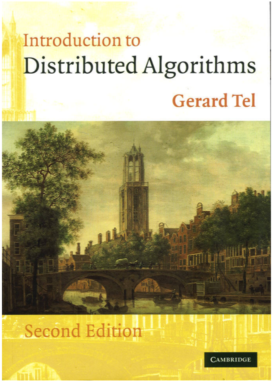
Distributed Algorithms



Gerard Tel
Introduction to Distributed Algorithms (2^{nd} edition)
Cambridge University Press, 2000

Set-Up of the Course

13 lectures:	Wan Fokkink room U342 email: wanf@few.vu.nl
11 exercise classes:	David van Moolenbroek one exercise on the board
bonus exercise sheet:	0.5 bonus for exam
exam:	written you may use (clean) copies of slides
homepage:	http://www.cs.vu.nl/∼tcs/da/ copies of the slides exercise sheets for the labs structure of the course schedule of lectures and exercise classes old exams

Distributed Systems

A distributed system is an interconnected collection of autonomous processes.

Motivation:

- information exchange (WAN)
- resource sharing (LAN)
- replication to increase reliability
- parallelization to increase performance
- modularity to improve design

Distributed Versus Uniprocessor

Distributed systems differ from uniprocessor systems in three aspects.

- *Lack of knowledge on the global state:* A process usually has no up-to-date knowledge on the local states of other processes. For example, deadlock detection becomes an issue.
- *Lack of a global time frame:* No total order on events by their temporal occurrence. For example, mutual exclusion becomes an issue.
- *Nondeterminism:* The execution of processes is usually nondeterministic, for example due to differences in execution speed.

Communication Paradigms

The two main paradigms to capture communication in a distributed system are message passing and variable sharing.
We will mainly consider message passing.

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.

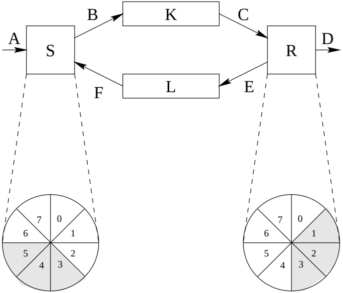
Communication Protocols

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

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

Examples:

- Alternating bit protocol
- Bounded retransmission protocol
- Sliding window protocols



Formal Framework

We first present a formal framework for distributed algorithms.

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

Transition Systems

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

The configuration evolves in discrete steps, called transitions.

A transition system consists of:

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

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

Executions

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

- $\gamma_0 \in \mathcal{I}$; and
- $\gamma_i \rightarrow \gamma_{i+1}$ for $i \geq 0$.

δ is reachable from γ if there is a sequence $\gamma = \gamma_0\,\gamma_1\,\gamma_2\,\cdots\,\gamma_k = \delta$ with $\gamma_i \rightarrow \gamma_{i+1}$ for $0 \leq i < k$.

δ is reachable if it is reachable from a $\gamma \in \mathcal{I}$.

States and Events

The configuration of a distributed algorithm is composed from the states at its processes.

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

Local Algorithm at a Process

For simplicity we assume different channels carry different messages.

The *local algorithm* at a process consists of:

- a set Z of states;
- a set I of initial states;
- a relation $\vdash^{\mathrm{i}}$ of internal events (c, d);
- a relation $\vdash^{\mathrm{s}}$ of send events (c, m, d); and
- a relation $\vdash^{\mathrm{r}}$ of receive events (c, m, d).

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

Asynchronous Communication

Let $p = (Z_p, I_p, \vdash^{\mathrm{i}}_p, \vdash^{\mathrm{s}}_p, \vdash^{\mathrm{r}}_p)$ for processes p.

Consider an asynchronous distributed algorithm $(p_1, \ldots, p_N)$.

$$\mathcal{C} = Z_{p_1} \times \cdots \times Z_{p_N} \times \mathbb{M}(\mathcal{M}) \qquad (\mathcal{M} \text{ is the set of messages})$$

$$\mathcal{I} = I_{p_1} \times \cdots \times I_{p_N} \times \{\emptyset\}$$

$$\begin{array}{lll} & (c_1, \ldots, c_j, \ldots, c_N, M) & \\ \to & (c_1, \ldots, d_j, \ldots, c_N, M) & \text{if } (c_j, d_j) \in \ \vdash^{\mathrm{i}}_{p_j} \end{array}$$

$$\begin{array}{lll} & (c_1, \ldots, c_j, \ldots, c_N, M) & \\ \to & (c_1, \ldots, d_j, \ldots, c_N, M \cup \{m\}) & \text{if } (c_j, m, d_j) \in \ \vdash^{\mathrm{s}}_{p_j} \end{array}$$

$$\begin{array}{lll} & (c_1, \ldots, c_j, \ldots, c_N, M) & \\ \to & (c_1, \ldots, d_j, \ldots, c_N, M \backslash \{m\}) & \text{if } (c_j, m, d_j) \in \ \vdash^{\mathrm{r}}_{p_j} \text{ and } m \in M \end{array}$$

Synchronous Communication

Consider a synchronous distributed algorithm $(p_1, \ldots, p_N)$.

$$\mathcal{C} = Z_{p_1} \times \cdots \times Z_{p_N}$$

$$\mathcal{I} = I_{p_1} \times \cdots \times I_{p_N}$$

$$\begin{array}{ll} & (c_1, \ldots, c_j, \ldots, c_N) \\ \to & (c_1, \ldots, d_j, \ldots, c_N) \end{array} \qquad \text{if } (c_j, d_j) \in \ \vdash^{\mathrm{i}}_{p_j}$$

$$\begin{array}{ll} & (c_1, \ldots, c_j, \ldots, c_k, \ldots, c_N) \\ \to & (c_1, \ldots, d_j, \ldots, d_k, \ldots, c_N) \end{array} \quad \begin{array}{l} \text{if } (c_j, m, d_j) \in \ \vdash^{\mathrm{s}}_{p_j} \text{ and} \\ (c_k, m, d_k) \in \ \vdash^{\mathrm{r}}_{p_k} \text{ for some } m \in \mathcal{M} \end{array}$$

$$\begin{array}{ll} & (c_1, \ldots, c_j, \ldots, c_k, \ldots, c_N) \\ \to & (c_1, \ldots, d_j, \ldots, d_k, \ldots, c_N) \end{array} \quad \begin{array}{l} \text{if } (c_j, m, d_j) \in \ \vdash^{\mathrm{r}}_{p_j} \text{ and} \\ (c_k, m, d_k) \in \ \vdash^{\mathrm{s}}_{p_k} \text{ for some } m \in \mathcal{M} \end{array}$$

Assertions

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

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

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

Invariants

Assertion P 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.

Let P be an assertion such that:

- there is a well-founded partial order $>$ on $\mathcal{C}$ such that for each $\gamma \to \delta$ either $\gamma > \delta$ or $P(\delta)$; and
- P is true in all terminal configurations.

Then P is a liveness property.

Invariants

Assertion P is an invariant if:

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

Each invariant is a safety property.

Let P be an assertion such that:

- there is a well-founded partial order $>$ on $\mathcal{C}$ such that for each $\gamma \rightarrow \delta$ either $\gamma > \delta$ or $P(\delta)$; and
- P is true in all terminal configurations.

Then P is a liveness property.

Question

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

Fairness

Intuitively, fairness means that if a state is visited infinitely often, then each event at this state is taken infinitely often.

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

Some assertions of the distributed algorithms that we will study are only liveness properties if we restrict to the fair executions.

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 event and b the corresponding receive event, then $a \prec b$.

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

Concurrency

An important challenge in the design of distributed algorithms is to cope with concurrent events (i.e., avoid *race conditions*).

Typical examples are:

Snapshots: Compute a configuration of the system during an execution.

Termination detection: Find out whether all processes have terminated.

Mutual exclusion: Guarantee that at any time, no more than one process is accessing the critical section.

Computations

A permutation of the events in an execution that respects the causal order, does not affect the result of the execution.

These permutations together form a computation.

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

Clocks

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

- Order in sequence: Order events based on a particular execution of the computation.

- Lamport's logical clock Θ_L: Assign to each event a the length k of the longest causality chain $b_1 \prec \cdots \prec b_k = a$.

A *distributed algorithm* to compute Θ_L is:

- ∗ if a is an internal or send event, and k the clock value of the previous event at the same process ($k = 0$ if there is no such previous event), then $\Theta_L(a) = k{+}1$;

- ∗ if a is a receive event, k the clock value of the previous event at the same process ($k = 0$ if there is no such previous event), and b the send event corresponding to a, then $\Theta_L(a) = \max\{k, \Theta_L(b)\}{+}1$.

Clocks

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

- Order in sequence: Order events based on a particular execution of the computation.
- Lamport's logical clock Θ_L: Assign to each event a the length k of the longest causality chain $b_1 \prec \cdots \prec b_k = a$.

A *distributed algorithm* to compute Θ_L is:

- if a is an internal or send event, and k the clock value of the previous event at the same process ($k = 0$ if there is no such previous event), then $\Theta_L(a) = k{+}1$;
- if a is a receive event, k the clock value of the previous event at the same process ($k = 0$ if there is no such previous event), and b the send event corresponding to a, then $\Theta_L(a) = \max\{k, \Theta_L(b)\}{+}1$.

Complexity Measures

Resource consumption of distributed algorithms can be computed in several ways.

Message complexity: Total number of messages exchanged by the algorithm.

Bit complexity: Total number of bits exchanged by the algorithm. *(Only interesting when messages are very long.)*

Time complexity: Amount of time consumed by the algorithm. *(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 space needed for the processes in the algorithm.

Different computations may give rise to different consumption of resources. We consider worst- and average-case complexity (the latter with a probability distribution over all computations).

Assumptions

Unless stated otherwise, we assume:

- a strongly connected network;
- each node knows only its neighbors;
- processes have unique identities;
- message passing communication;
- asynchronous communication;
- channels are non-FIFO;
- channels do not lose, duplicate or garbel messages;
- messages are received in finite time;
- execution times of events are abstracted away; and
- in-order execution of underlying processors.

Channels can be *directed* or *undirected*.

Question

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

Snapshots

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

A snapshot of a basic computation consists of local snapshots of the state of each process and the messages in transit in each channel.

A snapshot is meaningful if it is a configuration of an execution of the basic computation.

A snapshot may not be meaningful, if some process p takes a local snapshot, and sends a message m to a process q, where

- either q takes a local snapshot after the receipt of m;
- or m is included in the local snapshot of the channel pq.

Snapshots

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

A snapshot of a basic computation consists of local snapshots of the state of each process and the messages in transit in each channel.

A snapshot is meaningful if it is a configuration of an execution of the basic computation.

A snapshot may not be meaningful, if some process p takes a local snapshot, and sends a message m to a process q, where

- either q takes a local snapshot after the receipt of m;
- or m is included in the local snapshot of the channel pq.

Snapshots - Applications

Challenge: To take a snapshot without freezing the basic computation.

Snapshots can be used to determine stable properties, which remain true as soon as they have become true.
Examples: deadlock, garbage.

Snapshots can also be used for restarting after a failure, or for debugging.

Chandy-Lamport Algorithm

Consider a *directed* network with *FIFO* channels.

Initially, initiators take a local snapshot (of their state), and send a control message ⟨**mkr**⟩ to their neighbors.

When a non-initiator receives ⟨**mkr**⟩ for the first time, it takes a local snapshot (of its state), and sends ⟨**mkr**⟩ to its neighbors.

The channel state of pq consists of the messages via pq received by q after taking its local snapshot and before receiving ⟨**mkr**⟩ from p.

If channels are FIFO, then the Chandy-Lamport algorithm computes a meaningful snapshot.

Message complexity: $\Theta(|E|)$
Time complexity: $O(D)$

Chandy-Lamport Algorithm

Consider a *directed* network with *FIFO* channels.

Initially, initiators take a local snapshot (of their state), and send a control message $\langle \textbf{mkr} \rangle$ to their neighbors.

When a non-initiator receives $\langle \textbf{mkr} \rangle$ for the first time, it takes a local snapshot (of its state), and sends $\langle \textbf{mkr} \rangle$ to its neighbors.

The channel state of pq consists of the messages via pq received by q after taking its local snapshot and before receiving $\langle \textbf{mkr} \rangle$ from p.

If channels are FIFO, then the Chandy-Lamport algorithm computes a meaningful snapshot.

Message complexity: $\Theta(|E|)$
Time complexity: $O(D)$

Chandy-Lamport Algorithm

Consider a *directed* network with *FIFO* channels.

Initially, initiators take a local snapshot (of their state), and send a control message $\langle \textbf{mkr} \rangle$ to their neighbors.

When a non-initiator receives $\langle \textbf{mkr} \rangle$ for the first time, it takes a local snapshot (of its state), and sends $\langle \textbf{mkr} \rangle$ to its neighbors.

The channel state of pq consists of the messages via pq received by q after taking its local snapshot and before receiving $\langle \textbf{mkr} \rangle$ from p.

If channels are FIFO, then the Chandy-Lamport algorithm computes a meaningful snapshot.

Message complexity: $\Theta(|E|)$
Time complexity: $O(D)$

Question

Any ideas for a snapshot algorithm that works in case of non-FIFO channels?

Lai-Yang Algorithm

The Lai-Yang algorithm uses piggybacking.

Initially, each initiator takes a local snapshot.

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

When a non-initiator receives a message with *true* or a control message (see below) for the first time, it takes a local snapshot of its state *before reception of this message*.

All processes eventually take a snapshot.

The channel state of pq consists of the basic messages via pq without the tag *true* that are received by q *after its local snapshot*.

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

Lai-Yang Algorithm

The Lai-Yang algorithm uses piggybacking.

Initially, each initiator takes a local snapshot.

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

When a non-initiator receives a message with *true* or a control message (see below) for the first time, it takes a local snapshot of its state *before reception of this message*.

All processes eventually take a snapshot.

The channel state of *pq* consists of the basic messages via *pq* without the tag *true* that are received by *q after its local snapshot*.

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

Lai-Yang Algorithm

The Lai-Yang algorithm uses piggybacking.

Initially, each initiator takes a local snapshot.

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

When a non-initiator receives a message with *true* or a control message (see below) for the first time, it takes a local snapshot of its state *before reception of this message*.

All processes eventually take a snapshot.

The channel state of *pq* consists of the basic messages via *pq* without the tag *true* that are received by *q after its local snapshot*.

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

Question

Due to the control message from p, q knows when to take a local snapshot of channel pq.

Which information does q have to store exactly on incoming basic messages from p without the tag *true*?

Wave Algorithms

Decide events are special internal events.

A distributed algorithm is a wave algorithm if for each computation (also called wave) C:

- *Termination:* C is finite;
- *Decision:* C contains a decide event; and
- *Dependence:* for each decide event e in C and process p, $f \preceq e$ for an event f at p.

Examples: Ring algorithm, tree algorithm, echo algorithm.

Traversal Algorithms

A traversal algorithm is a centralized wave algorithm; i.e., there is one initiator, which sends around a token (representing a combined send and receive event).

In each computation, the token first visits all processes. Finally, a decide event happens at the initiator, who at that time holds the token.

In traversal algorithms, the father of a non-initiator is the neighbor from which it received the token first.

Tarry's Algorithm

$G = (V, E)$ is an undirected graph.

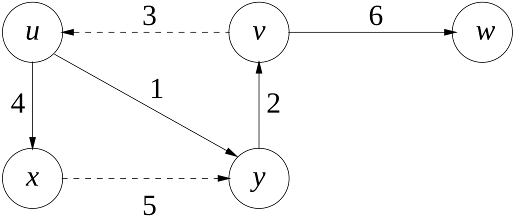
Tarry's traversal algorithm (from 1895):

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

R2 A node only forwards the token to its father when there is no other option.

Tarry's Algorithm - Example

The graph below is undirected and unweighted; *u* is the initiator.



Arrows mark the path of the token (so the father-child relation is reversed).

Edges of the spanning tree are solid.

Frond edges, which are not part of the spanning tree, are dashed.

Depth-First Search

A spanning tree is a depth-first search tree if each frond edge connects an ancestor and a descendant of the spanning tree.

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

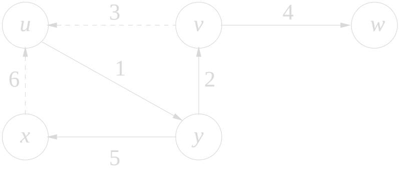
R3 When a node receives the token, it immediately sends it back through the same channel, if this is allowed by R1,2.

Message complexity: $2|E|$ messages

Time complexity: $\leq 2|E|$ time units

The spanning tree of a depth-first search is a depth-first search tree.

Example:



Depth-First Search

A spanning tree is a depth-first search tree if each frond edge connects an ancestor and a descendant of the spanning tree.

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

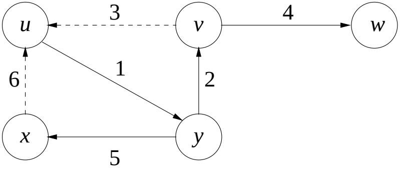
- R3 When a node receives the token, it immediately sends it back through the same channel, if this is allowed by R1,2.

Message complexity: $2|E|$ messages

Time complexity: $\leq 2|E|$ time units

The spanning tree of a depth-first search is a depth-first search tree.

Example:



Question

How can (the delay of) messages through frond edges be avoided?

Neighbor Knowledge

Prevents transmission of the token through a frond edge.

The visited nodes are included in the token.

The token is not forwarded to nodes in this list (except when a node sends the token to its father).

Message complexity: $2|V|-2$ messages

Tree edges carry 2 forwarded tokens.

Bit complexity: Up to $k|V|$ bits per message, where k bits are needed to represent one node.

Time complexity: $\leq 2|V|-2$ time units

Neighbor Knowledge

Prevents transmission of the token through a frond edge.

The visited nodes are included in the token.

The token is not forwarded to nodes in this list (except when a node sends the token to its father).

Message complexity: $2|V|-2$ messages

Tree edges carry 2 forwarded tokens.

Bit complexity: Up to $k|V|$ bits per message, where k bits are needed to represent one node.

Time complexity: $\leq 2|V|-2$ time units

Awerbuch's Algorithm

- A node holding the token for the first time informs all neighbors except its father (and the node to which it will forward the token).

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

- The token is only forwarded to nodes that were not yet visited by the token (except when a node sends the token to its father).

Awerbuch's Algorithm - Complexity

Message complexity: $< 4|E|$ messages

Frond edges carry 2 information and 2 acknowledgement messages.

Tree edges carry 2 forwarded tokens, and
possibly 1 information/acknowledgement pair.

Time complexity: $\leq 4|V|{-}2$ time units

Tree edges carry 2 forwarded tokens.

Nodes wait at most 2 time units before forwarding the token.

Awerbuch's Algorithm - Complexity

Message complexity: $< 4|E|$ messages

Frond edges carry 2 information and 2 acknowledgement messages.

Tree edges carry 2 forwarded tokens, and
possibly 1 information/acknowledgement pair.

Time complexity: $\leq 4|V|-2$ time units

Tree edges carry 2 forwarded tokens.

Nodes wait at most 2 time units before forwarding the token.

Question

Are the acknowledgements in Awerbuch's algorithm really needed?

Cidon's Algorithm

Abolishes acknowledgements from Awerbuch's algorithm.

- The token is forwarded without delay. Each node u records to which node mrs_u it forwarded the token last.
- Suppose node u receives the token from a node $v \neq mrs_u$. Then u marks the edge uv as used and *purges* the token.
- Suppose node v receives an information message from mrs_v. Then it continues forwarding the token.

Cidon's Algorithm - Complexity

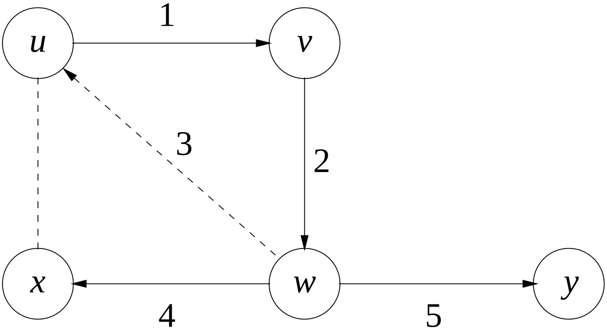
Message complexity: $< 4|E|$ messages

Each edge carries at most 2 information messages and 2 forwarded tokens.

Time complexity: $\leq 2|V|-2$ time units

At least once per time unit, a token is forwarded through a tree edge. Each tree edge carries 2 forwarded tokens.

Cidon's Algorithm - Example



Tree Algorithm

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

The local algorithm at a node u:

- u waits until it received messages from all neighbors except one, denoted Nb_u;
- then u sends a message to Nb_u;
- if u receives a *message* from Nb_u, it decides;

  in that case u sends the decision to all neighbors except Nb_u;
- if u receives a *decision* from Nb_u, it passes it on to all other neighbors.

Remark: Always two nodes decide.

Message complexity: $2|V|-2$ messages

Tree Algorithm

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

The local algorithm at a node u:

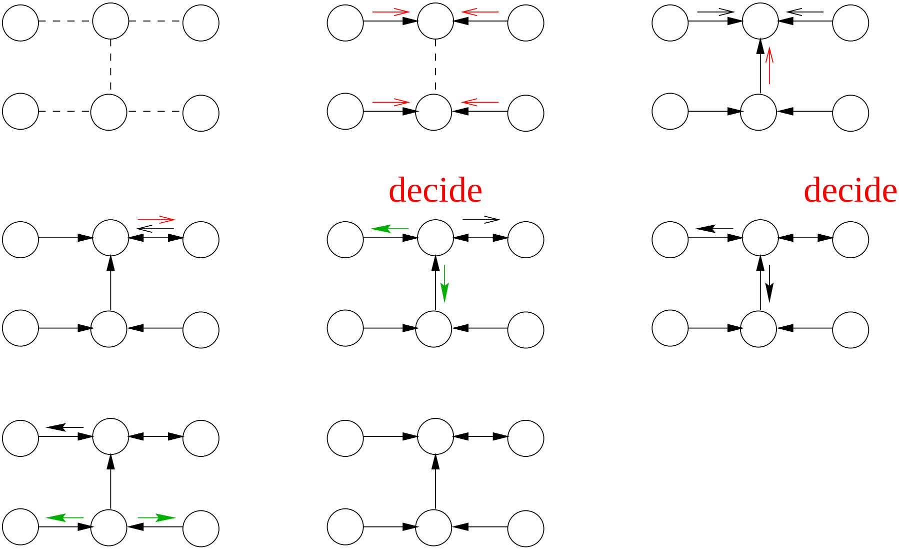
- u waits until it received messages from all neighbors except one, denoted Nb_u;
- then u sends a message to Nb_u;
- if u receives a *message* from Nb_u, it decides;

  in that case u sends the decision to all neighbors except Nb_u;
- if u receives a *decision* from Nb_u, it passes it on to all other neighbors.

Remark: Always two nodes decide.

Message complexity: $2|V|-2$ messages

Tree Algorithm - Example



Question

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

Echo Algorithm

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

- Initiator sends a message to all neighbors.
- The father of a non-initiator is the neighbor from which it receives the first message.
- A non-initiator sends a message to all neighbors except its father.
- When a non-initiator has received a message from all neighbors, it sends a message to its father.
- When the initiator has received a message from all neighbors, it decides.

Message complexity: $2|E|$ messages

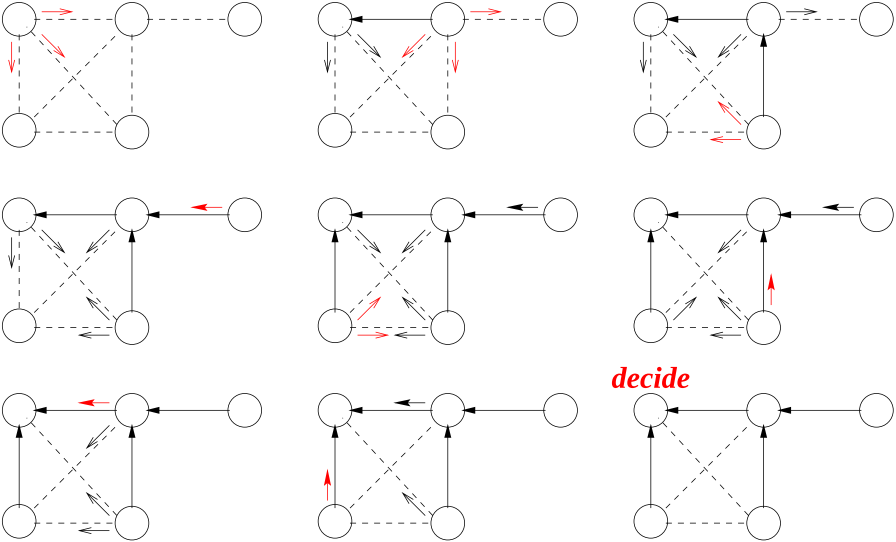
Echo Algorithm

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

- Initiator sends a message to all neighbors.
- The father of a non-initiator is the neighbor from which it receives the first message.
- A non-initiator sends a message to all neighbors except its father.
- When a non-initiator has received a message from all neighbors, it sends a message to its father.
- When the initiator has received a message from all neighbors, it decides.

Message complexity: $2|E|$ messages

Echo Algorithm - Example



Question

Let each process initially carry a random integer value.

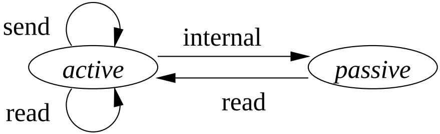
Adapt the echo algorithm to compute the sum of these integer values.

Termination Detection

In a distributed setting, detection of termination can be non-trivial.

The basic algorithm is terminated if each process is passive and no messages are in transit.

The control algorithm consists of termination detection and announcement. Announcement is simple; we focus on detection.



Termination detection should not (1) use further information on local states or internal events, or (2) influence basic computations.

Dijkstra-Scholten Algorithm

Requires (1) a centralized basic algorithm, and (2) an undirected graph.

A *tree* T is maintained, in which the initiator is the root, and all active processes are nodes of T. Initially, T consists of the initiator.

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

- When p sends a basic message, $sc_p := sc_p{+}1$.
- Suppose this message is received by q.
  - If q was not yet in T, q becomes a node in T with father p and $sc_q := 0$.
  - If q is already in T, it sends a message to p that it is not a new son of p. Upon receipt of this message, $sc_p := sc_p{-}1$.
- When a non-initiator p is passive and $sc_p = 0$, it informs its father it is no longer a son.
- When initiator p_0 is passive and $sc_{p_0} = 0$, it calls *Announce*.

Dijkstra-Scholten Algorithm

Requires (1) a centralized basic algorithm, and (2) an undirected graph.

A *tree* T is maintained, in which the initiator is the root, and all active processes are nodes of T. Initially, T consists of the initiator.

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

- When p sends a basic message, $sc_p := sc_p + 1$.
- Suppose this message is received by q.
  - If q was not yet in T, q becomes a node in T with father p and $sc_q := 0$.
  - If q is already in T, it sends a message to p that it is not a new son of p. Upon receipt of this message, $sc_p := sc_p - 1$.
- When a non-initiator p is passive and $sc_p = 0$, it informs its father it is no longer a son.
- When initiator p_0 is passive and $sc_{p_0} = 0$, it calls *Announce*.

Question

Any suggestions to make the Dijkstra-Scholten algorithm decentralized?

Shavit-Francez Algorithm

Allows a decentralized basic algorithm; requires an undirected graph.

A *forest* F of trees is maintained, rooted in the initiators. Each process is in at most one tree of F.

Initially, each initiator constitutes a tree in F.

- When a process p sends a basic message, $sc_p := sc_p+1$.
- Suppose this message is received by q.
  - If q was not yet in *some tree in F*, q becomes a node in F with father p and $sc_q := 0$.
  - If q is already in *some tree in F*, it sends a message to p that it is not a new son of p. Upon receipt, $sc_p := sc_p-1$.
- When a non-initiator p is passive and $sc_p = 0$, it informs its father it is no longer a son.

A wave algorithm is used on the side, in which only nodes that are not in a tree participate, and *decide* calls *Announce*.

Shavit-Francez Algorithm

Allows a decentralized basic algorithm; requires an undirected graph.

A *forest* F of trees is maintained, rooted in the initiators. Each process is in at most one tree of F.

Initially, each initiator constitutes a tree in F.

- When a process p sends a basic message, $sc_p := sc_p + 1$.
- Suppose this message is received by q.
  - If q was not yet in *some tree in F*, q becomes a node in F with father p and $sc_q := 0$.
  - If q is already in *some tree in F*, it sends a message to p that it is not a new son of p. Upon receipt, $sc_p := sc_p - 1$.
- When a non-initiator p is passive and $sc_p = 0$, it informs its father it is no longer a son.

A wave algorithm is used on the side, in which only nodes that are not in a tree participate, and *decide* calls *Announce*.

Shavit-Francez Algorithm

Allows a decentralized basic algorithm; requires an undirected graph.

A *forest* F of trees is maintained, rooted in the initiators. Each process is in at most one tree of F.

Initially, each initiator constitutes a tree in F.

- When a process p sends a basic message, $sc_p := sc_p{+}1$.
- Suppose this message is received by q.
  - If q was not yet in *some tree in F*, q becomes a node in F with father p and $sc_q := 0$.
  - If q is already in *some tree in F*, it sends a message to p that it is not a new son of p. Upon receipt, $sc_p := sc_p{-}1$.
- When a non-initiator p is passive and $sc_p = 0$, it informs its father it is no longer a son.

A wave algorithm is used on the side, in which only nodes that are not in a tree participate, and *decide* calls *Announce*.

Rana's Algorithm

A decentralized algorithm, for undirected graphs.

Let a logical clock provide *(basic and control)* events with a time stamp.

The time stamp of a process is the time stamp of its last event (initially it is 0).

Each basic message is acknowledged, and each process counts how many of the basic messages it sent have not yet been acknowledged.

If at some time t a process becomes quiet, meaning that (1) it is passive and (2) all basic messages it sent have been acknowledged, it starts a wave (of control messages) tagged with t.

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

If a wave completes, the initiator of the wave calls *Announce*.

Rana's Algorithm

A decentralized algorithm, for undirected graphs.

Let a logical clock provide *(basic and control)* events with a time stamp.

The time stamp of a process is the time stamp of its last event (initially it is 0).

Each basic message is acknowledged, and each process counts how many of the basic messages it sent have not yet been acknowledged.

If at some time t a process becomes quiet, meaning that (1) it is passive and (2) all basic messages it sent have been acknowledged, it starts a wave (of control messages) tagged with t.

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

If a wave completes, the initiator of the wave calls *Announce*.

Rana's Algorithm - Correctness

Suppose a wave, tagged with some t, does not complete.

Then some process did not take part in the wave, meaning that it was not quiet at a time $\geq t$.

So this process will start a new wave at a later time.

Rana's Algorithm - Correctness

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

Then this wave will not complete.

Namely, let the wave be tagged with t. When p takes part in the wave, its logical clock becomes $> t$.

By the resulting acknowledgement from p to q, the logical clock of q becomes $> t$.

So q will not take part in the wave (because it is tagged with t).

Mattern's Weight-Throwing Termination Detection

Requires a centralized basic algorithm; allows a directed graph.

