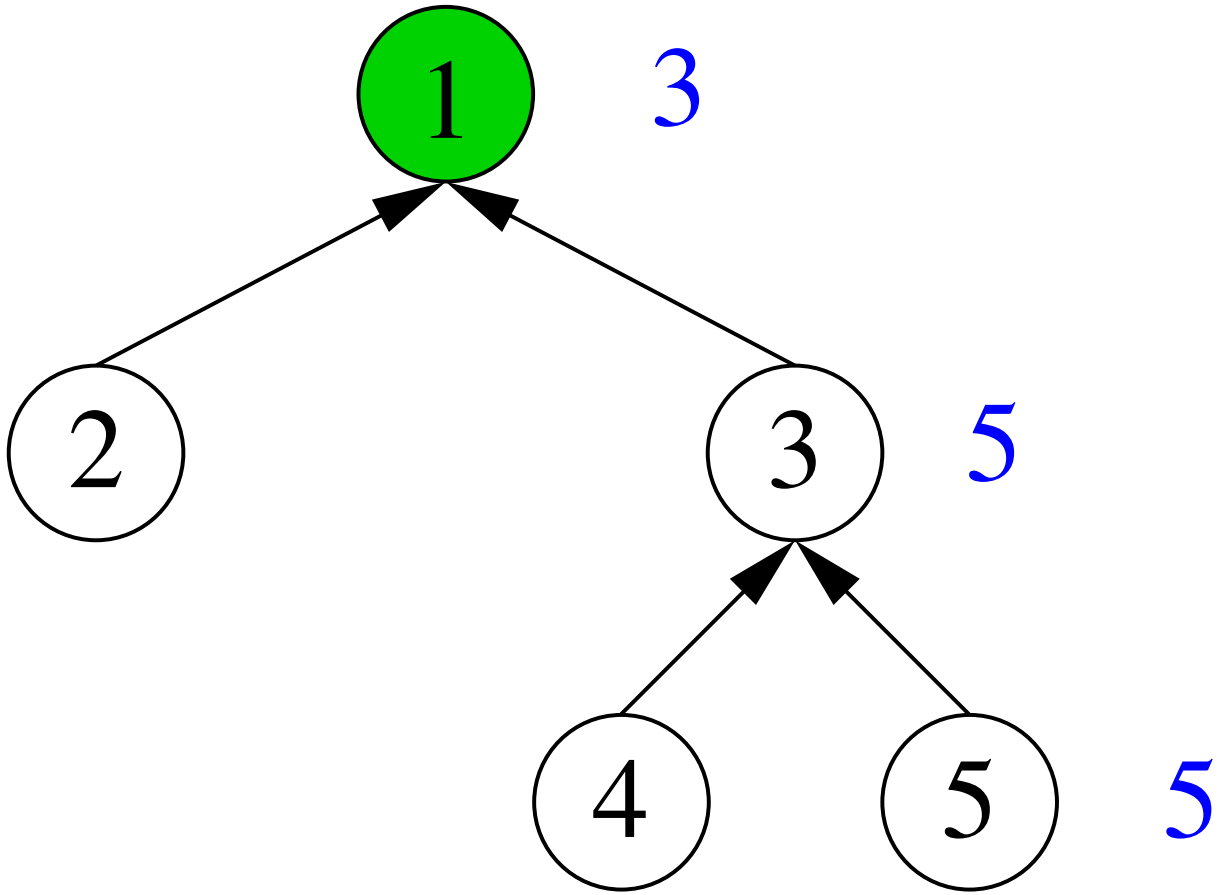
- If $q \neq p$, then p sends the token to q, and makes q its parent.
- If $q = p$, then p becomes the root
  (i.e., it has no parent, and is privileged).

In both cases, p removes q from the head of its queue.

