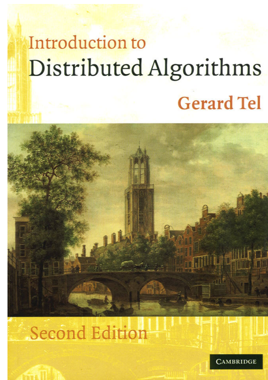
Distributed Algorithms



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

Distributed Systems

A distributed system is an interconnected collection of autonomous processes.

Motivation:

- information exchange (WAN)
- resource sharing (LAN)
- multicore programming
- parallelization to increase performance
- replication to increase reliability
- 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.

  Example: Termination and deadlock detection become an issue.

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

  Example: Mutual exclusion becomes an issue.

- *Nondeterminism:* Execution of processes is nondeterministic.

  Example: Running a system two times can give different results.

Communication Paradigms

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

We will only consider message passing.

(The course *Concurrency & Multithreading* focuses on shared memory.)

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

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

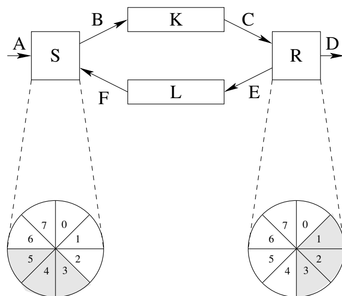
We will mainly consider asynchronous communication.

Communication Protocols

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

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

Example: Sliding window protocols.



Assumptions

Unless stated otherwise, we assume:

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

Directed Versus Undirected Channels

Channels can be *directed* or *undirected*.

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

Remarks:

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

Formal Framework

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

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

(The course *Protocol Validation* focuses on proving correctness
of distributed algorithms and communication protocols.)

Transition Systems

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

The configuration evolves in discrete steps, called transitions.

A transition system consists of:

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

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

Executions

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

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

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

States and Events

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

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

A process can perform internal, send and receive events.

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

An algorithm is centralized if there is exactly one initiator.

A decentralized algorithm can have multiple initiators.

Causal Order

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

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

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

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

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

Computations

A permutation of events in an execution that respects the causal order, doesn't 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 terminal configuration.

Lamport's Clock

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

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

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

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

Vector Clock

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

We consider $\mathbb{N}^N$ with a *partial* order defined by:

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

The vector clock VC maps occurrences of events in a computation to $\mathbb{N}^N$ such that $a \prec b \Leftrightarrow VC(a) < VC(b)$.

$VC(a) = (k_0, \ldots, k_{N-1})$ where each k_i is the length of a longest causality chain $a^i_1 \prec \cdots \prec a^i_{k_i}$ of events at process p_i with $a^i_{k_i} \preceq a$.

Question: Why does $VC(a) < VC(b)$ imply $a \prec b$?

VC can also be computed at run-time.

Complexity Measures

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

Message complexity: Total number of messages exchanged.

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

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

Space complexity: Amount of space needed for the processes.

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).

Big O Notation

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

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

Divide-and-conquer algorithms typically have a logarithm in their time complexity.

Namely, dividing a problem of input size 2^k into subproblems of size 1 takes k steps.

And $k = \log_2 2^k$.

Big O Notation

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

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

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

Examples: $n^a = O(n^b)$ for all $0 < a \leq b$

$n^a = O(b^n)$ for all $a > 0$ and $b > 1$

$\log_a n = O(n^b)$ for all $a, b > 0$

$\log_a n = \Theta(\log_b n)$ for all $a, b > 0$

By definition, $\log_a a^n = n$, which implies $a^{\log_a n} = n$. So

$$a^{\log_a b \cdot \log_b n} = b^{\log_b n} = n$$

Therefore, $\log_a b \cdot \log_b n = \log_a n$.

Assertions

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

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

"*something bad will never happen*"

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

"*something good will eventually happen*"

Invariants

Assertion P is an invariant if:

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

Each invariant is a safety property.

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

Fairness

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.

Snapshots

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

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

Snapshots can be used to determine off-line stable properties, which remain true as soon as they have become true.

Examples: deadlock, garbage.

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

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:

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

A snapshot is meaningful if it is a configuration of an execution in the 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 channel state of pq.

Chandy-Lamport Algorithm

Consider a directed network with *FIFO* channels.

Initiators can take a local snapshot of their state,
and send a control message $\langle \mathbf{mkr} \rangle$ to their neighbors.

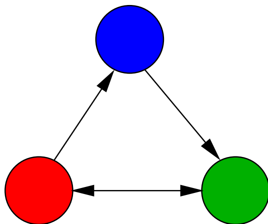
When a process that hasn't yet taken a snapshot receives $\langle \mathbf{mkr} \rangle$,
it takes a local snapshot of its state, and sends $\langle \mathbf{mkr} \rangle$ to its neighbors.

q computes as channel state of pq the messages it receives via pq
after taking its local snapshot and before receiving $\langle \mathbf{mkr} \rangle$ from p.

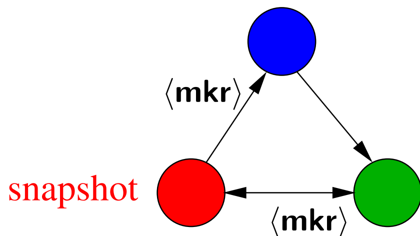
If channels are FIFO, this produces a meaningful snapshot.

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

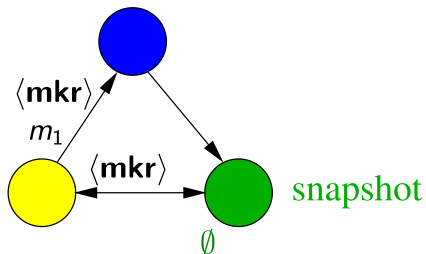
Chandy-Lamport Algorithm - Example



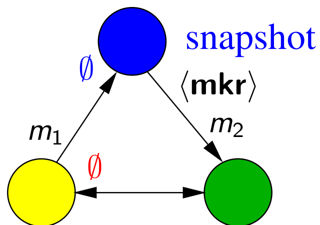
Chandy-Lamport Algorithm - Example



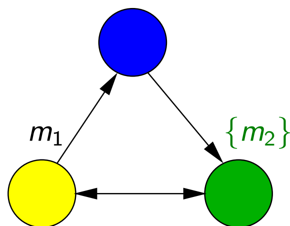
Chandy-Lamport Algorithm - Example



Chandy-Lamport Algorithm - Example



Chandy-Lamport Algorithm - Example



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

The send of m_1 and the internal event from red to yellow are both not causally before the send of m_2. So the snapshot is a configuration of an execution in the same computation as the actual execution.

Lai-Yang Algorithm

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

Initiators can take a local snapshot of their state.

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

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

All processes eventually take a local snapshot.

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

Lai-Yang Algorithm - Control Messages

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

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

Wave Algorithms

Decide events are special internal events.

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

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

Example: In the *ring algorithm*, the initiator sends a token, which is passed on by all other processes.
The initiator decides after the token has returned.

Traversal Algorithms

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

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

Traversal algorithms build a spanning tree:

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

Tarry's Algorithm (from 1895)

Consider an undirected network.

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

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

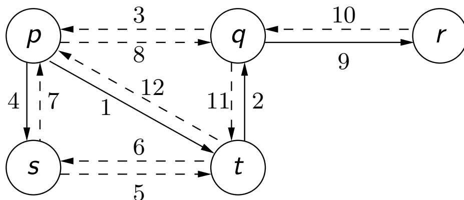
The token travels through each channel both ways,
and finally ends up at the initiator.

Message complexity: $2E$ messages

Time complexity: $\leq 2E$ time units

Tarry's Algorithm - Example

p is the initiator.

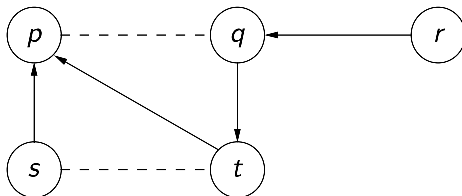


The network is undirected (and unweighted); arrows and numbers mark the path of the token.

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

Tarry's Algorithm - Example

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



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

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

Question: Could this spanning tree have been produced by a depth-first search ?

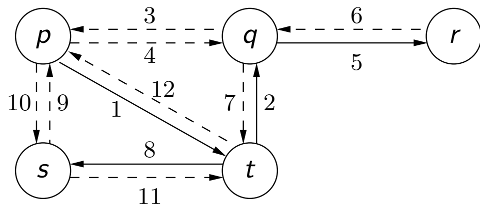
Depth-First Search

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

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

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

Example:



Depth-First Search with Neighbor Knowledge

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

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

Message complexity: $2N - 2$ messages

Each tree edge carries 2 tokens.

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

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

Awerbuch's Algorithm

- A process holding the token for the first time informs all neighbors except its parent and the process to which it forwards the token.
- The token is only forwarded when these neighbors have all acknowledged reception.
- The token is only forwarded to processes that weren't yet visited by the token (except when a process sends the token to its parent).

Awerbuch's Algorithm - Complexity

Message complexity: $\leq 4E$ messages

Each frond edge carries 2 info and 2 ack messages.

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

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

Each tree edge carries 2 tokens.

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

Cidon's Algorithm

Abolishes ack's from Awerbuch's algorithm.

- The token is forwarded without delay. Each process p records to which process $forward_p$ it forwarded the token last.
- Suppose process p receives the token from a process $q \neq forward_p$. Then p marks the channel pq as used and *purges* the token.
- Suppose process q receives an info message from $forward_q$. Then it continues forwarding the token.

Cidon's Algorithm - Complexity

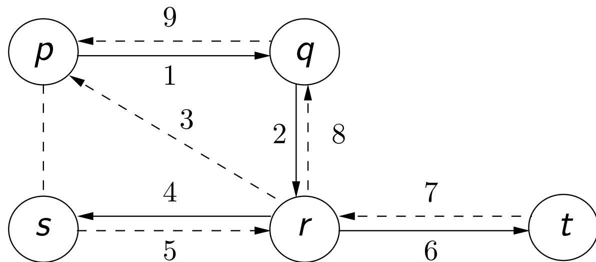
Message complexity: $< 4E$ messages

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

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

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

Cidon's Algorithm - Example



Tree Algorithm

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

The local algorithm at a process p:

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

  Then it sends a message to its parent.

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

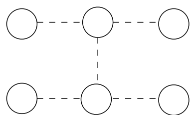
  It sends the decision to all neighbors except its parent.

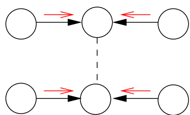
- If p receives a *decision* from its parent,
  it passes it on to all other neighbors.

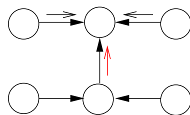
Remark: Always *two* processes decide.

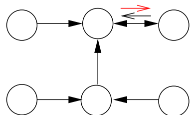
Message complexity: $2N-2$ messages

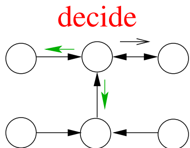
Tree Algorithm - Example

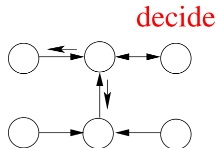


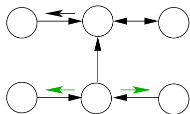


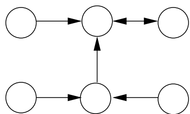












Question

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

Answer: Then the algorithm deadlocks without a decision.

Echo Algorithm

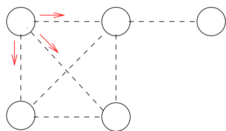
The echo algorithm is a centralized wave algorithm for undirected networks.