The initiator has weight 1, all non-initiators have weight 0.

When a process *sends* a basic message, it attaches part of its weight to this message (and subtracts this from its own weight).

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

When a non-initiator becomes passive, it returns its weight to the initiator, by means of a control message.

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

Mattern's Weight-Throwing Termination Detection

Requires a centralized basic algorithm; allows a directed graph.

The initiator has weight 1, all non-initiators have weight 0.

When a process *sends* a basic message, it attaches part of its weight to this message (and subtracts this from its own weight).

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

When a non-initiator becomes passive, it returns its weight to the initiator, by means of a control message.

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

Weight-Throwing Termination Detection - Drawback

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

Solution 1: This process can ask the initiator for extra weight.

Solution 2: This process can itself initiate a weight-throwing termination detection sub-call, and only return its weight to the initiator when it has become passive and this sub-call has terminated.

Weight-Throwing Termination Detection - Drawback

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

Solution 1: This process can ask the initiator for extra weight.

Solution 2: This process can itself initiate a weight-throwing termination detection sub-call, and only return its weight to the initiator when it has become passive and this sub-call has terminated.

Weight-Throwing Termination Detection - Drawback

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

Solution 1: This process can ask the initiator for extra weight.

Solution 2: This process can itself initiate a weight-throwing termination detection sub-call, and only return its weight to the initiator when it has become passive and this sub-call has terminated.

Token-Based Termination Detection

A centralized algorithm for directed graphs.

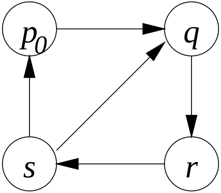
A process p_0 is initiator of a traversal algorithm to check whether all processes are passive.

Complication 1: Reception of basic messages cannot be acknowledged.

Solution: Synchronous communication.

Complication 2: A traversal of only passive processes still does not guarantee termination.

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*.

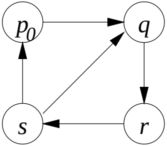
Dijkstra-Feijen-van Gasteren Algorithm

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

- When p_0 is passive, it sends a white token.
- Only passive processes forward the token.
- If a black process forwards the token, the token becomes black and the process white.
- Eventually, the token returns to p_0, and p_0 waits until it is passive.

- If both the token and p_0 are white, then p_0 calls *Announce*.
- Otherwise, p_0 sends a white token again.

Dijkstra-Feijen-van Gasteren Algorithm - Example



The token is at p_0; only s is active.

The token travels to r.

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

s becomes passive.

The token travels on to p_0, and is made black at s.

q becomes passive.

The token travels around the network, and p_0 calls *Announce*.

Question

Give an example to show that the Dijkstra-Feijen-van Gasteren algorithm does not work in case of asynchronous communication.

Safra's Algorithm

Allows a directed graph and asynchronous communication.

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

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

Still the token may compute a negative sum for a round trip, when a visited passive process receives a basic message, becomes active and sends basic messages that are received by an unvisited process.

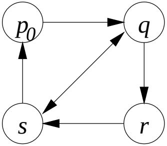
Safra's Algorithm

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.
- Only passive processes forward the token.
- When a black process forwards the token, the token becomes black and the process white.
- Eventually the token returns to p_0, and p_0 waits until it is passive.
  - If the token and p_0 are white and the sum of all counters is zero, p_0 calls *Announce*.
  - Otherwise, 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 and 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*.

Election Algorithms

- Each computation terminates in a configuration where *one* process is the leader.
- All processes have the same local algorithm.
- Identities of processes are totally ordered.
- The *initiators* are any non-empty subset.

Chang-Roberts Algorithm

Let $G = (V, E)$ be a directed ring.

- Each initiator u sends a token around the ring, containing its id.
- When u receives v with $v < u$, u becomes passive, and u passes on id v.
- When u receives v with $v > u$, it purges the message.
- When u receives u, u becomes the leader.

Passive processes (including all non-initiators) pass on incoming messages.

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

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

Chang-Roberts Algorithm

Let $G = (V, E)$ be a directed ring.

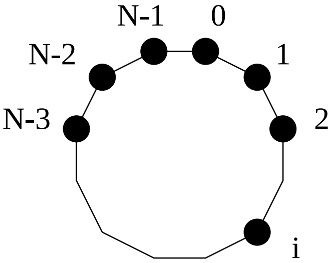
- Each initiator u sends a token around the ring, containing its id.
- When u receives v with $v < u$, u becomes passive, and u passes on id v.
- When u receives v with $v > u$, it purges the message.
- When u receives u, u becomes the leader.

Passive processes (including all non-initiators) pass on incoming messages.

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

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

Chang-Roberts Algorithm - Example



clockwise: N(N+1)/2 messages

anti-clockwise: 2N-1 messages

Franklin's Algorithm

Let G be an undirected ring.

In Franklin's algorithm, each *active* node compares its id with the identities of its nearest *active* neighbors. If its id is not the smallest, it becomes *passive*.

- Initially, initiators are active, and non-initiators are passive. Each active node sends its id to its neighbors on either side.
- Let active node u receive v and w:

  - if $\min\{v, w\} < u$, then u becomes passive;
  - if $\min\{v, w\} > u$, then u sends its id again;
  - if $\min\{v, w\} = u$, then u becomes the leader.

Passive nodes pass on incoming messages.

Franklin's Algorithm

Let G be an undirected ring.

In Franklin's algorithm, each *active* node compares its id with the identities of its nearest *active* neighbors. If its id is not the smallest, it becomes *passive*.

- Initially, initiators are active, and non-initiators are passive. Each active node sends its id to its neighbors on either side.
- Let active node u receive v and w:
  - if $\min\{v, w\} < u$, then u becomes passive;
  - if $\min\{v, w\} > u$, then u sends its id again;
  - if $\min\{v, w\} = u$, then u becomes the leader.

Passive nodes pass on incoming messages.

Franklin's Algorithm - Complexity

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

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

Each update round takes $2|V|$ messages.

Question

Any suggestions how to adapt Franklin's algorithm to a directed ring?

Dolev-Klawe-Rodeh Algorithm

Let G be a (clockwise) directed ring.

The difficulty is to obtain the clockwise next active id.

In the DKR algorithm, the comparison of identities of an active node u and its nearest active neighbors v and w is performed at w.

$$\dashrightarrow x \dashrightarrow v \dashrightarrow u \dashrightarrow w \dashrightarrow y \dashrightarrow$$

- If u has a smaller id than v and w, then w assumes the id of u.
- Otherwise, w becomes passive.

When u and w carry the same id, w concludes that the node with this id must become the leader;
then w broadcasts this id to all nodes.

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

Dolev-Klawe-Rodeh Algorithm

Let G be a (clockwise) directed ring.

The difficulty is to obtain the clockwise next active id.

In the DKR algorithm, the comparison of identities of an active node u and its nearest active neighbors v and w is performed at w.

$$\dashrightarrow x \dashrightarrow v \dashrightarrow u \dashrightarrow w \dashrightarrow y \dashrightarrow$$

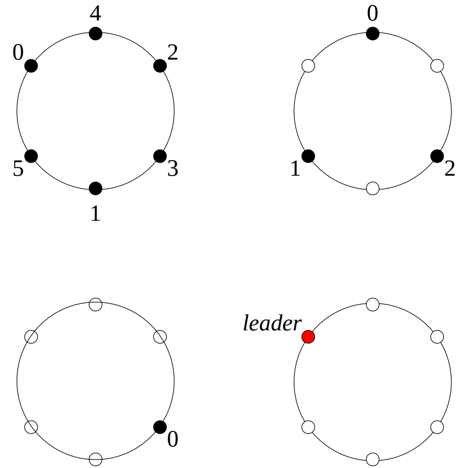
- If u has a smaller id than v and w, then w assumes the id of u.
- Otherwise, w becomes passive.

When u and w carry the same id, w concludes that the node with this id must become the leader;
then w broadcasts this id to all nodes.

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

Dolev-Klawe-Rodeh Algorithm - Example

Consider the following clockwise oriented ring.



Question

How could the tree algorithm be used to get an election algorithm for undirected, acyclic graphs?

Tree Election Algorithm

Let G be an undirected, acyclic graph.

The tree election algorithm starts with a wake-up phase, driven by the initiators.

The local algorithm at an awake node u:

- u waits until it received identities from all neighbors except one, denoted Nb_u;
- u computes the smallest id $\min_u$ among the received identities and its own id;
- u sends $\min_u$ to Nb_u;
- when u receives id v from Nb_u, it computes $\min'_u$, being the minimum of $\min_u$ and v;
- if $\min'_u = u$, then u becomes the leader;
- u sends $\min'_u$ to all neighbors except Nb_u.

Message complexity: $2|V|-2$ messages

Tree Election Algorithm

Let G be an undirected, acyclic graph.

The tree election algorithm starts with a wake-up phase, driven by the initiators.

The local algorithm at an awake node u:

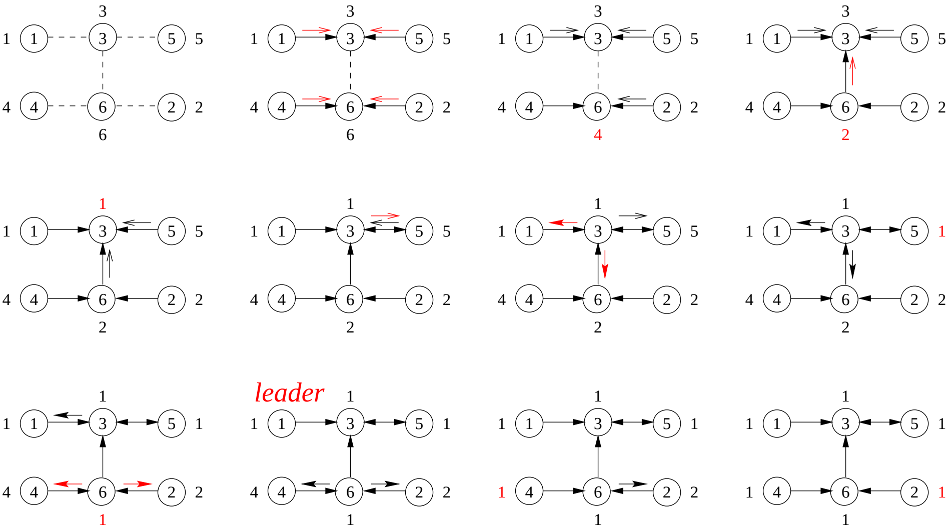
- u waits until it received identities from all neighbors except one, denoted Nb_u;
- u computes the smallest id $\min_u$ among the received identities and its own id;
- u sends $\min_u$ to Nb_u;
- when u receives id v from Nb_u, it computes $\min'_u$, being the minimum of $\min_u$ and v;
- if $\min'_u = u$, then u becomes the leader;
- u sends $\min'_u$ to all neighbors except Nb_u.

Message complexity: $2|V|-2$ messages

Question

Why does u compute the minimum of $\min_u$ and v?

Tree Election Algorithm - Example



Question

How could the echo algorithm be used to get an election algorithm for any undirected graph?

Echo Algorithm with Extinction

Election for undirected graphs, based on the echo algorithm:

- Each initiator starts a wave, tagged with its id.
- At any time, each node takes part in at most one wave.
- Suppose a node u in wave v is hit by a wave w:
  - if $v > w$, then u changes to wave w (it abandons all earlier messages);
  - if $v < w$, then u continues with wave v (it purges the incoming message);
  - if $v = w$, then the incoming message is treated according to the echo algorithm of wave w.
- If wave u executes a decide event (at u), u becomes leader.

Non-initiators join the first wave that hits them.

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

Echo Algorithm with Extinction

Election for undirected graphs, based on the echo algorithm:

- Each initiator starts a wave, tagged with its id.
- At any time, each node takes part in at most one wave.
- Suppose a node u in wave v is hit by a wave w:
  - if $v > w$, then u changes to wave w (it abandons all earlier messages);
  - if $v < w$, then u continues with wave v (it purges the incoming message);
  - if $v = w$, then the incoming message is treated according to the echo algorithm of wave w.
- If wave u executes a decide event (at u), u becomes leader.

Non-initiators join the first wave that hits them.

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

Minimal Spanning Trees

Let G be an undirected, weighted graph, in which *different edges have different weights*.

(Weights can be totally ordered by taking into account the identities of endpoints of an edge, and using a lexicographical order.)

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

Lemma: Let F be a fragment (i.e., a subtree of a minimal spanning tree M in G), and e 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.

Minimal Spanning Trees

Let G be an undirected, weighted graph, in which *different edges have different weights*.

(Weights can be totally ordered by taking into account the identities of endpoints of an edge, and using a lexicographical order.)

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

Lemma: Let F be a fragment (i.e., a subtree of a minimal spanning tree M in G), and e 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.

Prim's Algorithm

Centralized. Initially, F is a single node.

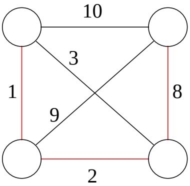
As long as F is not a spanning tree, add the lowest-weight outgoing edge of F to F.

Kruskal's Algorithm

Decentralized. Initially, each node in G forms a separate fragment.

In each step, two distinct fragments F and F' are joined by adding an outgoing edge of both F and F' which is lowest-weight for F.

Example:



Prim's and Kruskal's algorithm also work when edges have the same weight. But then the minimal spanning tree may not be unique.

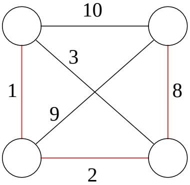
Complications in a distributed setting: Is an edge outgoing? Is it lowest-weight?

Kruskal's Algorithm

Decentralized. Initially, each node in G forms a separate fragment.

In each step, two distinct fragments F and F' are joined by adding an outgoing edge of both F and F' which is lowest-weight for F.

Example:



Prim's and Kruskal's algorithm also work when edges have the same weight. But then the minimal spanning tree may not be unique.

Complications in a distributed setting: Is an edge outgoing? Is it lowest-weight?

Gallager-Humblet-Spira Algorithm

G is an undirected, weighted graph, in which different edges have different weights.

Distributed computation of a minimal spanning tree in G:

- initially, each node is a fragment;
- the nodes in a fragment F together search for the lowest-weight outgoing edge e_F;
- when e_F is found, the fragment at the other end is asked to collaborate in a merge.

Level, name and core edge

Fragments carry a level $L : \mathbb{N}$ and a name FN.

Fragments $F = (L, FN)$ and $F' = (L', FN')$ are joined in the following cases:

$$
\begin{array}{ll}
L < L' \wedge F \xrightarrow{e_F} F': & F \cup F' = (L', FN') \\
L > L' \wedge F' \xrightarrow{e_{F'}} F: & F \cup F' = (L, FN) \\
L = L' \wedge e_F = e_{F'}: & F \cup F' = (L{+}1, \text{weight } e_F)
\end{array}
$$

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 node

Each node keeps track of:

- its state: sleep (for non-initiators), find or found;
- status of its edges: basic, branch or reject;
- level and name (i.e., the weight of the core edge) of its fragment;
- its father, toward the core edge.

Initialization

Each *initiator* u sets $level_u$ to 0, its lowest-weight outgoing edge uv to branch, and its other edges to basic.

u sends $\langle \textbf{connect}, 0 \rangle$ to v.

Each *non-initiator* is woken up when it receives a **connect** or **test** message.

Joining two fragments

Let fragments $F = (L, FN)$ and $F' = (L', FN')$ be joined via edge uv (after the exchange of 1 or 2 **connect** messages through this edge).

- If $L < L'$, then $\langle \textbf{initiate}, L', FN', \frac{\mathit{find}}{\mathit{found}} \rangle$ is sent by v to u, and forwarded through F;

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

- If $L > L'$, then vice versa.

- If $L = L'$, then $\langle \textbf{initiate}, L{+}1, \text{weight } uv, \mathit{find} \rangle$ is sent both ways;

  $F \cup F'$ has core edge uv.

Computing the lowest-weight outgoing edge

At reception of $\langle \textbf{initiate}, L, FN, \frac{find}{found} \rangle$, a node u stores L and FN, and adopts the sender as its father.

It passes on the message through the branch edges.

In case of *find*, u checks in increasing order of weight if one of its basic edges uv is outgoing, by sending $\langle \textbf{test}, L, FN \rangle$ to v.

1. If $L > level_v$, v postpones processing the incoming **test** message;
2. else, if v is in fragment FN, v replies **reject** (except when v was awaiting a reply of a **test** message to u), after which u and v set uv to reject;
3. else, v replies **accept**.

When a basic edge accepts, or all basic edges failed, u stops the search.

Computing the lowest-weight outgoing edge

At reception of $\langle \textbf{initiate}, L, FN, \frac{\mathit{find}}{\mathit{found}} \rangle$, a node u stores L and FN, and adopts the sender as its father.
It passes on the message through the branch edges.

In case of *find*, u checks in increasing order of weight if one of its basic edges uv is outgoing, by sending $\langle \textbf{test}, L, FN \rangle$ to v.

1. If $L > \mathit{level}_v$, v postpones processing the incoming **test** message;
2. else, if v is in fragment FN, v replies **reject** (except when v was awaiting a reply of a **test** message to u), after which u and v set uv to reject;
3. else, v replies **accept**.

When a basic edge accepts, or all basic edges failed, u stops the search.

Computing the lowest-weight outgoing edge

At reception of $\langle \textbf{initiate}, L, FN, \frac{\mathit{find}}{\mathit{found}} \rangle$, a node u stores L and FN, and adopts the sender as its father.
It passes on the message through the branch edges.

In case of *find*, u checks in increasing order of weight if one of its basic edges uv is outgoing, by sending $\langle \textbf{test}, L, FN \rangle$ to v.

1. If $L > \mathit{level}_v$, v postpones processing the incoming **test** message;
2. else, if v is in fragment FN, v replies **reject** (except when v was awaiting a reply of a **test** message to u), after which u and v set uv to reject;
3. else, v replies **accept**.

When a basic edge accepts, or all basic edges failed, u stops the search.

Question

In case 1, if $L > level_v$, why does v postpone processing the incoming **test** message from u?

Question

In case 1, if $L > \mathit{level}_v$, why does v postpone processing the incoming **test** message from u?

Answer: u and v might be in the same fragment, in which case $\langle \textbf{initiate}, L, FN, \frac{\mathit{find}}{\mathit{found}} \rangle$ is on its way to v.

Question

Why does this postponement not lead to a deadlock?

Question

Why does this postponement not lead to a deadlock?

Answer: Because there is always a fragment of minimal level.

Reporting to the core nodes

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

Termination or **changeroot** at the core nodes

A core node receives reports from *all* neighbors, including its father.

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

When a node u that reported its lowest-weight outgoing edge receives **changeroot**, it sets this edge to branch, and sends $\langle$**connect**, $level_u\rangle$ into it.

Starting the join of two fragments

When a node v receives $\langle \textbf{connect}, \mathit{level}_u \rangle$ from u, then $\mathit{level}_v \geq \mathit{level}_u$. Namely, either $\mathit{level}_u = 0$ or v earlier sent **accept** to u.

1. If $\mathit{level}_v > \mathit{level}_u$, then v sets vu to branch and sends $\langle \textbf{initiate}, \mathit{level}_v, \mathit{name}_v, \frac{\mathit{find}}{\mathit{found}} \rangle$ to u.

   (If v was awaiting a reply of a **test** message to u, it stops doing so.)

2. As long as $\mathit{level}_v = \mathit{level}_u$ and vu is not a branch of v, v postpones processing the **connect** message.

3. If $\mathit{level}_v = \mathit{level}_u$ and vu is a branch of v (i.e., v sent to u $\langle \textbf{connect}, \mathit{level}_v \rangle$), v sends to u $\langle \textbf{initiate}, \mathit{level}_v{+}1, \text{weight } vu, \mathit{find} \rangle$.
   Now v (and u) become the core nodes.

Question

In case 2, if $level_v = level_u$, why does v postpone processing the incoming **connect** message from u?

Question

In case 2, if $level_v = level_u$, why does v postpone processing the incoming **connect** message from u?

Answer: The fragment of v might be in the process of joining a fragment at level $\geq level_v$, in which case the fragment of u should subsume the name and level of that joined fragment, instead of joining the fragment of v at an equal level.

Question

Why does this postponement not give rise to a deadlock?

Question

Why does this postponement not give rise to a deadlock?

Answer: Since different edges have different weights, there cannot be a cycle of fragments that are waiting for a reply to a postponed **connect** message.

Question

In case 1, which problem could occur if at this time v was reporting uv as lowest-weight outgoing edge to its core nodes?

Why can we be sure that this is never the case?

Question

In case 1, which problem could occur if at this time v was reporting uv as lowest-weight outgoing edge to its core nodes?

Why can we be sure that this is never the case?

Answer: Then v could later receive a **changeroot** to set uv to branch, while this has already been done.

Since in case 1, $level_v > level_u$, we can be sure that u did not send an **accept** to v.

Gallager-Humblet-Spira Algorithm - Complexity

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

- At most one **test**-**reject** or **test**-**test** pair per edge.

Between two subsequent joins, each node in a fragment:

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

Each node experiences at most $\log|V|$ joins (because a fragment at level L contains $\geq 2^L$ nodes).

Question

How can the Gallager-Humblet-Spira algorithm be turned into an election algorithm for any undirected graph?

Back to Election

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

So Gallager-Humblet-Spira is an election algorithm for general graphs.

Lower bounds for the average-case message complexity of election algorithms, based on comparison of identities:

Rings: $\Omega(|V|\log|V|)$

General graphs: $\Omega(|E|+|V|\log|V|)$

Question

Consider a ring of size 3, in which all edges have weight 1.

Show that in this case, the Gallager-Humblet-Spira algorithm could get into a deadlock.

As said before, this deadlock can be avoided by imposing a total order on edges; take into account the identities of endpoints of an edge, and use a lexicographical order.

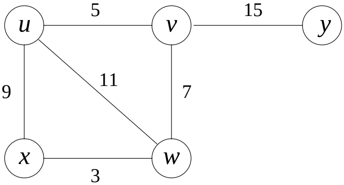
Question

Consider a ring of size 3, in which all edges have weight 1.

Show that in this case, the Gallager-Humblet-Spira algorithm could get into a deadlock.

As said before, this deadlock can be avoided by imposing a total order on edges; take into account the identities of endpoints of an edge, and use a lexicographical order.

Example



$uv \;\; vu$	$\langle \textbf{connect}, 0 \rangle$
$uv \;\; vu$	$\langle \textbf{initiate}, 1, 5, \mathit{find} \rangle$
$ux \;\; vw$	$\langle \textbf{test}, 1, 5 \rangle$
yv	$\langle \textbf{connect}, 0 \rangle$
vy	$\langle \textbf{initiate}, 1, 5, \mathit{find} \rangle$
yv	$\langle \textbf{report}, \infty \rangle$
$wx \;\; xw$	$\langle \textbf{connect}, 0 \rangle$
$wx \;\; xw$	$\langle \textbf{initiate}, 1, 3, \mathit{find} \rangle$
$xu \;\; wv$	**accept**
uv	$\langle \textbf{report}, 9 \rangle$
vu	$\langle \textbf{report}, 7 \rangle$
vw	$\langle \textbf{connect}, 1 \rangle$
$xu \;\; wv$	$\langle \textbf{test}, 1, 3 \rangle$
$ux \;\; vw$	**accept**
wx	$\langle \textbf{report}, 7 \rangle$
xw	$\langle \textbf{report}, 9 \rangle$
wv	$\langle \textbf{connect}, 1 \rangle$
$wv \;\; vu \;\; vy \;\; vw \;\; wx$	$\langle \textbf{initiate}, 2, 7, \mathit{find} \rangle$
uw	$\langle \textbf{test}, 2, 7 \rangle$
wu	**reject**
$yv \;\; uv \;\; vw \;\; xw \;\; wv$	$\langle \textbf{report}, \infty \rangle$

Anonymous Networks

Processes may be anonymous (e.g., Lego MindStorm chips), or transmitting identities may be too expensive (e.g., FireWire bus).

When a leader is known, all processes can be named (using for instance a traversal algorithm).

Assumptions: Processes have no identities and carry the same local algorithm.

Impossibility of Election in Anonymous Networks

Theorem: There is no terminating algorithm for electing a leader in an asynchronous anonymous graph.

Proof: Take a (directed) ring of size N.

In a symmetric configuration, all nodes are in the same state and all cannels carry the same messages.

- The initial configuration is symmetric.
- If γ_0 is symmetric and $\gamma_0 \rightarrow \gamma_1$, then $\gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_N$ where γ_N is symmetric.

So there is an infinite *fair* execution.

Impossibility of Election in Anonymous Networks

Theorem: There is no terminating algorithm for electing a leader in an asynchronous anonymous graph.

Proof: Take a (directed) ring of size N.

In a symmetric configuration, all nodes are in the same state and all cannels carry the same messages.

- The initial configuration is symmetric.
- If γ_0 is symmetric and $\gamma_0 \rightarrow \gamma_1$, then $\gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_N$ where γ_N is symmetric.

So there is an infinite *fair* execution.

Probabilistic Algorithms

In a probabilistic algorithm, each process p holds two local algorithms, $\vdash^0$ and $\vdash^1$.

Let $\rho_p : \mathbb{N} \rightarrow \{0, 1\}$ for each p. In a ρ-computation, the k-th event at p is performed according to $\vdash^{\rho_p(k)}$.

For a probabilistic algorithm where all computations terminate in a correct configuration, $\vdash^0$ is a correct non-probabilistic algorithm.

Monte Carlo and Las Vegas Algorithms

A probabilistic algorithm is Monte Carlo if:

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

It is Las Vegas if:

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

Monte Carlo and Las Vegas Algorithms

A probabilistic algorithm is Monte Carlo if:

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

It is Las Vegas if:

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

Question

Even if the probability that a Las Vegas algorithm terminates is 1, this does not imply termination. Why is that?

Question

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

Give a Las Vegas algorithm that terminates with probability 1.

Itai-Rodeh Election Algorithm

Let G be an anonymous, directed ring, in which *all nodes know the ring size* N.

The Itai-Rodeh election algorithm is based on the Chang-Roberts algorithm, where each process sends out its id, and the smallest 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 arrives at its source if and only if its hop count is N.

If several processes selected the same smallest id, they start a fresh election round, at a higher level.

The Itai-Rodeh election algorithm is a Las Vegas algorithm; it terminates with probability 1.

Itai-Rodeh Election Algorithm

Let G be an anonymous, directed ring, in which *all nodes know the ring size* N.

The Itai-Rodeh election algorithm is based on the Chang-Roberts algorithm, where each process sends out its id, and the smallest 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 arrives at its source if and only if its hop count is N.

If several processes selected the same smallest id, they start a fresh election round, at a higher level.

The Itai-Rodeh election algorithm is a Las Vegas algorithm; it terminates with probability 1.

Itai-Rodeh Election Algorithm

Initially, initiators are active at level 0, and non-initiators are passive. In each election round, if p is active at level ℓ:

- At the start of the round, p selects a random id id_p, and sends $(\ell, id_p, 1, true)$. The 3rd parameter is the hop count. The 4th parameter signals whether another process with the same id was encountered during the round trip.
- p gets (ℓ', u, h, b) with $\ell < \ell'$, or $\ell = \ell'$ and $id_p > u$: it becomes passive and sends $(\ell', u, h{+}1, b)$.
- p gets (ℓ', u, h, b) with $\ell > \ell'$, or $\ell = \ell'$ and $id_p < u$: it purges the message.
- p gets (ℓ, id_p, h, b) with $h < N$: it sends $(\ell, id_p, h{+}1, false)$.
- p gets $(\ell, id_p, N, false)$: it proceeds to an election at level $\ell{+}1$.
- p gets $(\ell, id_p, N, true)$: it becomes the leader.

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

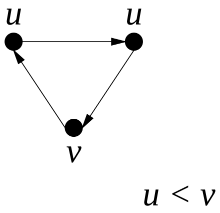
Itai-Rodeh Election Algorithm - Correctness + Complexity

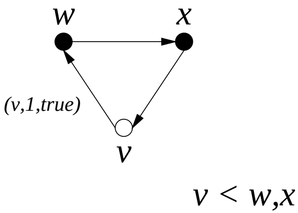
Correctness: Eventually one leader is elected, with probability 1.

Average-case message complexity: In the order of $N \log N$ messages.

Without levels, the algorithm would break down (if channels are non-FIFO).

Example:





Question

Any suggestions how to adapt the echo algorithm with extinction, to get an election algorithm for arbitrary anonymous undirected graphs?

Election in Arbitrary Anonymous Networks

We use the echo algorithm with extinction, with random selection of identities, for election in anonymous undirected graphs in which *all nodes know the network size*.

Initially, initiators are active at level 0, and non-initiators are passive.

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

Suppose process p in wave v at level ℓ is hit by wave w at level ℓ':

- if $\ell < \ell'$, or $\ell = \ell'$ and $v > w$, then p changes to wave w at level ℓ', and treats the message according to the echo algorithm;
- if $\ell > \ell'$, or $\ell = \ell'$ and $v < w$, then p purges the message;
- if $\ell = \ell'$ and $v = w$, then p treats the message according to the echo algorithm.

Election in Arbitrary Anonymous Networks

We use the echo algorithm with extinction, with random selection of identities, for election in anonymous undirected graphs in which *all nodes know the network size*.

Initially, initiators are active at level 0, and non-initiators are passive.

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

Suppose process p in wave v at level ℓ is hit by wave w at level ℓ':

- if $\ell < \ell'$, or $\ell = \ell'$ and $v > w$, then p changes to wave w at level ℓ', and treats the message according to the echo algorithm;
- if $\ell > \ell'$, or $\ell = \ell'$ and $v < w$, then p purges the message;
- if $\ell = \ell'$ and $v = w$, then p treats the message according to the echo algorithm.

Election in Arbitrary Networks

Each message sent upwards in the constructed tree reports the size of its subtree. All other messages report 0.

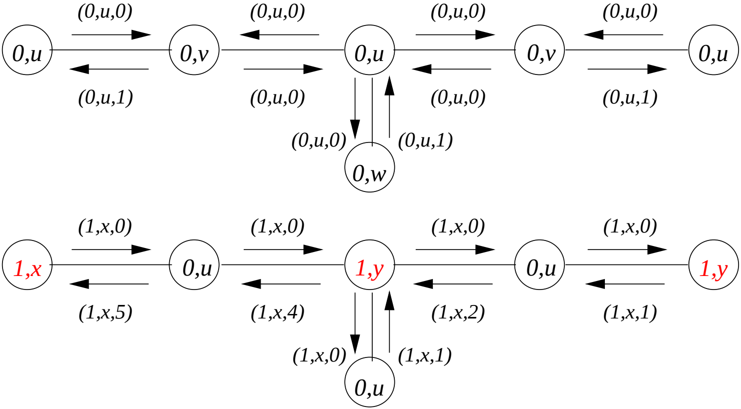
When a process p *decides*, it computes the size of the constructed tree.

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

Otherwise, it selects a new id, and initiates a new wave,
at a higher level.

Election in Arbitrary Networks - Example

$u < v < w < x < y$.   Only waves that complete are shown.