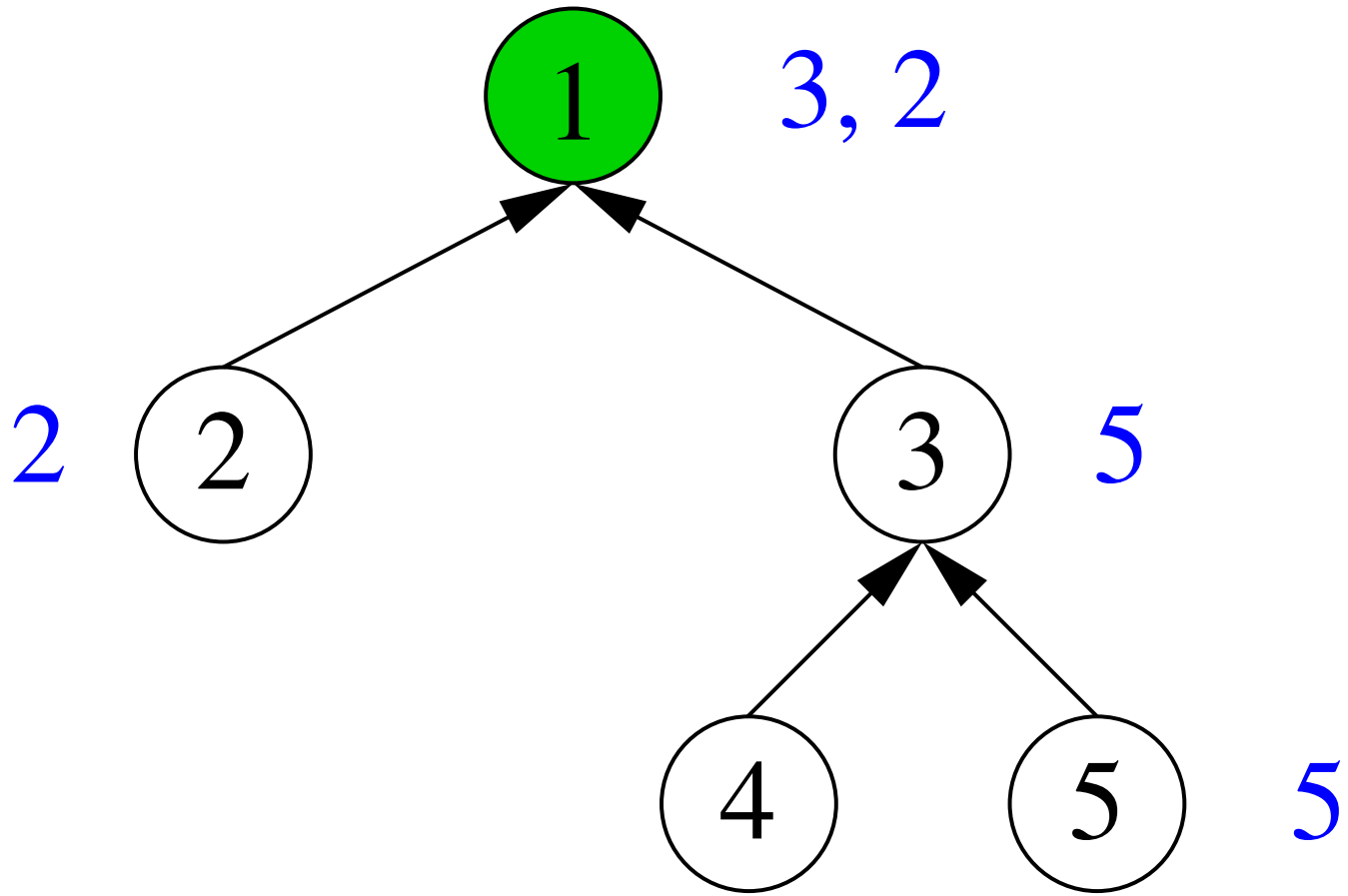
Raymond's algorithm - Example



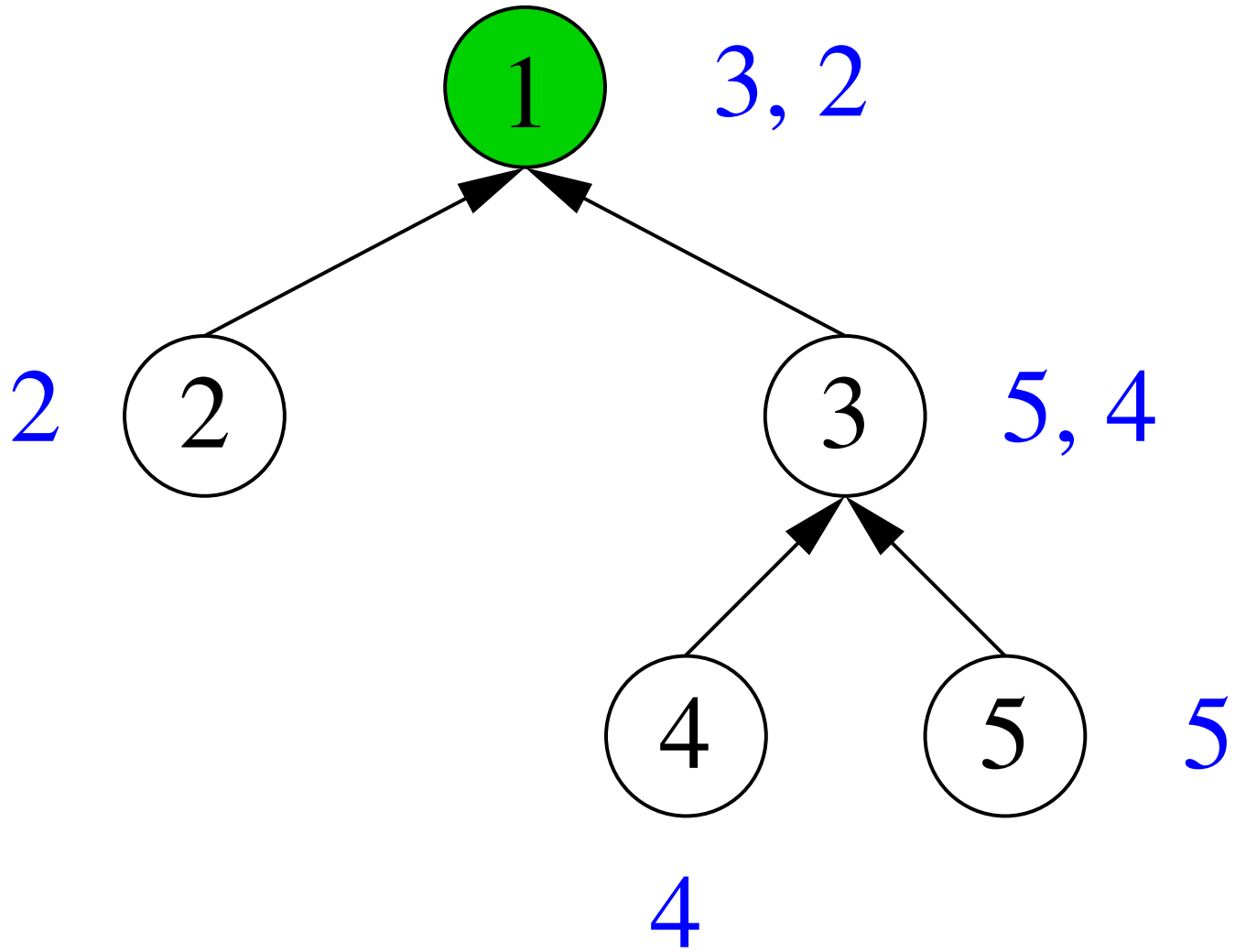
Raymond's algorithm - Example



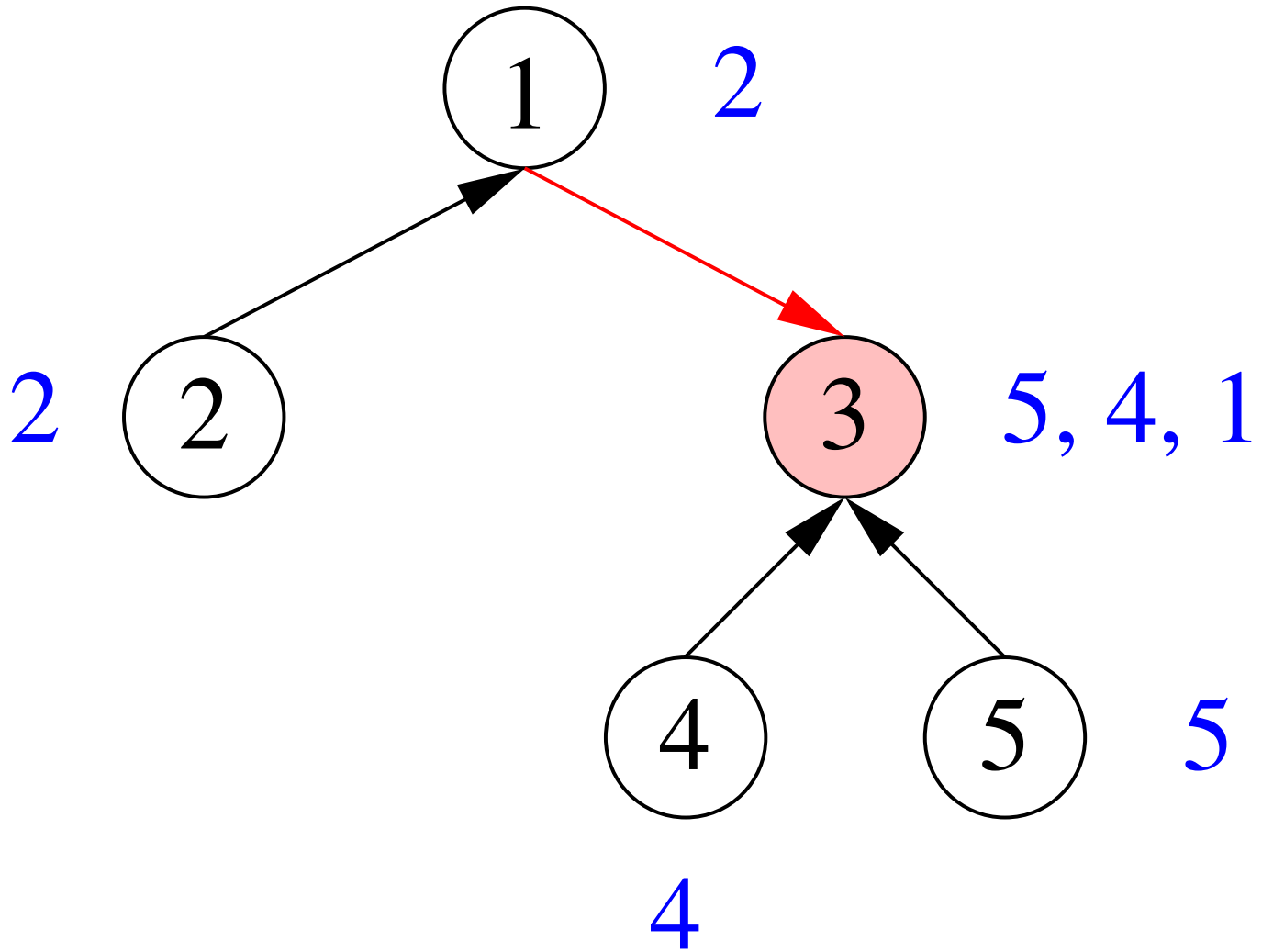
Raymond's algorithm - Example



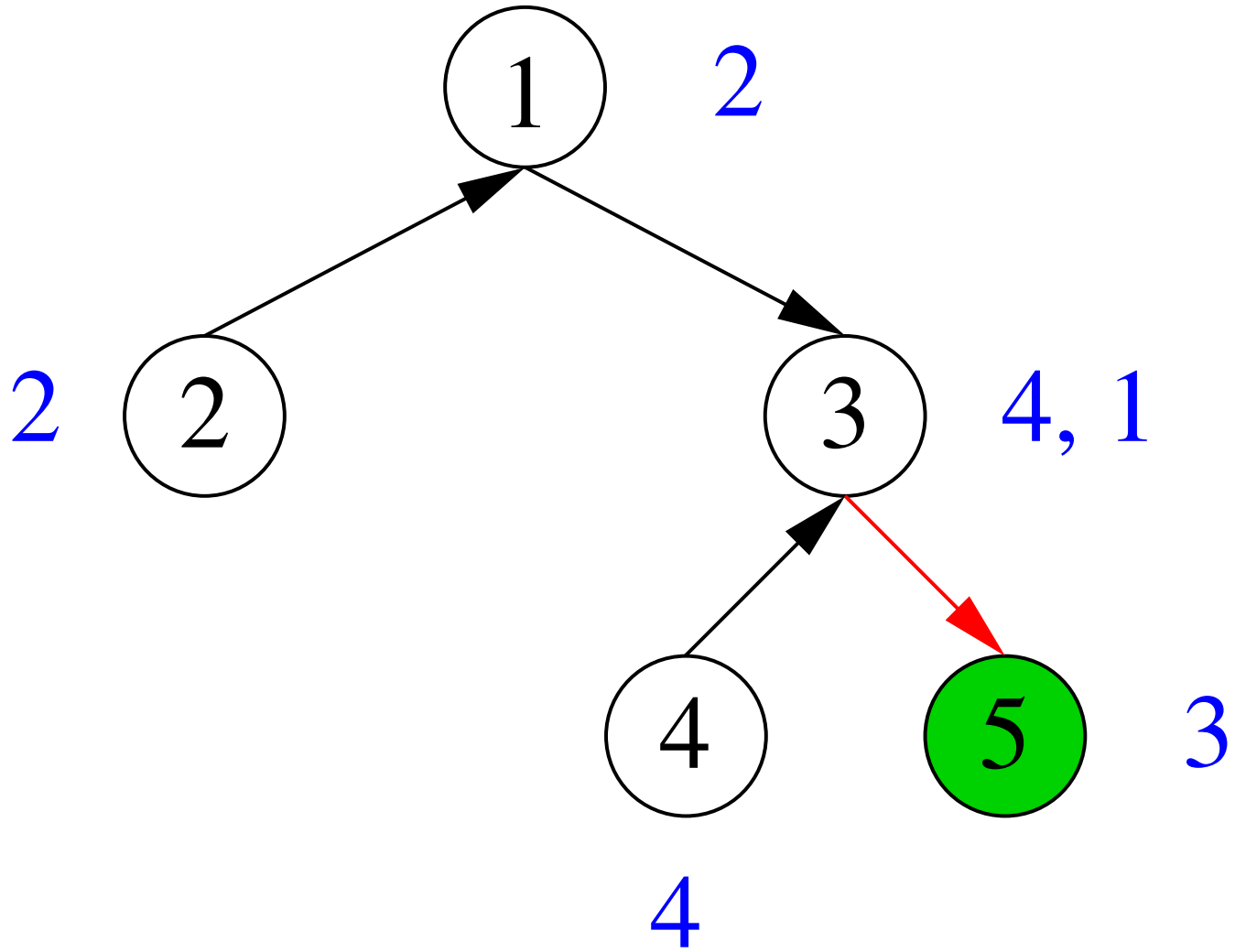
Raymond's algorithm - Example



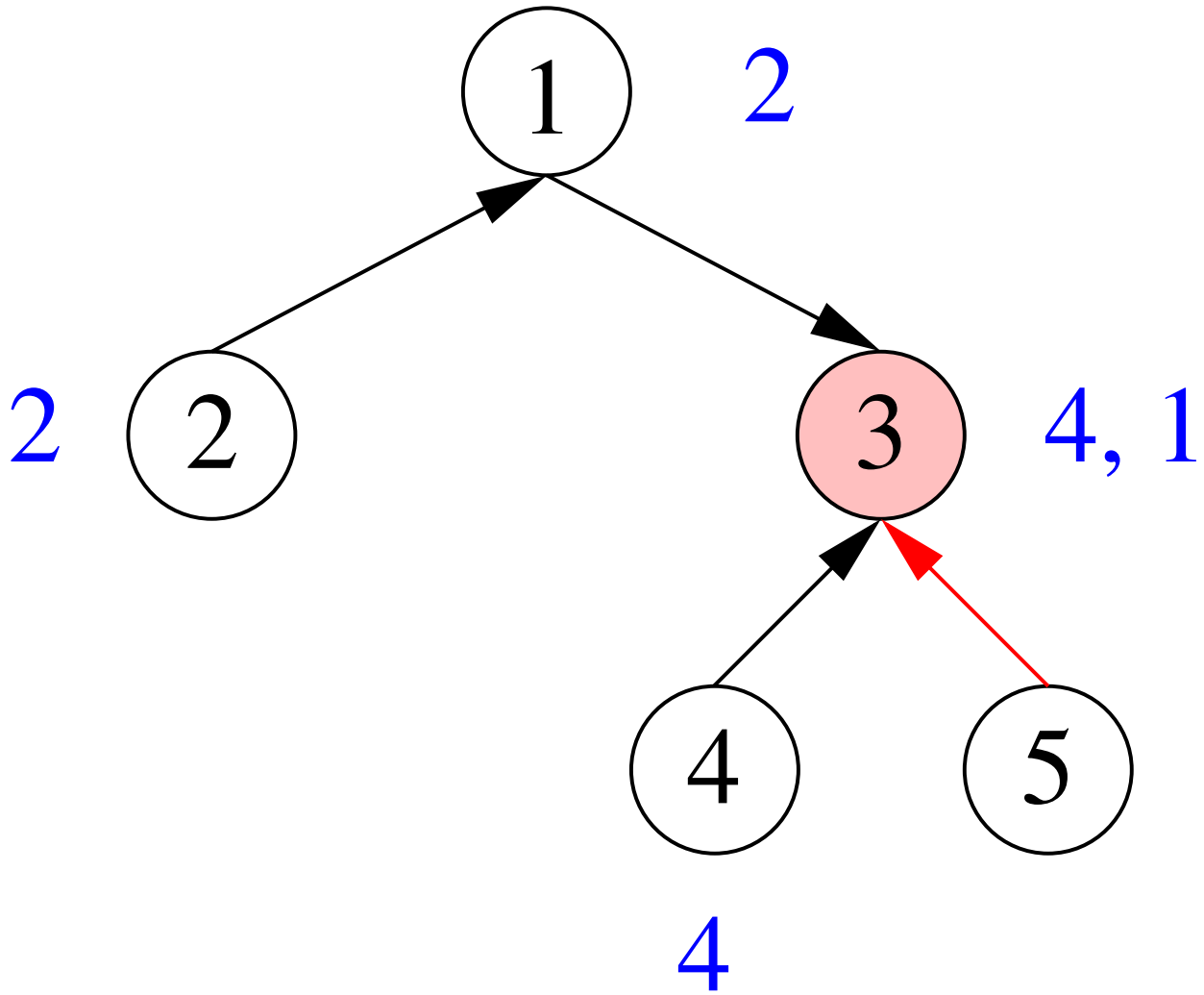
Raymond's algorithm - Example



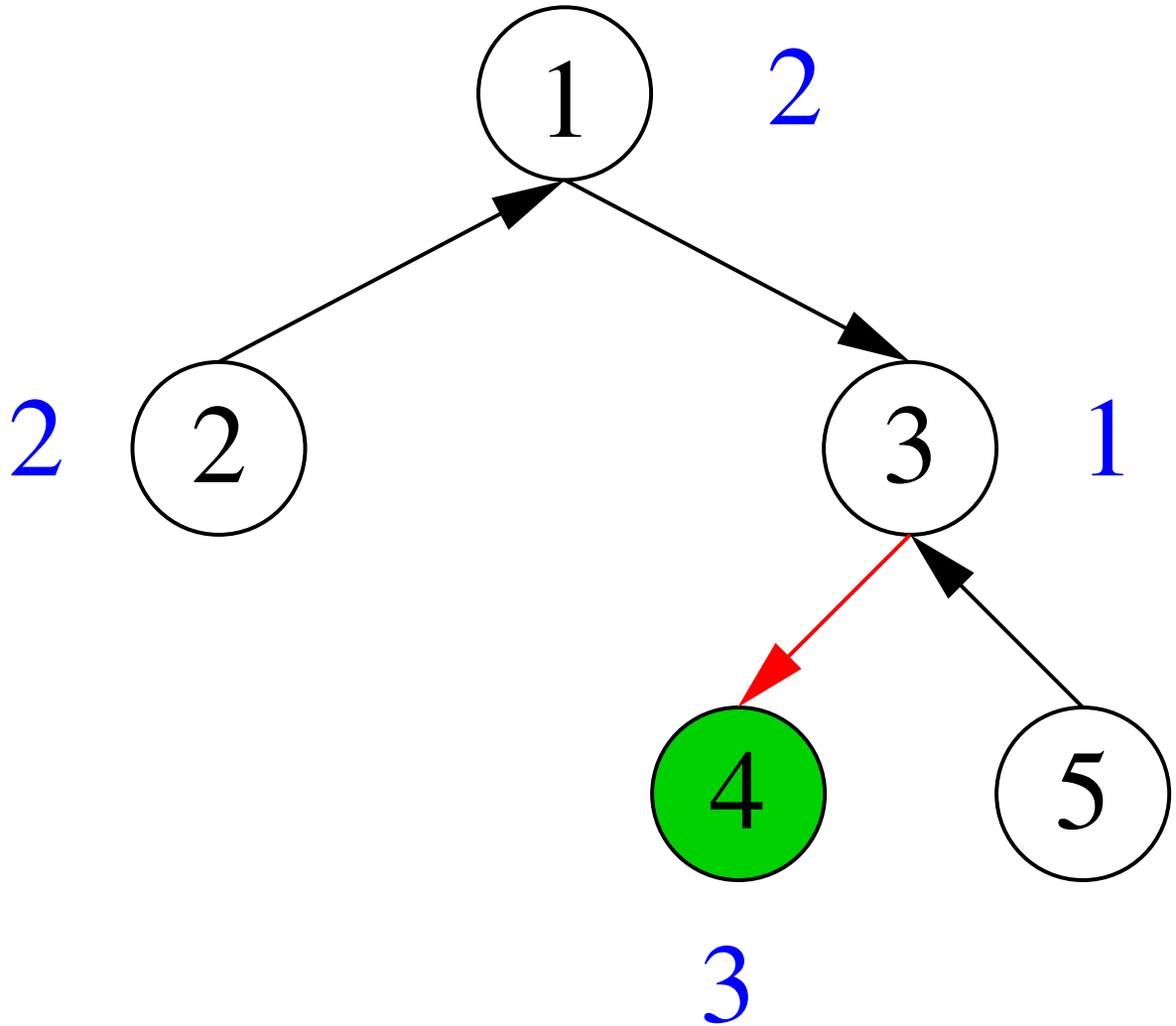
Raymond's algorithm - Example



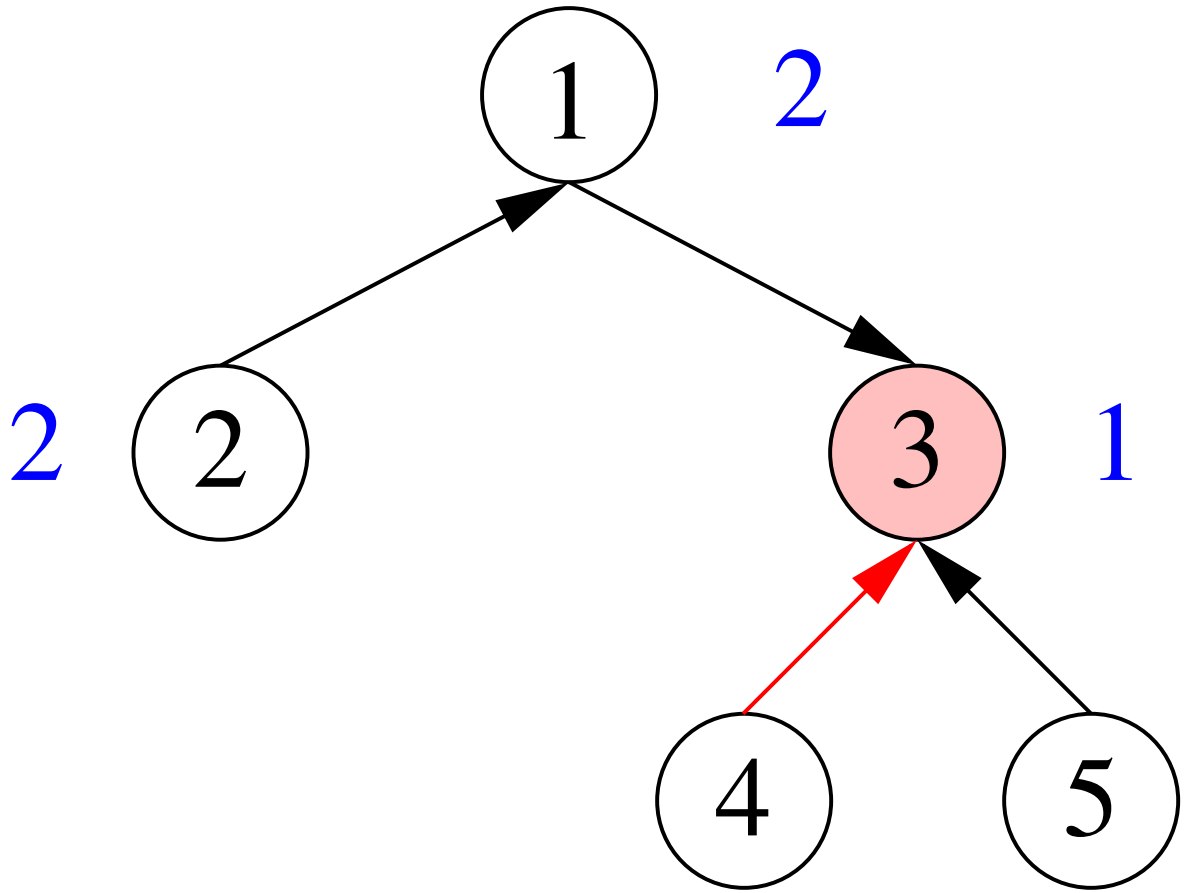
Raymond's algorithm - Example



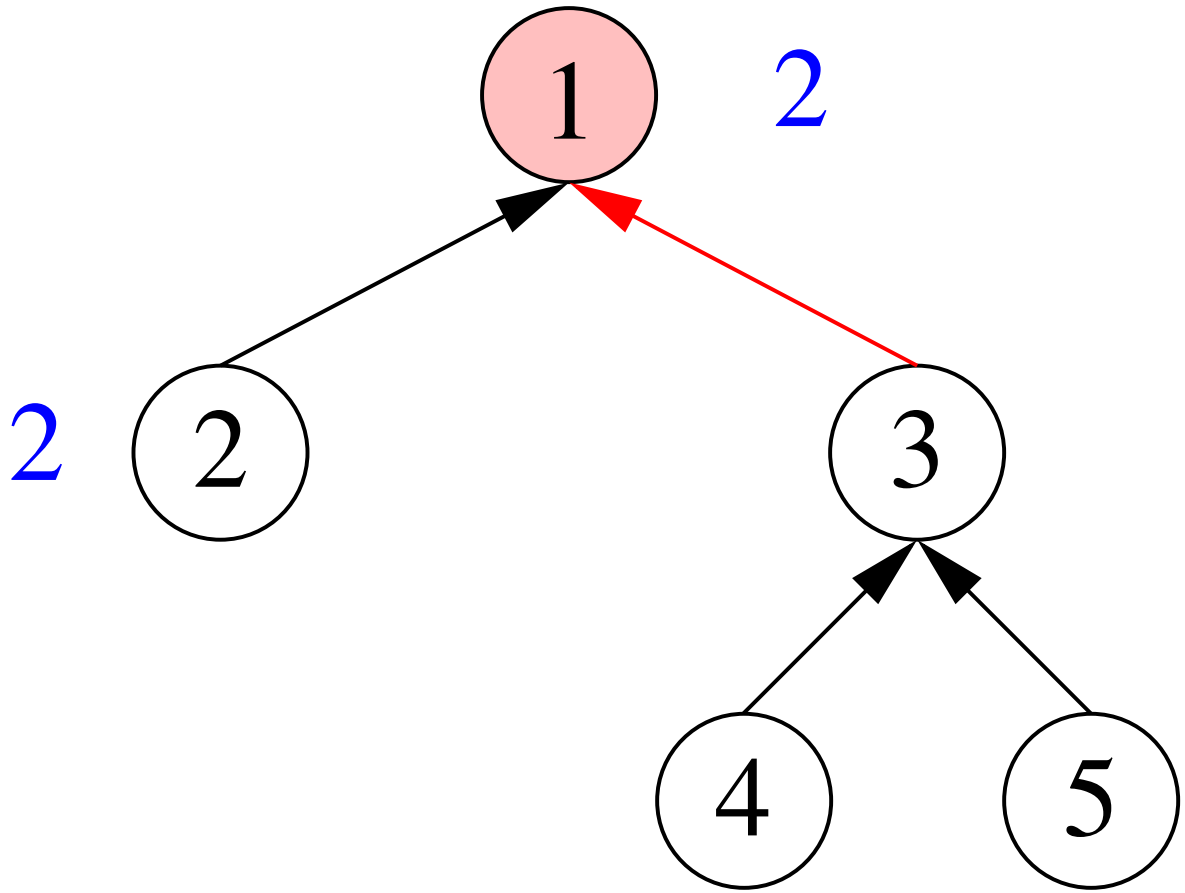
Raymond's algorithm - Example



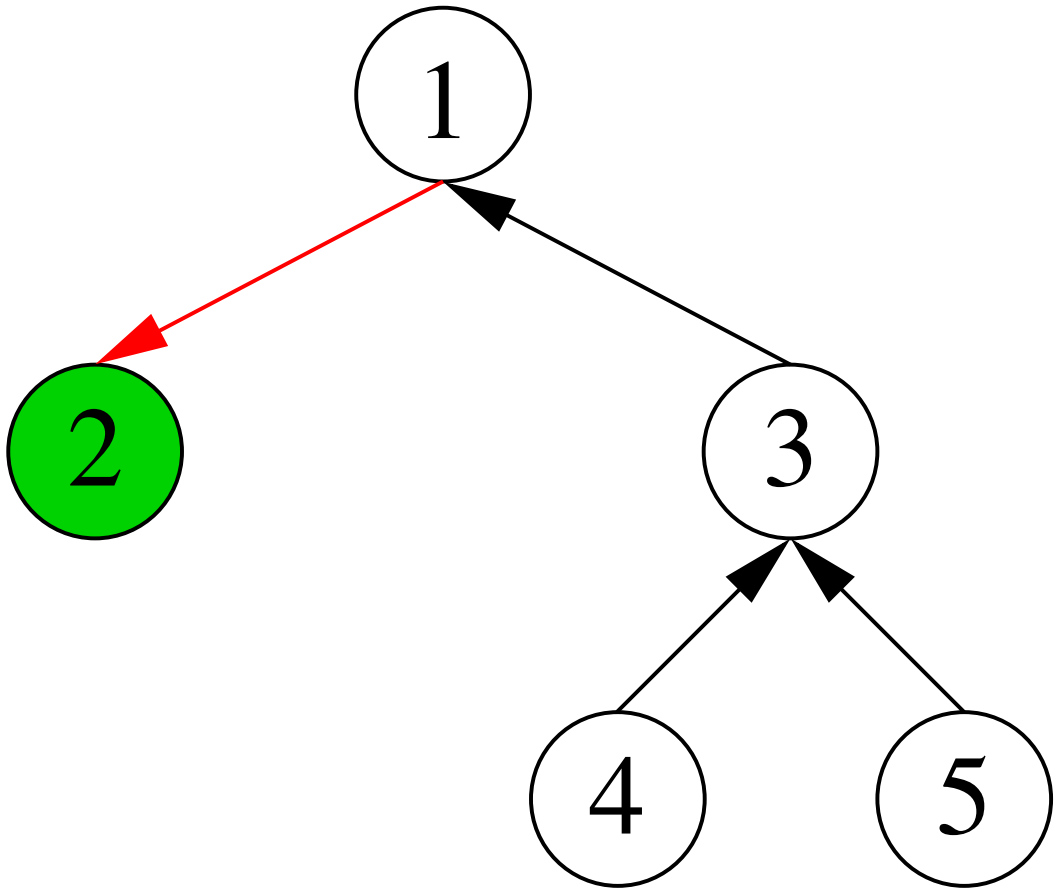
Raymond's algorithm - Example



Raymond's algorithm - Example



Raymond's algorithm - Example



Raymond's algorithm - Correctness

Raymond's algorithm provides mutual exclusion, because at all times there is at most one root.

Raymond's algorithm is starvation-free, because eventually each request in a queue moves to the head of this queue, and a chain of requests never contains a cycle.

Drawback: Sensitive to failures.

Question