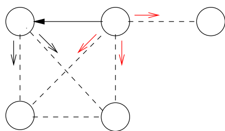
- The initiator sends a message to all neighbors.
- When a non-initiator receives a message for the first time, it makes the sender its *parent*.

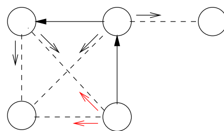
  Then it sends a message to all neighbors except its parent.
- When a non-initiator has received a message from all neighbors, it sends a message to its parent.
- When the initiator has received a message from all neighbors, it decides.

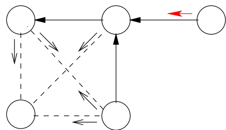
Message complexity: $2E$ messages

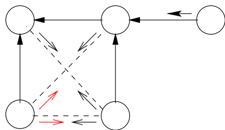
Echo Algorithm - Example

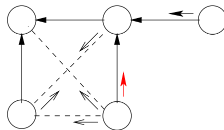


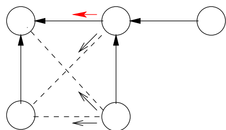


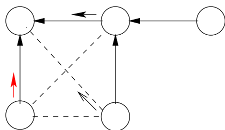




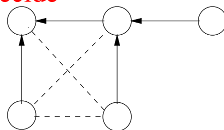








decide



Question

Let each process initially carry an integer value.

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

Communication and Resource Deadlock

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

- another process is ready to send some input, or receive an input (communication deadlock)
- or resources become available (resource deadlock)

Both types of deadlock are captured by the N-out-of-M model:
A process can wait for N grants out of M requests.

Examples:

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

Wait-For Graph

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

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

Only non-blocked processes can grant a request.

A directed wait-for graph captures dependencies between processes.

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

Wait-For Graph - Example 1

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

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

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

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

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

Wait-For Graph - Example 2

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

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

The resource is free, so w sends a grant to say u.
In the wait-for graph, the edge uw is removed.

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

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

Now w can grant the request from v.
The edge vw is removed and the edge wv is created.

Static Analysis on a Wait-For Graph

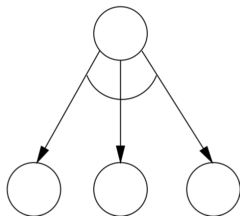
First a snapshot is taken of the wait-for graph.

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

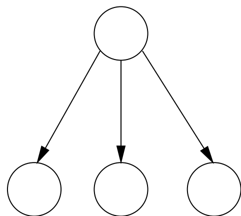
- Non-blocked nodes can grant requests.
- When a request is granted, the corresponding edge is removed.
- When an N-out-of-M request has received N grants, the requester becomes unblocked. (The remaining $M-N$ outgoing edges are purged.)

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

Drawing Wait-For Graphs

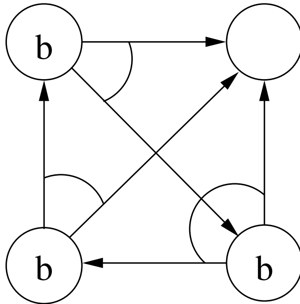


AND (3–out–of–3) request

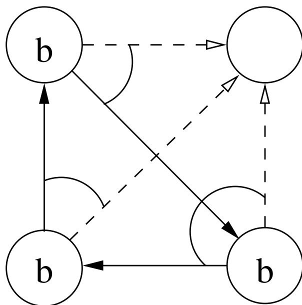


OR (1–out–of–3) request

Static Analysis - Example 1

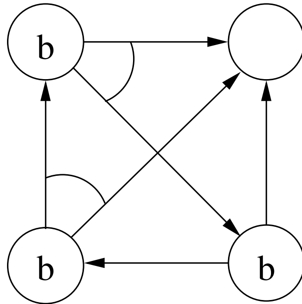


Static Analysis - Example 1

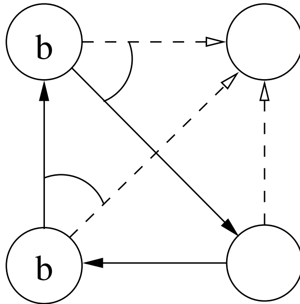


Deadlock

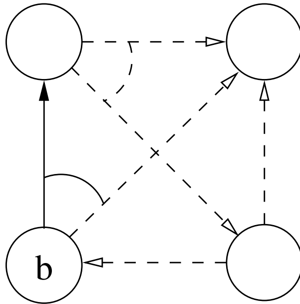
Static Analysis - Example 2

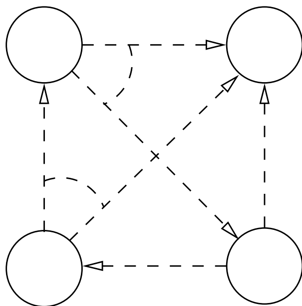


Static Analysis - Example 2



Static Analysis - Example 2





No deadlock

Bracha-Toueg Deadlock Detection Algorithm - Snapshot

Given an undirected network, and a basic algorithm.

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

Each node u takes a local snapshot of:

1. requests it sent or received that weren't yet granted or purged;
2. grant and purge messages in edges.

Then it computes:

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

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

Bracha-Toueg Deadlock Detection Algorithm

Initially, $requests_u$ is the number of grants u requires in the wait-for graph to become unblocked.

When $requests_u$ is (or becomes) 0, u sends grant messages to all nodes in In_u.

When u receives a grant message, $requests_u := requests_u - 1$.

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

Bracha-Toueg Deadlock Detection Algorithm

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

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

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

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

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

Bracha-Toueg Deadlock Detection Algorithm

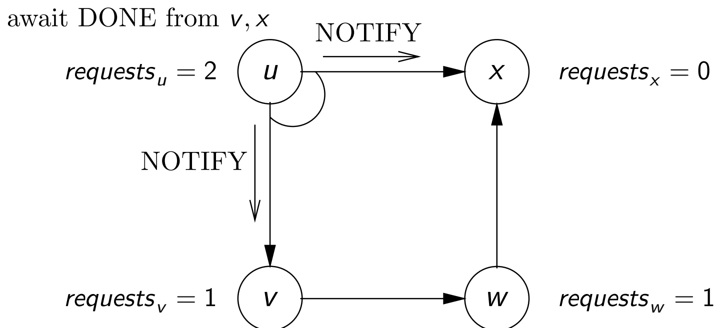
Let u receive NOTIFY.
If $notified_u = false$, then u executes $Notify_u$.
u sends back DONE.

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

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

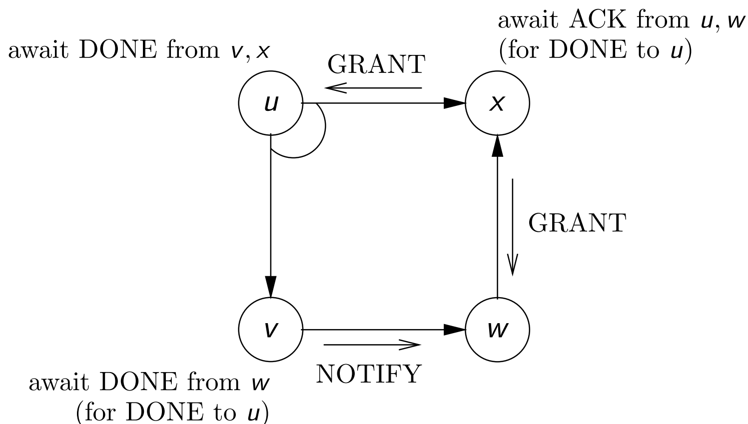
If it is still *false*, the node concludes it is deadlocked.

Bracha-Toueg Deadlock Detection Algorithm - Example

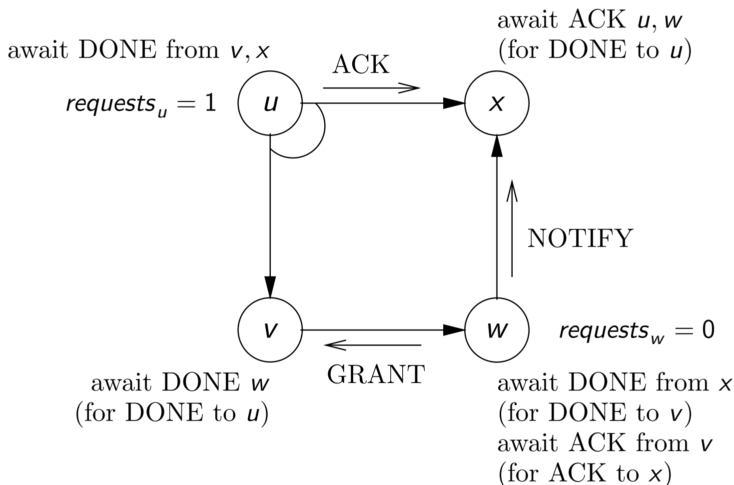


u is the initiator.

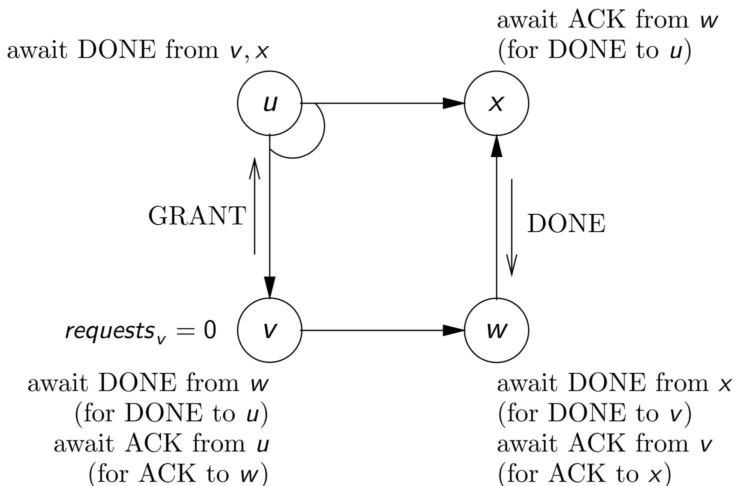
Bracha-Toueg Deadlock Detection Algorithm - Example



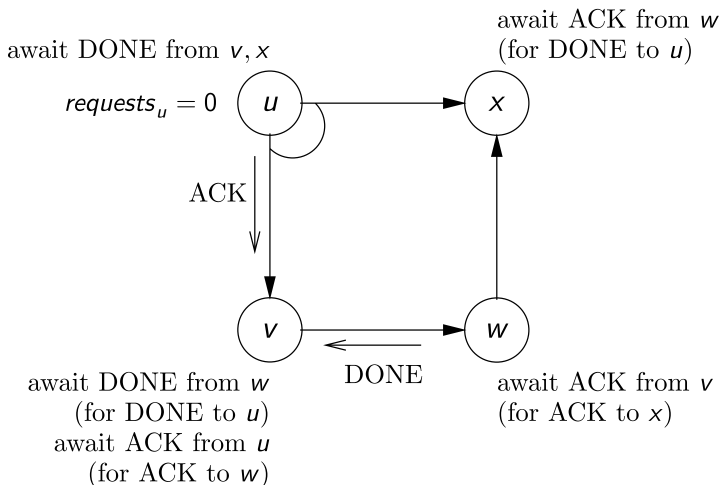
Bracha-Toueg Deadlock Detection Algorithm - Example