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

Computing the Size of a Network

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

When a leader is known, the network size can be computed by the echo algorithm.

Namely, each message sent upwards in the spanning tree reports the size of its subtree.

Hence there no Las Vegas algorithm for election in an anonymous ring if the nodes do not know the ring size.

Impossibility of Computing an Anonymous Network Size

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

Proof: Consider a directed ring $p_1, \ldots, p_N$.

Assume a probabilistic algorithm with a ρ-computation C that terminates with the correct outcome N.

Let each process execute at most L events in C.

Consider the ring $p_1, \ldots, p_{2N}$.

For $i = 1, \ldots, N$, let the first L bits of $\rho'(p_i)$ and $\rho'(p_{i+N})$ coincide with $\rho(p_i)$. (The probability of such an assignment is $(\frac{1}{2})^{NL}$.)

Let each event at a p_i in C be executed concurrently at p_i and p_{i+N}. This ρ'-computation terminates with the incorrect outcome N.

Itai-Rodeh Ring Size Algorithm

Each process p maintains an *estimate* est_p of the ring size. Always $est_p \leq N$; initially, $est_p = 2$.

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

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

- If $est < est_p$, p purges the message.
- Let $est > est_p$.
  - If $h < est$, then p sends $(est, id, h+1)$, and $est_p := est$.
  - If $h = est$, then $est_p := est+1$.
- Let $est = est_p$.
  - If $h < est$, then p sends $(est, id, h+1)$.
  - If $h = est$ and $id \neq id_p$, then $est_p := est+1$.
  - If $h = est$ and $id = id_p$, p purges the message (possibly its own token returned).

Itai-Rodeh Ring Size Algorithm

Each process p maintains an *estimate* est_p of the ring size. Always $est_p \leq N$; initially, $est_p = 2$.

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

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

- If $est < est_p$, p purges the message.
- Let $est > est_p$.
  - If $h < est$, then p sends $(est, id, h{+}1)$, and $est_p := est$.
  - If $h = est$, then $est_p := est{+}1$.
- Let $est = est_p$.
  - If $h < est$, then p sends $(est, id, h{+}1)$.
  - If $h = est$ and $id \neq id_p$, then $est_p := est{+}1$.
  - If $h = est$ and $id = id_p$, p purges the message (possibly its own token returned).

Question

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

Question

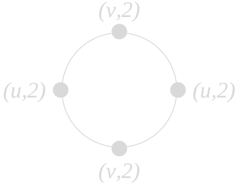
Why will est_p never become greater than N?

Itai-Rodeh Ring Size Algorithm - Correctness

Message-termination can be detected using a decentralized termination detection algorithm.

The Itai-Rodeh ring size algorithm is a Monte Carlo algorithm. Possibly, est_p is in the end smaller than the ring size.

Example:



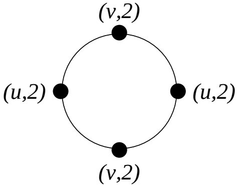
The probability of computing an *incorrect* ring size tends to zero when R tends to infinity.

Itai-Rodeh Ring Size Algorithm - Correctness

Message-termination can be detected using a decentralized termination detection algorithm.

The Itai-Rodeh ring size algorithm is a Monte Carlo algorithm. Possibly, est_p is in the end smaller than the ring size.

Example:



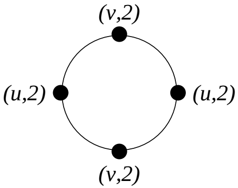
The probability of computing an *incorrect* ring size tends to zero when R tends to infinity.

Itai-Rodeh Ring Size Algorithm - Correctness

Message-termination can be detected using a decentralized termination detection algorithm.

The Itai-Rodeh ring size algorithm is a Monte Carlo algorithm. Possibly, est_p is in the end smaller than the ring size.

Example:



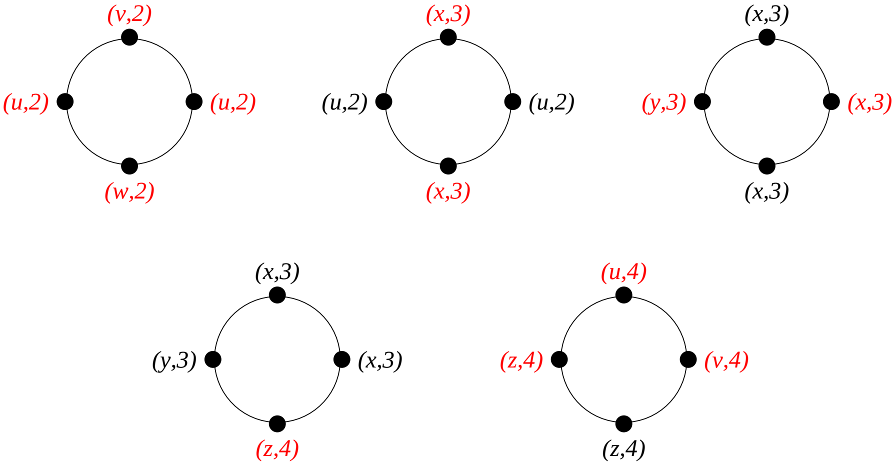
The probability of computing an *incorrect* ring size tends to zero when R tends to infinity.

Itai-Rodeh Ring Size Algorithm - Complexity

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

Each process starts at most $N-1$ estimate rounds; each round it sends out one message, which takes at most N steps.

Itai-Rodeh Ring Size Algorithm - Example



FireWire Election Algorithm

IEEE Standard 1394, called *FireWire*, is a serial multimedia bus. It connects digital devices, which can be added and removed dynamically.

It uses the *tree election algorithm* for undirected, acyclic graphs, adapted to *anonymous* networks. (Cyclic graphs give a time-out.)

The network size is unknown to the nodes!

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

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

Each of the two nodes 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 node.

This election algorithm is a Las Vegas algorithm for acyclic graphs; it terminates with probability 1.

FireWire Election Algorithm

IEEE Standard 1394, called *FireWire*, is a serial multimedia bus. It connects digital devices, which can be added and removed dynamically.

It uses the *tree election algorithm* for undirected, acyclic graphs, adapted to *anonymous* networks. (Cyclic graphs give a time-out.)

The network size is unknown to the nodes!

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

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

Each of the two nodes 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 node.

This election algorithm is a Las Vegas algorithm for acyclic graphs; it terminates with probability 1.

FireWire Election Algorithm

IEEE Standard 1394, called *FireWire*, is a serial multimedia bus. It connects digital devices, which can be added and removed dynamically.

It uses the *tree election algorithm* for undirected, acyclic graphs, adapted to *anonymous* networks. (Cyclic graphs give a time-out.)

The network size is unknown to the nodes!

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

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

Each of the two nodes 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 node.

This election algorithm is a Las Vegas algorithm for acyclic graphs; it terminates with probability 1.

FireWire Election Algorithm

IEEE Standard 1394, called *FireWire*, is a serial multimedia bus. It connects digital devices, which can be added and removed dynamically.

It uses the *tree election algorithm* for undirected, acyclic graphs, adapted to *anonymous* networks. (Cyclic graphs give a time-out.)

The network size is unknown to the nodes!

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

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

Each of the two nodes 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 node.

This election algorithm is a Las Vegas algorithm for acyclic graphs; it terminates with probability 1.

Question

In case of root contention, is it optimal to give an equal chance of 0.5 to both sending immediately and waiting for some time?

Question

Give a terminating algorithm for computing the network size of anonymous, acyclic graphs.

Routing Algorithms

See also Computer Networks (Chapter 5.2).

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

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

Criteria for good routing algorithms:

- use of optimal paths;
- robust with respect to topology changes in the network;
- computing routing tables is cheap;
- table adaptation to avoid busy channels;
- all nodes are served in the same degree.

All-Pairs Shortest-Path Problem

Let $G = (V, E)$ be a directed, weighted graph, with weights $\omega_{uv} > 0$.

We want to compute for each pair of nodes u, v a shortest path from u to v in G.

For $S \subseteq V$, $d^S(u, v)$ denotes the length of a shortest path in G with all intermediate nodes in S.

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

Note that d^V is the standard distance function.

All-Pairs Shortest-Path Problem

Let $G = (V, E)$ be a directed, weighted graph, with weights $\omega_{uv} > 0$.

We want to compute for each pair of nodes u, v a shortest path from u to v in G.

For $S \subseteq V$, $d^S(u, v)$ denotes the length of a shortest path in G with all intermediate nodes in S.

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

Note that d^V is the standard distance function.

Floyd-Warshall Algorithm

Exploits the last equality to compute d^S where S grows from $\emptyset$ to V.

```
$S := \emptyset$
**forall** $u, v \in V$ **do**
      **if** $u = v$ **then** $D_u[v] := 0$; $Nb_u[v] := \perp$
      **else if** $uv \in E$ **then** $D_u[v] := \omega_{uv}$; $Nb_u[v] := v$
            **else** $D_u[v] := \infty$; $Nb_u[v] := \perp$

**while** $S \neq V$ **do**
      pick $w$ from $V \backslash S$      ("$w$-pivot round")
      **forall** $u, v \in V$ **do**
            **if** $D_u[w] + D_w[v] < D_u[v]$ **then**
               $D_u[v] := D_u[w] + D_w[v]$;
               $Nb_u[v] := Nb_u[w]$
      $S := S \cup \{w\}$
```

Time complexity: $\Theta(|V|^3)$

Floyd-Warshall Algorithm

Exploits the last equality to compute d^S where S grows from $\emptyset$ to V.

```
$S := \emptyset$
forall $u, v \in V$ do
    if $u = v$ then $D_u[v] := 0$; $Nb_u[v] := \perp$
    else if $uv \in E$ then $D_u[v] := \omega_{uv}$; $Nb_u[v] := v$
        else $D_u[v] := \infty$; $Nb_u[v] := \perp$
while $S \neq V$ do
    pick $w$ from $V \backslash S$      (“$w$-pivot round”)
    forall $u, v \in V$ do
        if $D_u[w]+D_w[v] < D_u[v]$ then
            $D_u[v] := D_u[w]+D_w[v]$;
            $Nb_u[v] := Nb_u[w]$
    $S := S \cup \{w\}$
```

Time complexity: $\Theta(|V|^3)$

Floyd-Warshall Algorithm

Exploits the last equality to compute d^S where S grows from $\emptyset$ to V.

```
$S := \emptyset$
forall $u, v \in V$ do
    if $u = v$ then $D_u[v] := 0$; $Nb_u[v] := \perp$
    else if $uv \in E$ then $D_u[v] := \omega_{uv}$; $Nb_u[v] := v$
        else $D_u[v] := \infty$; $Nb_u[v] := \perp$

while $S \neq V$ do
    pick $w$ from $V \backslash S$      (“$w$-pivot round”)
    forall $u, v \in V$ do
        if $D_u[w] + D_w[v] < D_u[v]$ then
            $D_u[v] := D_u[w] + D_w[v]$;
            $Nb_u[v] := Nb_u[w]$
    $S := S \cup \{w\}$
```

Time complexity: $\Theta(|V|^3)$

Floyd-Warshall Algorithm - Example

pivot u

$$\begin{array}{llllll} D_x[v] & := & 5 & D_v[x] & := & 5 \\ N_x[v] & := & u & N_v[x] & := & u \end{array}$$

pivot v

$$\begin{array}{llllll} D_u[w] & := & 5 & D_w[u] & := & 5 \\ N_u[w] & := & v & N_w[u] & := & v \end{array}$$

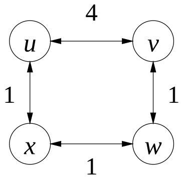
pivot w

$$\begin{array}{llllll} D_x[v] & := & 2 & D_v[x] & := & 2 \\ N_x[v] & := & w & N_v[x] & := & w \end{array}$$

pivot x

$$\begin{array}{llllll} D_u[w] & := & 2 & D_w[u] & := & 2 \\ N_u[w] & := & x & N_w[u] & := & x \end{array}$$

$$\begin{array}{llllll} D_u[v] & := & 3 & D_v[u] & := & 3 \\ N_u[v] & := & x & N_v[u] & := & w \end{array}$$



Question

How can the Floyd-Warshall algorithm be turned into a distributed algorithm?

Toueg's Algorithm

A distributed version of Floyd-Warshall computes the routing tables at their nodes.

Given an undirected, weighted graph.

Assumption: Each node knows from the start the identities of all nodes in V. (Because pivots must be picked uniformly at all nodes.)

At the w-pivot round, w broadcasts its values $D_w[v]$, for all $v \in V$.

If $Nb_u[w] = \perp$ with $u \neq w$ at the w-pivot round, then $D_u[w] = \infty$, so $D_u[w]+D_w[v] \geq D_u[v]$ for all $v \in V$. Hence the sink tree of w can be used to broadcast D_w.

Nodes u with $Nb_u[w] \neq \perp$ must tell $Nb_u[w]$ to pass on D_w:

- u sends $\langle ys, w \rangle$ to $Nb_u[w]$ if it is not $\perp$;
- u sends $\langle nys, w \rangle$ to its other neighbors.

Toueg's Algorithm

A distributed version of Floyd-Warshall computes the routing tables at their nodes.

Given an undirected, weighted graph.

Assumption: Each node knows from the start the identities of all nodes in V. (Because pivots must be picked uniformly at all nodes.)

At the w-pivot round, w broadcasts its values $D_w[v]$, for all $v \in V$.

If $Nb_u[w] = \perp$ with $u \neq w$ at the w-pivot round, then $D_u[w] = \infty$, so $D_u[w]+D_w[v] \geq D_u[v]$ for all $v \in V$. Hence the sink tree of w can be used to broadcast D_w.

Nodes u with $Nb_u[w] \neq \perp$ must tell $Nb_u[w]$ to pass on D_w:

- u sends $\langle ys, w \rangle$ to $Nb_u[w]$ if it is not $\perp$;
- u sends $\langle nys, w \rangle$ to its other neighbors.

Toueg's Algorithm - Local Algorithm at Node u

Initialization:

```
$S := \emptyset$
forall $v \in V$ do
    if $u = v$ then $D_u[v] := 0$; $Nb_u[v] := \perp$
    else if $v \in Neigh_u$ then $D_u[v] := \omega_{uv}$; $Nb_u[v] := v$
        else $D_u[v] := \infty$; $Nb_u[v] := \perp$
```

Toueg's Algorithm - Local Algorithm at Node u

```
while $S \neq V$ do
    pick $w$ from $V \backslash S$
    forall $x \in Neigh_u$ do
        if $Nb_u[w] = x$ then send $\langle ys, w \rangle$ to $x$
        else send $\langle nys, w \rangle$ to $x$
    $num\_rec_u := 0$
    while $num\_rec_u < |Neigh_u|$ do
        receive a $\langle ys, w \rangle$ or $\langle nys, w \rangle$ message;
        $num\_rec_u := num\_rec_u + 1$
    if $D_u[w] < \infty$ then
        if $u \neq w$ then receive $D_w$ from $Nb_u[w]$
        forall $x \in Neigh_u$ that sent $\langle ys, w \rangle$ do send $D_w$ to $x$
        forall $v \in V$ do
            if $D_u[w] + D_w[v] < D_u[v]$ then
                $D_u[v] := D_u[w] + D_w[v]$; $Nb_u[v] := Nb_u[w]$
    $S := S \cup \{w\}$
```

Toueg's Algorithm - Complexity and Drawbacks

Message complexity: $\Theta(|V|\cdot|E|)$

Drawbacks:

- uniform selection of pivot nodes requires that all nodes know V in advance;
- global broadcast of D_w at the w-pivot round;
- not robust with respect to topology changes.

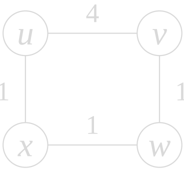
Toueg's Algorithm - Optimization

Let $Nb_x[w] = u$ with $u \neq w$ at the start of the w-pivot round. If $D_u[v]$ is not changed in this round, then neither is $D_x[v]$.

Upon reception of D_w, u can therefore first update D_u and N_u, and only forward values $D_w[v]$ for which $D_u[v]$ has changed.

Additional advantage: Cycle-free sink trees not only between but also during pivot rounds.

Example:



Subsequent pivots: u x w v.

At the w-pivot round, the sink tree toward v may temporarily contain a cycle: $Nb_x[v] = u$ and $Nb_u[v] = x$.

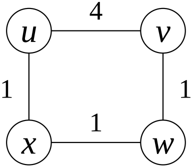
Toueg's Algorithm - Optimization

Let $Nb_x[w] = u$ with $u \neq w$ at the start of the w-pivot round. If $D_u[v]$ is not changed in this round, then neither is $D_x[v]$.

Upon reception of D_w, u can therefore first update D_u and N_u, and only forward values $D_w[v]$ for which $D_u[v]$ has changed.

Additional advantage: Cycle-free sink trees not only between but also during pivot rounds.

Example:



Subsequent pivots: u x w v.

At the w-pivot round, the sink tree toward v may temporarily contain a cycle: $Nb_x[v] = u$ and $Nb_u[v] = x$.

Chandy-Misra Algorithm

A centralized algorithm to compute all shortest paths to initiator v_0.

Again, an undirected, weighted graph is assumed.

Each node uses only $D_w[v_0]$ values from neighbors w.

Initially, $D_{v_0}[v_0] = 0$, $D_u[v_0] = \infty$ if $u \neq v_0$, and $Nb_u[v_0] = \perp$.

v_0 sends the message $\langle \mathbf{mydist}, 0 \rangle$ to its neighbors.

When a node u receives $\langle \mathbf{mydist}, d \rangle$ from a neighbor w, and if $d + \omega_{uw} < D_u[v_0]$, then:

- $D_u[v_0] := d + \omega_{uw}$ and $Nb_u[v_0] := w$;
- u sends $\langle \mathbf{mydist}, D_u[v_0] \rangle$ to its neighbors (except w).

Termination detection by the Dijkstra-Scholten algorithm.

Worst-case message complexity: Exponential

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

Chandy-Misra Algorithm

A centralized algorithm to compute all shortest paths to initiator v_0.

Again, an undirected, weighted graph is assumed.

Each node uses only $D_w[v_0]$ values from neighbors w.

Initially, $D_{v_0}[v_0] = 0$, $D_u[v_0] = \infty$ if $u \neq v_0$, and $Nb_u[v_0] = \perp$.

v_0 sends the message $\langle \textbf{mydist}, 0 \rangle$ to its neighbors.

When a node u receives $\langle \textbf{mydist}, d \rangle$ from a neighbor w, and if $d + \omega_{uw} < D_u[v_0]$, then:

- $D_u[v_0] := d + \omega_{uw}$ and $Nb_u[v_0] := w$;
- u sends $\langle \textbf{mydist}, D_u[v_0] \rangle$ to its neighbors (except w).

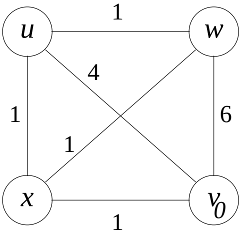
Termination detection by the Dijkstra-Scholten algorithm.

Worst-case message complexity: Exponential

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

Chandy-Misra Algorithm - Example

$$
\begin{array}{lllllll}
D_{v_0} & := & 0 & \qquad & Nb_{v_0} & := & \perp \\
D_w & := & 6 & & Nb_w & := & v_0 \\
D_u & := & 7 & & Nb_u & := & w \\
D_x & := & 8 & & Nb_x & := & u \\
D_x & := & 7 & & Nb_x & := & w \\
D_u & := & 4 & & Nb_u & := & v_0 \\
D_w & := & 5 & & Nb_w & := & u \\
D_x & := & 6 & & Nb_x & := & w \\
D_x & := & 5 & & Nb_x & := & u \\
D_x & := & 1 & & Nb_x & := & v_0 \\
D_w & := & 2 & & Nb_w & := & x \\
D_u & := & 3 & & Nb_u & := & w \\
D_u & := & 2 & & Nb_u & := & x
\end{array}
$$



Merlin-Segall Algorithm

A centralized algorithm to compute all shortest paths to initiator v_0.

Again, an undirected, weighted graph is assumed.

Initially, $D_{v_0}[v_0] = 0$, $D_u[v_0] = \infty$ if $u \neq v_0$, and the $Nb_u[v_0]$ form a sink tree with root v_0.

At each update round, for $u \neq v_0$:

1. v_0 sends $\langle \textbf{mydist}, 0 \rangle$ to its neighbors.
2. Let u get $\langle \textbf{mydist}, d \rangle$ from neighbor w.
   If $d + \omega_{uw} < D_u[v_0]$, $D_u[v_0] := d + \omega_{uw}$ (and u stores w as future value for $Nb_u[v_0]$).
   If $w = Nb_u[v_0]$, u sends $\langle \textbf{mydist}, D_u[v_0] \rangle$ to its neighbors *except* $Nb_u[v_0]$.
3. When u received a **mydist** message from all its neighbors, it sends $\langle \textbf{mydist}, D_u[v_0] \rangle$ to $Nb_u[v_0]$, and next updates $Nb_u[v_0]$.

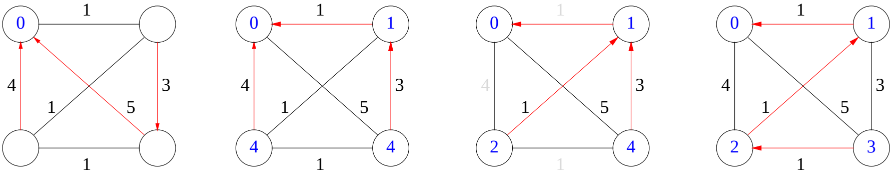
v_0 starts a new update round after receiving **mydist** from all neighbors.

Merlin-Segall Algorithm - Termination and Complexity

After i update rounds, all shortest paths of $\leq i$ hops have been computed. The algorithm terminates after $|V|-1$ update rounds.

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

Example:

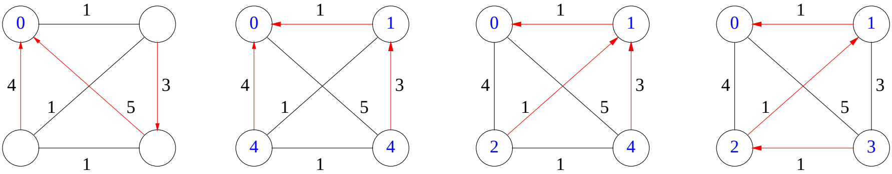


Merlin-Segall Algorithm - Termination and Complexity

After i update rounds, all shortest paths of $\leq i$ hops have been computed. The algorithm terminates after $|V|-1$ update rounds.

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

Example:



Merlin-Segall Algorithm - Topology Changes

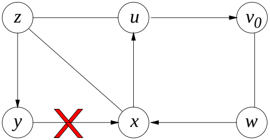
A number is attached to **mydist** messages.

When a channel fails or becomes operational, adjacent nodes send the number of the update round to v_0 via the sink tree.
(If the message meets a failed tree link, it is discarded.)

When v_0 receives such a message, it starts a new set of $|V|-1$ update rounds, with a *higher number*.

If a failed channel is part of the sink tree, the remaining tree is extended to a sink tree, similar to the initialization phase.

Example:



y signals to z that the failed channel was part of the sink tree.

Breadth-First Search

Consider an undirected graph.

A spanning tree is a breadth-first search tree if each tree path to the root is minimum-hop.

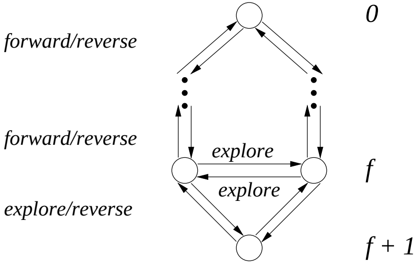
The Chandy-Misra algorithm for minimum-hop paths computed a breadth-first search tree using $O(|V|\cdot|E|)$ messages (for each root).

Breadth-First Search - A “Simple” Algorithm

Initially (after round 0), the initiator is at level 0, and all other nodes are at level ∞.

After round $f \geq 0$, each node at f hops from the node (1) is at level f, and (2) knows which neighbors are at level $f-1$.

We explain what happens in round $f+1$:



Breadth-First Search - A "Simple" Algorithm

- Messages $\langle \textbf{forward}, f \rangle$ travel down the tree, from the initiator to the nodes at level f.

- A node at level f sends $\langle \textbf{explore}, f{+}1 \rangle$ to all neighbors that are not at level $f{-}1$.

  It stores incoming messages $\langle \textbf{explore}, f{+}1 \rangle$ and $\langle \textbf{reverse}, b \rangle$ as answers.

- If a node at level ∞ gets $\langle \textbf{explore}, f{+}1 \rangle$, its level becomes $f{+}1$, and the sender becomes its father. It sends back $\langle \textbf{reverse}, \mathit{true} \rangle$.

- If a node at level $f{+}1$ gets $\langle \textbf{explore}, f{+}1 \rangle$, it stores that the sender is at level f. It sends back $\langle \textbf{reverse}, \mathit{false} \rangle$.

Breadth-First Search - A “Simple” Algorithm

- A non-initiator at level f (or $< f$) waits until all messages $\langle \textbf{explore}, f{+}1\rangle$ (resp. $\langle \textbf{forward}, f\rangle$) have been answered.

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

- The initiator waits until all $\langle \textbf{forward}, f\rangle$ (or, in round 1, $\langle \textbf{explore}, 1\rangle$) messages are answered.

  It continues with round $f{+}2$ if new nodes were added in round $f{+}1$. Otherwise it terminates.

Breadth-First Search - Complexity

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

There are at most $|V|$ rounds.

Each round, a tree edge carries at most one **forward** and one replying **reverse**.

In total, an edge carries one **explore** and one replying **reverse** or **explore**.

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

Round f is completed in at most $2f$ time units.

Frederickson's Algorithm

Computes $\ell \geq 1$ levels per round.

Initially, the initiator is at level 0, and all other nodes are at level ∞.

After round k, each node at $f \leq k\ell$ hops from the node (1) is at level f, and (2) knows which neighbors are at level $f{-}1$.

At round $k{+}1$:

- $\langle$**forward**$, k\ell\rangle$ travels down the tree, from the initiator to the nodes at level $k\ell$.

  forward *messages from non-fathers are replied with* $\langle$**no-child**$\rangle$*.*

Frederickson's Algorithm

- A node at level $k\ell$ sends $\langle \textbf{explore}, k\ell{+}1, \ell\rangle$ to neighbors that are not at level $k\ell{-}1$.

- If a node u receives $\langle \textbf{explore}, f, m\rangle$ with $f < \mathit{level}_u$, then $\mathit{level}_u := f$, and the sender becomes u's new father.

  If $m > 1$, u sends $\langle \textbf{explore}, f{+}1, m{-}1\rangle$ to those neighbors that are not already known to be at level $f{-}1$, f or $f{+}1$.

  If $m = 1$ (so $f = (k{+}1)\ell$), u sends back $\langle \textbf{reverse}, \mathit{true}\rangle$.

- If a node u receives $\langle \textbf{explore}, f, m\rangle$ with $f \geq \mathit{level}_u$, and u did not send $\langle \textbf{explore}, \mathit{level}_u, \_\rangle$ into this channel, it sends back $\langle \textbf{reverse}, \mathit{false}\rangle$.

Frederickson's Algorithm

- A non-initiator at level $k\ell \leq f < (k{+}1)\ell$ (or $f < k\ell$) waits until all $\langle$**explore**$, f{+}1\rangle$ (resp. $\langle$**forward**$, k\ell\rangle$) messages have been answered.

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

- The initiator waits until all $\langle$**forward**$, k\ell\rangle$ (or, in round 1, $\langle$**explore**$, 1, \ell\rangle$) messages are answered.

  It continues with round $k{+}2$ if new nodes were added in round $k{+}1$. Otherwise it terminates.

Frederickson's Algorithm - Complexity

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

There are at most $\lceil\frac{|V|}{\ell}\rceil$ rounds.

Each round, a tree edge carries at most one **forward** and one replying **reverse**.

In total, an edge carries at most 2ℓ **explore**'s and replying **reverse**'s.

In total, a frond edge carries at most one spurious **forward** and one replying **no-child**.

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

Levels $k\ell+1$ up to $(k+1)\ell$ are computed in $2(k+1)\ell$ time units.

Let $\ell = \lceil\frac{|V|}{\sqrt{|E|}}\rceil$. Then both message and time complexity are $O(|V|\sqrt{|E|})$.

Deadlock-Free Packet Switching

Let $G = (V, E)$ be a directed graph, supplied with routing tables.

Each node has buffers to store data packets on their way to their destination.

For simplicity we assume synchronous communication.

Possible events:

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

At a node with all buffers empty, generation of a packet must always be allowed.

Store-and-Forward Deadlocks

A store-and-forward deadlock occurs when a group of packets are all waiting for the use of a buffer 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 it is placed next.

Destination Scheme

$V = \{v_1, \ldots, v_N\}$, and T_i denotes the sink tree (with respect to the routing tables) with root v_i for $i = 1, \ldots, N$.

In the destination scheme, each node carries N buffers.

- When a packet with destination v_i is generated at u, it is placed in the ith buffer of u.
- If uv is an edge in T_i, then the ith buffer of u is linked to the ith buffer of v.

Hops-So-Far Scheme

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

In the hops-so-far scheme, each node carries $k+1$ buffers.

- When a packet is generated at u, it is placed in the first buffer of u.
- If uv is an edge in some T_i, then each jth buffer of u is linked to the $(j+1)$th buffer of v.

Forward-Count Controller

Suppose that for a packet p at a node u, the number $s_u(p)$ of hops that p still has to make to its destination is always known.

f_u is the number of free buffers at u.

k_u is the length of a longest path, starting in u, in any sink tree in G.

In the forward-count controller, each node u contains k_u+1 buffers. A packet p is accepted at node u if and only if $s_u(p) < f_u$.

If the buffers of a node u are all empty, u can accept any packet.

Unlike the previous controllers, an accepted packet can be placed in any buffer.

Forward-Count Controller

Suppose that for a packet p at a node u, the number $s_u(p)$ of hops that p still has to make to its destination is always known.

f_u is the number of free buffers at u.

k_u is the length of a longest path, starting in u, in any sink tree in G.

In the forward-count controller, each node u contains k_u+1 buffers. A packet p is accepted at node u if and only if $s_u(p) < f_u$.

If the buffers of a node u are all empty, u can accept any packet.

Unlike the previous controllers, an accepted packet can be placed in any buffer.

Forward-Count Controller - Correctness

Theorem: Forward-count controllers are deadlock-free.

Proof: Consider a reachable configuration δ where no forwarding or consumption is possible. Suppose, toward a contradiction, that in δ some buffer is occupied.

Select a packet p, at some node u, with $s_u(p)$ minimal. p must be forwarded to a node w, but is blocked:

$$s_w(p) \geq f_w$$

Then some buffer in w is occupied. Let q be the packet at w that arrived last. Let f_w^{old} be the number of free buffers before q's arrival.

$$s_w(q) < f_w^{old} \leq f_w{+}1$$

Hence, we get a contradiction:

$$s_u(p) = s_w(p){+}1 \geq f_w{+}1 > s_w(q)$$

So in δ, all buffers are empty.

Forward-Count Controller - Correctness

Theorem: Forward-count controllers are deadlock-free.