What is the Achilles' heel of a mutual exclusion algorithm based on a leader ?

Answer: The leader is a single point of failure.

Agrawal-El Abbadi algorithm

To enter a critical section, permission from a quorum is required.

For simplicity we assume that $N = 2^k - 1$, for some $k > 1$.

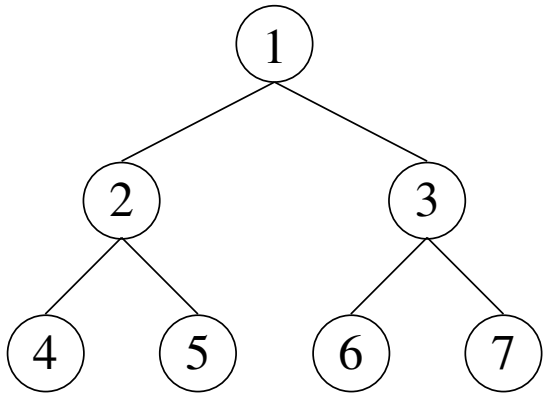
The processes are structured in a *binary tree* of depth $k - 1$.

A *quorum* consists of all processes on a path from the root to a leaf.

If a *non-leaf* p has crashed (or is unresponsive), permission is asked from all processes on two paths instead: from each child of p to a leaf.

Agrawal-El Abbadi algorithm - Example

Example: Let $N = 7$.



Possible quorums are:

- $\{1, 2, 4\}$, $\{1, 2, 5\}$, $\{1, 3, 6\}$, $\{1, 3, 7\}$
- if 1 crashed: $\{2, 4, 3, 6\}$, $\{2, 5, 3, 6\}$, $\{2, 4, 3, 7\}$, $\{2, 5, 3, 7\}$
- if 2 crashed: $\{1, 4, 5\}$ (and $\{1, 3, 6\}$, $\{1, 3, 7\}$)
- if 3 crashed: $\{1, 6, 7\}$ (and $\{1, 2, 4\}$, $\{1, 2, 5\}$)

Question: What are the quorums if 1,2 crashed? And if 1,2,3 crashed? And if 1,2,4 crashed?

Agrawal-El Abbadi algorithm