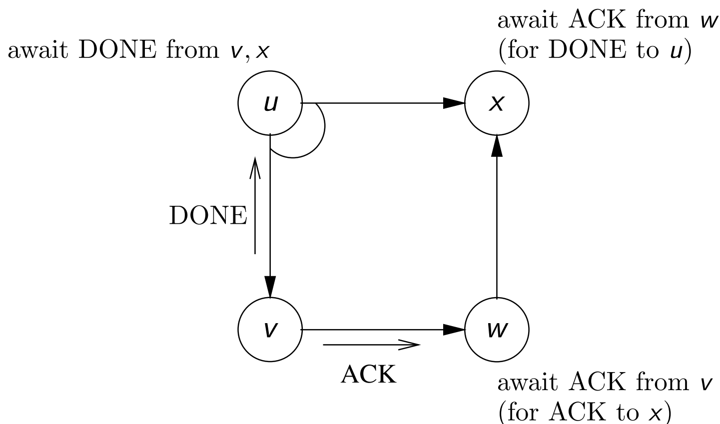
Bracha-Toueg Deadlock Detection Algorithm - Example



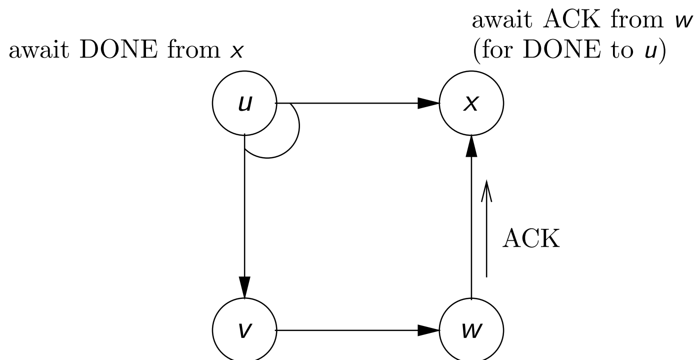
Bracha-Toueg Deadlock Detection Algorithm - Example



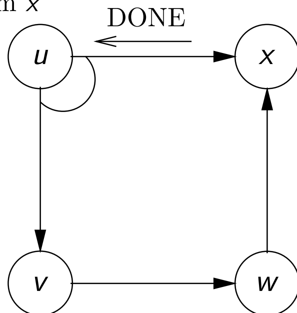
Bracha-Toueg Deadlock Detection Algorithm - Example



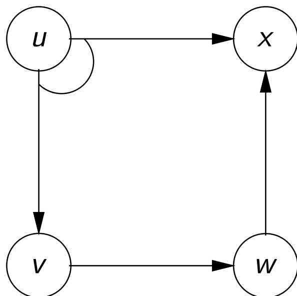
Bracha-Toueg Deadlock Detection Algorithm - Example



Bracha-Toueg Deadlock Detection Algorithm - Example



Bracha-Toueg Deadlock Detection Algorithm - Example



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

Bracha-Toueg Deadlock Detection Algorithm - Correctness

Replying with a DONE (to a NOTIFY) or ACK (to a GRANT) is only delayed by a node u if it is executing $Grant_u$ because $requests_u$ was 0 (in case of DONE) or has become 0 by the GRANT (in case of ACK), and it is awaiting ACK's.

A cycle of nodes awaiting ACK's always contains a node v of which $requests_v$ wasn't set to 0 by a GRANT from a node in this cycle. So this v can send back an ACK to its predecessor in this cycle.

This guarantees that the Bracha-Toueg algorithm is deadlock-free: The initiator eventually receives DONE's from all nodes in its *Out* set.

Bracha-Toueg Deadlock Detection Algorithm - Correctness

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

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

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

Original Bracha-Toueg Deadlock Detection Algorithm

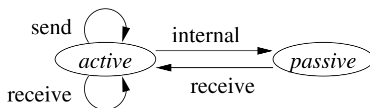
In their original algorithm, Bracha and Toueg don't take into account channel states in the snapshot of the wait-for graph.

As a result, they obtain an *underapproximation* of the wait-for graph.

Termination Detection

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

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



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

Termination detection shouldn't influence basic computations.

Dijkstra-Scholten Algorithm

Requires a centralized basic algorithm, and an undirected network.

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

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

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

Shavit-Francez Algorithm

Allows a decentralized basic algorithm; requires an undirected network.

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

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

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

When an initiator p is passive and $cc_p = 0$, it starts a wave in which only processes that aren't in a tree participate; *decide* calls *Announce*.

Weight-Throwing Termination Detection

Requires a centralized basic algorithm; allows a directed network.

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

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

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

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

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

Weight-Throwing Termination Detection - Underflow

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

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

An ack from the initiator is needed, to avoid race conditions.

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

Question

Why is the following termination detection algorithm not correct ?

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

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

Rana's Algorithm

Allows a decentralized basic algorithm; requires an undirected network.

A logical clock provides *(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.

If at 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 (and its id).

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

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

Rana's Algorithm - Correctness

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

Then some process doesn't take part in the wave,
meaning that at some time $> t$ it isn't quiet.

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

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 wasn't yet visited by this wave.

Then this wave won't complete.

Namely, let the wave be tagged with t.

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

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

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

Token-Based Termination Detection

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

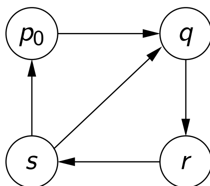
A process p_0 is initiator of a traversal algorithm to check whether all processes are passive.

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

Solution: Synchronous communication.

Complication 2: A traversal of only passive processes doesn't 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.

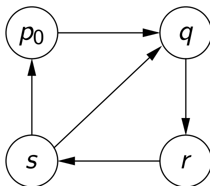
The token travels on to p_0, which falsely calls *Announce*.

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, who waits until it is passive:
  - If both the token and p_0 are white, p_0 calls *Announce*.
  - Else, p_0 becomes white and 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*.

Safra's Algorithm

Allows a directed network 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 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

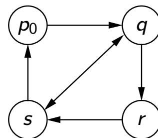
Processes are colored white or black. Initially they are white, and a process that receives a basic message becomes black.

- 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, who 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*.
  - Else, p_0 sends a white token again.

Safra's Algorithm - Example

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



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

s becomes passive.

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

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

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

s becomes passive.

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

q becomes passive.

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

Garbage Collection

Processes are provided with memory.

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

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

Aim of garbage collection: To reclaim inaccessible objects.

Three operations by processes to build or delete a reference:

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

Reference Counting

Reference counting tracks the number of references to an object.
If it drops to zero, and there are no pointers, the object is garbage.

Advantage: Can be performed at run-time.

Drawbacks:

- Reference counting can't reclaim *cyclic* garbage.
- In a distributed setting, duplicating or deleting a reference may induce a message.

Reference Counting - Race Conditions

Reference counting may give rise to *race conditions*.

Example: Process p holds a reference to object O on process r.
The reference counter of O is 1, and there are no pointers.

p sends a message to process q, duplicating its reference to O.

p deletes its 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 - Increment Messages

Race conditions can be avoided by *increment messages* and *ack's*.

In the previous example, before p duplicates the O-reference to q, let it first send an increment message for O to r.

Upon reception of this increment message, r increments O's counter, and sends an ack to p.

Upon reception of this ack, p duplicates the O-reference to q.

Drawback: High synchronization delays.

Indirect Reference Counting

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

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

Processes store where a reference is duplicated from.

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

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

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

Weighted Reference Counting

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

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

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

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

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

Weighted Reference Counting - Underflow

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

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

An ack from the object owner to the reference is needed, to avoid race conditions.

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

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

Question

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

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

Garbage Collection ⇒ Termination Detection

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

Given a basic algorithm. Let each process p host one artificial root object O_p. There is also a special non-root object Z.

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

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

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

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

Garbage Collection $\Rightarrow$ Termination Detection - Examples

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

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

Mark-Scan

Mark-scan garbage collection consists of two phases:

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

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

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

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

Copying is significantly faster than compaction,
but leads to fragmentation of the memory space.

Generational Garbage Collection

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

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

- Garbage in the young generation is collected *frequently* using mark-*copy*.
- Garbage in the old generation is collected *sporadically* using mark-*compact*.

Routing

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

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

Criteria for good routing algorithms:

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

Chandy-Misra Algorithm

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

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

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

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

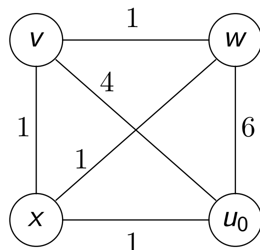
When a node v receives $\langle d \rangle$ from a neighbor w,
and if $d + \omega_{vw} < dist_v(u_0)$, then:

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

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

Chandy-Misra Algorithm - Example

$dist_{u_0}$	$:=$	0	$next_{u_0}$	$:=$	$\perp$
$dist_w$	$:=$	6	$next_w$	$:=$	u_0
$dist_v$	$:=$	7	$next_v$	$:=$	w
$dist_x$	$:=$	8	$next_x$	$:=$	v
$dist_x$	$:=$	7	$next_x$	$:=$	w
$dist_v$	$:=$	4	$next_v$	$:=$	u_0
$dist_w$	$:=$	5	$next_w$	$:=$	v
$dist_x$	$:=$	6	$next_x$	$:=$	w
$dist_x$	$:=$	5	$next_x$	$:=$	v
$dist_x$	$:=$	1	$next_x$	$:=$	u_0
$dist_w$	$:=$	2	$next_w$	$:=$	x
$dist_v$	$:=$	3	$next_v$	$:=$	w
$dist_v$	$:=$	2	$next_v$	$:=$	x



Chandy-Misra Algorithm - Complexity

Worst-case message complexity: Exponential

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

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

Toueg's Algorithm

We compute for *each* pair u, v a shortest path from u to v.

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

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

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

Floyd-Warshall Algorithm

We first discuss a *uniprocessor* algorithm.

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

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

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

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

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

Question

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

Answer:

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

Toueg's Algorithm

Assumption: Each node knows from the start the id's of all nodes.
(Because pivots must be picked uniformly at all nodes.)

Initially, at each node u:

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

Toueg's Algorithm

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

If $next_u(w) = \perp$ for a node $u \neq w$ at the w-pivot round,
then $dist_u(w) = \infty$, so $dist_u(w) + dist_w(v) \geq dist_u(v)$ for all nodes v.
Hence the sink tree of w can be used to broadcast $dist_w$.

Each node u sends:

- $\langle yes, w\rangle$ to $next_u(w)$ if it isn't $\perp$, to let it pass on $dist_w$ to u.
- $\langle no, w\rangle$ to its other neighbors, so that they don't send $dist_w$ to u.

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

Toueg's Algorithm

If u is in the sink tree of w, it waits until it has received:

- $\langle yes/no, w \rangle$ from each neighbor, and
- if $u \neq w$, the values $dist_w(v)$ from $next_u(w)$.

u forwards the values $dist_w(v)$ to the neighbors that sent $\langle yes, w \rangle$.

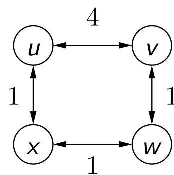
u checks for each node v whether

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

If so, $dist_u(v) := dist_u(w) + dist_w(u)$ and $next_u(v) := next_u(w)$.

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

Toueg's Algorithm - Example



pivot u	$dist_x(v)$	$:=$	5	$dist_v(x)$	$:=$	5
	$next_x(v)$	$:=$	u	$next_v(x)$	$:=$	u
pivot v	$dist_u(w)$	$:=$	5	$dist_w(u)$	$:=$	5
	$next_u(w)$	$:=$	v	$next_w(u)$	$:=$	v
pivot w	$dist_x(v)$	$:=$	2	$dist_v(x)$	$:=$	2
	$next_x(v)$	$:=$	w	$next_v(x)$	$:=$	w
pivot x	$dist_u(w)$	$:=$	2	$dist_w(u)$	$:=$	2
	$next_u(w)$	$:=$	x	$next_w(u)$	$:=$	x
	$dist_u(v)$	$:=$	3	$dist_v(u)$	$:=$	3
	$next_u(v)$	$:=$	x	$next_v(u)$	$:=$	w

Toueg's Algorithm - Complexity + Drawbacks

Message complexity: $\Theta(N{\cdot}E)$

Drawbacks:

- uniform selection of pivots requires that
  all nodes know the nodes in the network in advance
- global broadcast of $dist_w$ at the w-pivot round
- not robust with respect to topology changes

Toueg's Algorithm - Optimization

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

If $dist_u(v)$ isn't changed in this round, then neither is $dist_x(v)$.