Proof: Consider a reachable configuration δ where no forwarding or consumption is possible. Suppose, toward a contradiction, that in δ some buffer is occupied.

Select a packet p, at some node u, with $s_u(p)$ minimal. p must be forwarded to a node w, but is blocked:

$$s_w(p) \geq f_w$$

Then some buffer in w is occupied. Let q be the packet at w that arrived last. Let f_w^{old} be the number of free buffers before q's arrival.

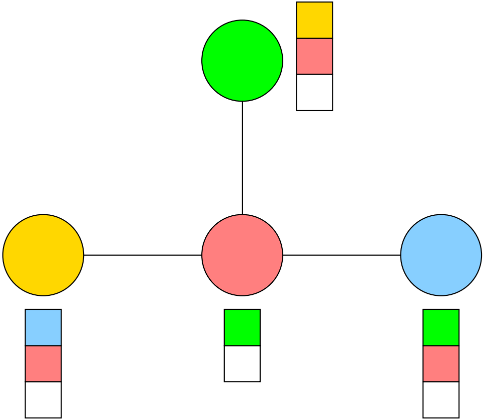
$$s_w(q) < f_w^{old} \leq f_w+1$$

Hence, we get a contradiction:

$$s_u(p) = s_w(p)+1 \geq f_w+1 > s_w(q)$$

So in δ, all buffers are empty.

Example



Acyclic Orientation Cover Controller

Let G be undirected. An acyclic orientation of G is a directed, acyclic graph obtained by directing all edges of G.

Acyclic orientations $G_1, \ldots, G_n$ of G are an acyclic orientation cover of a set $\mathcal{P}$ of paths in G if each path in $\mathcal{P}$ is the concatenation of paths $P_1, \ldots, P_n$ in $G_1, \ldots, G_n$.

Given an acyclic orientation cover $G_1, \ldots, G_n$ of the sink trees. In the acyclic orientation cover controller, each node has n buffers.

- A packet generated at v, is placed in the first buffer of v.
- If vw is an edge in G_i, then the ith buffer of v is linked to the ith buffer of w, and if $i < n$, the ith buffer of w is linked to the $(i+1)$th buffer of v.

Acyclic Orientation Cover Controller

Let G be undirected. An acyclic orientation of G is a directed, acyclic graph obtained by directing all edges of G.

Acyclic orientations $G_1, \ldots, G_n$ of G are an acyclic orientation cover of a set $\mathcal{P}$ of paths in G if each path in $\mathcal{P}$ is the concatenation of paths $P_1, \ldots, P_n$ in $G_1, \ldots, G_n$.

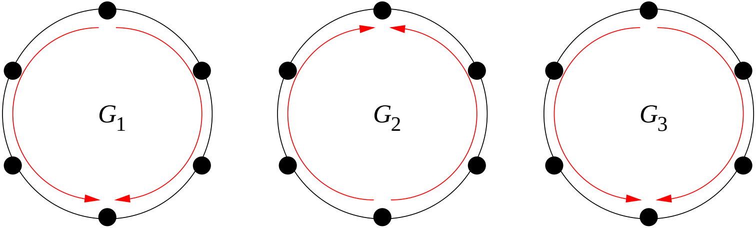
Given an acyclic orientation cover $G_1, \ldots, G_n$ of the sink trees. In the acyclic orientation cover controller, each node has n buffers.

- A packet generated at v, is placed in the first buffer of v.
- If vw is an edge in G_i, then the ith buffer of v is linked to the ith buffer of w, and if $i < n$, the ith buffer of w is linked to the $(i+1)$th buffer of v.

Example

For each undirected ring there exists a deadlock-free controller that uses three buffers per node and allows packets to travel via minimum-hop paths.

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



Acyclic Orientation Cover Controller - Correctness

Theorem: Acyclic orientation cover controllers are 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 is acyclic, packets in nth buffers can travel to their destinations. So in δ, nth buffers are empty.

Suppose all $(i{+}1)$th buffers are empty in δ, for some $i < n$. Then all ith buffers must also be empty in δ. For else, since G_i is acyclic, some packet in an ith buffer could be forwarded or consumed.

Concluding, in δ all buffers are empty.

Acyclic Orientation Cover Controller - Correctness

Theorem: Acyclic orientation cover controllers are 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 is acyclic, packets in nth buffers can travel to their destinations. So in δ, nth buffers are empty.

Suppose all $(i{+}1)$th buffers are empty in δ, for some $i < n$. Then all ith buffers must also be empty in δ. For else, since G_i is acyclic, some packet in an ith buffer could be forwarded or consumed.

Concluding, in δ all buffers are empty.

Fault Tolerance

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

Assumptions: The graph is *complete*, i.e., there is an undirected channel between each pair of different processes. Thus, failing processes never make the network disconnected.

Crashing of processes cannot be observed.

Consensus: Correct processes must uniformly decide 0 or 1.

Assumption: Initial configurations are bivalent, meaning that both decisions occur in some terminal configurations (that are reachable by *correct* transitions).

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

Fault Tolerance

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

Assumptions: The graph is *complete*, i.e., there is an undirected channel between each pair of different processes. Thus, failing processes never make the network disconnected.

Crashing of processes cannot be observed.

Consensus: Correct processes must uniformly decide 0 or 1.

Assumption: Initial configurations are bivalent, meaning that both decisions occur in some terminal configurations (that are reachable by *correct* transitions).

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

Impossibility of 1-Crash Consensus

Theorem: There is no terminating algorithm for 1-crash consensus (i.e., only one process may crash).

Proof: Suppose, toward a contradiction, there is such an algorithm. Let γ be a reachable bivalent configuration. Then $\gamma \rightarrow \gamma_0$ and $\gamma \rightarrow \gamma_1$, where γ_0 can lead to decision 0 and γ_1 to decision 1.

- Suppose these transitions correspond to events at different processes. Then $\gamma_0 \rightarrow \delta \leftarrow \gamma_1$ for some δ. So γ_0 or γ_1 is bivalent.
- Suppose these two transitions correspond to events at the same process *p*. In γ, *p* can crash, so the other processes are *b*-potent for some *b*. Then in γ_0 and γ_1, the processes except *p* are *b*-potent. So γ_0 or γ_1 is bivalent.

Concluding, each reachable bivalent configuration can make a transition to a bivalent configuration. Since initial configurations are bivalent, there is an execution visiting only bivalent configurations.

Note: There even exists a *fair* infinite execution.

Impossibility of 1-Crash Consensus

Theorem: There is no terminating algorithm for 1-crash consensus (i.e., only one process may crash).

Proof: Suppose, toward a contradiction, there is such an algorithm. Let γ be a reachable bivalent configuration. Then $\gamma \rightarrow \gamma_0$ and $\gamma \rightarrow \gamma_1$, where γ_0 can lead to decision 0 and γ_1 to decision 1.

- Suppose these transitions correspond to events at different processes. Then $\gamma_0 \rightarrow \delta \leftarrow \gamma_1$ for some δ. So γ_0 or γ_1 is bivalent.
- Suppose these two transitions correspond to events at the same process *p*. In γ, *p* can crash, so the other processes are *b*-potent for some *b*. Then in γ_0 and γ_1, the processes except *p* are *b*-potent. So γ_0 or γ_1 is bivalent.

Concluding, each reachable bivalent configuration can make a transition to a bivalent configuration. Since initial configurations are bivalent, there is an execution visiting only bivalent configurations.

Note: There even exists a *fair* infinite execution.

Impossibility of 1-Crash Consensus

Theorem: There is no terminating algorithm for 1-crash consensus (i.e., only one process may crash).

Proof: Suppose, toward a contradiction, there is such an algorithm. Let γ be a reachable bivalent configuration. Then $\gamma \rightarrow \gamma_0$ and $\gamma \rightarrow \gamma_1$, where γ_0 can lead to decision 0 and γ_1 to decision 1.

- Suppose these transitions correspond to events at different processes. Then $\gamma_0 \rightarrow \delta \leftarrow \gamma_1$ for some δ. So γ_0 or γ_1 is bivalent.
- Suppose these two transitions correspond to events at the same process p. In γ, p can crash, so the other processes are b-potent for some b. Then in γ_0 and γ_1, the processes except p are b-potent. So γ_0 or γ_1 is bivalent.

Concluding, each reachable bivalent configuration can make a transition to a bivalent configuration. Since initial configurations are bivalent, there is an execution visiting only bivalent configurations.

Note: There even exists a *fair* infinite execution.

Impossibility of $\lceil \frac{N}{2} \rceil$-Crash Consensus

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

Proof: Suppose, toward a contradiction, there is such an algorithm. Partition the set of processes into S and T, with $|S| = \lfloor \frac{N}{2} \rfloor$ and $|T| = \lceil \frac{N}{2} \rceil$.

In reachable configurations, S and T are either both 0-potent or both 1-potent. For else, since $t \geq \frac{N}{2}$, S and T could independently decide for different values.

Since the initial configuration is bivalent, there is a reachable configuration γ and a transition $\gamma \to \delta$ with S and T both only b-potent in γ and only $(1-b)$-potent in δ. Such a transition cannot exist.

Impossibility of $\lceil \frac{N}{2} \rceil$-Crash Consensus

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

Proof: Suppose, toward a contradiction, there is such an algorithm. Partition the set of processes into S and T, with $|S| = \lfloor \frac{N}{2} \rfloor$ and $|T| = \lceil \frac{N}{2} \rceil$.

In reachable configurations, S and T are either both 0-potent or both 1-potent. For else, since $t \geq \frac{N}{2}$, S and T could independently decide for different values.

Since the initial configuration is bivalent, there is a reachable configuration γ and a transition $\gamma \rightarrow \delta$ with S and T both only b-potent in γ and only $(1-b)$-potent in δ. Such a transition cannot exist.

Question

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

Bracha-Toueg Crash Consensus Algorithm

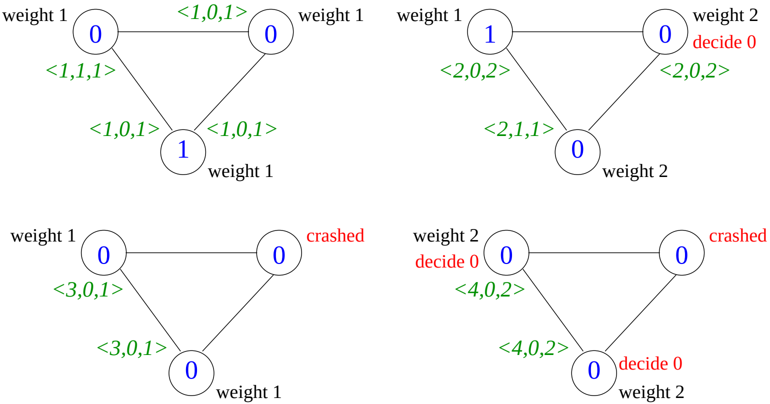
Let $t < \frac{N}{2}$. Initially, each correct process randomly chooses a value 0 or 1, with weight 1. In round k, at each correct, undecided p:

- p sends $\langle k, value_p, weight_p \rangle$ to all processes (including itself).
- p waits till $N{-}t$ messages $\langle k, b, w \rangle$ arrived. (p purges/stores messages from earlier/future rounds.)
  If $w > \frac{N}{2}$ for a $\langle k, b, w \rangle$, then $value_p := b$. *(This b is unique.)*
  Else, $value_p := 0$ if most messages voted 0, and $value_p := 1$ otherwise.
  $weight_p$ is changed into the number of incoming votes for $value_p$ in round k.
- If $w > \frac{N}{2}$ for more than t incoming messages $\langle k, b, w \rangle$, then p decides b. *(Note that $t < N{-}t$.)*

When p decides b, it broadcasts $\langle k{+}1, b, N{-}t \rangle$ and $\langle k{+}2, b, N{-}t \rangle$, and terminates.

Bracha-Toueg Crash Consensus Algorithm - Example

$N = 3$ and $t = 1$. 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 are not depicted.)



Bracha-Toueg Crash Consensus Algorithm - Correctness

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

Proof (part I): Suppose a process decides b in round k. Then in this round, $value_q = b$ and $weight_q > \frac{N}{2}$ for more than t processes q.

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

Hence, in round $k{+}1$, all correct processes vote b.

Then, after round $k{+}2$, all correct processes have decided b.

Concluding, all correct processes decide for the same value.

Bracha-Toueg Crash Consensus Algorithm - Correctness

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

Proof (part I): Suppose a process decides b in round k. Then in this round, $value_q = b$ and $weight_q > \frac{N}{2}$ for more than t processes q.

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

Hence, in round $k{+}1$, all correct processes vote b.

Then, after round $k{+}2$, 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.

Let S be a set of $N-t$ processes that do not crash.

Due to fair scheduling, there is a chance $\rho > 0$ that in a round each process in S receives its first $N-t$ messages from processes in S.

So with chance ρ^3 this happens in consecutive rounds $k, k+1, k+2$.

After round k, all processes in S have the same value b.

After round $k+1$, all processes in S have weight $N-t > \frac{N}{2}$.

Since $N-t > t$, after round $k+2$, all processes in S have decided b.

Concluding, the algorithm terminates with probability 1.

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

Theorem: Let $t \geq \frac{N}{3}$. There is no t-Byzantine consensus algorithm.

Proof: Suppose, toward a contradiction, that there is such an algorithm. Since $t \geq \frac{N}{3}$, we can choose sets S and T of processes with $|S| = |T| = N{-}t$ and $|S \cap T| \leq t$.

Let configuration γ be reachable by a sequence of *correct* transitions (so that still any process can become Byzantine). In γ, S and T are either both 0-potent or both 1-potent. For else, since the processes in $S \cap T$ may become Byzantine, S and T could independently decide for different values.

Since the initial configuration is bivalent, there is a configuration γ, reachable by correct transitions, and a correct transition $\gamma \rightarrow \delta$, with S and T both only b-potent in γ and only $(1{-}b)$-potent in δ. Such a transition cannot exist.

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

Theorem: Let $t \geq \frac{N}{3}$. There is no t-Byzantine consensus algorithm.

Proof: Suppose, toward a contradiction, that there is such an algorithm. Since $t \geq \frac{N}{3}$, we can choose sets S and T of processes with $|S| = |T| = N{-}t$ and $|S \cap T| \leq t$.

Let configuration γ be reachable by a sequence of *correct* transitions (so that still any process can become Byzantine). In γ, S and T are either both 0-potent or both 1-potent. For else, since the processes in $S \cap T$ may become Byzantine, S and T could independently decide for different values.

Since the initial configuration is bivalent, there is a configuration γ, reachable by correct transitions, and a correct transition $\gamma \rightarrow \delta$, with S and T both only b-potent in γ and only $(1{-}b)$-potent in δ. Such a transition cannot exist.

Bracha-Toueg Byzantine Consensus Algorithm

Let $t < \frac{N}{3}$. Bracha and Toueg gave a t-Byzantine consensus algorithm.

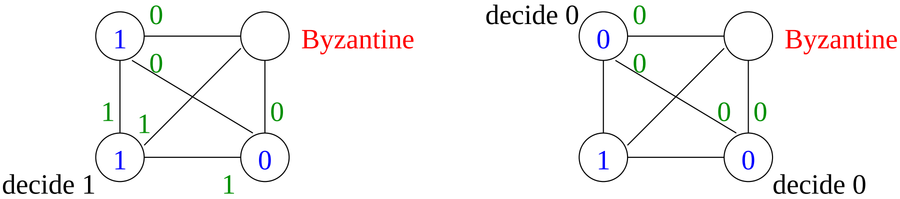
Again, in every round, correct processes broadcast their value, and wait for $N{-}t$ incoming messages. *(No weights are needed.)*

A correct process decides b upon receiving more than $\frac{N-t}{2}+t = \frac{N+t}{2}$ b-votes in one round. *(Note that $\frac{N+t}{2} < N{-}t$.)*

Echo Mechanism

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

Example: Let $N = 4$ and $t = 1$. Each round, a correct process waits for 3 votes, and needs 3 b-votes to decide b.



Solution: Each incoming vote is verified using an echo mechanism. A vote is accepted after more than $\frac{N+t}{2}$ confirming echos.

Bracha-Toueg Byzantine Consensus Algorithm

Initially, each correct process randomly chooses 0 or 1.
In round k, at each correct, undecided p:

- p broadcasts $\langle \textbf{in}, k, value_p \rangle$.
- If p receives $\langle \textbf{in}, \ell, b \rangle$ with $\ell \leq k$ from q, it broadcasts $\langle \textbf{ec}, q, \ell, b \rangle$.
- p counts incoming $\langle \textbf{ec}, q, k, b \rangle$ messages for each q, b. When more than $\frac{N+t}{2}$ such messages arrived, p accepts q's b-vote.
- p *purges* $\langle \textbf{ec}, q, \ell, b \rangle$ with $\ell < k$, and *stores* $\langle \textbf{in}, \ell, b \rangle$ and $\langle \textbf{ec}, q, \ell, b \rangle$ with $\ell > k$.
- The round is completed when p has accepted $N{-}t$ votes.
  If most votes are for 0, then $value_p := 0$. Else, $value_p := 1$.

Bracha-Toueg Byzantine Consensus Algorithm

p keeps track whether it received multiple messages $\langle \mathbf{in}, \ell, \_ \rangle$ or $\langle \mathbf{ec}, q, \ell, \_ \rangle$ via the same channel. (The sender must be Byzantine.) p only takes into account the first of these messages.

If more than $\frac{N+t}{2}$ of the accepted votes were for b, then p decides b.

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

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

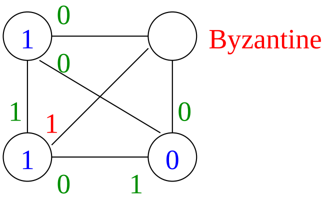
Question

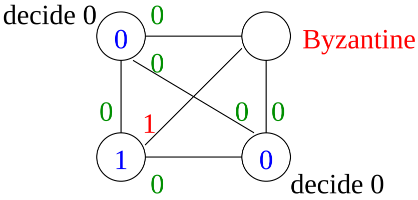
If an undecided process receives $\langle \textbf{decide}, b \rangle$, why can it in general not immediately decide b?

Example

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

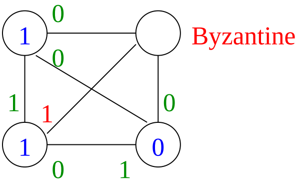
Let $N = 4$ and $t = 1$. Each round, a correct process needs 3 confirmations to accept a vote, 3 accepted votes, and 3 b-votes to decide b.

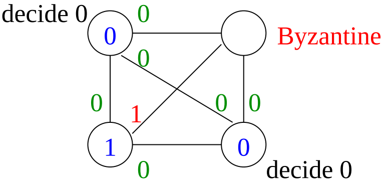




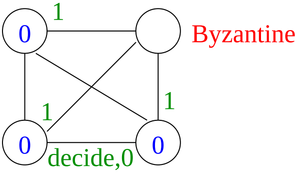
Only relevant **in** messages are depicted (without their round number).

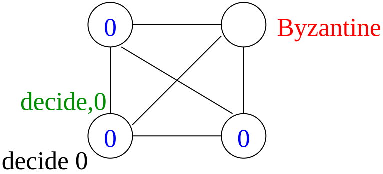
Example





In the first round, the left bottom node does *not* 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 node, and thus does *not* decide 1 in the first round.





Bracha-Toueg Byzantine Consensus Alg. - Correctness

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

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

Suppose in round k, correct processes p and q accept votes for b and b', respectively, from a process r. Then p and q received more than $\frac{N+t}{2}$ messages $\langle \mathbf{ec}, r, k, b \rangle$ and $\langle \mathbf{ec}, r, k, b' \rangle$, respectively. More than t processes, so at least one correct process, sent such messages to both p and q. Then $b = b'$.

Bracha-Toueg Byzantine Consensus Alg. - Correctness

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

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

Suppose in round k, correct processes p and q accept votes for b and b', respectively, from a process r. Then p and q received more than $\frac{N+t}{2}$ messages $\langle \mathbf{ec}, r, k, b \rangle$ and $\langle \mathbf{ec}, r, k, b' \rangle$, respectively. More than t processes, so at least one correct process, sent such messages to both p and q. Then $b = b'$.

Bracha-Toueg Byzantine Consensus Alg. - Correctness

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

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

Suppose in round k, correct processes p and q accept votes for b and b', respectively, from a process r. Then p and q received more than $\frac{N+t}{2}$ messages $\langle \mathbf{ec}, r, k, b \rangle$ and $\langle \mathbf{ec}, r, k, b' \rangle$, respectively. More than t processes, so at least one correct process, sent such messages to both p and q. Then $b = b'$.

Bracha-Toueg Byzantine Consensus Alg. - Correctness

Suppose a correct process decides b in round k. In this round it accepts more than $\frac{N+t}{2}$ b-votes. So in round k, each correct process accepts more than $\frac{N+t}{2} - t = \frac{N-t}{2}$ b-votes. Hence, in round $k{+}1$, $value_q = b$ for each correct q. This implies that the correct processes vote b in all rounds $\ell > k$. (Namely, in rounds $\ell > k$, each correct process accepts at least $N{-}2t > \frac{N-t}{2}$ b-votes.)

Let S be a set of $N{-}t$ processes that do not become Byzantine. Due to fair scheduling, there is a chance $\rho > 0$ that in a round each process in S accepts $N{-}t$ votes from processes in S. So there is a chance ρ^2 that this happens in consecutive rounds $k, k{+}1$. After round k, all processes in S have the same value b. After round $k{+}1$, all processes in S have decided b.

Bracha-Toueg Byzantine Consensus Alg. - Correctness

Suppose a correct process decides b in round k. In this round it accepts more than $\frac{N+t}{2}$ b-votes. So in round k, each correct process accepts more than $\frac{N+t}{2} - t = \frac{N-t}{2}$ b-votes. Hence, in round $k{+}1$, $value_q = b$ for each correct q. This implies that the correct processes vote b in all rounds $\ell > k$. (Namely, in rounds $\ell > k$, each correct process accepts at least $N{-}2t > \frac{N-t}{2}$ b-votes.)

Let S be a set of $N{-}t$ processes that do not become Byzantine. Due to fair scheduling, there is a chance $\rho > 0$ that in a round each process in S accepts $N{-}t$ votes from processes in S. So there is a chance ρ^2 that this happens in consecutive rounds $k, k{+}1$. After round k, all processes in S have the same value b. After round $k{+}1$, all processes in S have decided b.

Failure Detectors and Synchronous Systems

A failure detector at a process keeps track which processes have (or may have) crashed.

Given a (known or unknown) upper bound on network delay, and heartbeat messages by each process, one can implement a failure detector.

In a synchronous system, processes execute in *lock step*.

Given local clocks that have a known bounded inaccuracy, and a known upper bound on network delay, one can transform a system based on asynchronous communication into a synchronous system.

With a failure detector, and for a synchronous system, the proof for impossibility of 1-crash consensus no longer applies. Consensus algorithms have been developed for these settings.

Failure Detectors and Synchronous Systems

A failure detector at a process keeps track which processes have (or may have) crashed.

Given a (known or unknown) upper bound on network delay, and heartbeat messages by each process, one can implement a failure detector.

In a synchronous system, processes execute in *lock step*.

Given local clocks that have a known bounded inaccuracy, and a known upper bound on network delay, one can transform a system based on asynchronous communication into a synchronous system.

With a failure detector, and for a synchronous system, the proof for impossibility of 1-crash consensus no longer applies. Consensus algorithms have been developed for these settings.

Failure Detectors and Synchronous Systems

A failure detector at a process keeps track which processes have (or may have) crashed.

Given a (known or unknown) upper bound on network delay, and heartbeat messages by each process, one can implement a failure detector.

In a synchronous system, processes execute in *lock step*.

Given local clocks that have a known bounded inaccuracy, and a known upper bound on network delay, one can transform a system based on asynchronous communication into a synchronous system.

With a failure detector, and for a synchronous system, the proof for impossibility of 1-crash consensus no longer applies. Consensus algorithms have been developed for these settings.

Failure Detection

Aim: To detect *crashed* processes.

T is the time domain. $F(\tau)$ is the set of crashed processes at time τ.

A process cannot observe τ and $F(\tau)$.

$\tau_1 \leq \tau_2 \Rightarrow F(\tau_1) \subseteq F(\tau_2)$ (i.e., no restart).

$\mathit{Crash}(F) = \cup_{\tau \in T} F(\tau)$, and $H(p,\tau)$ is the set of processes *suspected* to be crashed by process p at time τ.

Each execution is decorated with a failure pattern F and a failure detector history H.

We require that a failure detector is complete: From some time onward, every crashed process is suspected by every correct process.

$$\exists \tau \; \forall p \in \mathit{Crash}(F) \; \forall q \notin \mathit{Crash}(F) \; \forall \tau' \geq \tau \quad p \in H(q,\tau')$$

Failure Detection

Aim: To detect *crashed* processes.

T is the time domain. $F(\tau)$ is the set of crashed processes at time τ.

A process cannot observe τ and $F(\tau)$.

$\tau_1 \leq \tau_2 \Rightarrow F(\tau_1) \subseteq F(\tau_2)$ (i.e., no restart).

$Crash(F) = \cup_{\tau \in T} F(\tau)$, and $H(p,\tau)$ is the set of processes *suspected* to be crashed by process p at time τ.

Each execution is decorated with a failure pattern F and a failure detector history H.

We require that a failure detector is complete: From some time onward, every crashed process is suspected by every correct process.

$$\exists \tau \; \forall p \in Crash(F) \; \forall q \notin Crash(F) \; \forall \tau' \geq \tau \quad p \in H(q,\tau')$$

Strongly Accurate Failure Detection

A failure detector is strongly accurate if only crashed processes are suspected:

$$\forall \tau\ \forall p, q \notin F(\tau)\quad p \notin H(q, \tau)$$

Assumptions:

- Each correct process broadcasts **alive** every σ time units.
- μ is an upper bound on communication delay.

Each process from which no message is received for $\sigma + \mu$ time units has crashed.

This failure detector is *complete* and *strongly accurate*.

Strongly Accurate Failure Detection

A failure detector is strongly accurate if only crashed processes are suspected:

$$\forall \tau \ \forall p, q \notin F(\tau) \ \ p \notin H(q, \tau)$$

Assumptions:

- Each correct process broadcasts **alive** every σ time units.
- μ is an upper bound on communication delay.

Each process from which no message is received for $\sigma+\mu$ time units has crashed.

This failure detector is *complete* and *strongly accurate*.

Weakly Accurate Failure Detection

With a failure detector, the proof for impossibility of 1-crash consensus no longer applies, if for instance from some time on some process is never suspected by any process.

A failure detector is weakly accurate if some process is never suspected:

$$\exists p \ \forall \tau \ \forall q \notin F(\tau) \ \ p \notin H(q,\tau)$$

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: $p_1, \ldots, p_N$. Initially, each process has value 0 or 1. In round k:

- p_k (if not crashed) broadcasts its value.
- Each process waits:
  - either for an incoming message from p_k, in which case it adopts the value of p_k;
  - or until it suspects that p_k crashed.

After round N, each correct process decides for its value at that time.

Correctness: Let p_j never be suspected. After round j, all correct processes have the same value b. Hence, after round N, all correct processes decide b.

Consensus with Weakly Accurate Failure Detection

Processes are numbered: $p_1, \ldots, p_N$. Initially, each process has value 0 or 1. In round k:

- p_k (if not crashed) broadcasts its value.
- Each process waits:
  - either for an incoming message from p_k, in which case it adopts the value of p_k;
  - or until it suspects that p_k crashed.

After round N, each correct process decides for its value at that time.

Correctness: Let p_j never be suspected. After round j, all correct processes have the same value b. Hence, after round N, all correct processes decide b.

Eventually Strongly Accurate Failure Detection

A failure detector is eventually strongly accurate if from some time onward, only crashed processes are suspected:

$$\exists\tau \ \forall\tau' \geq \tau \ \forall p, q \notin \mathit{Crash}(F) \ \ p \notin H(q, \tau')$$

Assumptions:

- Each correct process broadcasts **alive** every σ time units.
- There is an (unknown) upper bound on communication delay.

Each process p initially takes $\mu_p = 1$.

If p receives no message from q for $\sigma + \mu_p$ time units, then p *suspects* that q has crashed.

When p receives a message from a suspected process q, then q is no longer suspected and $\mu_p := \mu_p + 1$.

This failure detector is *complete* and *eventually strongly accurate*.

Eventually Strongly Accurate Failure Detection

A failure detector is eventually strongly accurate if from some time onward, only crashed processes are suspected:

$$\exists\tau\ \forall\tau' \geq \tau\ \forall p, q \notin \mathit{Crash}(F)\ \ p \notin H(q, \tau')$$

Assumptions:

- Each correct process broadcasts **alive** every σ time units.
- There is an (unknown) upper bound on communication delay.

Each process p initially takes $\mu_p = 1$.

If p receives no message from q for $\sigma{+}\mu_p$ time units, then p *suspects* that q has crashed.

When p receives a message from a suspected process q, then q is no longer suspected and $\mu_p := \mu_p{+}1$.

This failure detector is *complete* and *eventually strongly accurate*.

Consensus with Failure Detection

Theorem: Let $t \geq \frac{N}{2}$. There is no t-crash consensus algorithm based on an *eventually strongly accurate* failure detector.

Proof: Suppose, toward a contradiction, there is such an algorithm. Partition the set of processes into S and T, with $|S| = \lfloor \frac{N}{2} \rfloor$ and $|T| = \lceil \frac{N}{2} \rceil$.

In reachable configurations, S and T are either both 0-potent or both 1-potent. For else, the processes in S could suspect for a sufficiently long period that the processes in T have crashed, and vice versa. Then, since $t \geq \frac{N}{2}$, S and T could independently decide for different values.

Since the initial configuration is bivalent, there is a reachable configuration γ and a transition $\gamma \rightarrow \delta$ with S and T both only b-potent in γ and only $(1-b)$-potent in δ. Such a transition cannot exist.

Consensus with Failure Detection

Theorem: Let $t \geq \frac{N}{2}$. There is no t-crash consensus algorithm based on an *eventually strongly accurate* failure detector.

Proof: Suppose, toward a contradiction, there is such an algorithm. Partition the set of processes into S and T, with $|S| = \lfloor \frac{N}{2} \rfloor$ and $|T| = \lceil \frac{N}{2} \rceil$.

In reachable configurations, S and T are either both 0-potent or both 1-potent. For else, the processes in S could suspect for a sufficiently long period that the processes in T have crashed, and vice versa. Then, since $t \geq \frac{N}{2}$, S and T could independently decide for different values.

Since the initial configuration is bivalent, there is a reachable configuration γ and a transition $\gamma \rightarrow \delta$ with S and T both only b-potent in γ and only $(1{-}b)$-potent in δ. Such a transition cannot exist.

Chandra-Toueg Algorithm

A failure detector is eventually weakly accurate if from some time onward some process is never suspected:

$$\exists p \; \exists \tau \; \forall \tau' \geq \tau \; \forall q \notin \mathit{Crash}(F) \quad p \notin H(q, \tau')$$

Let $t < \frac{N}{2}$. Assume a complete and *eventually weakly accurate* failure detector. Chandra and Toueg gave a rotating coordinator algorithm for t-crash consensus.