A process p that wants to enter its critical section, places the root of the tree in a queue.

p repeatedly tries to get permission from the head r of its queue.

If successful, r is removed from p's queue.

If r is a non-leaf, *one* of r's children is appended to p's queue.

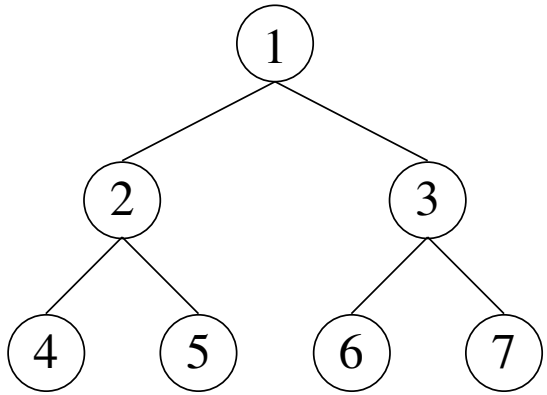
If non-leaf r has crashed, it is removed from p's queue, and *both* of r's children are appended at the end of the queue (in a fixed order, to avoid deadlocks).

If leaf r has crashed, p *aborts* its attempt to become privileged.

When p's queue becomes empty, it enters its critical section.

After exiting its critical section, p informs all processes in the quorum that their permission to p can be withdrawn.

Agrawal-El Abbadi algorithm - Example



p and q concurrently want to enter their critical section.

p gets permission from 1, and wants permission from 3.

1 crashes, and q now wants permission from 2 and 3.

q gets permission from 2, and appends 4 to its queue.

q obtains permission from 3, and appends 7 to its queue.

3 crashes, and p now wants permission from 6 and 7.

q gets permission from 4, and now wants permission from 7.