Upon reception of $dist_w$, u can therefore first update $dist_u$ and $next_u$, and only forward values $dist_w(v)$ for which $dist_u(v)$ has changed.

Merlin-Segall Algorithm

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

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

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

1. Let a node v get $\langle d \rangle$ from a neighbor w.
   If $d+\omega_{vw} < dist_v(u_0)$, then $dist_v(u_0) := d+\omega_{vw}$
   (and v stores w as future value for $next_v(u_0)$).
   If $w = next_v(u_0)$, then v sends $\langle dist_v(u_0) \rangle$
   to its neighbors *except* $next_v(u_0)$.

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

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

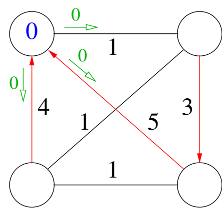
Merlin-Segall Algorithm - Termination + Complexity

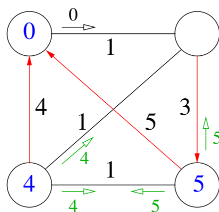
After i rounds, all shortest paths of $\leq i$ hops have been computed.
The algorithm terminates after $N-1$ rounds.

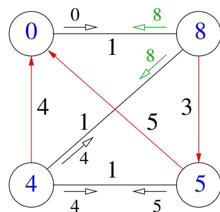
Message complexity: $\Theta(N^2 \cdot E)$

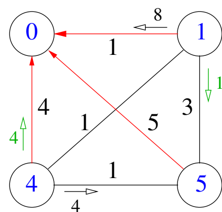
For each root, the algorithm requires $\Theta(N \cdot E)$ messages.

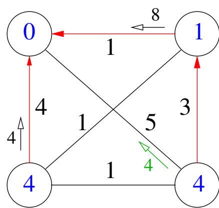
Merlin-Segall Algorithm - Example (Round 1)



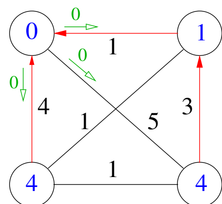


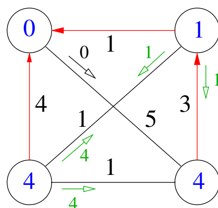


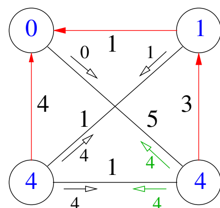


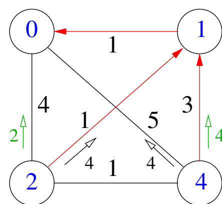


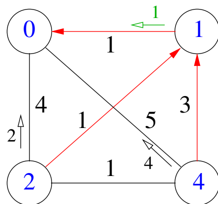
Merlin-Segall Algorithm - Example (Round 2)



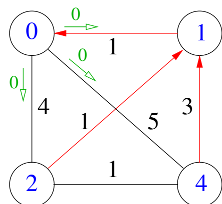


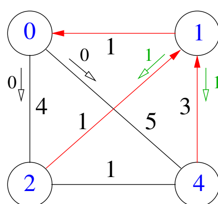


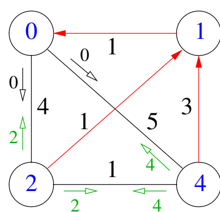


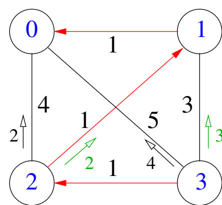


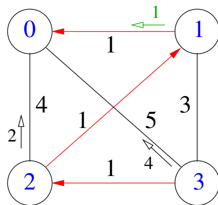
Merlin-Segall Algorithm - Example (Round 3)

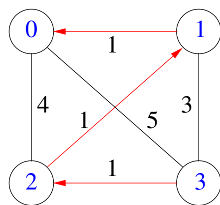












Merlin-Segall Algorithm - Topology Changes

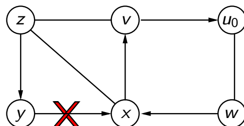
A number is attached to distance messages.

When an edge fails or becomes operational,
adjacent nodes send a message to u_0 via the sink tree.
(If the message meets a failed tree link, it is discarded.)

If the failed edge is part of the sink tree,
the remaining tree is extended to a sink tree.

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

Example:



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

Link-State Routing for Dynamic Topologies

Each node periodically sends out a message, reporting

- the node's edges and their weights (based on latency, bandwidth)
- a sequence number
  (increased with each broadcast of such messages)

This message is flooded through the network.

Nodes store information of such messages,
so that they obtain a local view of the entire network.

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

The sequence numbers avoid that new information is overwritten
by old information.

Routing on the Internet

Link-state routing underlies the OSPF protocol for routing on the Internet.

However, link-state routing doesn't scale up to the size of the Internet, because it uses flooding.

Therefore the Internet is divided into "Autonomous Systems", each of which uses link-state routing.

Routing between Autonomous Systems is performed with the Border Gateway Protocol.

Breadth-First Search

Consider an undirected, unweighted network.

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

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

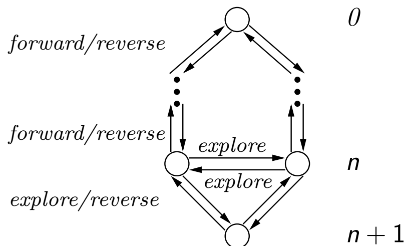
The following algorithm requires only $O(N{\cdot}\sqrt{E})$ messages (for each root).

Breadth-First Search - A “Simple” Algorithm

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

After round $n \geq 0$, each node at n hops from the root knows
(1) it is at distance n, and (2) which neighbors are at distance $n{-}1$.

We explain what happens in round $n{+}1$:



Breadth-First Search - A "Simple" Algorithm

- Messages $\langle \textbf{forward}, n \rangle$ travel down the tree, from the initiator to nodes at distance n.
- A node at distance n sends $\langle \textbf{explore}, n{+}1 \rangle$ to its neighbors that aren't at distance $n{-}1$.
- If a node v with $dist_v = \infty$ gets $\langle \textbf{explore}, n{+}1 \rangle$, $dist_v := n{+}1$, and the sender becomes its parent. It sends back $\langle \textbf{reverse}, true \rangle$.
- If a node v with $dist_v = n{+}1$ gets $\langle \textbf{explore}, n{+}1 \rangle$, it stores that the sender is at distance n. It sends back $\langle \textbf{reverse}, false \rangle$.
- If a node v with $dist_v = n$ gets $\langle \textbf{explore}, n{+}1 \rangle$, it takes this as a negative ack for the $\langle \textbf{explore}, n{+}1 \rangle$ that v sent into this edge.

Breadth-First Search - A “Simple” Algorithm

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

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

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

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

  Else, it continues with round $n{+}2$; nodes only send a **forward** to children that reported new nodes in round $n{+}1$.

Breadth-First Search - Complexity

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

There are at most N rounds.

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

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

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

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

Frederickson's Algorithm

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

Initially, the initiator is at distance 0, all other nodes at distance ∞.

After round n, each node at $k \leq \ell n$ hops from the root knows (1) it is at distance k, and (2) which neighbors are at distance $k-1$.

In round $n+1$:

- $\langle$**forward**$, \ell n\rangle$ travels down the tree, from the initiator to nodes at distance ℓn.
- Processes at distance ℓn send $\langle$**explore**$, \ell n+1, \ell\rangle$ to neighbors that aren't at distance $\ell n-1$.

Frederickson's Algorithm

Complication 1: A node may send multiple **explore**'s into an edge.

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

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

Solution: Such messages can be purged.

Frederickson's Algorithm

Let a node v receive $\langle \textbf{explore}, k, m \rangle$.

- Suppose $k < dist_v$:

  $dist_v := k$, and the sender becomes v's parent.

  If $m > 1$, v sends $\langle \textbf{explore}, k{+}1, m{-}1 \rangle$ to its other neighbors.

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

- Suppose $k \geq dist_v$:

  If $k > dist_v$ or $m > 1$, v doesn't send a reply
  (because v sent $\langle \textbf{explore}, dist_v + 1, _ \rangle$ into this edge).

  If $k = dist_v$ and $m = 1$, v sends back $\langle \textbf{reverse}, k, false \rangle$.

Frederickson's Algorithm

- A node at distance $\ell n < k < \ell(n{+}1)$ waits until it has received $\langle \textbf{reverse}, k{+}1, _\rangle$ or $\langle \textbf{explore}, j, _\rangle$ with $j \in \{k, k{+}1, k{+}2\}$ from all neighbors.

  Then it sends $\langle \textbf{reverse}, k, \mathit{true}\rangle$ to its parent.

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

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

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

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

  Else, it continues with round $n{+}2$; nodes only send a **forward** to children that reported new nodes in round $n{+}1$.

Frederickson's Algorithm - Complexity

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

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

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

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

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

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

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

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

Deadlock-Free Packet Switching

Consider a directed network, supplied with routing tables.

Processes store data packets traveling to their destination in buffers.

Possible events:

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

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

Store-and-Forward Deadlocks

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

For simplicity we assume synchronous communication.

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

Destination Controller

$V = \{u_0, \ldots, u_{N-1}\}$, and T_i denotes the sink tree (with respect to the routing tables) with root u_i for $0 = 1, \ldots, N-1$.

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

- When a packet with destination u_i is generated at v, it is placed in the ith buffer place of v.
- If vw is an edge in T_i, then the ith buffer place of v is linked to the ith buffer place of w.

Destination Controller - Correctness

Theorem: The destination controller is deadlock-free.

Proof: Consider a reachable configuration γ.
Make forwarding and consumption transitions to a configuration δ where no forwarding or consumption is possible.

For each i, since T_i is acyclic, packets in an ith buffer place can travel to their destination.

So in δ, all buffers are empty.

Hops-So-Far Controller

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

In the hops-so-far controller, each node carries $K+1$ buffer places.

- When a packet is generated at v, it is placed in the 0th buffer place of v.
- If vw is an edge in some T_i, then each jth buffer place of v is linked to the $(j+1)$th buffer place of w.

Hops-So-Far Controller - Correctness

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

Proof: Consider a reachable configuration γ.
Make forwarding and consumption transitions to a configuration δ where no forwarding or consumption is possible.

Packets in a Kth buffer place are at their destination.
So in δ, Kth buffer places are all empty.

Suppose all ith buffer places are empty in δ, for some $1 \leq i \leq K$.
Then all $(i-1)$th buffer places must also be empty in δ.
For else some packet in an $(i-1)$th buffer place could be forwarded or consumed.

Concluding, in δ all buffers are empty.

Acyclic Orientation Cover

Consider an undirected network.

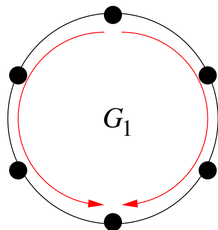
An acyclic orientation is a directed, acyclic network, obtained by directing all edges.

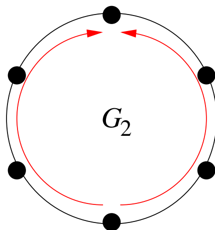
Acyclic orientations $G_0, \ldots, G_{n-1}$ are an acyclic orientation cover of a set $\mathcal{P}$ of paths in the network if each path in $\mathcal{P}$ is the concatenation of paths $P_0, \ldots, P_{n-1}$ in $G_0, \ldots, G_{n-1}$.