Each process q records the last round ts_q in which it updated $value_q$. Initially, $value_q \in \{0, 1\}$ and $ts_q = 0$.

Processes are numbered: $p_1, \ldots, p_N$. Round k is coordinated by p_c with $c = (k \bmod N)+1$.

Note: Tel presents a simplified version of the Chandra-Toueg algorithm (without acknowledgements and time stamps), which only works for $t < \frac{N}{3}$.

Chandra-Toueg Algorithm

A failure detector is eventually weakly accurate if from some time onward some process is never suspected:

$$\exists p \; \exists \tau \; \forall \tau' \geq \tau \; \forall q \notin \mathit{Crash}(F) \;\; p \notin H(q, \tau')$$

Let $t < \frac{N}{2}$. Assume a complete and *eventually weakly accurate* failure detector. Chandra and Toueg gave a rotating coordinator algorithm for t-crash consensus.

Each process q records the last round ts_q in which it updated $value_q$. Initially, $value_q \in \{0, 1\}$ and $ts_q = 0$.

Processes are numbered: $p_1, \ldots, p_N$. Round k is coordinated by p_c with $c = (k \bmod N)+1$.

Note: Tel presents a simplified version of the Chandra-Toueg algorithm (without acknowledgements and time stamps), which only works for $t < \frac{N}{3}$.

Chandra-Toueg Algorithm

A failure detector is eventually weakly accurate if from some time onward some process is never suspected:

$$\exists p\ \exists \tau\ \forall \tau' \geq \tau\ \forall q \notin \mathit{Crash}(F)\quad p \notin H(q, \tau')$$

Let $t < \frac{N}{2}$. Assume a complete and *eventually weakly accurate* failure detector. Chandra and Toueg gave a rotating coordinator algorithm for t-crash consensus.

Each process q records the last round ts_q in which it updated $value_q$. Initially, $value_q \in \{0, 1\}$ and $ts_q = 0$.

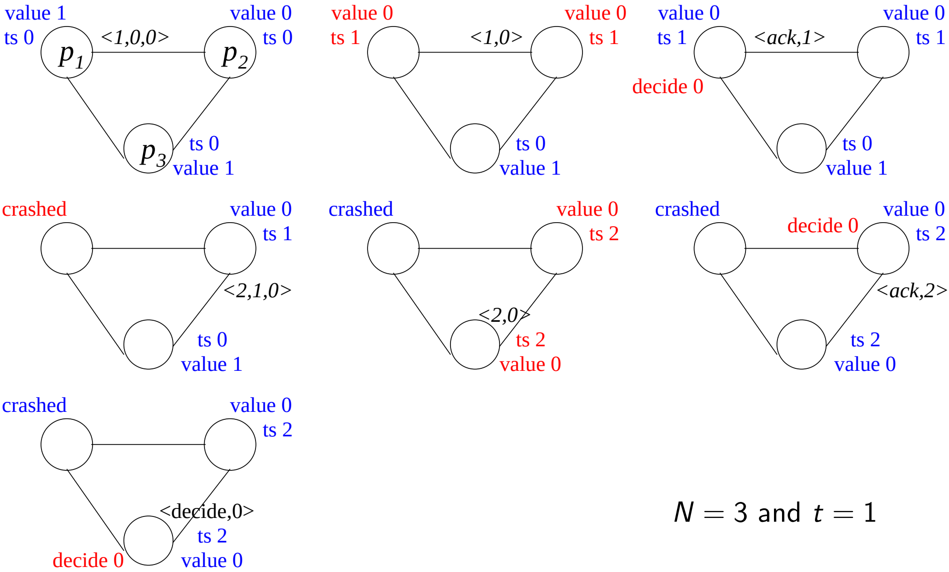
Processes are numbered: $p_1, \ldots, p_N$. Round k is coordinated by p_c with $c = (k \bmod N) + 1$.

Note: Tel presents a simplified version of the Chandra-Toueg algorithm (without acknowledgements and time stamps), which only works for $t < \frac{N}{3}$.

Chandra-Toueg Algorithm

- Correct q send $\langle k, value_q, ts_q \rangle$ to p_c.
- p_c (if not crashed) waits until $N{-}t$ such messages arrived, and selects one, say $\langle k, b, ts \rangle$, with ts as large as possible. p_c broadcasts $\langle k, b \rangle$.
- Each correct q waits:
  - until $\langle k, b \rangle$ arrives: then $value_q := b$, $ts_q := k$, and q sends $\langle \textbf{ack}, k \rangle$ to p_c;
  - or until it suspects p_c crashed: then q sends $\langle \textbf{nack}, k \rangle$ to p_c.
- p_c (if not crashed) waits until $N{-}t$ acknowledgements arrived. If more than t of them are **ack**, then p_c decides b, and broadcasts $\langle \textbf{decide}, b \rangle$. *(Note that $N{-}t > t$.)*
- When a process did not yet decide and receives $\langle \textbf{decide}, b \rangle$, it decides b.

Chandra-Toueg Algorithm - Example



$N = 3$ and $t = 1$

Chandra-Toueg Algorithm - Correctness

Theorem: Let $t < \frac{N}{2}$. The Chandra-Toueg algorithm is a terminating algorithm for t-crash consensus.

Proof: If the coordinator in some round k receives $> t$ **ack**'s, then (for some $b \in \{0,1\}$):

(1) there are $> t$ processes q with $ts_q \geq k$; and

(2) $ts_q \geq k$ implies $value_q = b$.

These properties are preserved in rounds $\ell > k$.

This follows by induction on $\ell - k$.

By (1), in round ℓ the coordinator receives at least one message with time stamp $\geq k$.

Hence, by (2), the coordinator of round ℓ broadcasts $\langle \ell, b \rangle$.

Chandra-Toueg Algorithm - Correctness

Theorem: Let $t < \frac{N}{2}$. The Chandra-Toueg algorithm is a terminating algorithm for t-crash consensus.

Proof: If the coordinator in some round k receives $> t$ **ack**'s, then (for some $b \in \{0, 1\}$):

(1) there are $> t$ processes q with $ts_q \geq k$; and

(2) $ts_q \geq k$ implies $value_q = b$.

These properties are preserved in rounds $\ell > k$.

This follows by induction on $\ell - k$.

By (1), in round ℓ the coordinator receives at least one message with time stamp $\geq k$.

Hence, by (2), the coordinator of round ℓ broadcasts $\langle \ell, b \rangle$.

Chandra-Toueg Algorithm - Correctness

Theorem: Let $t < \frac{N}{2}$. The Chandra-Toueg algorithm is a terminating algorithm for t-crash consensus.

Proof: If the coordinator in some round k receives $> t$ **ack**'s, then (for some $b \in \{0,1\}$):

(1) there are $> t$ processes q with $ts_q \geq k$; and

(2) $ts_q \geq k$ implies $value_q = b$.

These properties are preserved in rounds $\ell > k$.

This follows by induction on $\ell - k$.

By (1), in round ℓ the coordinator receives at least one message with time stamp $\geq k$.

Hence, by (2), the coordinator of round ℓ broadcasts $\langle \ell, b \rangle$.

Chandra-Toueg Algorithm - Correctness

Theorem: Let $t < \frac{N}{2}$. The Chandra-Toueg algorithm is a terminating algorithm for t-crash consensus.

Proof: If the coordinator in some round k receives $> t$ **ack**'s, then (for some $b \in \{0, 1\}$):

(1) there are $> t$ processes q with $ts_q \geq k$; and

(2) $ts_q \geq k$ implies $value_q = b$.

These properties are preserved in rounds $\ell > k$.

So from round k onward, processes can only decide b.

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 $N{-}t$ **ack**'s.
Since $N{-}t > t$, it decides.

All correct processes eventually receive the decide message of p, and also decide.

Chandra-Toueg Algorithm - Correctness

Theorem: Let $t < \frac{N}{2}$. The Chandra-Toueg algorithm is a terminating algorithm for t-crash consensus.

Proof: If the coordinator in some round k receives $> t$ **ack**'s, then (for some $b \in \{0,1\}$):

(1) there are $> t$ processes q with $ts_q \geq k$; and

(2) $ts_q \geq k$ implies $value_q = b$.

These properties are preserved in rounds $\ell > k$.

So from round k onward, processes can only decide b.

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 $N{-}t$ **ack**'s.
Since $N{-}t > t$, it decides.

All correct processes eventually receive the decide message of p, and also decide.

Question

Why is it difficult to implement a failure detector for Byzantine processes?

Local Clocks with Bounded Drift

Suppose we have a dense time domain.

Let each process p have a local clock $C_p(\tau)$, which returns a time value at real time τ.

We assume that each local clock has bounded drift, compared to real time:

$$\frac{1}{1+\rho}(\tau_2 - \tau_1) \ \leq \ C_p(\tau_2) - C_p(\tau_1) \ \leq \ (1+\rho)(\tau_2 - \tau_1)$$

for some $\rho > 0$.

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 $\delta > 0$,

$$|C_p(\tau) - C_q(\tau)| \;\leq\; \delta$$

for all τ.

Due to drift, this precision may degrade over time, necessitating repeated synchronizations.

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, at real time $\tau{+}R$,

$$|C_p(\tau{+}R) - C_q(\tau{+}R)| \ \leq \ \delta_0 + (1{+}\rho - \frac{1}{1{+}\rho})R \ < \ \delta_0 + 2\rho R$$

So there should be a synchronization every $\frac{\delta - \delta_0}{2\rho}$ time units.

We assume a bound $\delta_{\max}$ on network delay. For simplicity, let $\delta_{\max}$ be much smaller than δ (so that this delay can be ignored).

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, at real time $\tau{+}R$,

$$|C_p(\tau{+}R) - C_q(\tau{+}R)| \;\leq\; \delta_0 + (1{+}\rho - \frac{1}{1{+}\rho})R \;<\; \delta_0 + 2\rho R$$

So there should be a synchronization every $\frac{\delta-\delta_0}{2\rho}$ time units.

We assume a bound $\delta_{\max}$ on network delay. For simplicity, let $\delta_{\max}$ be much smaller than δ (so that this delay can be ignored).

Mahaney-Schneider Synchronizer

Consider a complete network of N processes, where at most t processes can become Byzantine.

In the Mahaney-Schneider synchronizer, each correct process at a synchronization:

1. Collects the clock values of all processes (waiting for $2\delta_{\max}$).
2. Discards those reported values τ for which less than $N-t$ processes report a value in the interval $[\tau-\delta, \tau+\delta]$ (they are from Byzantine processes).
3. Replaces all discarded and non-received values with some value v such that there are accepted reported values v_1 and v_2 with $v_1 \leq v \leq v_2$.
4. Takes the average of these N values as its new clock value.

Mahaney-Schneider Synchronizer - Correctness

Lemma: Let $t < \frac{N}{3}$. If values a_p and a_q pass the filters of correct processes p and q, respectively, in some synchronization round, then

$$|a_p - a_q| \leq 2\delta$$

Proof: At least $N{-}t$ processes reported a value in $[a_p{-}\delta, a_p{+}\delta]$ to p. And at least $N{-}t$ processes reported a value in $[a_q{-}\delta, a_q{+}\delta]$ to q.

Since $N{-}2t > t$, 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 $t < \frac{N}{3}$. The Mahaney-Schneider synchronizer is t-Byzantine robust.

Proof: Let a_{pr} (resp. a_{qr}) be the value that correct process p (resp. q) accepted or computed for 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,

$$|\frac{1}{N}(\sum_{\text{processes } r} a_{pr}) - \frac{1}{N}(\sum_{\text{processes } r} a_{qr})| \ \leq \ \frac{1}{N}t2\delta \ < \ \frac{2}{3}\delta$$

So we can take $\delta_0 = \frac{2}{3}\delta$, and there should be a synchronization every $\frac{\delta}{6\rho}$ time units.

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

Theorem: Let $t \geq \frac{N}{3}$. There is no t-Byzantine robust synchronizer.

Proof: Let $N = 3$, $t = 1$. Processes are p, q, r; r is Byzantine.
(The construction below easily extends to general N and $t \geq \frac{N}{3}$.)

Let the local clock of p run faster than the local 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 cannot recognize that r is Byzantine. So they have to stay within range δ of the value reported by r. Hence p cannot decrease its clock value, and q cannot increase its clock value.

By repeating this scenario at each synchronization round, the clock values of p and q get further and further apart.

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

Theorem: Let $t \geq \frac{N}{3}$. There is no t-Byzantine robust synchronizer.

Proof: Let $N = 3$, $t = 1$. Processes are p, q, r; r is Byzantine.
(The construction below easily extends to general N and $t \geq \frac{N}{3}$.)

Let the local clock of p run faster than the local 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 cannot recognize that r is Byzantine. So they have to stay within range δ of the value reported by r. Hence p cannot decrease its clock value, and q cannot increase its clock value.

By repeating this scenario at each synchronization round, the clock values of p and q get further and further apart.

Synchronous Networks

A synchronous network proceeds in pulses. In one pulse, each process:

1. sends messages;
2. receives messages; and
3. performs internal events.

A message is sent and received in the same pulse.

Such synchrony is called lockstep.

From Synchronizer to Synchronous Network

Assume ρ-bounded local clocks, and a synchronizer with precision δ.

For simplicity, let the maximum network delay $\delta_{\max}$, and the time to perform internal events in a pulse, be much smaller than δ.

When a process reads clock value $(i-1)(1+\rho)^2\delta$, it starts pulse i.

Key question: Does each process receive all messages for pulse i before the start of pulse $i+1$?

From Synchronizer to Synchronous Network

Assume ρ-bounded local clocks, and a synchronizer with precision δ.

For simplicity, let the maximum network delay $\delta_{\max}$, and the time to perform internal events in a pulse, be much smaller than δ.

When a process reads clock value $(i{-}1)(1{+}\rho)^2\delta$, it starts pulse i.

Key question: Does each process receive all messages for pulse i before the start of pulse $i{+}1$?

From Synchronizer to Synchronous Network

When a process reads clock value $(i{-}1)(1{+}\rho)^2\delta$, it starts pulse i.

Since the synchronizer has precision δ, and the clock of q is ρ-bounded (from below), for all τ,

$$C_q^{-1}(\tau) \ \leq \ C_p^{-1}(\tau) \ + \ (1{+}\rho)\delta$$

And since the clock of p is ρ-bounded (from above), for all τ, υ,

$$C_p^{-1}(\tau) \ + \ \upsilon \ \leq \ C_p^{-1}(\tau \ + \ (1{+}\rho)\upsilon)$$

Hence $C_q^{-1}((i{-}1)(1{+}\rho)^2\delta) \leq C_p^{-1}(i(1{+}\rho)^2\delta)$, so p receives the message from q for pulse i before the start of pulse $i{+}1$.

Byzantine Broadcast

Consider a synchronous network of N processes, where at most t processes can become Byzantine.

One process g, called the general, is given an input $x_g \in V$.

The other processes are called lieutenants.

Requirements for t-Byzantine broadcast:

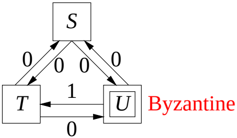
- Termination: Every correct process decides a value in V.
- Dependence: If the general is correct, it decides x_g.
- Agreement: All correct processes decide the same value.

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

Theorem: Let $t \geq \frac{N}{3}$. There is no t-Byzantine broadcast algorithm for synchronous networks (unless authentication is used).

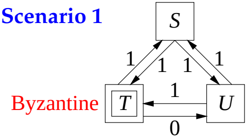
Proof: Divide the processes into three sets S, T and U with each $\leq t$ elements. Let $g \in S$.

Scenario 0



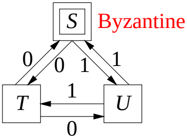
The processes in S and T decide 0

Scenario 1



The processes in S and U decide 1

Scenario 2



The processes in T decide 0 and in U decide 1

Lamport-Shostak-Pease Byzantine Broadcast

Let $t < \frac{N}{3}$. $Broadcast_g(N, t)$ is a t-Byzantine broadcast algorithm for synchronous networks.

```
Pulse 1: General $g$: decide and broadcast $x_g$

Lieutenant $p$: **if** $v$ is received from $g$ **then** $x_p := v$ **else** $x_p := \perp$;

if $t = 0$: decide $x_p$

if $t > 0$: perform $Broadcast_p(N-1, t-1)$ in pulse 2 ($g$ is excluded)

Pulse $t+1$: $(t > 0)$

Lieutenant $p$: for each lieutenant $q$, $p$ has taken a decision in
$Broadcast_q(N-1, t-1)$; store this decision in $M_p[q]$;

$x_p := major(M_p)$; decide $x_p$
```

(*major* maps each multiset over V to a value in V, such that if more than half of the elements in m are v, then $major(m) = v$.)

Lamport-Shostak-Pease Byzantine Broadcast

Let $t < \frac{N}{3}$. $\mathit{Broadcast}_g(N,t)$ is a t-Byzantine broadcast algorithm for synchronous networks.

Pulse 1: General g: decide and broadcast x_g

Lieutenant p: **if** v is received from g **then** $x_p := v$ **else** $x_p := \bot$;

if $t = 0$: decide x_p

if $t > 0$: perform $\mathit{Broadcast}_p(N{-}1, t{-}1)$ in pulse 2 (g is excluded)

Pulse $t{+}1$: $(t > 0)$

Lieutenant p: for each lieutenant q, p has taken a decision in $\mathit{Broadcast}_q(N{-}1, t{-}1)$; store this decision in $M_p[q]$;

$x_p := \mathit{major}(M_p)$; decide x_p

(*major* maps each multiset over V to a value in V, such that if more than half of the elements in m are v, then $\mathit{major}(m) = v$.)

Lamport-Shostak-Pease Byzantine Broadcast

Let $t < \frac{N}{3}$. $\mathit{Broadcast}_g(N,t)$ is a t-Byzantine broadcast algorithm for synchronous networks.

Pulse 1: General g: decide and broadcast x_g

Lieutenant p: **if** v is received from g **then** $x_p := v$ **else** $x_p := \perp$;

if $t = 0$: decide x_p

if $t > 0$: perform $\mathit{Broadcast}_p(N{-}1, t{-}1)$ in pulse 2 (g is excluded)

Pulse $t{+}1$: ($t > 0$)

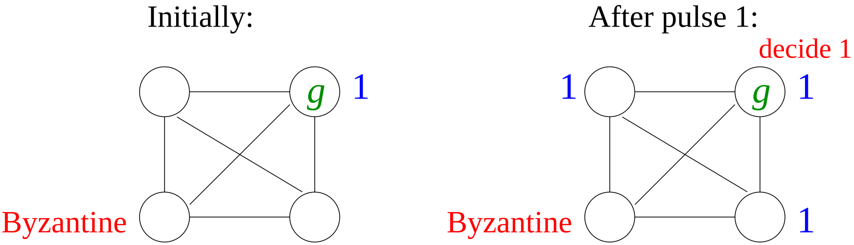
Lieutenant p: for each lieutenant q, p has taken a decision in $\mathit{Broadcast}_q(N{-}1, t{-}1)$; store this decision in $M_p[q]$;

$x_p := \mathit{major}(M_p)$; decide x_p

(*major* maps each multiset over V to a value in V, such that if more than half of the elements in m are v, then $\mathit{major}(m) = v$.)

Example

$N = 4$ and $t = 1$; general correct.



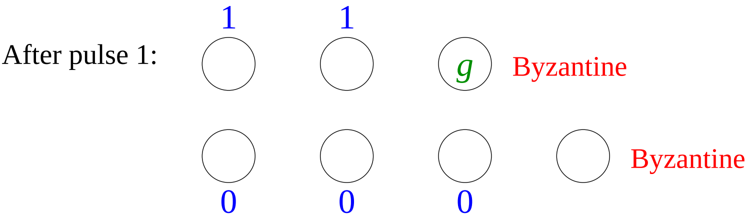
Consider the sub-network without g.

After pulse 1, all correct processes carry the value 1.

So, since $N-1 > 2f$ (i.e., $3 > 2$), the correct lieutenants will all decide 1 (even though one third of the lieutenants is Byzantine).

Example

$N = 7$ and $t = 2$; general Byzantine. (Channels are omitted.)



All correct lieutenants p build, in the recursive call $\mathit{Broadcast}_p(6, 1)$, the same *multiset* $m = \{0, 0, 0, 1, 1, v\}$, for some $v \in V$.

So in $\mathit{Broadcast}_g(7, 2)$, they all decide $\mathit{major}(m)$.

Lamport-Shostak-Pease Byzantine Broadcast - Correctness

Lemma: If general g is correct, and $N > 2f+t$ (with f the number of Byzantine processes; $f > t$ is allowed here), then in $Broadcast_g(N,t)$ all correct processes decide x_g.

Proof: By induction on t. Case $t = 0$ is trivial. Let $t > 0$.

Since g is correct, in pulse 1, at all correct lieutenants q, $x_q := x_g$.

Since $N{-}1 > 2f+(t{-}1)$, by induction, for all correct lieutenants q, in $Broadcast_q(N{-}1, t{-}1)$, the decision $x_q = x_g$ is taken.

Since a majority of the lieutenants is correct ($N{-}1 > 2f$), in pulse $t{+}1$, at each correct lieutenant p, $x_p := major(M_p) = x_g$.

Lamport-Shostak-Pease Byzantine Broadcast - Correctness

Lemma: If general g is correct, and $N > 2f+t$ (with f the number of Byzantine processes; $f > t$ is allowed here), then in $\mathit{Broadcast}_g(N,t)$ all correct processes decide x_g.

Proof: By induction on t. Case $t = 0$ is trivial. Let $t > 0$.

Since g is correct, in pulse 1, at all correct lieutenants q, $x_q := x_g$.

Since $N{-}1 > 2f+(t{-}1)$, by induction, for all correct lieutenants q, in $\mathit{Broadcast}_q(N{-}1, t{-}1)$, the decision $x_q = x_g$ is taken.

Since a majority of the lieutenants is correct $(N{-}1 > 2f)$, in pulse $t{+}1$, at each correct lieutenant p, $x_p := \mathit{major}(M_p) = x_g$.

Lamport-Shostak-Pease Byzantine Broadcast - Correctness

Theorem: Let $t < \frac{N}{3}$. $Broadcast_g(N,t)$ is a t-Byzantine broadcast algorithm for synchronous networks.

Proof: By induction on t.

If g is correct, then consensus follows from the lemma.

Let g be Byzantine (so $t > 0$). Then at most $t{-}1$ lieutenants are Byzantine.

Since $t{-}1 < \frac{N-1}{3}$, by induction, for every lieutenant q, all correct lieutenants take in $Byzantine_q(N{-}1, t{-}1)$ the same decision v_q.

Hence, all correct lieutenants p compute the same multiset M_p.

So in pulse $t{+}1$, all correct lieutenants p decide the same value $major(M_p)$.

Lamport-Shostak-Pease Byzantine Broadcast - Correctness

Theorem: Let $t < \frac{N}{3}$. $\mathit{Broadcast}_g(N, t)$ is a t-Byzantine broadcast algorithm for synchronous networks.

Proof: By induction on t.

If g is correct, then consensus follows from the lemma.

Let g be Byzantine (so $t > 0$). Then at most $t{-}1$ lieutenants are Byzantine.

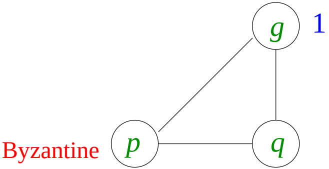
Since $t{-}1 < \frac{N-1}{3}$, by induction, for every lieutenant q, all correct lieutenants take in $\mathit{Byzantine}_q(N{-}1, t{-}1)$ the same decision v_q.

Hence, all correct lieutenants p compute the same multiset M_p.

So in pulse $t{+}1$, all correct lieutenants p decide the same value $\mathit{major}(M_p)$.

Incorrectness for $t \geq \frac{N}{3}$ - Example

$N = 3$ and $t = 1$.



g decides 1.

On the other hand, calling $Broadcast_q(2,0)$, q builds the multiset $\{0,1\}$ (assuming that p communicates 0 to q).

As a result, in $Broadcast_g(3,1)$, q decides 0
(assuming that $major(\{0,1\}) = 0$).

Partial Synchrony

A synchronous system can be obtained if local clocks have known *bounded drift*, and there is a known *upper bound on network delay*.

Dwork, Lynch and Stockmeyer showed that a t-Byzantine broadcast algorithm, for $t < \frac{N}{3}$, exists for partially synchronous systems, in which either

- the bounds on the inaccuracy of local clocks and network delay are unknown; or
- these bounds are known, but only valid from some unknown point in time.

Public-Key Cryptosystems

A public-key cryptosystem consists of a finite message domain $\mathcal{M}$ and, for each process q, functions $S_q, P_q : \mathcal{M} \to \mathcal{M}$ with

$$S_q(P_q(m)) = P_q(S_q(m)) = m \qquad \text{for } m \in \mathcal{M}.$$

S_q is kept *secret*, P_q is made *public*.

Underlying assumption: Computing S_q from P_q is expensive.

p sends *secret* message m to q: $P_q(m)$

p sends *signed* message m to q: $\langle m, S_p(m) \rangle$

Example: RSA cryptosystem.

Lamport-Shostak-Pease Authenticating 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, and
- $P_{p_k}(\sigma_k) = v$ for $k = 1, \ldots, i$,

then q includes v in the *set* W_q.

If $i \leq t$ and $|W_q| \leq 2$, then in pulse $i{+}1$, q broadcasts

$$\langle v, (\sigma_1, p_1) : \cdots : (\sigma_i, p_i) : (S_q(v), q)\rangle$$

After pulse $t{+}1$, each correct lieutenant p decides

v	if W_p is a singleton $\{v\}$, or
$\perp$	otherwise (the general is Byzantine)

Lamport-Shostak-Pease Authenticating Alg. - Correctness

Theorem: The Lamport-Shostak-Pease authenticating algorithm is a t-Byzantine broadcast algorithm, for any t.

Proof: If the general is correct, then owing to authentication, correct lieutenants only add x_g to W_q. So they all decide x_g.

Suppose a lieutenant receives a valid message $\langle v, \ell' \rangle$ in pulse $t{+}1$. Since ℓ' has length $t{+}1$, it contains a correct q. Then q received a valid message $\langle v, \ell \rangle$ in a pulse $\leq t$.

When a correct lieutenant q receives a valid message $\langle v, \ell \rangle$ in a pulse $\leq t$, then either it broadcasts v, or it already broadcast two other values before, with valid messages.

Lamport-Shostak-Pease Authenticating Alg. - Correctness

Theorem: The Lamport-Shostak-Pease authenticating algorithm is a t-Byzantine broadcast algorithm, for any t.

Proof: If the general is correct, then owing to authentication, correct lieutenants only add x_g to W_q. So they all decide x_g.

Suppose a lieutenant receives a valid message $\langle v, \ell' \rangle$ in pulse $t{+}1$. Since ℓ' has length $t{+}1$, it contains a correct q. Then q received a valid message $\langle v, \ell \rangle$ in a pulse $\leq t$.

When a correct lieutenant q receives a valid message $\langle v, \ell \rangle$ in a pulse $\leq t$, then either it broadcasts v, or it already broadcast two other values before, with valid messages.

Lamport-Shostak-Pease Authenticating Alg. - Correctness

We conclude that for all correct lieutenants p,

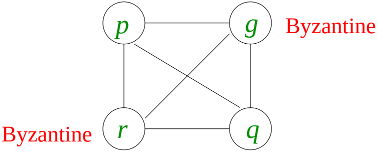
- either $W_p = \emptyset$ for all p,
- or $|W_p| \geq 2$ for all p,
- or $W_p = \{v\}$ for all p, for some $v \in V$.

In the first two cases, all correct processes decide $\perp$ (the general is Byzantine).

In the third case, they all decide v.

Example

$N = 4$ and $t = 2$.



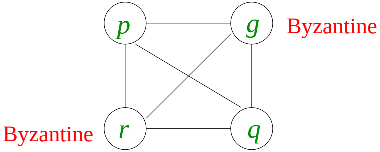
pulse 1: g sends $\langle 0, (S_g(0), g)\rangle$ to p and q
g sends $\langle 1, (S_g(1), g)\rangle$ to r
$W_p = W_q = \{0\}$

pulse 2: p broadcasts $\langle 0, (S_g(0), g) : (S_p(0), p)\rangle$
q broadcasts $\langle 0, (S_g(0), g) : (S_q(0), q)\rangle$
r sends $\langle 1, (S_g(1), g) : (S_r(1), r)\rangle$ to q
$W_p = \{0\}$ and $W_q = \{0, 1\}$

pulse 3: q broadcasts $\langle 1, (S_g(1), g) : (S_r(1), r) : (S_q(1), q)\rangle$
$W_p = W_q = \{0, 1\}$
p and q decide $\perp$

Example

$N = 4$ and $t = 2$.



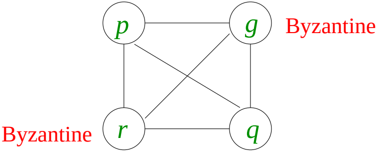
pulse 1: g sends $\langle 0, (S_g(0), g)\rangle$ to p and q
g sends $\langle 1, (S_g(1), g)\rangle$ to r
$W_p = W_q = \{0\}$

pulse 2: p broadcasts $\langle 0, (S_g(0), g) : (S_p(0), p)\rangle$
q broadcasts $\langle 0, (S_g(0), g) : (S_q(0), q)\rangle$
r sends $\langle 1, (S_g(1), g) : (S_r(1), r)\rangle$ to q
$W_p = \{0\}$ and $W_q = \{0, 1\}$

pulse 3: q broadcasts $\langle 1, (S_g(1), g) : (S_r(1), r) : (S_q(1), q)\rangle$
$W_p = W_q = \{0, 1\}$
p and q decide $\perp$

Example

$N = 4$ and $t = 2$.



pulse 1: g sends $\langle 0, (S_g(0), g)\rangle$ to p and q

g sends $\langle 1, (S_g(1), g)\rangle$ to r

$W_p = W_q = \{0\}$

pulse 2: p broadcasts $\langle 0, (S_g(0), g) : (S_p(0), p)\rangle$

q broadcasts $\langle 0, (S_g(0), g) : (S_q(0), q)\rangle$

r sends $\langle 1, (S_g(1), g) : (S_r(1), r)\rangle$ to q

$W_p = \{0\}$ and $W_q = \{0, 1\}$

pulse 3: q broadcasts $\langle 1, (S_g(1), g) : (S_r(1), r) : (S_q(1), q)\rangle$

$W_p = W_q = \{0, 1\}$

p and q decide $\perp$

Mutual Exclusion

Processes $p_0, \ldots, p_{N-1}$ contend for the critical section.

A process that can enter the critical section is called privileged.

For each execution, we require mutual exclusion and no starvation:

- in every configuration at most one process is privileged;
- if a process p_i tries to enter the critical section, and no process remains privileged forever, then p_i will eventually become privileged.

Raymond's Algorithm

Requires an undirected graph, which must, also initially, form a sink tree. At any time, the root, holding a token, is privileged.

Each process maintains a FIFO queue, which may contain identities of its children, and its own id. Initially, this queue is empty.