p gets permission from both 6 and 7, and enters its critical section.

Agrawal-El Abbadi algorithm - Mutual exclusion

We prove, by induction on depth k, that each pair of quorums has a non-empty intersection, and so mutual exclusion is guaranteed.

A quorum with 1 contains a quorum in one of the subtrees below 1, while a quorum without 1 contains a quorum in both subtrees below 1.

- If two quorums both contain 1, we are done.
- If two quorums both don't contain 1, then by induction they have elements in common in the two subtrees below process 1.
- Suppose quorum Q contains 1, while quorum Q' doesn't.

  Then Q contains a quorum in one of the subtrees below 1, and Q' also contains a quorum in this subtree.

  By induction, they have an element in common in this subtree.

Agrawal-El Abbadi algorithm - Deadlock-freeness

In case of a crashed process, let its left child be put before its right child in the queue of a process that wants to become privileged.

Let a process p at depth d be greater than any process

- at a depth $> d$ in the binary tree, or
- at depth d and more to the right than p in the binary tree.

A process with permission from r, never needs permission from a $q < r$.

This guarantees that, in case some leave is responsive, eventually some process will become privileged.

Starvation can happen, if a process waits for a permission infinitely long. (This can be easily resolved.)

Self-stabilization

All configurations are initial configurations.

An algorithm is self-stabilizing if every computation eventually reaches a correct configuration.

Advantages:

- fault tolerance
- straightforward initialization

Self-stabilizing *operating systems* and *databases* have been developed.

Self-stabilization - Shared memory

In a *message-passing* setting, processes might all be initialized in a state where they are waiting for a message.

Then the self-stabilizing algorithm wouldn't exhibit any behavior.

Therefore, in self-stabilizing algorithms, processes communicate via variables in *shared memory*.

We assume that a process can read the variables of its neighbors.

Dijkstra's self-stabilizing token ring

Processes $p_0, \ldots, p_{N-1}$ form a directed ring.