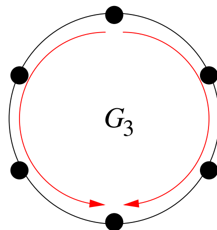
Acyclic Orientation Cover - Example

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

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







Acyclic Orientation Cover Controller

Given an acyclic orientation cover $G_0, \ldots, G_{n-1}$ of a set $\mathcal{P}$ of paths.

In the acyclic orientation cover controller, each node has n buffer places, numbered from 0 to $n-1$.

- A packet generated at v, is placed in the 0th buffer place of v.
- Let vw be an edge in G_i.
  The ith buffer place of v is linked to the ith buffer place of w, and if $i < n{-}1$, the ith buffer place of w is linked to the $(i{+}1)$th buffer place of v.

Acyclic Orientation Cover Controller - Correctness

Theorem: If all packets are routed via paths in $\mathcal{P}$,
the acyclic orientation cover controller is deadlock-free.

Proof: Consider a reachable configuration γ.
Make forwarding and consumption transitions to a configuration δ
where no forwarding or consumption is possible.

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

Suppose all ith buffer places are empty in δ, for some $i = 1, \ldots, n-1$.
Then all $(i-1)$th buffer places must also be empty in δ.
For else, since G_{i-1} is acyclic, some packet in an $(i-1)$th buffer place could be forwarded or consumed.

Concluding, in δ all buffers are empty.

Acyclic Orientation Cover Controller - Example

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

Slow-Start Algorithm in TCP

Back to the asynchronous, pragmatic world of the Internet.

To control congestion in the network, in TCP, nodes maintain a congestion window for each outgoing edge. It provides an upper bound on the number of unacknowledged packets a node can have on this edge.

The congestion window grows linearly with each received ack, up to some threshold. The congestion window may effectively double with every "round trip time".

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

Election Algorithms

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

Assumptions:

- All processes have the *same local algorithm*.
- Process *id's* are *unique*, and from a *totally ordered set*.
- The algorithm is *decentralized*:
  The initiators can be any non-empty set of processes.

Chang-Roberts Algorithm

Consider a directed ring.

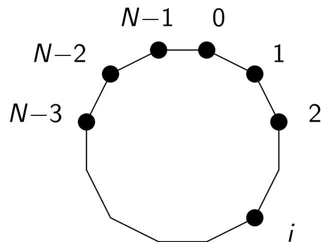
- Each initiator sends a message, tagged with its id.
- When p receives q with $q < p$, it *purges* the message.
- When p receives q with $q > p$, it becomes passive, and *passes on* the message.
- When p receives p, it becomes the leader.

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

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

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

Chang-Roberts Algorithm - Example



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

clockwise: $2N-1$ messages

Franklin's Algorithm

Consider an undirected ring.

Each *active* process compares its id with the id's of its nearest *active* neighbors on both sides.
If its id isn't the largest, it becomes *passive*.

- Initially, initiators are active, and non-initiators are passive.
  Each active process sends its id to its neighbors on either side.
- Let active process p receive q and r:

- if $\max\{q, r\} < p$, then p *sends its id again*
- if $\max\{q, r\} > p$, then p becomes passive
- if $\max\{q, r\} = p$, then p becomes the leader

Passive processes pass on incoming messages.

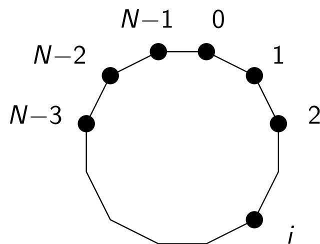
Franklin's Algorithm - Complexity

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

In each round, at least half of the active processes become passive.
So there are at most $\log_2 N$ rounds.

Each round takes $2N$ messages.

Franklin's Algorithm - Example



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

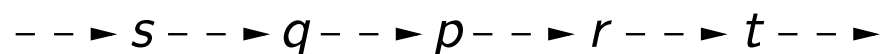
after 2 rounds node $N-1$ is the leader

Remark: Suppose the ring is directed with a clockwise orientation. If a process would only compare its id with the one of its predecessor, then this ring would take N rounds to complete.

Dolev-Klawe-Rodeh Algorithm

Consider a directed ring.

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



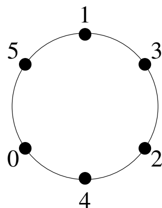
- If $\max\{q, r\} < p$, then r changes its id to p, and sends out p.
- If $\max\{q, r\} > p$, then r becomes passive.
- If $\max\{q, r\} = p$, then r announces this id to all processes.

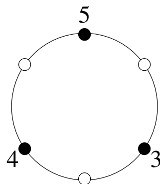
  The process that originally had the id p becomes the leader.

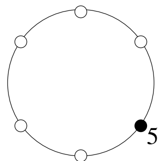
Worst-case message complexity: $O(N \log N)$

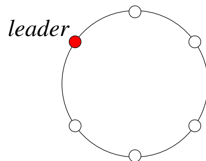
Dolev-Klawe-Rodeh Algorithm - Example

Consider the following clockwise oriented ring.









Election in Acyclic Networks

Given an undirected, acyclic network.

The tree algorithm can be used as an election algorithm
for undirected, acyclic networks:

Let the two *deciding* processes determine which one of them
becomes the leader (e.g. the one with the largest id).

Question: How can the tree algorithm be used to make
the process with the largest id in the network the leader ?

Tree Election Algorithm - Wake-up Phase

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

- initially, initiators send a wake-up message to all neighbors
- non-initiators send a wake-up message to all neighbors after they receive a first wake-up message
- a process wakes up when it has received wake-up messages from all neighbors

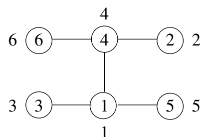
Tree Election Algorithm

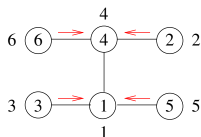
The local algorithm at an awake process p:

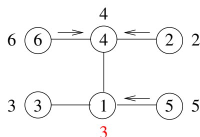
- p waits until it received id's from all neighbors except one, which becomes its parent
- p computes the largest id max_p among the received id's and its own id
- p sends max_p to its parent
- when p receives id q from its parent, it computes max'_p, being the maximum of max_p and q
- if $\mathrm{max}'_p = p$, then p becomes the leader
- p sends max'_p to all neighbors except its parent

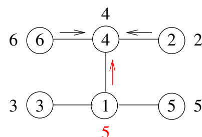
Message complexity: $2N-2$ messages

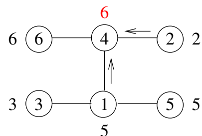
Tree Election Algorithm - Example

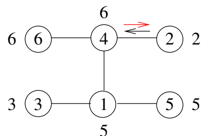


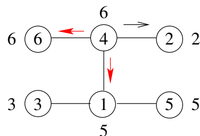


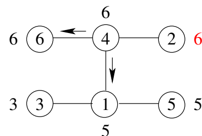


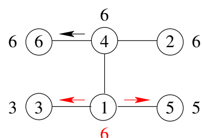


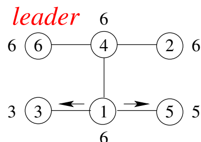


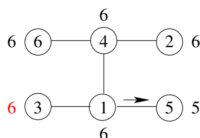


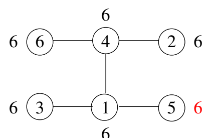












Echo Algorithm with Extinction

Election for undirected networks, based on the echo algorithm:

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

Non-initiators join the first wave that hits them.

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

Minimum Spanning Trees

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

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

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

Fragments

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

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

Then e is in M.

Proof: Suppose not.

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

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

Prim's Algorithm

Centralized. Initially, F is a single process.

As long as F isn't a spanning tree,
add the lowest-weight outgoing edge of F to F.

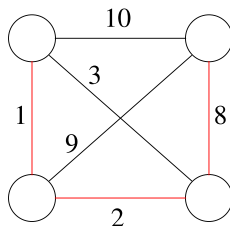
Remark: Prim's (and Kruskal's) algorithm also work when edges may have the same weight.
But then the minimum spanning tree isn't always unique.

Kruskal's Algorithm

Decentralized. Initially, each process forms a separate fragment.

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

Example:



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

Gallager-Humblet-Spira Algorithm

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

Distributed computation of a minimum spanning tree:

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

Level, name and core edge

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

Each fragment has a unique name. Its level is the maximum number of joins any process in the fragment has experienced.

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

$$L < L' \;\wedge\; F \xrightarrow{e_F} F': \quad F \cup F' = (FN', L')$$

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

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

Parameters of a process

Its *state*:

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

The *status* of its *channels*:

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

Name and *level* of its fragment.

Its *parent* (toward the core edge).

Initialization

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

Each *initiator*, and *non-initiator* after it has woken up:

- sets its level to 0
- sets its *lowest-weight* edge to branch,
  and sends $\langle$**connect**$, 0\rangle$ into it
- sets its other channels to basic
- sets its state to found

Joining two fragments

Let fragments $F = (FN, L)$ and $F' = (FN', L')$ be joined via channel pq.

- If $L < L'$, then p sent $\langle \textbf{connect}, L \rangle$ to q, and p sends $\langle \textbf{initiate}, FN', L', \frac{find}{found} \rangle$ to q.

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

- If $L = L'$, then p and q sent $\langle \textbf{connect}, L \rangle$ to each other, and send $\langle \textbf{initiate}, \text{weight } pq, L{+}1, find \rangle$ to each other.

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

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

It passes on the message through its other branch edges.

Computing the lowest-weight outgoing edge

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

1. If $L > level_q$, q *postpones* processing the incoming **test** message.
2. Else, if q is in fragment FN, q replies **reject**, after which p and q set pq to rejected.
3. Else, q replies **accept**.

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

Questions

In case 1, if $L > level_q$, why does q postpone processing the incoming **test** message from p?

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

Why does this postponement not lead to a deadlock?

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

Reporting to the core nodes

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

Termination or **changeroot** at the core nodes

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

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

When a process *p* that reported its lowest-weight outgoing basic edge receives **changeroot**, it sets this channel to branch, and sends $\langle$**connect**, $level_p\rangle$ into it.

Starting the join of two fragments

When a process q receives $\langle \textbf{connect}, level_p \rangle$ from p, $level_q \geq level_p$. Namely, either $level_p = 0$, or q earlier sent **accept** to p.

1. If $level_q > level_p$, then q sets qp to *branch* and sends $\langle \textbf{initiate}, name_q, level_q, \frac{find}{found} \rangle$ to p.

2. As long as $level_q = level_p$ and qp isn't a branch edge, q *postpones* processing the **connect** message.

3. If $level_q = level_p$ and qp is a branch edge (meaning that q sent $\langle \textbf{connect}, level_q \rangle$ to p), then q sends $\langle \textbf{initiate}, \text{weight } qp, level_q + 1, find \rangle$ to p (and vice versa).

In case 3, pq becomes the core edge.

Questions

In case 2, if $level_q = level_p$, why does q postpone processing the incoming **connect** message from p ?

Answer: The fragment of q might be in the process of joining a fragment at level $\geq level_q$, in which case the fragment of p should subsume the name and level of that joint fragment, instead of joining the fragment of q at an equal level.

Why does this postponement not give rise to a deadlock ?

Answer: Since different channels have different weights, there can't be a cycle of fragments waiting for a reply to a postponed **connect** message.