Queue maintenance:

- When a process wants to enter the critical section, it adds its id to its own queue.
- When a process that is not the root gets a new head at its (non-empty) queue, it asks its father 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

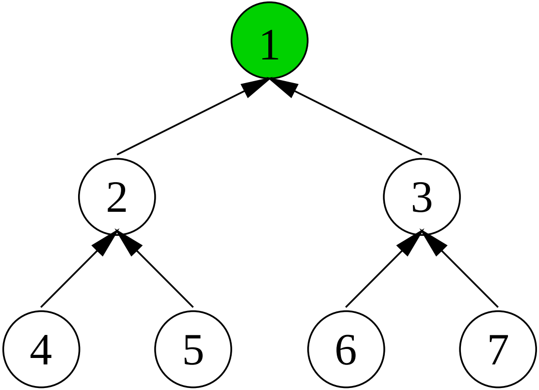
When the root exits the critical section (and its queue is non-empty), it sends the token to the process q at the head of its queue, makes q its father, and removes q from the head of its queue.

Let a process p get the token from its father, with process q at the head of its queue.

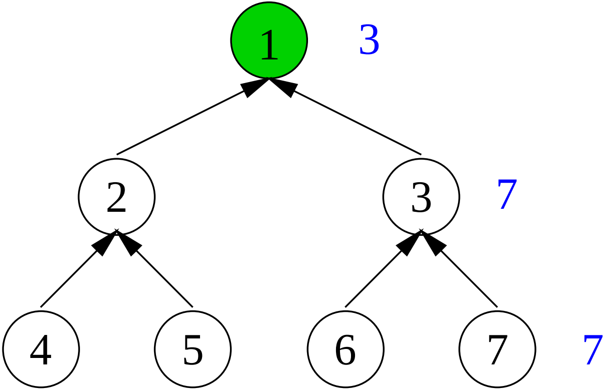
- if $p \neq q$, then p sends the token to q, and makes q its father;
- if $p = q$, then p becomes the root (i.e., it has no father, and becomes privileged).

In both cases, p removes q from the head of its queue.

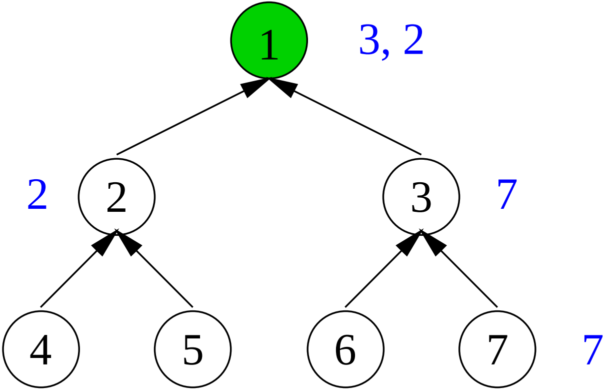
Raymond's Algorithm - Example



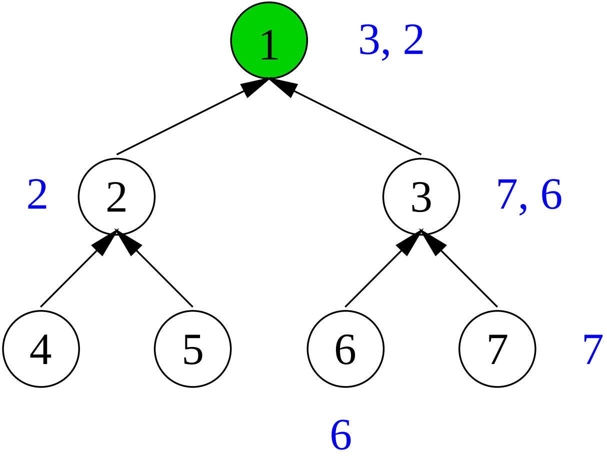
Raymond's Algorithm - Example



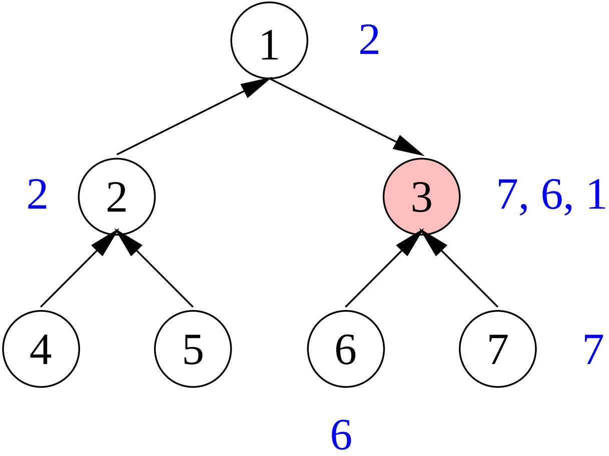
Raymond’s Algorithm - Example



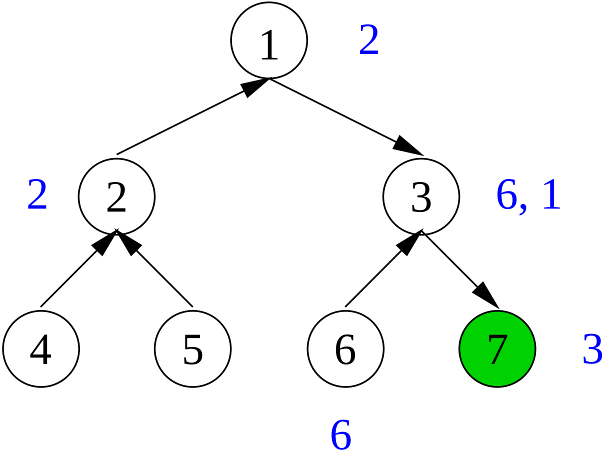
Raymond's Algorithm - Example



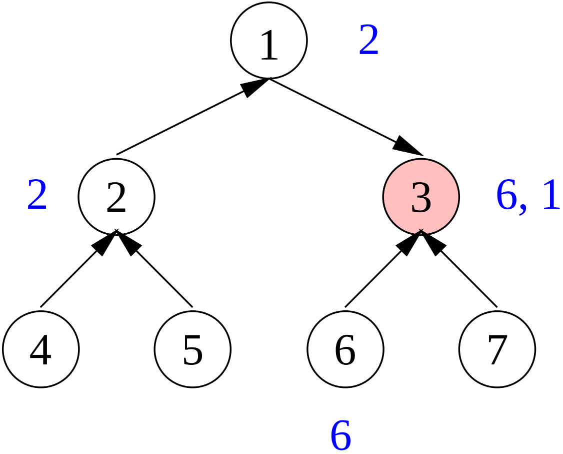
Raymond's Algorithm - Example



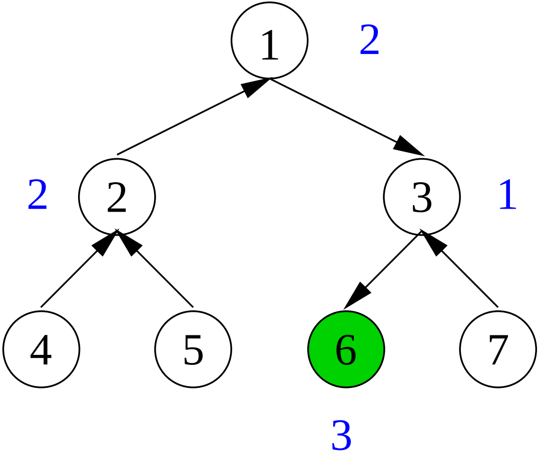
Raymond's Algorithm - Example



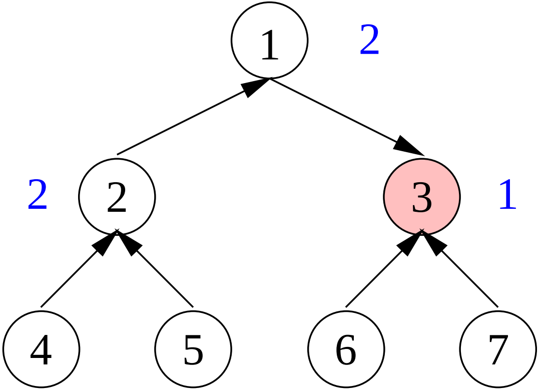
Raymond's Algorithm - Example



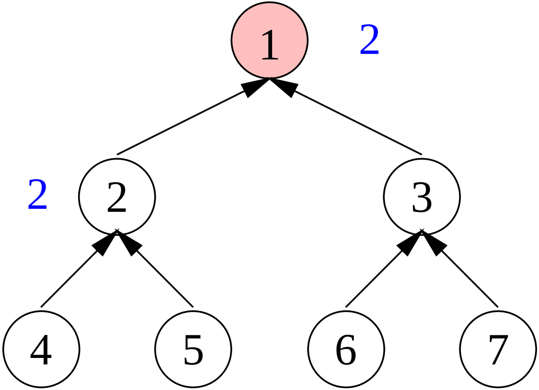
Raymond's Algorithm - Example



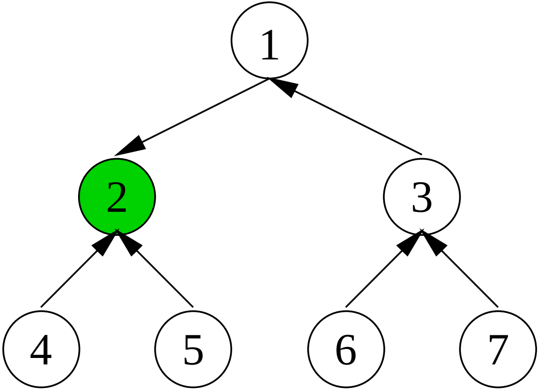
Raymond's Algorithm - Example



Raymond's Algorithm - Example



Raymond's Algorithm - Example



Raymond's Algorithm - Correctness

Raymond's algorithm provides mutual exclusion, because at all times there is only one root.

Raymond's algorithm provides no starvation, because eventually each request in a queue moves to the head of this queue.

However, note that in the example, process 2 requests the token before process 6, but process 6 receives the token before process 2.

Drawback: Sensitive to failures.

Raymond's Algorithm - Correctness

Raymond's algorithm provides mutual exclusion, because at all times there is only one root.

Raymond's algorithm provides no starvation, because eventually each request in a queue moves to the head of this queue.

However, note that in the example, process 2 requests the token before process 6, but process 6 receives the token before process 2.

Drawback: Sensitive to failures.

Ricart-Agrawala Algorithm

When a process p_i wants to access the 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 a permission if:

- p_j is neither inside nor trying to enter the critical section; or
- p_j sent a request with time stamp ts_j, and either $ts_i < ts_j$ or $ts_i = ts_j \wedge i < j$.

p_i enters the critical section when it received permission from all other processes.

Ricart-Agrawala Algorithm - Correctness

Mutual exclusion: Two processes cannot send permission to each other concurrently.

Because when a process p sends permission to q, p has not issued a request, and the logical time of p is greater than the time stamp of q's request.

No starvation: Eventually a request will have the smallest time stamp of all requests in the network.

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 sets its logical time to 1.

p_0 sends permission to p_1.

p_0 sends $request(2, 0)$ to p_1. When p_1 receives this message, it does not send permission to p_0, because $(1, 1) < (2, 0)$.

p_1 receives permission from p_0, and enters the critical section.

Ricart-Agrawala Algorithm - Example 2

$N = 2$, and p_0 and p_1 both are at logical time 0.

p_1 sends $\mathit{request}(1, 1)$ to p_0, and p_0 sends $\mathit{request}(1, 0)$ to p_1.

When p_0 receives the request from p_1, it does not 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.

p_0 receives permission from p_1, and enters the critical section.

Ricart-Agrawala Algorithm - Optimization

Drawback: High message complexity.

Carvalho-Roucairol optimization: After a first entry of the critical section, a process only needs to send a request to processes that it sent permission to (since its last exit from the critical section).

Question

Suppose a leader has been elected in the network. Give a mutual exclusion algorithm, with no starvation.

What is a drawback of such a mutual exclusion algorithm?

Mutual Exclusion with Shared Variables

Hagit Attiya and Jennifer Welch, *Distributed Computing*, McGraw Hill, 1998 (Chapter 4)

See also Chapter 10 of: Nancy Lynch, *Distributed Algorithms*, 1996

Processes communicate via shared variables (called registers) in *shared memory*.

read/write registers allow a process to perform an atomic read or write.

A single-reader (or single-writer) register is readable (or writable) by one process.

A multi-reader (or multi-writer) register is readable (or writable) by all processes.

An Incorrect Solution for Mutual Exclusion

Let *flag* be a multi-reader/multi-writer register, with range $\{0, 1\}$.

A process wanting to enter the critical section waits until $\mathit{flag} = 0$.

Then it writes $\mathit{flag} := 1$, and becomes privileged.

When it exits the critical section, it writes $\mathit{flag} := 0$.

The problem is that there is a time delay between reading $\mathit{flag} = 0$ and writing $\mathit{flag} := 1$, so that multiple processes can perform this read and write in parallel.

An Incorrect Solution for Mutual Exclusion

Let *flag* be a multi-reader/multi-writer register, with range $\{0,1\}$.

A process wanting to enter the critical section waits until $flag = 0$.

Then it writes $flag := 1$, and becomes privileged.

When it exits the critical section, it writes $flag := 0$.

The problem is that there is a time delay between reading $flag = 0$ and writing $flag := 1$, so that multiple processes can perform this read and write in parallel.

Dijkstra's Mutual Exclusion Algorithm

turn is a multi-reader/multi-writer register with range $\{0, \ldots, N-1\}$.

flag[i] a multi-reader/ single-writer register, only writable by p_i, with range $\{0, 1, 2\}$.

Initially they all have value 0.

We present the pseudocode for process p_i.

```
⟨Entry⟩:   L:   $flag[i] := 1$
                **while** $turn \neq i$ **do**
                        **if** $flag[turn] = 0$ **then** $turn := i$
                $flag[i] := 2$
                **for** $j \neq i$ **do**
                        **if** $flag[j] = 2$ **then goto** L

⟨Exit⟩:         $flag[i] := 0$
```

Dijkstra's Mutual Exclusion Algorithm - Correctness

This algorithm provides mutual exclusion.

And if a process p_i tries to enter the critical section, and no process remains in the critical section forever, then *some* process will eventually become privileged (no deadlock).

However, there can be starvation.

Dijkstra's Mutual Exclusion Algorithm - Example

Let $N = 3$.

$\mathit{flag}[1] := 1$

$\mathit{flag}[2] := 1$

p_1 and p_2 read $\mathit{turn} = 0$

p_1 and p_2 read $\mathit{flag}[0] = 0$

$\mathit{turn} := 1$

$\mathit{turn} := 2$

$\mathit{flag}[1] := 2$

$\mathit{flag}[2] := 2$

$\mathit{flag}[1] := 1$

$\mathit{flag}[2] := 1$

p_1 and p_2 read $\mathit{turn} = 2$

p_1 reads $\mathit{flag}[2] \neq 0$

$\mathit{flag}[2] := 2$

p_2 enters the critical section

p_2 exits the critical section

$\mathit{flag}[2] := 0$

$\mathit{flag}[2] := 1$

p_1 reads $\mathit{flag}[2] \neq 0$

$\mathit{flag}[2] := 2$

p_2 enters the critical section

Fischer's Algorithm

Uses time delays, and the assumption that an operation can be performed within one time unit.

turn is a multi-reader/multi-writer register with range $\{-1, 0, \ldots, N{-}1\}$. Initially it has value -1.

We present the pseudocode for process p_i.

```
⟨Entry⟩:   L:   wait until $turn = -1$
                $turn := i$ (takes less than one time unit)
                delay of more than one time unit
                if $turn \neq i$ then goto L

⟨Exit⟩:         $turn := -1$
```

Fischer's algorithm guarantees mutual exclusion and no deadlock.

Lamport's Bakery Algorithm

Multi-reader/single-writer registers *number*$[i]$ and *choosing*$[i]$ range over $\mathbb{N}$ and $\{0,1\}$, respectively; they are only writeable by p_i. Initially they all have value 0.

When p_i wants to enter the critical section, it writes a number to *number*$[i]$ that is greater than *number*$[j]$ for all $j \neq i$.

Different processes can concurrently obtain the same number; therefore the *ticket* of p_i is the pair $(\mathit{number}[i], i)$.

choosing$[i] = 1$ while p_i is obtaining a number.

When the critical section is empty, and no process is obtaining a number, the process with the smallest ticket (n, i) with $n > 0$ enters.

When p_i exits the critical section, *number*$[i]$ is set to 0.

Lamport's Bakery Algorithm

We present the pseudocode for process p_i.

```
⟨Entry⟩ :   $choosing[i] := 1$
            $number[i] := \max\{number[0], \ldots, number[N-1]\} + 1$
            $choosing[i] := 0$
            **for** $j \neq i$ **do**
               **wait until** $choosing[j] = 0$
               **wait until** $number[j] = 0$ **or** $(number[j], j) > (number[i], i)$

⟨Exit⟩ :    $number[i] := 0$
```

The bakery algorithm provides mutual exclusion and no starvation.

Drawback: Can have high synchronization delays.

Lamport's Bakery Algorithm

We present the pseudocode for process p_i.

```
⟨Entry⟩ :   $choosing[i] := 1$
            $number[i] := \max\{number[0], \ldots, number[N-1]\} + 1$
            $choosing[i] := 0$
            for $j \neq i$ do
               wait until $choosing[j] = 0$
               wait until $number[j] = 0$ or $(number[j], j) > (number[i], i)$

⟨Exit⟩ :    $number[i] := 0$
```

The bakery algorithm provides mutual exclusion and no starvation.

Drawback: Can have high synchronization delays.

Lamport's Bakery Algorithm - Example

Let $N = 2$.

$choosing[1] := 1$

$choosing[0] := 1$

p_0 and p_1 read $number[0]$ and $number[1]$

$number[1] := 1$

$choosing[1] := 0$

p_1 reads $choosing[0] := 1$

$number[0] := 1$

$choosing[0] := 0$

p_0 reads $choosing[1] = 0$ and $(number[0], 0) < (number[1], 1)$

p_0 enters the critical section

p_0 exits the critical section

$number[0] := 0$

$choosing[0] := 1$

p_0 reads $number[0]$ and $number[1]$

$number[0] := 2$

$choosing[0] := 0$

p_1 reads $choosing[0] = 0$ and $(number[1], 1) < (number[0], 0)$

p_1 enters the critical section

Mutual Exclusion for Two Processes

Assume processes p_0 and p_1.

flag$[i]$ is a multi-reader/single-writer register, only writeable by p_i.
Its range is $\{0, 1\}$; initially it has value 0.

```
             code for $p_0$                        code for $p_1$
⟨Entry⟩:                                      L:   $flag[1] := 0$
                                                   wait until $flag[0] = 0$
             $flag[0] := 1$                        $flag[1] := 1$
             wait until $flag[1] = 0$              if $flag[0] = 1$ then goto L

⟨Exit⟩:      $flag[0] := 0$                        $flag[1] := 0$
```

This algorithm provides mutual exclusion and no deadlock.

However, p_1 may never progress from Entry to the critical section, while p_0 enters the critical section infinitely often (starvation of p_1).

Mutual Exclusion for Two Processes

Assume processes p_0 and p_1.

$flag[i]$ is a multi-reader/single-writer register, only writeable by p_i. Its range is $\{0, 1\}$; initially it has value 0.

```
            code for $p_0$                   code for $p_1$
⟨Entry⟩:                               L:   $flag[1] := 0$
                                            wait until $flag[0] = 0$
            $flag[0] := 1$                  $flag[1] := 1$
            wait until $flag[1] = 0$        if $flag[0] = 1$ then goto L

⟨Exit⟩:     $flag[0] := 0$                  $flag[1] := 0$
```

This algorithm provides mutual exclusion and no deadlock.

However, p_1 may never progress from Entry to the critical section, while p_0 enters the critical section infinitely often (starvation of p_1).

Question

How can this mutual exlusion algorithm for two processes be adapted so that it provides no starvation?

Peterson2P Algorithm

flag$[i]$ is a multi-reader/single-writer register, only writeable by p_i.
priority is a multi-reader/ multi-writer register.
They all have range $\{0, 1\}$; initially they have value 0.

```
            code for $p_0$
⟨Entry⟩: L: $flag[0] := 0$
            wait until ($flag[1] = 0$
                    or $priority = 0$)
            $flag[0] := 1$
            if $priority = 1$ then
                if $flag[1] = 1$ then goto L
            else wait until $flag[1] = 0$

⟨Exit⟩:     $priority := 1$
            $flag[0] := 0$
```

```
   code for $p_1$
L: $flag[1] := 0$
   wait until ($flag[0] = 0$
           or $priority = 1$)
   $flag[1] := 1$
   if $priority = 0$ then
       if $flag[0] = 1$ then goto L
   else wait until $flag[0] = 0$

   $priority := 0$
   $flag[1] := 0$
```

Peterson2P algorithm provides mutual exclusion and no starvation.

Peterson2P Algorithm

flag[*i*] is a multi-reader/single-writer register, only writeable by p_i.
priority is a multi-reader/ multi-writer register.
They all have range $\{0, 1\}$; initially they have value 0.

code for p_0

```
⟨Entry⟩:  L: $flag[0] := 0$
             wait until ($flag[1] = 0$
                     or $priority = 0$)
             $flag[0] := 1$
             if $priority = 1$ then
                 if $flag[1] = 1$ then goto L
             else wait until $flag[1] = 0$

⟨Exit⟩:      $priority := 1$
             $flag[0] := 0$
```

code for p_1

```
L: $flag[1] := 0$
   wait until ($flag[0] = 0$
           or $priority = 1$)
   $flag[1] := 1$
   if $priority = 0$ then
       if $flag[0] = 1$ then goto L
   else wait until $flag[0] = 0$

   $priority := 0$
   $flag[1] := 0$
```

Peterson2P algorithm provides mutual exclusion and no starvation.

Peterson2P Algorithm - Example

$\mathit{flag}[1] := 0$ (L)

$\mathit{flag}[1] := 1$

$\mathit{flag}[0] := 0$ (L)

$\mathit{flag}[0] := 1$

$\mathit{flag}[1] := 0$

p_0 enters the critical section

p_0 exits the critical section

$\mathit{priority} := 1$

$\mathit{flag}[0] := 0$

$\mathit{flag}[0] := 0$ (L)

$\mathit{flag}[0] := 1$

p_0 enters the critical section

$\mathit{flag}[1] := 1$

p_0 exits the critical section

$\mathit{priority} := 1$

$\mathit{flag}[0] := 0$

$\mathit{flag}[0] := 0$ (L)

p_1 enters the critical section

Question

How can the Peterson2P algorithm be transformed into a mutual exclusion algorithm for $N \geq 2$ processes?

PetersonNP Algorithm

Assume processes $p_0, \dots, p_{N-1}$.

Processes compete pairwise, using the Peterson2P algorithm, in a tournament tree, which is a complete binary tree.

- Each process begins in a leaf of the tree.
- The winner proceeds to the next higher level, where it competes with the winner of the competition on the other side of the subtree.
- The process that wins at the root becomes privileged.

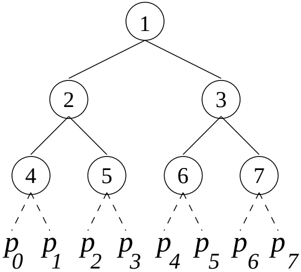
PetersonNP Algorithm - Tournament Tree

Let $k = \lceil {}^2\log N \rceil - 1$.

Consider the complete binary tree of depth k. The root is numbered 1, and the left and right child of a node v are numbered $2v$ and $2v+1$, respectively.

Each node has two sides, 0 and 1 (corresponding to priorities).

Initially, process p_i is associated to node $2^k + \lfloor i/2 \rfloor$ and side $i \bmod 2$.



PetersonNP Algorithm

Each node v has shared variables $flag_v[0]$, $flag_v[1]$ and $priority_v$.

They all have range $\{0,1\}$; initially they have value 0.

A process p_i repeatedly applies the procedure $Node(2^k+\lfloor i/2 \rfloor, i \bmod 2)$.

```
     procedure Node($v : \mathbb{N}, side : 0..1$)
L:   $flag_v[side] := 0$
     wait until ($flag_v[1-side] = 0$ or $priority_v = side$)
     $flag_v[side] := 1$
     if $priority_v = 1-side$ then if $flag_v[1-side] = 1$ then goto L
     else wait until $flag_v[1-side] = 0$
     if $v = 1$ then ⟨Enter Critical Section⟩ ⟨Exit⟩
     else Node($\lfloor v/2 \rfloor, v \bmod 2$)
     $priority_v := 1-side$
     $flag_v[side] := 0$
```

PetersonNP algorithm provides mutual exclusion and no starvation.

PetersonNP Algorithm

Each node v has shared variables $flag_v[0]$, $flag_v[1]$ and $priority_v$.

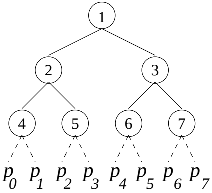
They all have range $\{0,1\}$; initially they have value 0.

A process p_i repeatedly applies the procedure $Node(2^k+\lfloor i/2\rfloor, i \bmod 2)$.

```
    procedure Node($v : \mathbb{N}, side : 0..1$)
L:  $flag_v[side] := 0$
    wait until ($flag_v[1-side] = 0$ or $priority_v = side$)
    $flag_v[side] := 1$
    if $priority_v = 1-side$ then if $flag_v[1-side] = 1$ then goto L
    else wait until $flag_v[1-side] = 0$
    if $v = 1$ then ⟨Enter Critical Section⟩ ⟨Exit⟩
    else $Node(\lfloor v/2\rfloor, v \bmod 2)$
    $priority_v := 1-side$
    $flag_v[side] := 0$
```

PetersonNP algorithm provides mutual exclusion and no starvation.

PetersonNP Algorithm - Example



$N = 8$.

p_1 starts in $Node(4, 1)$ and p_6 in $Node(7, 0)$.

Redundant L events ($flag_v[side] := 0$) are omitted.

$flag_7[1] := 1$
$flag_7[0] := 1$
$flag_7[1] := 0$
p_6 continues with $Node(3, 1)$
$flag_3[1] := 1$
p_6 continues with $Node(1, 1)$
$flag_1[1] := 1$
$flag_4[1] := 1$
p_1 continues with $Node(2, 0)$
$flag_2[0] := 1$
p_1 continues with $Node(1, 0)$
$flag_1[0] := 1$

$flag_1[1] := 0$
p_1 enters the critical section
p_1 exits the critical section
$priority_1 := 1$
$flag_1[0] := 0$
$flag_1[1] := 1$
p_6 enters the critical section
$priority_2 := 1$
$flag_2[0] := 0$
$priority_4 := 0$
$flag_4[1] := 0$
p_1 continues with $Node(4, 1)$

Read-Modify-Write Registers

A read-modify-write register allows a process to (1) read its value, (2) compute a new value, and (3) assign this new value to the register, all in one instantaneous atomic operation.

In case of a read-modify-write register, mutual exclusion with no starvation can be achieved with a single register.

The register maintains a FIFO queue of process identities. A process that wants to enter the critical section, adds its id at the end of the queue. A process at the head of the queue can enter the critical section. When a process exits the critical section, it deletes itself from the queue.

By contrast, mutual exclusion with no deadlock for N processes can only be achieved with $\geq N$ read/write registers.

Read-Modify-Write Registers

A read-modify-write register allows a process to (1) read its value, (2) compute a new value, and (3) assign this new value to the register, all in one instantaneous atomic operation.

In case of a read-modify-write register, mutual exclusion with no starvation can be achieved with a single register.

The register maintains a FIFO queue of process identities. A process that wants to enter the critical section, adds its id at the end of the queue. A process at the head of the queue can enter the critical section. When a process exits the critical section, it deletes itself from the queue.

By contrast, mutual exclusion with no deadlock for N processes can only be achieved with $\geq N$ read/write registers.

Read-Modify-Write Registers

A read-modify-write register allows a process to (1) read its value, (2) compute a new value, and (3) assign this new value to the register, all in one instantaneous atomic operation.

In case of a read-modify-write register, mutual exclusion with no starvation can be achieved with a single register.

The register maintains a FIFO queue of process identities. A process that wants to enter the critical section, adds its id at the end of the queue. A process at the head of the queue can enter the critical section. When a process exits the critical section, it deletes itself from the queue.

By contrast, mutual exclusion with no deadlock for N processes can only be achieved with $\geq N$ read/write registers.

Wait-Free Consensus

Two read-modify-write registers:

Fetch-and-Φ: Fetch the value of a register, and modify this value using the function Φ.

Compare-and-swap(v_1, v_2): Compare the value of the register with v_1, and if they are equal, change this value to v_2

Wait-free algorithm: Each process can complete any operation in a finite number of steps, even if other processes do not respond.

Herlihy showed that wait-free ($N-1$)-crash consensus can be achieved with *compare-and-swap*, but not with *fetch-and-*Φ.

Wait-Free Consensus

Two read-modify-write registers:

Fetch-and-Φ: Fetch the value of a register, and modify this value using the function Φ.

Compare-and-swap(v_1, v_2): Compare the value of the register with v_1, and if they are equal, change this value to v_2

Wait-free algorithm: Each process can complete any operation in a finite number of steps, even if other processes do not respond.

Herlihy showed that wait-free ($N-1$)-crash consensus can be achieved with *compare-and-swap*, but not with *fetch-and*-Φ.

Question

How can wait-free consensus be achieved with compare-and-swap?

Ticket Lock

A ticket lock L maintains two multi-reader/multi-writer counters:

- the number of requests to *acquire* L; and
- the number of times L has been *released*.

A process p that want to enter the critical section, performs a fetch-and-increment on L's request counter; p stores the value of this counter as its ticket, and increments this counter by 1.

Next p keeps polling L's release counter; when this counter equals p's ticket, p enters the critical section.

When p exits the critical section, it increments L's release counter by 1.

The ticket counter is an optimization of Lamport's bakery algorithm.

Drawback: In large-scale systems, excessive polling of L's release counter by remote processes becomes a bottleneck.

Ticket Lock

A ticket lock L maintains two multi-reader/multi-writer counters:

- the number of requests to *acquire* L; and
- the number of times L has been *released*.

A process p that want to enter the critical section, performs a fetch-and-increment on L's request counter; p stores the value of this counter as its ticket, and increments this counter by 1.

Next p keeps polling L's release counter; when this counter equals p's ticket, p enters the critical section.

When p exits the critical section, it increments L's release counter by 1.

The ticket counter is an optimization of Lamport's bakery algorithm.

Drawback: In large-scale systems, excessive polling of L's release counter by remote processes becomes a bottleneck.

Mellor-Crummey-Scott Lock

Lock L maintains a multi-reader/multi-writer register *last*, containing the last process that requested L (or $\perp$ if L is not held).

Each process p maintains multi-reader/multi-writer registers $locked_p$ of type boolean (initially *false*), and $next_p$ containing a process id (initially $\perp$).

A process p that wants to enter the critical section, performs fetch-and-store(p) on *last*, to fetch the process that requested L last, and write its own id into *last*.