Each p_i holds a value $x_i \in \{0, \ldots, K-1\}$ with $K \geq N$.

- p_i for each $i = 1, \ldots, N-1$ is privileged if $x_i \neq x_{i-1}$.
- p_0 is privileged if $x_0 = x_{N-1}$.

Each privileged process is allowed to change its value, causing the loss of its privilege:

- $x_i \leftarrow x_{i-1}$ when $x_i \neq x_{i-1}$, for each $i = 1, \ldots, N-1$
- $x_0 \leftarrow (x_0 + 1) \bmod K$ when $x_0 = x_{N-1}$

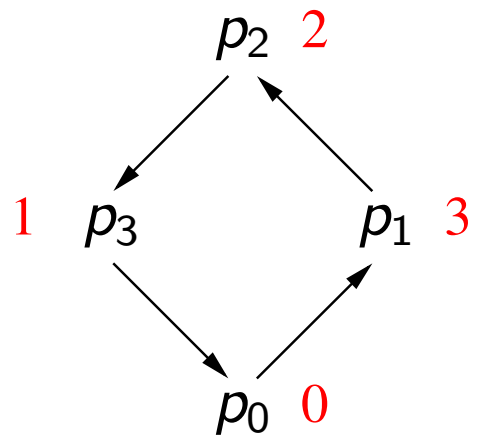
If $K \geq N$, then Dijkstra's token ring *self-stabilizes*.

That is, each computation eventually satisfies *mutual exclusion*.

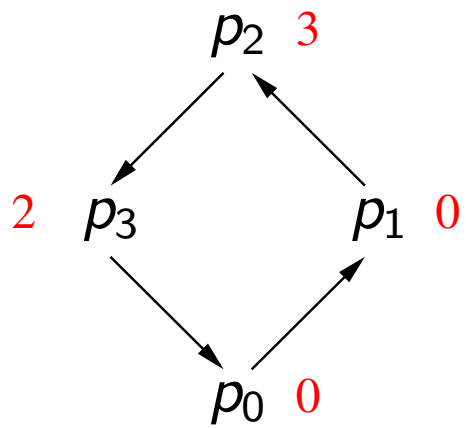
Moreover, Dijkstra's token ring is *starvation-free*.

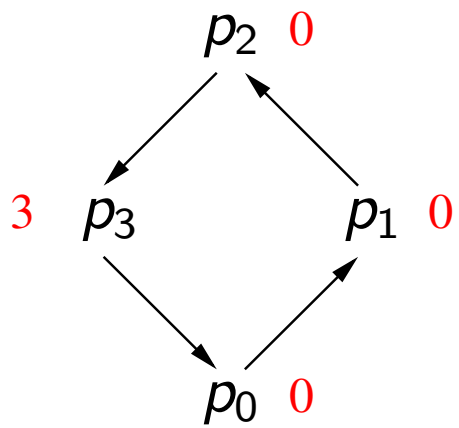
Dijkstra's token ring - Example

Let $N = K = 4$. Consider the initial configuration



It isn't hard to see that the ring self-stabilizes. For instance,





Dijkstra's token ring - Correctness

Theorem: If $K \geq N$, then Dijkstra's token ring self-stabilizes.

Proof: In each configuration at least one process is privileged.

An event never increases the number of privileged processes.

Consider an (infinite) computation. After at most $\frac{1}{2}(N-1)N$ events at $p_1, \ldots, p_{N-1}$, an event must happen at p_0.

So during the computation, x_0 ranges over all values in $\{0, \ldots, K-1\}$.

Since $p_1, \ldots, p_{N-1}$ only copy values, they stick to their $\leq N-1$ values as long as x_0 equals x_i for some $i = 1, \ldots, N-1$.

Since $K \geq N$, at some point, $x_0 \neq x_i$ for all $i = 1, \ldots, N-1$.

The next time p_0 becomes privileged, clearly $x_i = x_0$ for all i.

So then mutual exclusion has been achieved.

Question

Let $N \geq 3$. Argue that Dijkstra's token ring self-stabilizes if $K = N - 1$.

This lower bound for K is sharp! (See the next slide.)

Answer: Consider any computation.

At some moment, p_{N-1} copies the value from p_{N-2}.

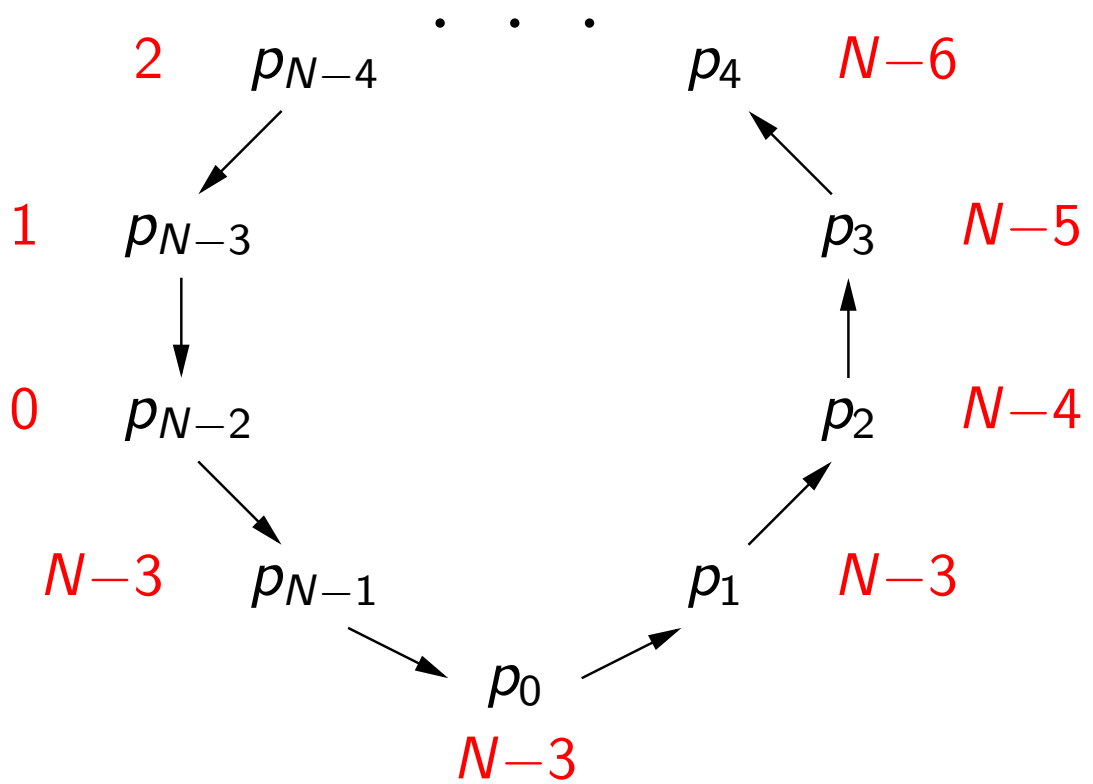
Then $p_1, \ldots, p_{N-1}$ hold $\leq N - 2$ different values (because $N \geq 3$).

Since $p_1, \ldots, p_{N-1}$ only copy values, they hold these $\leq N - 2$ values as long as x_0 equals x_i for some $i = 1, \ldots, N - 1$.