Question

Suppose a process reported a lowest-weight outgoing basic edge, and in return receives $\langle \textbf{initiate}, FN, L, \mathit{find} \rangle$.

Why must it test its lowest-weight outgoing basic edge anew ?

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

Gallager-Humblet-Spira Algorithm - Complexity

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

- Each channel is rejected at most once, by a **test-reject** pair.

Between two subsequent joins, each process in a fragment:

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

Each process experiences at most $\log_2 N$ joins
(because a fragment at level L contains $\geq 2^L$ processes).

Back to Election

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

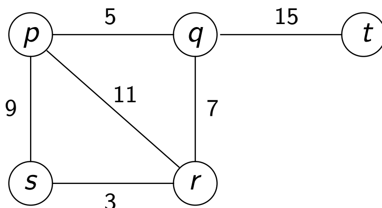
So Gallager-Humblet-Spira is an election algorithm
for undirected networks (with a total order on the channels).

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

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

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

Gallager-Humblet-Spira Algorithm - Example



pq qp	$\langle \textbf{connect}, 0\rangle$
pq qp	$\langle \textbf{initiate}, 5, 1, \mathit{find}\rangle$
ps qr	$\langle \textbf{test}, 5, 1\rangle$
tq	$\langle \textbf{connect}, 0\rangle$
qt	$\langle \textbf{initiate}, 5, 1, \mathit{find}\rangle$
tq	$\langle \textbf{report}, \infty\rangle$
rs sr	$\langle \textbf{connect}, 0\rangle$
rs sr	$\langle \textbf{initiate}, 3, 1, \mathit{find}\rangle$
sp rq	**accept**
pq	$\langle \textbf{report}, 9\rangle$
qp	$\langle \textbf{report}, 7\rangle$
qr	$\langle \textbf{connect}, 1\rangle$
sp rq	$\langle \textbf{test}, 3, 1\rangle$
ps qr	**accept**

rs	$\langle \textbf{report}, 7\rangle$
sr	$\langle \textbf{report}, 9\rangle$
rq	$\langle \textbf{connect}, 1\rangle$
rq qp qt qr rs	$\langle \textbf{initiate}, 7, 2, \mathit{find}\rangle$
ps sp	$\langle \textbf{test}, 7, 2\rangle$
ps sp	**reject**
pr	$\langle \textbf{test}, 7, 2\rangle$
rp	**reject**
tq pq qr sr rq	$\langle \textbf{report}, \infty\rangle$

Anonymous Networks

Processes may be anonymous for several reasons:

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

Assumptions:

- Processes (and channels) don't have a unique id.
- All processes have the same local algorithm.
- The initiators can be any non-empty set of processes.

If there is one leader, all processes can be given a unique id (using a traversal algorithm).

Impossibility of Election in Anonymous Rings

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

Proof: Consider an anonymous ring of size N.

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

- There is a symmetric initial configuration.
- If γ_0 is symmetric and $\gamma_0 \to \gamma_1$, then there is an execution $\gamma_1 \to \gamma_2 \to \cdots \to \gamma_N$ where γ_N is symmetric.

In a symmetric configuration there isn't one leader.

So there is an *infinite*, *fair* execution.

Probabilistic Algorithms

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

For a probabilistic algorithm where all computations terminate
in a *correct* configuration, letting the coin e.g. always flip heads
yields 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.

Questions

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

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

Give a Las Vegas algorithm that terminates with probability 1.

Itai-Rodeh Election Algorithm

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

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

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

Complication: Different processes may select the same id.

Solution: Each message is supplied with a hop count.
A message arrives at its source if and only if its hop count is N.

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

Itai-Rodeh Election Algorithm

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

Let p be active. At the start of an *election round* n, p selects a random id_p, sends $(n, id_p, 1, true)$, and waits for a message (n', i, h, b). The 3rd value is the hop count. The 4th value signals if another process with the same id was encountered during the round trip.

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

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

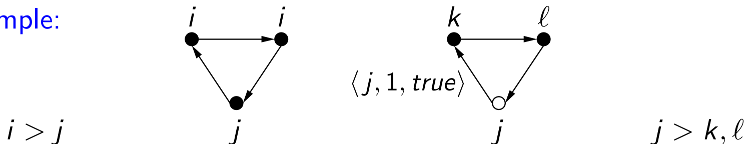
Itai-Rodeh Election Algorithm - Correctness + Complexity

Correctness: The Itai-Rodeh election algorithm is Las Vegas: eventually one leader is elected, with probability 1.

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

Without rounds, the algorithm would be flawed (for non-FIFO channels).

Example:



$i > j$ $\qquad$ $j > k, \ell$

Election in Arbitrary Anonymous Networks

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

Initially, initiators are active in round 0, and non-initiators are passive.

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

Suppose process p in wave i from round n is hit by wave j from round n':

- if $n' > n$, or $n' = n$ and $j > i$, then p changes to wave j from round n', and treats the message according to the echo algorithm
- if $n' < n$, or $n' = n$ and $j < i$, then p purges the message
- if $n' = n$ and $j = i$, then p treats the message according to the echo algorithm

Election in Arbitrary Anonymous Networks

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

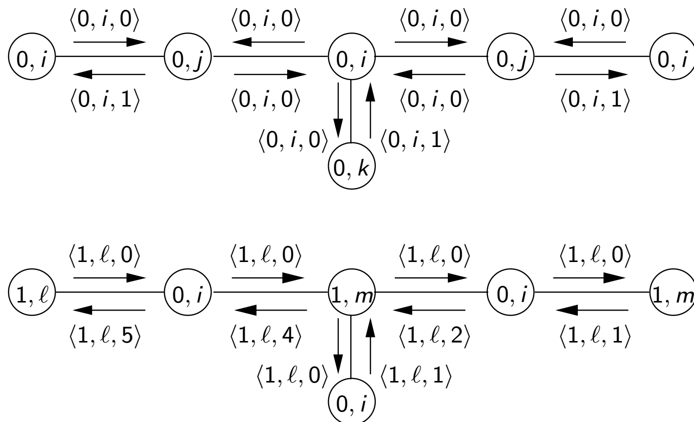
When a process *decides*, it computes the size of the constructed tree.

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

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

Election in Arbitrary Anonymous Networks - Example

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



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

Computing the Size of a Network

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

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

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

Impossibility of Computing an Anonymous Network Size

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

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

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

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

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

Itai-Rodeh Ring Size Algorithm

Each process p maintains an *estimate* est_p of the ring size.
Initially $est_p = 2$. (Always $est_p \leq N$.)

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

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

- 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 $id = id_p$, then p *purges* the message (possibly its own message returned).

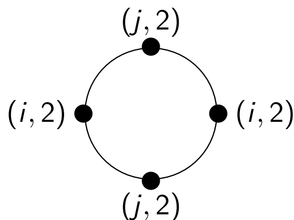
Itai-Rodeh Ring Size Algorithm: Termination + Correctness

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

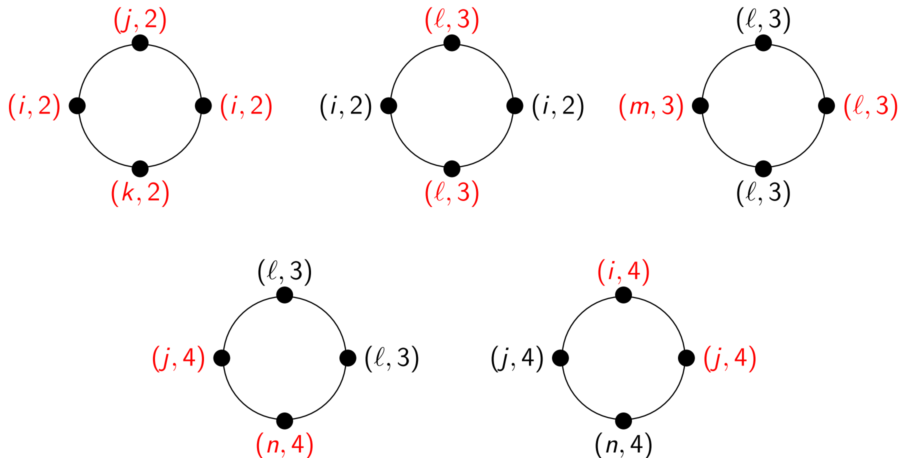
When the algorithm terminates, $est_p \leq N$ for all p.

The Itai-Rodeh ring size algorithm is a Monte Carlo algorithm.
Possibly, in the end $est_p < N$.

Example:



Itai-Rodeh Ring Size Algorithm - Example



Itai-Rodeh Ring Size Algorithm: Complexity

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

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.

Question

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

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

IEEE 1394 Election Algorithm

The IEEE 1394 standard is a serial multimedia bus.
It connects digital devices, which can be added/removed dynamically.

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

The network size is unknown to the processes.

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

IEEE 1394 Election Algorithm - Root Contention

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

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

Question: Is it optimal to give an equal chance of 0.5 to both sending immediately and waiting for some time ?

Fault Tolerance

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

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

So failing processes never make the remaining network disconnected.

Assumption: Crashing of processes can't be observed.

Consensus

Binary consensus: Correct processes must uniformly decide 0 or 1.

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

Validity rules out trivial solutions where e.g. all processes always decide 0.

Validity implies that there is a bivalent initial configuration, meaning that it can reach terminal configurations with a decision 0 as well as with a decision 1.

Impossibility of 1-Crash Consensus

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

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: No algorithm for 1-crash consensus always terminates.

Proof: Consider a 1-crash consensus algorithm.
Let γ be a reachable bivalent configuration. Then $\gamma \to \gamma_0$ and $\gamma \to \gamma_1$, where γ_0 can lead to decision 0 and γ_1 to decision 1.

- Suppose the transitions correspond to events at *different* processes.
  Then $\gamma_0 \to \delta \leftarrow \gamma_1$ for some δ. So γ_0 or γ_1 is bivalent.
- Suppose the transitions correspond to events at *one* 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 still b-potent.
  So γ_{1-b} is bivalent.

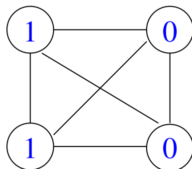
Concluding, each reachable bivalent configuration can make a transition to a bivalent configuration.
So bivalent initial configurations yield an infinite execution.

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

Impossibility of 1-Crash Consensus - Example

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

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



The left processes might all the time receive two 1-votes and one 0-vote, while the right processes receive two 0-votes and one 1-vote.

Question: Give a fair infinite execution.

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

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

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

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

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

Since there is a bivalent initial configuration,
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, however, can't exist.

Question

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

Answer: Let any process randomly choose 0 or 1.

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

Bracha-Toueg Crash Consensus Algorithm

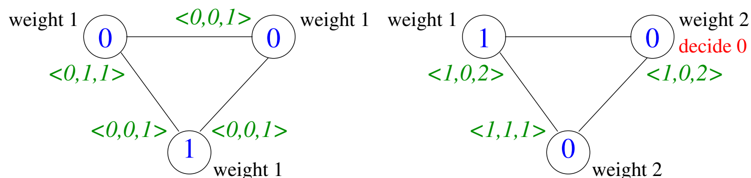
Let $k < \frac{N}{2}$. Initially, each correct process randomly chooses a value 0 or 1, with weight 1. In round n, at each correct, undecided p:

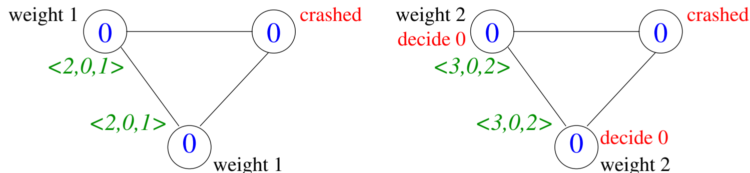
- p sends $\langle n, value_p, weight_p \rangle$ to all processes (including itself).
- p waits till $N{-}k$ messages $\langle n, b, w \rangle$ arrived.
  (p purges/stores messages from earlier/future rounds.)
  If $w > \frac{N}{2}$ for a $\langle n, b, w \rangle$, then $value_p := b$. *(This b is unique.)*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

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

Bracha-Toueg Crash Consensus Algorithm - Example

$N = 3$ and $k = 1$. Each round a correct process requires two incoming messages, and two b-votes with weight 2 to decide b. (Messages of a process to itself aren't depicted.)