- If $last = \perp$, then p enters the critical section.
- If $last = q$, then p sets $locked_p := true$ and $next_q := p$.

Now p must wait until $locked_p = false$.

Mellor-Crummey-Scott Lock

Lock L maintains a multi-reader/multi-writer register $last$, containing the last process that requested L (or $\perp$ if L is not held).

Each process p maintains multi-reader/multi-writer registers $locked_p$ of type boolean (initially $false$), and $next_p$ containing a process id (initially $\perp$).

A process p that wants to enter the critical section, performs fetch-and-store(p) on $last$, to fetch the process that requested L last, and write its own id into $last$.

- If $last = \perp$, then p enters the critical section.
- If $last = q$, then p sets $locked_p := true$ and $next_q := p$.

Now p must wait until $locked_p = false$.

Mellor-Crummey-Scott Lock

Let process q exit the critical section.

- If $next_q = p$, then q sets $locked_p := false$, upon which p can enter the critical section.

- If $next_q = \perp$, then q performs $compare(q, \perp)$ on $last$.

  If q finds that $last = p \neq q$, it waits until $next_q = p$, and then sets $locked_p := false$.

Note that p only needs to repeatedly poll its *local* variables $locked_p$ and (sometimes, for a short period) $next_p$.

Mellor-Crummey-Scott Lock

Let process q exit the critical section.

- If $next_q = p$, then q sets $locked_p := false$, upon which p can enter the critical section.
- If $next_q = \perp$, then q performs $compare(q, \perp)$ on *last*.

  If q finds that $last = p \neq q$, it waits until $next_q = p$, and then sets $locked_p := false$.

Note that p only needs to repeatedly poll its *local* variables $locked_p$ and (sometimes, for a short period) $next_p$.

Mellor-Crummey-Scott Lock - Example

q performs fetch-and-store(q) on *last*: $last := q$

q enters the critical section

p performs fetch-and-store(p) on *last*: $last := p$

p performs $locked_p := true$

q exits the critical section, and reads $next_q = \bot$

q performs $compare(q, \bot)$ on *last*

Since $last = p \neq q$, q must wait until $next_q = p$

p performs $next_q := p$

q performs $locked_p := false$

p enters the critical section

Self-Stabilization

All configurations are initial configurations.

An algorithm is self-stabilizing if every execution reaches a "correct" configuration.

Advantages:

- fault tolerance
- robustness for dynamic topologies
- straightforward initialization

Processes communicate via registers in *shared memory*.

Self-Stabilization

All configurations are initial configurations.

An algorithm is self-stabilizing if every execution reaches a "correct" configuration.

Advantages:

- fault tolerance
- robustness for dynamic topologies
- straightforward initialization

Processes communicate via registers in *shared memory*.

Dijkstra's Self-Stabilizing Token Ring

Let $p_0, \ldots, p_{N-1}$ form a directed ring, where each p_i holds a value $\sigma_i \in \{0, \ldots, K-1\}$ with $K \geq N$.

- p_i with $0 < i < N$ is privileged if $\sigma_i \neq \sigma_{i-1}$.
- p_0 is privileged if $\sigma_0 = \sigma_{N-1}$.

Each privileged process is allowed to change its value, causing the loss of its privilege:

- $\sigma_i := \sigma_{i-1}$ when $\sigma_i \neq \sigma_{i-1}$, for $0 < i < N$;
- $\sigma_0 := (\sigma_{N-1}+1) \bmod K$ when $\sigma_0 = \sigma_{N-1}$.

If $K \geq N$, then Dijkstra's token ring *self-stabilizes*. That is, each execution will reach a configuration where *mutual exclusion* is satisfied.

Moreover, Dijkstra's token ring guarantees *no starvation*.

Dijkstra's Self-Stabilizing Token Ring

Let $p_0, \ldots, p_{N-1}$ form a directed ring, where each p_i holds a value $\sigma_i \in \{0, \ldots, K-1\}$ with $K \geq N$.

- p_i with $0 < i < N$ is privileged if $\sigma_i \neq \sigma_{i-1}$.
- p_0 is privileged if $\sigma_0 = \sigma_{N-1}$.

Each privileged process is allowed to change its value, causing the loss of its privilege:

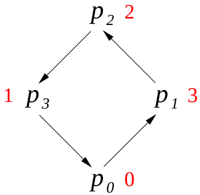
- $\sigma_i := \sigma_{i-1}$ when $\sigma_i \neq \sigma_{i-1}$, for $0 < i < N$;
- $\sigma_0 := (\sigma_{N-1}+1) \bmod K$ when $\sigma_0 = \sigma_{N-1}$.

If $K \geq N$, then Dijkstra's token ring *self-stabilizes*. That is, each execution will reach a configuration where *mutual exclusion* is satisfied.

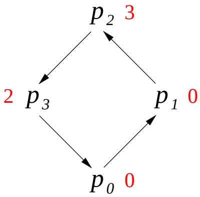
Moreover, Dijkstra's token ring guarantees *no starvation*.

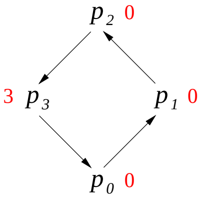
Dijkstra's Token Ring - Example

Let $N = K = 4$. Consider the initial configuration



It is not hard to see that it 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. A transition never increases the number of privileged processes.

Consider an execution. 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 execution, σ_0 ranges over all values in $\{0, \ldots, K-1\}$. Since $p_1, \ldots, p_{N-1}$ only copy values, and $K \geq N$, in some configuration of the execution, $\sigma_0 \neq \sigma_i$ for all $0 < i < N$.

The next time p_0 becomes privileged, clearly $\sigma_i = \sigma_0$ for all $0 < i < N$. So then mutual exclusion has been achieved.

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.
A transition never increases the number of privileged processes.

Consider an execution. 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 execution, σ_0 ranges over all values in $\{0, \ldots, K-1\}$. Since $p_1, \ldots, p_{N-1}$ only copy values, and $K \geq N$, in some configuration of the execution, $\sigma_0 \neq \sigma_i$ for all $0 < i < N$.

The next time p_0 becomes privileged, clearly $\sigma_i = \sigma_0$ for all $0 < i < N$. So then mutual exclusion has been achieved.

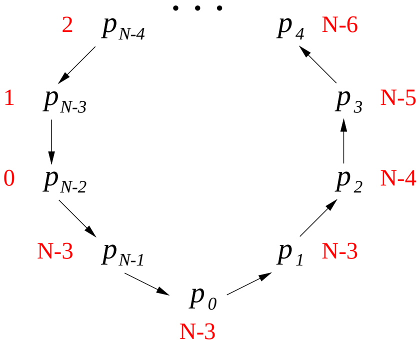
Question

Can you argue why, if $N \geq 3$, Dijkstra's token ring also self-stabilizes when $K = N - 1$?

This lower bound for K is sharp!

Dijkstra's Token Ring - Lower Bound for K

Example: Let $N \geq 4$ and $K = N-2$, and consider the following initial configuration.



It does not always self-stabilize.

Dijkstra's Token Ring - Message Complexity

Worst-case message complexity: Mutual exclusion is achieved after at most $O(N^2)$ transitions.

p_i for $0 < i < N$ can copy the initial values of $p_0, \ldots, p_{i-1}$. (Total: $\leq \frac{1}{2}(N-1)N$ events.)

p_0 takes on at most N new values to attain a *fresh* value. These values can be copied by $p_1, \ldots, p_{N-1}$. (Total: $\leq N^2$ events.)

Arora-Gouda Self-Stabilizing Election Algorithm

Given an undirected network.

Let an *upper bound* K on the *network size* be known to all processes.

The process with the *largest* id becomes the *leader*.

Each process p_i maintains the following variables:

$Neigh_i$:	the set of identities of its neighbors
$father_i$:	its father in the sink tree
$leader_i$:	the root of the sink tree
$dist_i$:	its distance from the root

Arora-Gouda Election Algorithm - Complications

Due to arbitrary initialization, there are three complications.

Complication 1: Multiple processes may consider themselves root of the sink tree.

Complication 2: There may be cycles in the sink tree.

Complication 3: $leader_i$ may not be the id of any process in the network.

Arora-Gouda Election Algorithm

A process p_i declares itself *leader*, i.e.

$$leader_i := i \qquad father_i := \perp \qquad dist_i := 0$$

if it detects an inconsistency in its local variables:

- $leader_i < i$; or
- $father_i = \perp$, and $leader_i \neq i$ or $dist_i > 0$; or
- $father_i \notin Neigh_i \cup \{\perp\}$; or
- $dist_i \geq K$.

Suppose $father_i = j$ with $j \in Neigh_i$ and $dist_j < K$.

If $leader_i \neq leader_j$, then $leader_i := leader_j$.

If $dist_i \neq dist_j + 1$, then $dist_i := dist_j + 1$.

Arora-Gouda Election Algorithm

A process p_i declares itself *leader*, i.e.

$$leader_i := i \qquad father_i := \perp \qquad dist_i := 0$$

if it detects an inconsistency in its local variables:

- $leader_i < i$; or
- $father_i = \perp$, and $leader_i \neq i$ or $dist_i > 0$; or
- $father_i \notin Neigh_i \cup \{\perp\}$; or
- $dist_i \geq K$.

Suppose $father_i = j$ with $j \in Neigh_i$ and $dist_j < K$.

If $leader_i \neq leader_j$, then $leader_i := leader_j$.

If $dist_i \neq dist_j + 1$, then $dist_i := dist_j + 1$.

Arora-Gouda Election Algorithm

If $leader_i < leader_j$ where $j \in Neigh_i$ and $dist_j < K$, then

$leader_i := leader_j \qquad father_i := j \qquad dist_i := dist_j + 1$

To obtain a *breadth-first search tree*, one can add:

If $leader_i = leader_j$ where $j \in Neigh_i$ and $dist_j + 1 < dist_i$, then

$leader_i := leader_j \qquad father_i := j \qquad dist_i := dist_j + 1$

Arora-Gouda Election Algorithm

If $leader_i < leader_j$ where $j \in Neigh_i$ and $dist_j < K$, then

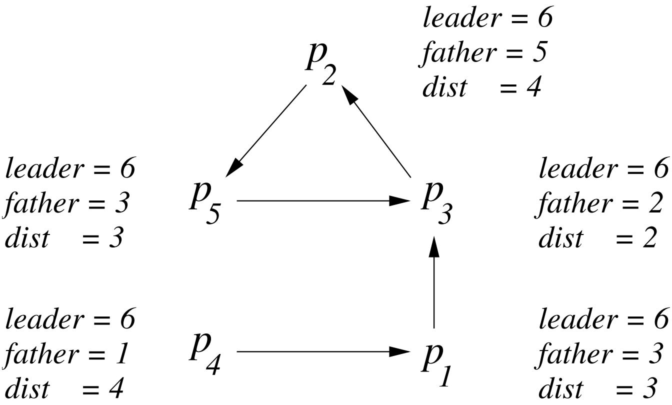
$$leader_i := leader_j \qquad father_i := j \qquad dist_i := dist_j + 1$$

To obtain a *breadth-first search tree*, one can add:

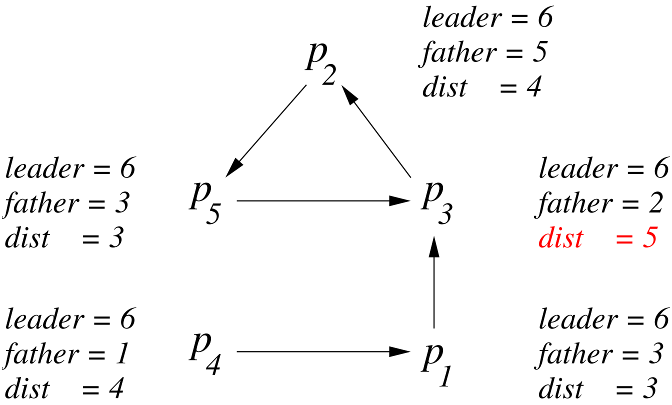
If $leader_i = leader_j$ where $j \in Neigh_i$ and $dist_j + 1 < dist_i$, then

$$leader_i := leader_j \qquad father_i := j \qquad dist_i := dist_j + 1$$

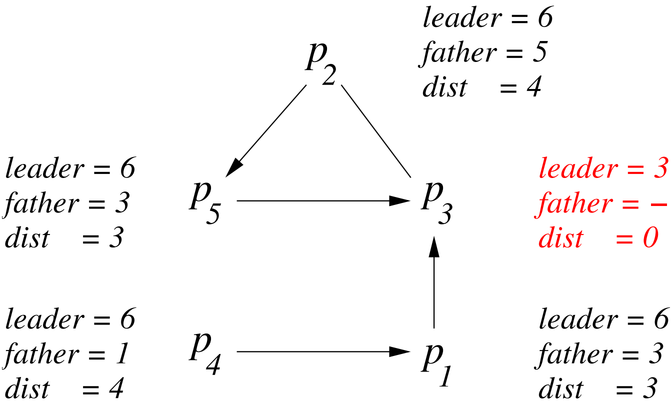
Arora-Gouda Election Algorithm - Example



Arora-Gouda Election Algorithm - Example



Arora-Gouda Election Algorithm - Example



leader = 6
father = 5
dist = 4

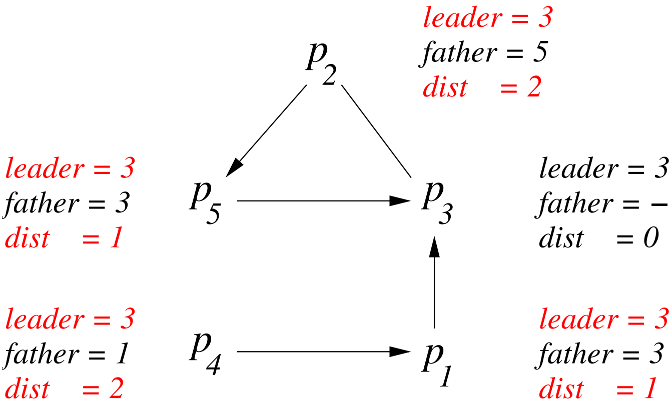
leader = 6
father = 3
dist = 3

leader = 3
father = –
dist = 0

leader = 6
father = 1
dist = 4

leader = 6
father = 3
dist = 3

Arora-Gouda Election Algorithm - Example



leader = 3
father = 5
dist = 2

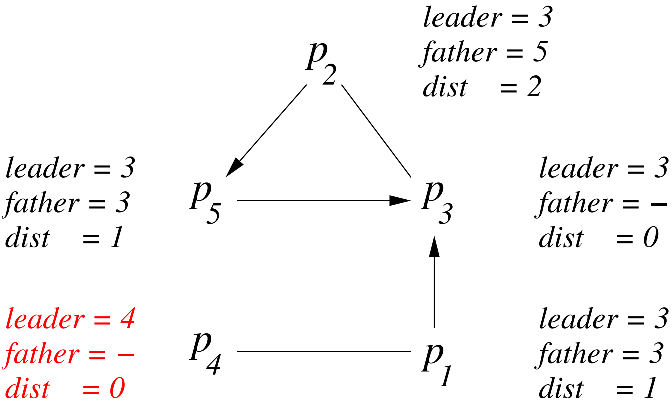
leader = 3
father = 3
dist = 1

leader = 3
father = –
dist = 0

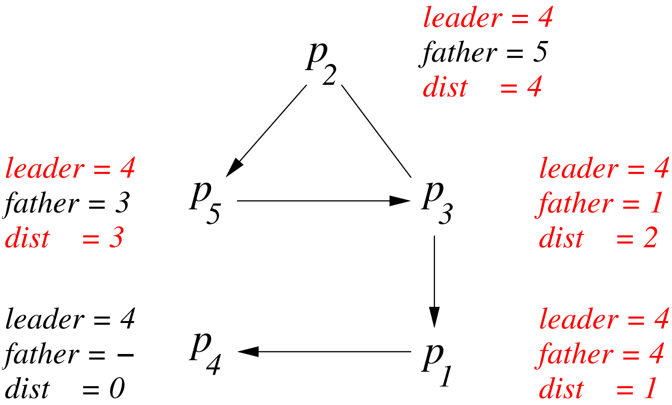
leader = 3
father = 1
dist = 2

leader = 3
father = 3
dist = 1

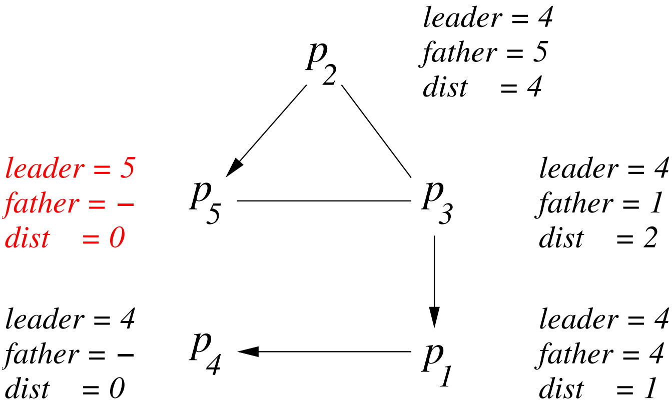
Arora-Gouda Election Algorithm - Example



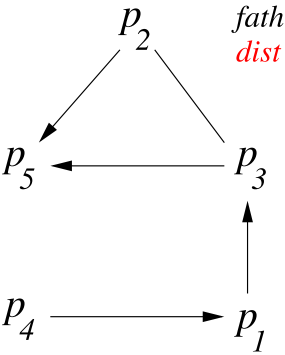
Arora-Gouda Election Algorithm - Example



Arora-Gouda Election Algorithm - Example



Arora-Gouda Election Algorithm - Example



leader = 5
father = 5
dist = 1

leader = 5
father = –
dist = 0

leader = 5
father = 5
dist = 1

leader = 5
father = 1
dist = 3

leader = 5
father = 3
dist = 2

Arora-Gouda Election Algorithm - Correctness

A subgraph in the network with a leader value j that is not an id of any node in this subgraph, contains an inconsistency or a cycle.

Such an inconsistency or cycle will eventually cause a process in this subgraph to declare itself leader.

Let i be the largest id of any process in the network.

- Leader values greater than i will eventually disappear; and
- p_i will eventually declare itself leader.

After p_i has declared itself leader, the algorithm will eventually converge to a spanning tree with root p_i.

Arora-Gouda Election Algorithm - Correctness

A subgraph in the network with a leader value j that is not an id of any node in this subgraph, contains an inconsistency or a cycle.

Such an inconsistency or cycle will eventually cause a process in this subgraph to declare itself leader.

Let i be the largest id of any process in the network.

- Leader values greater than i will eventually disappear; and
- p_i will eventually declare itself leader.

After p_i has declared itself leader, the algorithm will eventually converge to a spanning tree with root p_i.

Arora-Gouda Election Algorithm - Correctness

A subgraph in the network with a leader value j that is not an id of any node in this subgraph, contains an inconsistency or a cycle.

Such an inconsistency or cycle will eventually cause a process in this subgraph to declare itself leader.

Let i be the largest id of any process in the network.

- Leader values greater than i will eventually disappear; and
- p_i will eventually declare itself leader.

After p_i has declared itself leader, the algorithm will eventually converge to a spanning tree with root p_i.

Afek-Kutten-Yung Self-Stabilizing Election Algorithm

No upper bound on the network size needs to be known.

The process with the *largest* id becomes the *leader*.

A process p_i declares itself *leader*, i.e.

$$leader_i := i \qquad father_i := \perp \qquad dist_i := 0$$

if these three variables do not yet all have these values, and p_i detects even the slightest inconsistency in its local variables:

$$leader_i \leq i \quad \text{or} \quad father_i \notin Neigh_i \cup \{\perp\} \quad \text{or}$$

$$leader_i \neq leader_{father_i} \quad \text{or} \quad dist_i \neq dist_{father_i} + 1$$

p_i can make a neighbor p_j its *father* if $leader_i < leader_j$:

$$leader_i := leader_j \qquad father_i := j \qquad dist_i := dist_j + 1$$

Afek-Kutten-Yung Self-Stabilizing Election Algorithm

No upper bound on the network size needs to be known.
The process with the *largest* id becomes the *leader*.

A process p_i declares itself *leader*, i.e.

$$leader_i := i \qquad father_i := \perp \qquad dist_i := 0$$

if these three variables do not yet all have these values, and p_i detects even the slightest inconsistency in its local variables:

$$leader_i \leq i \quad \text{or} \quad father_i \notin Neigh_i \cup \{\perp\} \quad \text{or}$$

$$leader_i \neq leader_{father_i} \quad \text{or} \quad dist_i \neq dist_{father_i} + 1$$

p_i can make a neighbor p_j its *father* if $leader_i < leader_j$:

$$leader_i := leader_j \qquad father_i := j \qquad dist_i := dist_j + 1$$

Afek-Kutten-Yung Self-Stabilizing Election Algorithm

No upper bound on the network size needs to be known.

The process with the *largest* id becomes the *leader*.

A process p_i declares itself *leader*, i.e.

$$leader_i := i \qquad father_i := \perp \qquad dist_i := 0$$

if these three variables do not yet all have these values, and p_i detects even the slightest inconsistency in its local variables:

$$leader_i \leq i \quad \text{or} \quad father_i \notin Neigh_i \cup \{\perp\} \quad \text{or}$$

$$leader_i \neq leader_{father_i} \quad \text{or} \quad dist_i \neq dist_{father_i} + 1$$

p_i can make a neighbor p_j its *father* if $leader_i < leader_j$:

$$leader_i := leader_j \qquad father_i := j \qquad dist_i := dist_j + 1$$

Question

Suppose that during an application of the Afek-Kutten-Yung leader election algorithm, the created subgraph contains a cycle.

Why will at least one of the processes on this cycle declare itself leader?

Afek-Kutten-Yung Election Algorithm - Complication

Processes can infinitely often join a component of the created subgraph with a "false leader".

Example: Given two adjacent processes p_0 and p_1.

$leader_0 = leader_1 = 2$; $father_0 = 1$ and $father_1 = 0$; $dist_0 = dist_1 = 0$.

Since $dist_0 \neq dist_1 + 1$, p_0 declares itself *leader*:
$leader_0 := 0$, $father_0 := \perp$ and $dist_0 := 0$.

Since $leader_0 < leader_1$, p_0 makes p_1 its *father*:
$leader_0 := 2$, $father_0 := 1$ and $dist_0 := 2$.

Since $dist_1 \neq dist_0 + 1$, p_1 declares itself *leader*:
$leader_1 := 1$, $father_1 := \perp$ and $dist_1 := 0$.

Since $leader_1 < leader_0$, p_1 makes p_0 its *father*:
$leader_1 := 2$, $father_1 := 0$ and $dist_1 := 3$.

Et cetera

Afek-Kutten-Yung Election Algorithm - Complication

Processes can infinitely often join a component of the created subgraph with a "false leader".

Example: Given two adjacent processes p_0 and p_1.

$\mathit{leader}_0 = \mathit{leader}_1 = 2$; $\mathit{father}_0 = 1$ and $\mathit{father}_1 = 0$; $\mathit{dist}_0 = \mathit{dist}_1 = 0$.

Since $\mathit{dist}_0 \neq \mathit{dist}_1 + 1$, p_0 declares itself *leader*:
$\mathit{leader}_0 := 0$, $\mathit{father}_0 := \perp$ and $\mathit{dist}_0 := 0$.

Since $\mathit{leader}_0 < \mathit{leader}_1$, p_0 makes p_1 its *father*:
$\mathit{leader}_0 := 2$, $\mathit{father}_0 := 1$ and $\mathit{dist}_0 := 2$.

Since $\mathit{dist}_1 \neq \mathit{dist}_0 + 1$, p_1 declares itself *leader*:
$\mathit{leader}_1 := 1$, $\mathit{father}_1 := \perp$ and $\mathit{dist}_1 := 0$.

Since $\mathit{leader}_1 < \mathit{leader}_0$, p_1 makes p_0 its *father*:
$\mathit{leader}_1 := 2$, $\mathit{father}_1 := 0$ and $\mathit{dist}_1 := 3$.

Et cetera

Afek-Kutten-Yung Election Algorithm - Complication

Processes can infinitely often join a component of the created subgraph with a "false leader".

Example: Given two adjacent processes p_0 and p_1.

$\mathit{leader}_0 = \mathit{leader}_1 = 2$; $\mathit{father}_0 = 1$ and $\mathit{father}_1 = 0$; $\mathit{dist}_0 = \mathit{dist}_1 = 0$.

Since $\mathit{dist}_0 \neq \mathit{dist}_1 + 1$, p_0 declares itself *leader*:
$\mathit{leader}_0 := 0$, $\mathit{father}_0 := \perp$ and $\mathit{dist}_0 := 0$.

Since $\mathit{leader}_0 < \mathit{leader}_1$, p_0 makes p_1 its *father*:
$\mathit{leader}_0 := 2$, $\mathit{father}_0 := 1$ and $\mathit{dist}_0 := 2$.

Since $\mathit{dist}_1 \neq \mathit{dist}_0 + 1$, p_1 declares itself *leader*:
$\mathit{leader}_1 := 1$, $\mathit{father}_1 := \perp$ and $\mathit{dist}_1 := 0$.

Since $\mathit{leader}_1 < \mathit{leader}_0$, p_1 makes p_0 its *father*:
$\mathit{leader}_1 := 2$, $\mathit{father}_1 := 0$ and $\mathit{dist}_1 := 3$.

Et cetera

Afek-Kutten-Yung Election Algorithm - Complication

Processes can infinitely often join a component of the created subgraph with a "false leader".

Example: Given two adjacent processes p_0 and p_1.

$leader_0 = leader_1 = 2$; $father_0 = 1$ and $father_1 = 0$; $dist_0 = dist_1 = 0$.

Since $dist_0 \neq dist_1 + 1$, p_0 declares itself *leader*:
$leader_0 := 0$, $father_0 := \bot$ and $dist_0 := 0$.

Since $leader_0 < leader_1$, p_0 makes p_1 its *father*:
$leader_0 := 2$, $father_0 := 1$ and $dist_0 := 2$.

Since $dist_1 \neq dist_0 + 1$, p_1 declares itself *leader*:
$leader_1 := 1$, $father_1 := \bot$ and $dist_1 := 0$.

Since $leader_1 < leader_0$, p_1 makes p_0 its *father*:
$leader_1 := 2$, $father_1 := 0$ and $dist_1 := 3$.

Et cetera

Afek-Kutten-Yung Election Algorithm - Complication

Processes can infinitely often join a component of the created subgraph with a “false leader”.

Example: Given two adjacent processes p_0 and p_1.

$leader_0 = leader_1 = 2$; $father_0 = 1$ and $father_1 = 0$; $dist_0 = dist_1 = 0$.

Since $dist_0 \neq dist_1 + 1$, p_0 declares itself *leader*:
$leader_0 := 0$, $father_0 := \bot$ and $dist_0 := 0$.

Since $leader_0 < leader_1$, p_0 makes p_1 its *father*:
$leader_0 := 2$, $father_0 := 1$ and $dist_0 := 2$.

Since $dist_1 \neq dist_0 + 1$, p_1 declares itself *leader*:
$leader_1 := 1$, $father_1 := \bot$ and $dist_1 := 0$.

Since $leader_1 < leader_0$, p_1 makes p_0 its *father*:
$leader_1 := 2$, $father_1 := 0$ and $dist_1 := 3$.

Et cetera

Afek-Kutten-Yung Election Algorithm - Join Requests

Let $leader_i < leader_j$ for some $j \in Neigh_i$.

Before p_i makes p_j its father, first it sends a *join request* to p_j.

This request is forwarded through p_j's component, toward the root (if any) of this component.

The root sends back a *grant* toward p_i, which travels the reverse path of the request.

When p_i receives this grant, it makes p_j its *father*:
$leader_i := leader_j$, $father_i := j$ and $dist_i := dist_j + 1$.

If p_j's component has no root, p_j will never join this component.

Communication is performed using *shared variables*, so join requests and grants are encoded in shared variables.

Afek-Kutten-Yung Election Algorithm - Join Requests

Let $leader_i < leader_j$ for some $j \in Neigh_i$.

Before p_i makes p_j its father, first it sends a *join request* to p_j.

This request is forwarded through p_j's component, toward the root (if any) of this component.

The root sends back a *grant* toward p_i, which travels the reverse path of the request.

When p_i receives this grant, it makes p_j its *father*:
$leader_i := leader_j$, $father_i := j$ and $dist_i := dist_j + 1$.

If p_j's component has no root, p_j will never join this component.

Communication is performed using *shared variables*, so
join requests and grants are encoded in shared variables.

Afek-Kutten-Yung Election Algorithm - Join Requests

A process can only be forwarding (and awaiting a grant for) at most one request message at a time.

Join requests and grants between "inconsistent" nodes are not forwarded.

Example: Given a ring with nodes u, v, w, and let $x > u, v, w$.

Initially, u and v consider themselves leader, while w considers u its father and x the leader.

Since $leader_w > leader_v$, v sends a join req to w.

Without the aformentioned consistency check, w would forward this join req to u. Since u considers itself leader, it would send back an ack to v (via w), and v would make w its father.

Since $leader_w \neq leader_u$, w would make itself leader.

Now we would have a symmetrical configuration to the initial one.

Afek-Kutten-Yung Election Algorithm - Join Requests

A process can only be forwarding (and awaiting a grant for) at most one request message at a time.

Join requests and grants between “inconsistent” nodes are not forwarded.

Example: Given a ring with nodes u, v, w, and let $x > u, v, w$.

Initially, u and v consider themselves leader, while w considers u its father and x the leader.

Since $leader_w > leader_v$, v sends a join req to w.

Without the aformentioned consistency check, w would forward this join req to u. Since u considers itself leader, it would send back an ack to v (via w), and v would make w its father.

Since $leader_w \neq leader_u$, w would make itself leader.

Now we would have a symmetrical configuration to the initial one.

Afek-Kutten-Yung Election Algorithm - Join Requests

A process can only be forwarding (and awaiting a grant for) at most one request message at a time.

Join requests and grants between "inconsistent" nodes are not forwarded.

Example: Given a ring with nodes u, v, w, and let $x > u, v, w$.

Initially, u and v consider themselves leader, while w considers u its father and x the leader.

Since $leader_w > leader_v$, v sends a join req to w.

Without the aformentioned consistency check, w would forward this join req to u. Since u considers itself leader, it would send back an ack to v (via w), and v would make w its father.

Since $leader_w \neq leader_u$, w would make itself leader.

Now we would have a symmetrical configuration to the initial one.

Afek-Kutten-Yung Election Algorithm - Example

Given two adjacent processes p_0 and p_1.

$leader_0 = leader_1 = 2$; $father_0 = 1$ and $father_1 = 0$; $dist_0 = dist_1 = 0$.

Since $dist_0 \neq dist_1 + 1$, p_0 declares itself *leader*:
$leader_0 := 0$, $father_0 := \perp$ and $dist_0 := 0$.

Since $leader_0 < leader_1$, p_0 sends a *join request* to p_1.

This join request does *not* immediately trigger a grant.

Since $dist_1 \neq dist_0 + 1$, p_1 declares itself *leader*:
$leader_1 := 1$, $father_1 := \perp$ and $dist_1 := 0$.