Since $K \geq N - 1$, at some point, $x_0 \neq x_i$ for all $i = 1, \ldots, N - 1$.

Dijkstra's token ring - Non-stabilization if $K = N - 2$

Example: Let $N \geq 4$ and $K = N - 2$, and consider the following initial configuration.



It doesn't always self-stabilize.

Afek-Kutten-Yung self-stabilizing spanning tree algorithm

We compute a spanning tree in an undirected network.

As always, each process is supposed to have a unique id.

The process with the largest id becomes the *root*.

Each process p maintains the following variables :

$parent_p$: its parent in the spanning tree

$root_p$: the root of the spanning tree

$dist_p$: its distance from the root via the spanning tree

Afek-Kutten-Yung spanning tree algorithm - Complications

Due to arbitrary initialization, there are three complications.

Complication 1: Multiple processes may consider themselves root.

Complication 2: There may be a cycle in the spanning tree.

Complication 3: $root_p$ may not be the id of any process in the network.

Afek-Kutten-Yung spanning tree algorithm

A *non-root* p declares itself *root*, i.e.

$$parent_p \leftarrow \perp \qquad root_p \leftarrow p \qquad dist_p \leftarrow 0$$

if it detects an inconsistency in its *root* or *parent* value, or with the *root* or *dist* value of its parent :

- $root_p \leq p$, or
- $parent_p = \perp$, or
- $parent_p \neq \perp$, and $parent_p$ isn't a neighbor of p or $root_p \neq root_{parent_p}$ or $dist_p \neq dist_{parent_p} + 1$.

Question

Suppose that during an application of the Afek-Kutten-Yung algorithm, the created directed network contains a cycle with a "false" root.

Why is such a cycle always broken ?

Answer: At some p on this cycle, $dist_p \neq dist_{parent_p} + 1$.

So p declares itself root.

Afek-Kutten-Yung spanning tree algorithm

A *root* p makes a neighbor q its parent if $p < root_q$:

$$parent_p \leftarrow q \qquad root_p \leftarrow root_q \qquad dist_p \leftarrow dist_q + 1$$

Complication: Processes can infinitely often rejoin a component with a false root.

Afek-Kutten-Yung spanning tree alg. - Example

Given two processes 0 and 1.

$parent_0 = 1 \quad parent_1 = 0 \quad root_0 = root_1 = 2 \quad dist_0 = 0 \quad dist_1 = 1$

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:

$parent_0 \leftarrow \perp \qquad root_0 \leftarrow 0 \qquad dist_0 \leftarrow 0$

Since $root_0 < root_1$, 0 makes 1 its *parent*:

$parent_0 \leftarrow 1 \qquad root_0 \leftarrow 2 \qquad dist_0 \leftarrow 2$

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:

$parent_1 \leftarrow \perp \qquad root_1 \leftarrow 1 \qquad dist_1 \leftarrow 0$

Since $root_1 < root_0$, 1 makes 0 its *parent*:

$parent_1 \leftarrow 0 \qquad root_1 \leftarrow 2 \qquad dist_1 \leftarrow 3$ et cetera

Afek-Kutten-Yung spanning tree alg. - Join Requests

Before p makes q its parent, it must wait until q's component has a proper root. Therefore p first sends a *join request* to q.

This request is forwarded through q's component, toward the root of this component.

The root sends back an *ack* toward p, which retraces the path of the request.

Only when p receives this ack, it makes q its parent :

$$parent_p \leftarrow q \qquad root_p \leftarrow root_q \qquad dist_p \leftarrow dist_q + 1$$

Join requests are only forwarded between "consistent" processes.

Afek-Kutten-Yung spanning tree alg. - Example

Given two processes 0 and 1.

$parent_0 = 1 \quad parent_1 = 0 \quad root_0 = root_1 = 2 \quad dist_0 = dist_1 = 0$

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:

$parent_0 \leftarrow \perp \qquad root_0 \leftarrow 0 \qquad dist_0 \leftarrow 0$

Since $root_0 < root_1$, 0 sends a *join request* to 1.

This join request doesn't immediately trigger an ack.

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:

$parent_1 \leftarrow \perp \qquad root_1 \leftarrow 1 \qquad dist_1 \leftarrow 0$

Since 1 is now a proper root, it replies to the join request of 0 with an ack, and 0 makes 1 its *parent*:

$parent_0 \leftarrow 1 \qquad root_0 \leftarrow 1 \qquad dist_0 \leftarrow 1$

Afek-Kutten-Yung spanning tree alg. - Shared memory

A process can only be forwarding and awaiting an ack for at most one join request at a time.

(That's why in the previous example 1 can't send 0's join request on to 0.)

Communication is performed using *shared memory*, so join requests and ack's are encoded in shared variables.

The path of a join request is remembered in local variables.

For simplicity, join requests are here presented in a message passing framework with synchronous communication.

Afek-Kutten-Yung spanning tree alg. - Consistency check

Given a ring with processes p, q, r, and $s > p, q, r$.

Initially, p and q consider themselves root; r has p as parent and considers s the root.

Since $root_r > q$, q sends a join request to r.

Without the consistency check, r would forward this join request to p.

Since p considers itself root, it would send back an ack to q (via r), and q would make r its parent and consider s the root.

Since $root_r \neq root_p$, r makes itself root.

Now we would have a symmetrical configuration to the initial one.

Afek-Kutten-Yung spanning tree alg. - Correctness

Each component in the network with a false root has an inconsistency, so a process in this component will declare itself root.

Since processes can only be involved in one join request at a time, each join request is eventually acknowledged.

Since join requests are only passed on between consistent processes, processes can only finitely often join a component with a false root (each time due to improper initial values of local variables).

These observations imply that eventually false roots will disappear, the process with the largest id in the network will declare itself root, and the network converges to a spanning tree with this process as root.

Lecture in a nutshell

mutual exclusion

Ricart-Agrawala algorithm with a logical clock

Raymond's algorithm with token passing

Agrawal-El Abbadi algorithm with quorums

self-stabilization

Dijkstra's self-stabilizing mutual exclusion algorithm

Afek-Kutten-Yung self-stabilizing spanning tree algorithm

Edsger W. Dijkstra prize in distributed computing

2000: Lamport, *Time, clocks, and the ordering of events in a distributed system*, 1978

2001: Fischer, Lynch, Paterson, *Impossibility of distributed consensus with one faulty process*, 1985

2002: Dijkstra, *Self-stabilizing systems in spite of distributed control*, 1974

2004: Gallager, Humblet, Spira, *A distributed algorithm for minimum-weight spanning trees*, 1983

2005: Pease, Shostak, Lamport, *Reaching agreement in the presence of faults*, 1980

2007: Dwork, Lynch, Stockmeyer, *Consensus in the presence of partial synchrony*, 1988

2010: Chandra, Toueg, *Unreliable failure detectors for reliable distributed systems*, 1996

2014: Chandy, Lamport, *Distributed snapshots: determining global states of distributed systems*, 1985

2015: Ben-Or, *Another advantage of free choice: completely asynchronous agreement protocols*, 1983

The 2003, 2006, 2012 award winners are treated in *Concurrency & Multithreading*.