Bracha-Toueg Crash Consensus Algorithm - Correctness

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

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

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

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

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

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

Concluding, all correct processes decide for the same value.

Bracha-Toueg Crash Consensus Algorithm - Correctness

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

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

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

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

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

Concluding, the algorithm terminates with probability 1.

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

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

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

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

In reachable configurations, S and T are either both 0-potent or both 1-potent. For else, since the processes in $S \cap T$ may be Byzantine, S and T could independently decide for different values.

Since there is a bivalent initial configuration,
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, however, can't exist.

Bracha-Toueg Byzantine Consensus Algorithm

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

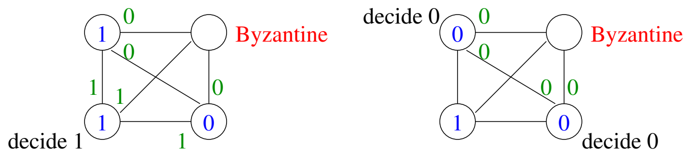
Again, in every round, correct processes broadcast their value, and wait for $N - k$ incoming messages. *(No weights are needed.)*

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

Echo Mechanism

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

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



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

Bracha-Toueg Byzantine Consensus Algorithm

Initially, each correct process randomly chooses 0 or 1.

In round n, at each correct, undecided p:

- p sends $\langle \textbf{vote}, n, value_p \rangle$ to all processes (including itself).
- If p receives $\langle \textbf{vote}, m, b \rangle$ from q,
  it sends $\langle \textbf{echo}, q, m, b \rangle$ to all processes (including itself).
- p stores $\langle \textbf{vote}, m, b \rangle$ and $\langle \textbf{echo}, q, m, b \rangle$ with $m > n$.
- p counts incoming $\langle \textbf{echo}, q, n, b \rangle$ messages for each q, b.
  When $> \frac{N+k}{2}$ such messages arrived, p *accepts* q's b-vote.
- The round is completed when p has accepted $N-k$ votes.
  If most votes are for 0, then $value_p := 0$. Else, $value_p := 1$.

Bracha-Toueg Byzantine Consensus Algorithm

p keeps track whether multiple messages $\langle \textbf{vote}, m, _ \rangle$ or $\langle \textbf{echo}, q, m, _ \rangle$ arrive via the same channel. (The sender must be Byzantine.)
p only takes into account the first of these messages.

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

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

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

Bracha-Toueg Byzantine Consensus Alg. - Example

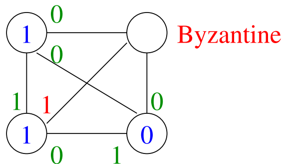
We study the previous example again, now with verification of votes.

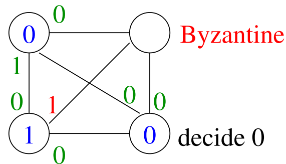
$N = 4$ and $k = 1$, so each round a correct process needs:

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

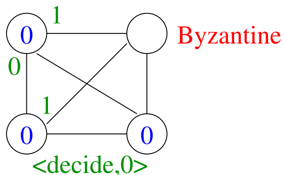
Only relevant **vote** messages are depicted (without their round number).

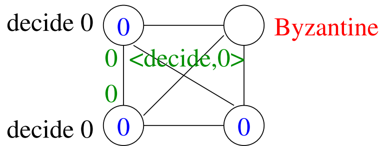
Bracha-Toueg Byzantine Consensus Alg. - Example





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





Bracha-Toueg Byzantine Consensus Alg. - Correctness

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

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

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

Then p and q received $> \frac{N+k}{2}$ messages $\langle \mathbf{echo}, r, n, b \rangle$ and $\langle \mathbf{echo}, r, n, b' \rangle$, respectively

More than k processes, so at least one correct process, sent such messages to both p and q. So $b = b'$.

Bracha-Toueg Byzantine Consensus Alg. - Correctness

Suppose a correct process decides b in round n.

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

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

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

So correct processes will vote b in all rounds $m > n$
(because they will accept $\geq N{-}2k > \frac{N-k}{2}$ b-votes).

Let S be a set of $N{-}k$ processes that aren't Byzantine.

Assuming fair scheduling, there is a chance $\rho > 0$ that in a round each process in S accepts $N{-}k$ votes from the processes in S.

With chance ρ^2 this happens in consecutive rounds $n, n{+}1$.

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

After round $n{+}1$, all processes in S have decided b.

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 latency, and heartbeat messages, 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 latency, 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, with a total order.

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

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

Assumption: Processes can't observe $F(\tau)$.

$H(q, \tau)$ is the set of processes that q *suspects* to be crashed at time τ.

Each execution is decorated with

- a failure pattern F
- a failure detector history H

Complete Failure Detector

We require that every failure detector is complete.

From some time onward, every crashed process is suspected by every correct process.

Strongly Accurate Failure Detector

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

Assumptions:

- Each correct process broadcasts **alive** every ν time units.
- $d_{\max}$ is a (known) upper bound on network latency.

Each process from which no message is received for $\nu + d_{\max}$ time units has crashed.

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

Weakly Accurate Failure Detector

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

Assume a complete and *weakly accurate* failure detector.

We give a rotating coordinator algorithm for $(N-1)$-crash consensus.

Consensus with Weakly Accurate Failure Detection

Processes are numbered: $p_0, \ldots, p_{N-1}$.

Initially, each process has value 0 or 1. In round n:

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

After round $N-1$, each correct process decides for its value.

Correctness: Let p_j never be suspected.

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

Hence, after round $N-1$, all correct processes decide b.

Eventually Strongly Accurate Failure Detector

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

Assumptions:

- Each correct process broadcasts **alive** every ν time units.
- There is an (unknown) upper bound on network latency.

Each process q initially guesses as network latency $\mu_q = 1$.

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

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

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

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

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

Proof: Suppose, toward a contradiction, there is such an algorithm. Split 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 $k \geq \frac{N}{2}$, S and T could independently decide for different values.

Since there is a bivalent initial configuration,
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, however, can't exist.

Chandra-Toueg Algorithm

A failure detector is eventually weakly accurate if
from some time onward some process is never suspected.

Let $k < \frac{N}{2}$. A complete and *eventually weakly accurate* failure detector can be used for k-crash consensus.

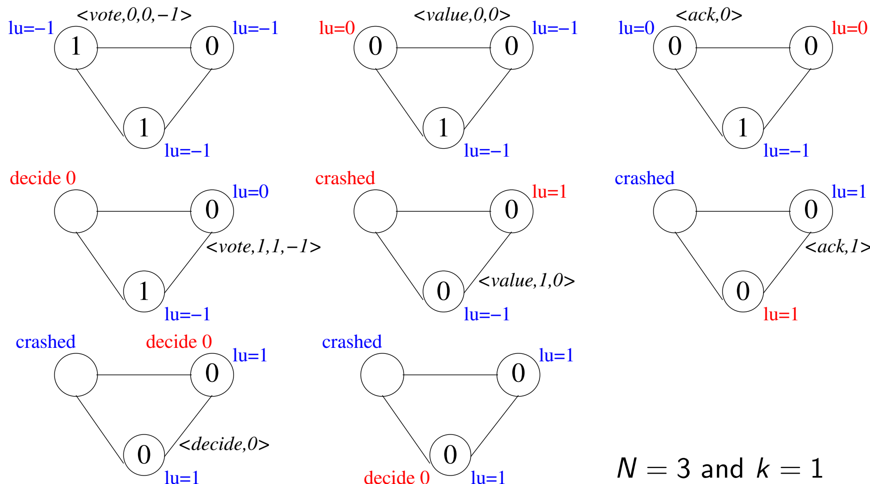
Each process q records the last round $\mathit{last\text{-}update}_q$ in which it updated value_q. Initially, $\mathit{value}_q \in \{0, 1\}$ and $\mathit{last\text{-}update}_q = -1$.

Processes are numbered: $p_0, \ldots, p_{N-1}$.
Round n is coordinated by p_c with $c = n \bmod N$.

Chandra-Toueg Algorithm

- In round n, each correct q sends $\langle \textbf{vote}, n, value_q, last\text{-}update_q \rangle$ to p_c.
- p_c (if not crashed) waits until $N{-}k$ such messages arrived, and selects one, say $\langle \textbf{vote}, n, b, \ell \rangle$, with ℓ as large as possible.
  $value_{p_c} := b$, $last\text{-}update_{p_c} := n$, and p_c broadcasts $\langle \textbf{value}, n, b \rangle$.
- Each correct q waits:
  - until $\langle \textbf{value}, n, b \rangle$ arrives: then $value_q := b$, $last\text{-}update_q := n$, and q sends $\langle \textbf{ack}, n \rangle$ to p_c;
  - or until it suspects p_c crashed: then q sends $\langle \textbf{nack}, n \rangle$ to p_c.
- p_c (if not crashed) waits until $N{-}k$ ack's and nack's have arrived.
  If $> k$ of them are **ack**, p_c decides b, and broadcasts $\langle \textbf{decide}, b \rangle$.
- An undecided process that receives $\langle \textbf{decide}, b \rangle$, decides b.

Chandra-Toueg Algorithm - Example



$N = 3$ and $k = 1$

Messages and ack's that a process sends to itself are omitted.

Chandra-Toueg Algorithm - Correctness

Theorem: Let $k < \frac{N}{2}$. The Chandra-Toueg algorithm is an (always terminating) algorithm for k-crash consensus.

Proof (part I): If the coordinator in some round n receives $> k$ **ack**'s, then (for some $b \in \{0,1\}$):

(1) there are $> k$ processes q with $\mathit{last\text{-}update}_q \geq n$, and

(2) $\mathit{last\text{-}update}_q \geq n$ implies $\mathit{value}_q = b$.

Properties (1) and (2) are preserved in rounds $m > n$.

This follows by induction on $m-n$.

By (1), in round m the coordinator receives at least one message with $\mathit{last\text{-}update} \geq n$.

Hence, by (2), the coordinator of round m sets its value to b, and broadcasts $\langle m, b \rangle$.

So from round n onward, processes can only decide b.

Chandra-Toueg Algorithm - Correctness

Proof (part II):

Since the failure detector is eventually weakly accurate,
from some round onward, some process p will never be suspected.

So when p becomes the coordinator, it receives $N-k$ **ack**'s.

Since $N-k > k$, 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 ?

Answer: Failure detectors are usually based on the *absence* of events.

Local Clocks with Bounded Drift

Let's forget about Byzantine processes for a moment.

The time domain is $\mathbb{R}_{\geq 0}$.

Each process p has a local clock $C_p(\tau)$, which returns a time value at *real* time τ.

Let local clocks have bounded drift, compared to real time.
If C_p isn't adjusted between times τ_1 and τ_2, then

$$\frac{1}{1+\rho}(\tau_2 - \tau_1) \ \leq \ C_p(\tau_2) - C_p(\tau_1) \ \leq \ (1+\rho)(\tau_2 - \tau_1)$$

for some (known) $\rho > 0$.