Since p_1 is now a proper root, it grants the join request of p_0, which makes p_1 its *father*:
$leader_0 := 1$, $father_0 := 1$ and $dist_0 := 1$.

Afek-Kutten-Yung Election Algorithm - Example

Given two adjacent processes p_0 and p_1.

$leader_0 = leader_1 = 2$; $father_0 = 1$ and $father_1 = 0$; $dist_0 = dist_1 = 0$.

Since $dist_0 \neq dist_1 + 1$, p_0 declares itself *leader*:
$leader_0 := 0$, $father_0 := \perp$ and $dist_0 := 0$.

Since $leader_0 < leader_1$, p_0 sends a *join request* to p_1.

This join request does *not* immediately trigger a grant.

Since $dist_1 \neq dist_0 + 1$, p_1 declares itself *leader*:
$leader_1 := 1$, $father_1 := \perp$ and $dist_1 := 0$.

Since p_1 is now a proper root, it grants the join request of p_0, which makes p_1 its *father*:
$leader_0 := 1$, $father_0 := 1$ and $dist_0 := 1$.

Afek-Kutten-Yung Election Algorithm - Example

Given two adjacent processes p_0 and p_1.

$leader_0 = leader_1 = 2$; $father_0 = 1$ and $father_1 = 0$; $dist_0 = dist_1 = 0$.

Since $dist_0 \neq dist_1 + 1$, p_0 declares itself *leader*:
$leader_0 := 0$, $father_0 := \perp$ and $dist_0 := 0$.

Since $leader_0 < leader_1$, p_0 sends a *join request* to p_1.

This join request does *not* immediately trigger a grant.

Since $dist_1 \neq dist_0 + 1$, p_1 declares itself *leader*:
$leader_1 := 1$, $father_1 := \perp$ and $dist_1 := 0$.

Since p_1 is now a proper root, it grants the join request of p_0, which makes p_1 its *father*:
$leader_0 := 1$, $father_0 := 1$ and $dist_0 := 1$.

Afek-Kutten-Yung Election Algorithm - Example

Given two adjacent processes p_0 and p_1.

$\mathit{leader}_0 = \mathit{leader}_1 = 2$; $\mathit{father}_0 = 1$ and $\mathit{father}_1 = 0$; $\mathit{dist}_0 = \mathit{dist}_1 = 0$.

Since $\mathit{dist}_0 \neq \mathit{dist}_1 + 1$, p_0 declares itself *leader*:
$\mathit{leader}_0 := 0$, $\mathit{father}_0 := \perp$ and $\mathit{dist}_0 := 0$.

Since $\mathit{leader}_0 < \mathit{leader}_1$, p_0 sends a *join request* to p_1.

This join request does *not* immediately trigger a grant.

Since $\mathit{dist}_1 \neq \mathit{dist}_0 + 1$, p_1 declares itself *leader*:
$\mathit{leader}_1 := 1$, $\mathit{father}_1 := \perp$ and $\mathit{dist}_1 := 0$.

Since p_1 is now a proper root, it grants the join request of p_0,
which makes p_1 its *father*:
$\mathit{leader}_0 := 1$, $\mathit{father}_0 := 1$ and $\mathit{dist}_0 := 1$.

Afek-Kutten-Yung Election Algorithm - Example

Given two adjacent processes p_0 and p_1.

$leader_0 = leader_1 = 2$; $father_0 = 1$ and $father_1 = 0$; $dist_0 = dist_1 = 0$.

Since $dist_0 \neq dist_1 + 1$, p_0 declares itself *leader*:
$leader_0 := 0$, $father_0 := \perp$ and $dist_0 := 0$.

Since $leader_0 < leader_1$, p_0 sends a *join request* to p_1.

This join request does *not* immediately trigger a grant.

Since $dist_1 \neq dist_0 + 1$, p_1 declares itself *leader*:
$leader_1 := 1$, $father_1 := \perp$ and $dist_1 := 0$.

Since p_1 is now a proper root, it grants the join request of p_0, which makes p_1 its *father*:
$leader_0 := 1$, $father_0 := 1$ and $dist_0 := 1$.

Afek-Kutten-Yung Election Algorithm - Correctness

A subgraph in the network with a leader value j that is not an id of any node in this subgraph, contains an inconsistency, so a process in this subgraph will declare itself leader.

Each process can only finitely often (each time due to incorrect initial register values) join a subgraph with a "false leader".

Let i be the largest id of any process in the network.

- Leader values greater than i will eventually disappear; and
- p_i will eventually declare itself leader.

After p_i has declared itself leader, the algorithm will eventually converge to a spanning tree with as root p_i.

Afek-Kutten-Yung Election Algorithm - Correctness

A subgraph in the network with a leader value j that is not an id of any node in this subgraph, contains an inconsistency, so a process in this subgraph will declare itself leader.

Each process can only finitely often (each time due to incorrect initial register values) join a subgraph with a “false leader”.

Let i be the largest id of any process in the network.

- Leader values greater than i will eventually disappear; and
- p_i will eventually declare itself leader.

After p_i has declared itself leader, the algorithm will eventually converge to a spanning tree with as root p_i.

Afek-Kutten-Yung Election Algorithm - Correctness

A subgraph in the network with a leader value j that is not an id of any node in this subgraph, contains an inconsistency, so a process in this subgraph will declare itself leader.

Each process can only finitely often (each time due to incorrect initial register values) join a subgraph with a "false leader".

Let i be the largest id of any process in the network.

- Leader values greater than i will eventually disappear; and
- p_i will eventually declare itself leader.

After p_i has declared itself leader, the algorithm will eventually converge to a spanning tree with as root p_i.

Garbage Collection

Processes are provided with memory, and *root objects* carry references to (local or remote) *heap objects*.

Also heap objects can carry references to each other.

Processes can perform three operations related to references:

- reference *creation* by the object owner;
- *duplication* of a remote reference to another processes;
- reference *deletion*.

Aim of garbage collection is to reclaim inaccessible heap objects.

Garbage Collection - Reference Counting

Reference counting is based on keeping track of the number of references to an object. If it drops to zero, the object is garbage.

Drawbacks:

- Reference counting cannot reclaim *cyclic* garbage.
- In a distributed setting, each operation on a remote reference induces a message.

Garbage Collection - Race Conditions

In a distributed setting, garbage collection suffers from race conditions.

Example: Process p holds a reference to object O on process r.

p duplicates this reference to process q.

p deletes the reference to O, and sends a dereference message to r.

r receives this message from p, and marks O as garbage.

O is reclaimed prematurely by the garbage collector.

q receives from p the reference to O.

Reference Counting - Acknowledgements

Consider *reference counting*. One way to avoid race conditions is to use acknowledgements.

In the previous example, before p duplicates the O-reference to q, it first sends an increment message for O to r.

Upon reception of this increment message, r increments O's counter, and sends an acknowledgement to p.

Upon reception of this acknowledgement, p duplicates the O-reference to q.

Drawback: High synchronization delays.

Reference Counting - Acknowledgements

Consider *reference counting*. One way to avoid race conditions is to use acknowledgements.

In the previous example, before p duplicates the O-reference to q, it first sends an increment message for O to r.

Upon reception of this increment message, r increments O's counter, and sends an acknowledgement to p.

Upon reception of this acknowledgement, p duplicates the O-reference to q.

Drawback: High synchronization delays.

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.

When the total weight of the object becomes equal to its partial weight, the object can be reclaimed.

Weighted Reference Counting - Drawback

When the weight of a reference (or object) has become 1, no more duplication (or creation) is possible.

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

An acknowledgement from the object owner to the reference is needed, to avoid race conditions.

Solution 2: The duplicated reference is to an artificial "object" with a new total weight, so that the reference to the original object becomes *indirect*.

Weighted Reference Counting - Drawback

When the weight of a reference (or object) has become 1, no more duplication (or creation) is possible.

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

An acknowledgement from the object owner to the reference is needed, to avoid race conditions.

Solution 2: The duplicated reference is to an artificial "object" with a new total weight, so that the reference to the original object becomes *indirect*.

Weighted Reference Counting - Drawback

When the weight of a reference (or object) has become 1, no more duplication (or creation) is possible.

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

An acknowledgement from the object owner to the reference is needed, to avoid race conditions.

Solution 2: The duplicated reference is to an artificial "object" with a new total weight, so that the reference to the original object becomes *indirect*.

Question

Why is the possibility of an underflow (weight 1) in weighted reference counting a much more serious problem than the possibility of an overflow of a reference counter?

Piquer's Indirect Reference Counting

The target object maintains a counter how many references have been *created*.

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

Process store where duplicated reference were duplicated from.

When a process receives a duplicated reference, but already holds a reference to this object, it sends a decrement to the sender of the duplicated reference, to decrease its counter for this reference.

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

When the counter of the object becomes zero, it can be reclaimed.

Indirect Reference Listing

Instead of a counter, the object and references keep track at which process a reference has been created or duplicated.

Advantage: Resilience against process failures (at the expense of some memory overhead).

Garbage Collection ⇒ Termination Detection

Tel and Mattern showed that garbage collection algorithms can be *tranformed* 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 for active processes p, there is 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-references.

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

Garbage Collection $\Rightarrow$ Termination Detection

Tel and Mattern showed that garbage collection algorithms can be *tranformed* 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 for active processes p, there is 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-references.

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

Garbage Collection ⇒ Termination Detection - Examples

Weighted *reference counting* yields weight-throwing *termination detection*.

Indirect *reference counting* yields Dijkstra-Scholten *termination detection*.

Garbage Collection ⇒ Termination Detection - Examples

Weighted *reference counting* yields weight-throwing *termination detection*.

Indirect *reference counting* yields Dijkstra-Scholten *termination detection*.

Garbage Collection - Mark-Scan

Mark-scan 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.

Generational Garbage Collection in Java

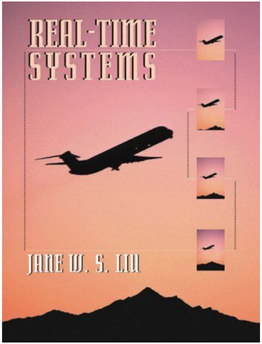
In practice, most objects can be reclaimed shortly after their creation.

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

- Garbage in the young generation is collected *frequently* using mark-*copy*.
- Garbage in the old generation is collected *less frequently* using mark-*compact*.

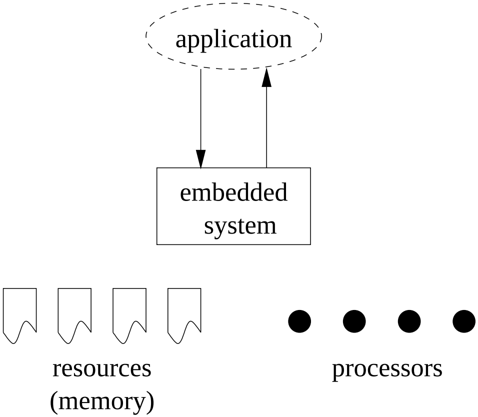
On-Line Scheduling Algorithms

Jane Liu, Real-Time Systems, Prentice Hall, 2000



See also Chapter 2.4 of: Andrew Tanenbaum and Albert Woodhull, *Operating Systems: Design and Implementation*, Prentice Hall, 1997

General Picture



Resources are allocated to processors.

Jobs

A job is a unit of work, scheduled and executed by the system.

Parameters of jobs are:

- functional behavior
- time constraints
- resource requirements

Job are divided over processors, and are competing for resources.

A scheduler decides in which order jobs are performed on a processor, and which resources they can claim.

Terminology

arrival time: when a job arrives at a processor

release time: when a job becomes available for execution

execution time: amount of processor time needed to perform the job (assuming it executes alone and all resources are available)

absolute deadline: when a job is required to be completed

relative deadline: maximum allowed length of time from arrival until completion of a job

hard deadline: late completion not allowed

soft deadline: late completion allowed

slack: available idle time of a job until the next deadline

A preemptive job can be suspended at any time of its execution

Out of scope

- jitter: imprecise release and execution times
- overrun management
- penalty for missing a soft deadline
- performance
- communication between jobs
- migration of jobs
- use of distant resources
- different processor and resource types

Types of Tasks

A task is a set of related jobs.

A processor distinguishes three types of tasks:

- periodic: known input before the start of the system, with hard deadlines
- aperiodic: executed in response to some external event, with soft deadlines
- sporadic: executed in response to some external event, with hard deadlines

Periodic Tasks

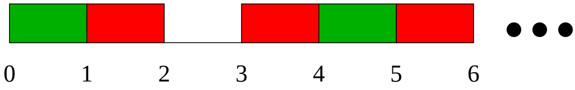
A periodic task is defined by:

- release time r (of the first periodic job)
- period p (regular time interval, at the start of which a periodic job is released)
- execution time e

For simplicity we assume that the relative deadline of each periodic job is equal to its period.

Periodic Tasks - Example

$T_1 = (1, 2, 1)$ and $T_2 = (0, 3, 1)$.

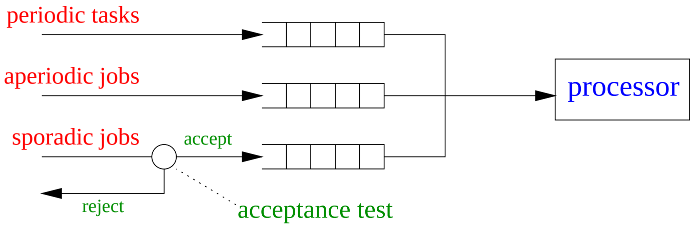


The hyperperiod is 6.

The conflict at time 3 must be resolved by some scheduler.

Job Queues at a Processor

We focus on individual aperiodic and sporadic *jobs*.



Sporadic jobs are only accepted when they can be completed in time.

Aperiodic jobs are always accepted, and performed such that periodic and accepted sporadic jobs do not miss their deadlines.

The queueing discipline of aperiodic jobs tries to minimize e.g. average tardiness (completion time monus deadline), or the number of missed soft deadlines.

Utilization

Utilization of a *periodic task* (r, p, e) is $\frac{e}{p}$.

Utilization of a *processor* is the sum of utilizations of its periodic tasks.

Assumptions: Jobs preemptive, no resource competition.

Theorem: Utilization of a processor is ≤ 1 if and only if scheduling its periodic tasks is feasible.

Scheduler

The scheduler of a processor schedules and allocates resources to jobs (according to some scheduling algorithms and resource access control algorithms).

A schedule is valid if:

- jobs are not scheduled before their release times, and
- the total amount of processor time assigned to a job equals its (maximum) execution time.

A (valid) schedule is feasible if all hard deadlines are met.

A scheduler is optimal if it produces a feasible schedule whenever possible.

Clock-Driven Scheduling

Off-line scheduling: The schedule for periodic tasks is computed beforehand (typically with an algorithm for an NP-complete graph problem).

Time is divided into regular time intervals called frames.

In each frame, a predetermined set of periodic tasks is executed.

Jobs may be sliced into subjobs, to accommodate frame length.

Clock-driven scheduling is conceptually simple, but cannot cope well with:

- jitter
- system modifications
- nondeterminism

Priority-Driven Scheduling

On-line scheduling: The schedule is computed at run-time.

Scheduling decisions are taken when:

- periodic jobs are released or aperiodic/sporadic jobs arrive
- jobs are completed
- resources are required or released

Released jobs are placed in priority queues, e.g. ordered by:

- release time (FIFO, LIFO)
- execution time (SETF, LETF)
- period of the task (RM)
- deadline (EDF)
- slack (LST)

We will focus on *EDF* scheduling.

Periodic tasks and jobs are assumed to be *preemptive*.

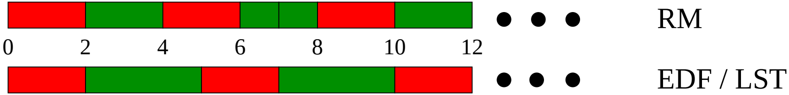
RM Scheduler

Rate Monotonic: Shorter period gives a higher priority.

Advantage: Priority on the level of tasks makes RM easier to analyze than EDF/LST.

Example: Non-optimality of the RM scheduler
(one processor, preemptive jobs, no competition for resources).

Let $T_1 = (0, 4, 2)$ and $T_2 = (0, 6, 3)$.



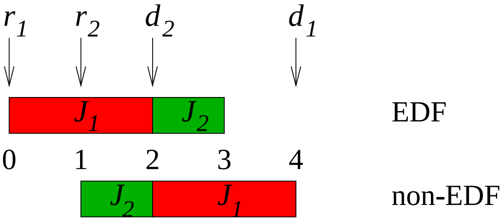
Remark: If for periods $p < p'$, p is always a divisor of p', then the RM scheduler is optimal.

EDF Scheduler

Earliest Deadline First: The earlier the deadline, the higher the priority.

Theorem: Given one processor, and preemptive jobs. When jobs do not compete for resources, the EDF scheduler is optimal.

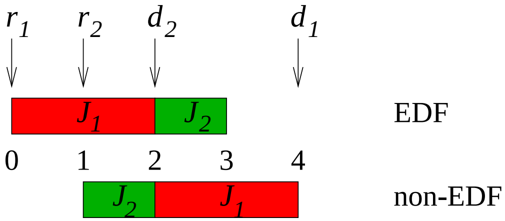
Example: Non-optimality in case of non-preemption.



EDF Scheduler

Example: Non-optimality in case of resource competition.

Let J_1 and J_2 both require resource R.



EDF Scheduler - Drawbacks

Dynamic priority of periodic tasks makes it difficult to analyze which deadlines are met in case of overloads.

Late jobs can cause other jobs to miss their deadlines (good overrun management is needed).

LST Scheduler

Least Slack Time first: less slack gives a job higher priority.

With the LST scheduler, priorities of jobs change dynamically.

Remark: Continuous scheduling decisions would lead to context switch overhead in case of two jobs with the same slack.

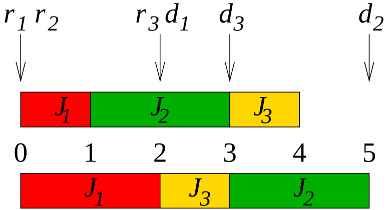
Theorem: Given one processor, and preemptive jobs. When jobs do not compete for resources, the LST scheduler is optimal.

Drawback of LST: computationally expensive

Scheduling Anomaly

Let jobs be non-preemptive. Then shorter execution times can lead to violation of deadlines.

Example: Consider the EDF (or LST) scheduler.



If jobs are preemptive, and there is no competition for resources, then there is no scheduling anomaly.

Scheduling Aperiodic Jobs

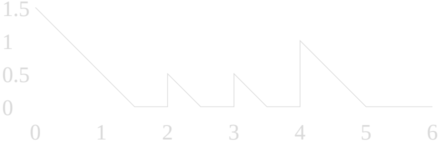
(For the moment, we ignore sporadic jobs.)

Background: Aperiodic jobs are only scheduled in idle time.

Drawback: Needless delay of aperiodic jobs.

Slack stealing: Periodic tasks and accepted sporadic jobs may be interrupted if there is sufficient slack.

Example: $T_1 = (0, 2, \frac{1}{2})$ and $T_2 = (0, 3, \frac{1}{2})$. Aperiodic jobs available in $[0, 6]$.



Drawback: Difficult to compute in case of jitter.

Scheduling Aperiodic Jobs

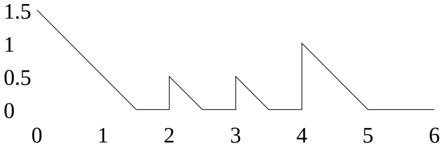
(For the moment, we ignore sporadic jobs.)

Background: Aperiodic jobs are only scheduled in idle time.

Drawback: Needless delay of aperiodic jobs.

Slack stealing: Periodic tasks and accepted sporadic jobs may be interrupted if there is sufficient slack.

Example: $T_1 = (0, 2, \frac{1}{2})$ and $T_2 = (0, 3, \frac{1}{2})$. Aperiodic jobs available in $[0, 6]$.



Drawback: Difficult to compute in case of jitter.

Polling Server

Given a period p_s, and an execution time e_s for aperiodic jobs in such a period.

At the start of a new period, *the first e_s time units* can be used to execute aperiodic jobs.

Consider periodic tasks $T_k = (r_k, p_k, e_k)$ for $k = 1, \ldots, n$.
The polling server works if

$$\sum_{k=1}^{n} \frac{e_k}{p_k} + \frac{e_s}{p_s} \leq 1$$

Drawback: Aperiodic jobs released just after a polling may be delayed needlessly.

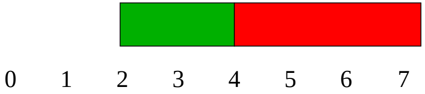
Deferrable Server

Allows a polling server to save its execution time within a period p_s (but not after this period!) if the aperiodic queue is empty.

The EDF scheduler treats the deadline of a deferrable server at the end of a period p_s as a hard deadline.

Remark: $\sum_{k=1}^{n} \frac{e_k}{p_k} + \frac{e_s}{p_s} \leq 1$ does not guarantee that periodic jobs meet their deadlines.

Example: $T = (2, 5, 3\frac{1}{3})$ and $p_s = 3$, $e_s = 1$. An aperiodic job with $e = 2$ arrives at 2.



T misses its deadline at 7

Drawbacks: Partial use of available bandwidth.

Difficult to determine good values for p_s and e_s.

Total Bandwidth Server

Fix an allowed utilization rate $\tilde{u}_s$ for the server, such that

$$\sum_{k=1}^{n} \frac{e_k}{p_k} + \tilde{u}_s \leq 1$$

When the aperiodic queue is non-empty, a deadline d is determined for the head of the queue, according to the rules below.

Let the head of the aperiodic queue have execution time e.

When, at a time t, either a job arrives at the *empty* aperiodic queue, or an aperiodic job *completes* and the *tail* of the aperiodic queue is *non-empty*:

$$d := \max(d, t) + \frac{e}{\tilde{u}_s}$$

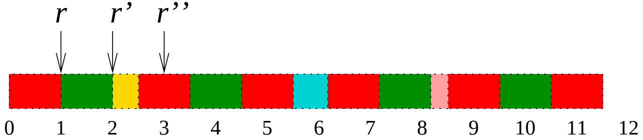
Initially, $d = 0$.

Total Bandwidth Server

Aperiodic jobs can now be treated in the same way as periodic jobs, by the EDF scheduler.

In the absence of sporadic jobs, aperiodic jobs meet the deadlines assigned to them (which may differ from their actual soft deadlines).

Example: $T_1 = (0, 2, 1)$ and $T_2 = (0, 3, 1)$. We fix $\tilde{u}_s = \frac{1}{6}$.



A, released at 1 with $e = \frac{1}{2}$, gets (at 1) deadline $1 + 3 = 4$.

A', released at 2 with $e' = \frac{2}{3}$, gets (at $2\frac{1}{2}$) deadline $4 + 4 = 8$.

A'', released at 3 with $e'' = \frac{1}{3}$, gets (at $6\frac{1}{3}$) deadline $8 + 2 = 10$.

Acceptance Test for Sporadic Jobs

A sporadic job with deadline d and execution time e is accepted at time t if utilization (of the periodic and accepted sporadic jobs) in the time interval $[t, d]$ is never more than $1 - \frac{e}{d-t}$.

If accepted, utilization in $[t, d]$ is increased with $\frac{e}{d-t}$.

Example: Periodic task $T = (0, 2, 1)$.

Sporadic job with $r = 1$, $e = 2$ and $d = 6$ is accepted.
Utilization in $[1, 6]$ is increased to $\frac{9}{10}$.

Sporadic job with $r = 2$, $e = 2$ and $d = 20$ is rejected *(although it could be scheduled).*

Sporadic job with $r = 3$, $e = 1$ and $d = 13$ is accepted.
Utilization in $[3, 6]$ is increased to 1, and utilization in $[6, 13]$ to $\frac{3}{5}$.

Acceptance Test for Sporadic Jobs

A sporadic job with deadline d and execution time e is accepted at time t if utilization (of the periodic and accepted sporadic jobs) in the time interval $[t, d]$ is never more than $1 - \frac{e}{d-t}$.

If accepted, utilization in $[t, d]$ is increased with $\frac{e}{d-t}$.

Example: Periodic task $T = (0, 2, 1)$.

Sporadic job with $r = 1$, $e = 2$ and $d = 6$ is accepted.
Utilization in $[1, 6]$ is increased to $\frac{9}{10}$.

Sporadic job with $r = 2$, $e = 2$ and $d = 20$ is rejected *(although it could be scheduled)*.

Sporadic job with $r = 3$, $e = 1$ and $d = 13$ is accepted.
Utilization in $[3, 6]$ is increased to 1, and utilization in $[6, 13]$ to $\frac{3}{5}$.

Acceptance Test for Sporadic Jobs

Periodic and accepted sporadic jobs are scheduled by the EDF scheduler.

The acceptance test may reject schedulable sporadic jobs.

The total bandwidth server can be integrated with an acceptance test for sporadic jobs (e.g. by making the allowed utilization rate $\tilde{u}_s$ dynamic).

Acceptance Test for Sporadic Jobs

Periodic and accepted sporadic jobs are scheduled by the EDF scheduler.

The acceptance test may reject schedulable sporadic jobs.

The total bandwidth server can be integrated with an acceptance test for sporadic jobs (e.g. by making the allowed utilization rate $\tilde{u}_s$ dynamic).

Remote Access Control Algorithms

Resource units can be requested by jobs during their execution, and are allocated to jobs in a mutually exclusive fashion.

When a requested resource is refused, the job is preempted (blocked).

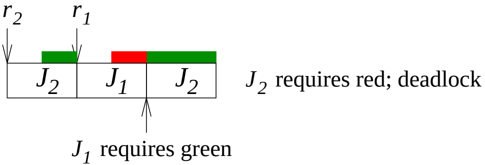
Resource sharing gives rise to scheduling anomaly.

Remote Access Control Algorithms

Dangers of resource sharing:

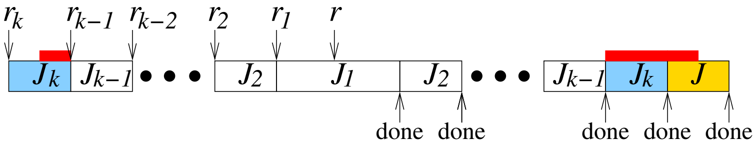
(1) Deadlock can occur.

Example: $J_1 > J_2$.



(2) A job J can be blocked by lower-priority jobs.

Example: $J > J_1 > \cdots > J_k$, and J, J_k require the red resource.



Question

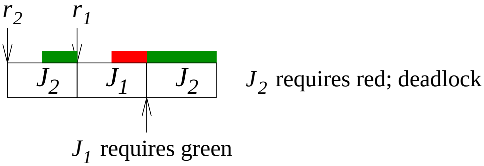
How can we avoid blocking by lower-priority jobs?

Priority Inheritance

When a job J requires a resource R and becomes blocked because a lower-priority job J' holds R, then J' inherits the priority of J until it releases R.

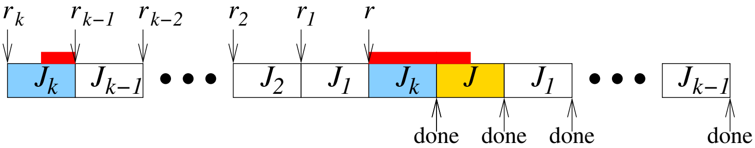
(1) Deadlock can still occur.

Example: $J_1 > J_2$.



(2) Blocking by lower-priority jobs becomes less likely.

Example: $J > J_1 > \cdots > J_k$, and J, J_k require red.



Priority Ceiling

The priority ceiling of a *resource* R at time t is the highest priority of (known) jobs that will require R at some time $\geq t$.

The priority ceiling of the *system* at time t is the highest priority ceiling of resources that are in use at time t. (It has a special bottom value Ω when no resources are in use.)

In the priority ceiling algorithm, from the *arrival* of a job, this job is *not released* until its priority is higher than the priority ceiling of the system.

There is also priority inheritance: A job inherits the priority of a higher-priority job that it is blocking.

Assumption: The resources required by a job are known beforehand.

Note: In the pictures to follow, *a* denotes the *arrival* of a job (different from its release).

Priority Ceiling

The priority ceiling of a *resource* R at time t is the highest priority of (known) jobs that will require R at some time $\geq t$.

The priority ceiling of the *system* at time t is the highest priority ceiling of resources that are in use at time t. (It has a special bottom value Ω when no resources are in use.)

In the priority ceiling algorithm, from the *arrival* of a job, this job is *not released* until its priority is higher than the priority ceiling of the system.

There is also priority inheritance: A job inherits the priority of a higher-priority job that it is blocking.

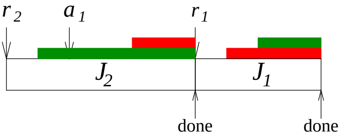
Assumption: The resources required by a job are known beforehand.

Note: In the pictures to follow, *a* denotes the *arrival* of a job (different from its release).

Priority Ceiling

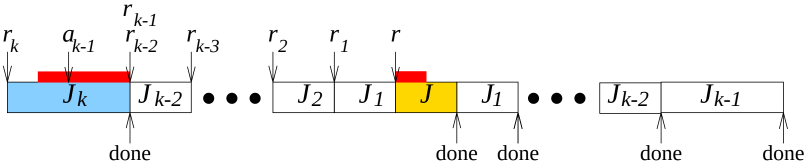
(1) No deadlocks, if job priorities are fixed (e.g. EDF, but not LST). Because a job can only start executing when all resources it will require are free.

Example: $J_1 > J_2$.



(2) Blocking by lower-priority jobs becomes less likely.

Example: $J > J_1 > \cdots > J_k$, and J, J_k require the red resource.



In this example, the future arrival of J is known when J_{k-1} arrives.

Questions

What would happen if J were only known at its arrival?

How can J get blocked by $J_1, \ldots, J_k$?

Questions

What would happen if J were only known at its arrival?

How can J get blocked by $J_1, \ldots, J_k$?

Priority Ceiling - Multiple Resource Units

Priority ceiling assumed only *one unit* per resource type.

In case of multiple units of the same resource type, the definition of priority ceiling needs to be adapted:

The priority ceiling of a resource R with k free units at time t is the highest priority level of known jobs that require $> k$ units of R at some time $\geq t$.

PODC Influential Paper Award

2000: Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, CACM 1978

2001: Fischer, Lynch, Paterson, *Impossibility of Distributed Consensus with One Faulty Process*, JACM 1985

2002: Dijkstra, *Self-Stabilizing Systems in Spite of Distributed Control*, CACM 1974

2003: Herlihy, *Wait-Free Synchronization*, ACM TOPLAS 1991

2004: Gallager, Humblet, Spira, *A Distributed Algorithm for Minimum-Weight Spanning Trees*, ACM TOPLAS 1983

2005: Pease, Shostak, Lamport, *Reaching Agreement in the Presence of Faults*, JACM 1980

2006: Mellor-Crummey, Scott, *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors*, ACM TOCS 1991

2007: Dwork, Lynch, Stockmeyer, *Consensus in the Presence of Partial Synchrony*, JACM 1988

2008: Awerbuch, Peleg, *Sparse Partitions*, FOCS 1990