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 $\delta > 0$, and for all τ,

$$|C_p(\tau) - C_q(\tau)| \ \leq \ \delta$$

Due to drift, this precision may degrade over time, necessitating repeated clock synchronizations.

We assume a (known) bound $d_{\max}$ on network latency.

For simplicity, let $d_{\max}$ be much smaller than δ,
so that this latency can be ignored in the clock synchronization.

Clock Synchronization

Suppose that after each synchronization, at say real time τ, for all processes p, q:

$$|C_p(\tau) - C_q(\tau)| \ \leq \ \delta_0$$

for some $\delta_0 < \delta$.

Due to ρ-bounded drift of local clocks, at real time $\tau{+}R$,

$$|C_p(\tau{+}R) - C_q(\tau{+}R)| \ \leq \ \delta_0 + (1{+}\rho - \frac{1}{1{+}\rho})R \ < \ \delta_0 + 2\rho R$$

So there should be a synchronization every $\frac{\delta-\delta_0}{2\rho}$ time units.

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

Theorem: Let $k \geq \frac{N}{3}$. There is no k-Byzantine clock synchronizer.

Proof: Let $N = 3$, $k = 1$. Processes are p, q, r; r is Byzantine.
(The construction below easily extends to general N and $k \geq \frac{N}{3}$.)

Let the clock of p run faster than the clock of q.

Suppose a synchronization takes place at real time τ.
r sends $C_p(\tau) + \delta$ to p, and $C_q(\tau) - \delta$ to q.

p and q can't recognize that r is Byzantine.
So they have to stay within range δ of the value reported by r.
Hence p can't decrease, and q can't increase its clock value.

By repeating this scenario at each synchronization round,
the clock values of p and q get further and further apart.

Mahaney-Schneider Synchronizer

Consider a complete network of N processes, in which at most $k < \frac{N}{3}$ processes can be Byzantine.

Each correct process in a synchronization round:

1. Collects the clock values of all processes (waiting for $2d_{\max}$).
2. Discards those reported values τ for which $< N-k$ processes report a value in the interval $[\tau-\delta, \tau+\delta]$
(they are from Byzantine processes).
3. Replaces all discarded/non-received values by an accepted value.
4. Takes the average of these N values as its new clock value.

Mahaney-Schneider Synchronizer - Correctness

Lemma: If in some synchronization round values a_p and a_q pass the filters of correct processes p and q, respectively, then

$$|a_p - a_q| \leq 2\delta$$

Proof: $\geq N-k$ processes reported a value in $[a_p-\delta, a_p+\delta]$ to p.
And $\geq N-k$ processes reported a value in $[a_q-\delta, a_q+\delta]$ to q.

Since $N-2k > k$, at least one correct process r reported a value in $[a_p-\delta, a_p+\delta]$ to p, and in $[a_q-\delta, a_q+\delta]$ to q.

Since r reports the same value to p and q, it follows that

$$|a_p - a_q| \leq 2\delta$$

Mahaney-Schneider Synchronizer - Correctness

Theorem: The Mahaney-Schneider synchronizer is k-Byzantine for all $k < \frac{N}{3}$.

Proof: Let a_{pr} (resp. a_{qr}) be the value that correct process p (resp. q) accepted from or assigned to process r, in some synchronization round.

By the lemma, for all r, $|a_{pr} - a_{qr}| \leq 2\delta$.

Moreover, $a_{pr} = a_{qr}$ for all *correct* r.

Hence, for all correct p and q,

$$|\frac{1}{N}(\sum_{\text{processes } r} a_{pr}) - \frac{1}{N}(\sum_{\text{processes } r} a_{qr})| \;\leq\; \frac{1}{N}k2\delta \;<\; \frac{2}{3}\delta$$

So we can take $\delta_0 = \frac{2}{3}\delta$.

There should be a synchronization every $\frac{\delta}{6\rho}$ time units.

Synchronous Networks

Let's again forget about Byzantine processes for a moment.

A synchronous network proceeds in pulses. In one pulse, each process:

1. sends messages
2. receives messages
3. performs internal events

A message is sent and received in the same pulse.

Such synchrony is called lockstep.

Building a Synchronous Network

Assume ρ-bounded local clocks with precision δ.

For simplicity, we ignore the network latency.

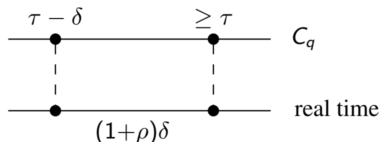
When a process reads clock value $(i{-}1)(1{+}\rho)^2\delta$, it starts pulse i.

Key question: Does a process p receive all messages for pulse i before it starts pulse $i{+}1$? That is, for all q,

$$C_q^{-1}((i{-}1)(1{+}\rho)^2\delta) \;\leq\; C_p^{-1}(i(1{+}\rho)^2\delta)$$

Because then q starts pulse i no later than p starts pulse $i{+}1$.

Building a Synchronous Network



Since the clock of q is ρ-bounded from below,

$$C_q^{-1}(\tau) \leq C_q^{-1}(\tau-\delta) + (1+\rho)\delta$$

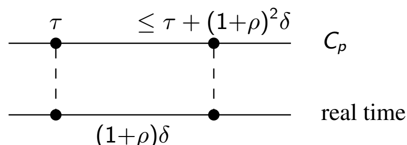
Since local clocks have precision δ, $\tau-\delta \leq C_q(C_p^{-1}(\tau))$, so

$$C_q^{-1}(\tau-\delta) \leq C_p^{-1}(\tau)$$

Hence, for all τ,

$$C_q^{-1}(\tau) \;\leq\; C_p^{-1}(\tau) + (1+\rho)\delta$$

Building a Synchronous Network



Since the clock of p is ρ-bounded from above,

$$C_p^{-1}(\tau) + (1+\rho)\delta \;\leq\; C_p^{-1}(\tau + (1+\rho)^2\delta)$$

Hence,

$$\begin{aligned} C_q^{-1}((i-1)(1+\rho)^2\delta) &\leq C_p^{-1}((i-1)(1+\rho)^2\delta) + (1+\rho)\delta \\ &\leq C_p^{-1}(i(1+\rho)^2\delta) \end{aligned}$$

Byzantine Broadcast

Consider a synchronous network of N processes, in which at most $k < \frac{N}{3}$ processes can be Byzantine.

One process g, called the general, is given an input $x_g \in \{0, 1\}$.

The other processes, called lieutenants, know who is the general.

Requirements for k-Byzantine broadcast:

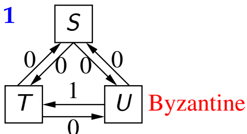
- Termination: Every correct process decides 0 or 1.
- Agreement: All correct processes decide the same value.
- Dependence: If the general is correct, it decides x_g.

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

Theorem: Let $k \geq \frac{N}{3}$. There is no k-Byzantine broadcast algorithm for synchronous networks.

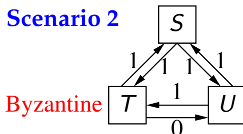
Proof: Divide the processes into three sets S, T and U with each $\leq k$ elements. Let $g \in S$.

Scenario 1



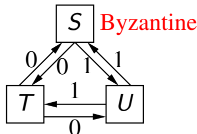
The processes in S and T decide 0

Scenario 2



The processes in S and U decide 1

Scenario 3



The processes in T decide 0 and in U decide 1

Lamport-Shostak-Pease Broadcast Algorithm

$Broadcast_g(N, k)$ is a k-Byzantine broadcast algorithm for a synchronous network of size N, with $k < \frac{N}{3}$.

Pulse 1: The general g decides and broadcasts x_g.

If a lieutenant p receives b from g then $x_p := b$, else $x_p := 0$.

If $k = 0$: each lieutenant p decides x_p.

If $k > 0$: for each lieutenant q, all lieutenants take part in $Broadcast_q(N-1, k-1)$ in pulse 2 (g is excluded).

Pulse $k+1$: $(k > 0)$

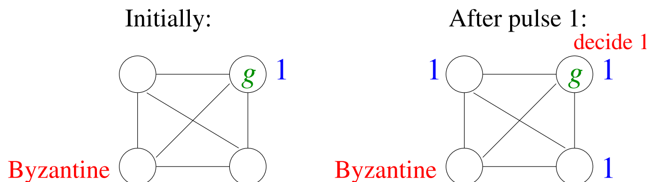
Lieutenant p has, for each lieutenant q, computed a value in $Broadcast_q(N-1, k-1)$; store these values in $M_p[q]$.

$x_p := major(M_p)$; lieutenant p decides x_p.

(*major* maps each *multiset* over $\{0, 1\}$ to 0 or 1, such that if more than half of the elements in M are b, then $major(M) = b$.)

Lamport-Shostak-Pease Broadcast Alg. - Example 1

$N = 4$ and $k = 1$; general correct.



g decides 1. Consider the sub-network without g.

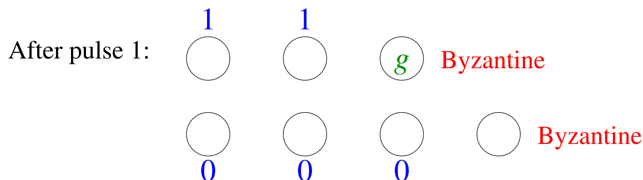
After pulse 1, all correct processes carry the value 1.

The two correct lieutenants build a multiset $\{1, 1, _\}$, and also decide 1.

Although one third of the lieutenants is Byzantine, the correct ones make the right decision, as they all carry the value 1, and $k < \frac{N-1}{2}$.

Lamport-Shostak-Pease Broadcast Alg. - Example 2

$N = 7$ and $k = 2$; general Byzantine. (Channels are omitted.)



Since $k-1 < \frac{N-1}{3}$, by induction, all correct lieutenants p build, in the recursive call $Broadcast_p(6,1)$, the same multiset $M = \{1,1,0,0,0,b\}$, for some $b \in \{0,1\}$.

So in $Broadcast_g(7,2)$, they all decide $major(M)$.

Lamport-Shostak-Pease Broadcast Alg. - Correctness

Lemma: If the general g is correct, and $f < \frac{N-k}{2}$
(with f the number of Byzantine processes; $f > k$ is allowed here),
then in $Broadcast_g(N, k)$ all correct processes decide x_g.

Proof: By induction on k. Case $k = 0$ is trivial, because g is correct.

Let $k > 0$.

Since g is correct, in pulse 1, at all correct lieutenants q, $x_q := x_g$.

Since $f < \frac{(N-1)-(k-1)}{2}$, by induction, for all correct lieutenants q,
in $Broadcast_q(N-1, k-1)$ the value $x_q = x_g$ is computed.

Since a majority of the lieutenants is correct ($f < \frac{N-1}{2}$), in pulse $k+1$,
at each correct lieutenant p, $x_p := major(M_p) = x_g$.

Lamport-Shostak-Pease Broadcast Alg. - Correctness

Theorem: Let $k < \frac{N}{3}$. $Broadcast_g(N, k)$ is a k-Byzantine broadcast algorithm for synchronous networks.

Proof: By induction on k.

If g is correct, then consensus follows from the lemma and $k < \frac{N}{3}$.

Let g be Byzantine (so $k > 0$). Then $\leq k-1$ lieutenants are Byzantine.

Since $k-1 < \frac{N-1}{3}$, by induction, for every lieutenant q, all correct lieutenants take in $Broadcast_q(N-1, k-1)$ the *same* decision b_q.

Hence, all correct lieutenants compute the *same* multiset M.

So in pulse $k+1$, all correct lieutenants decide the *same* value $major(M)$.

Partial Synchrony

A synchronous system can be obtained if local clocks have known *bounded drift*, and there is a known *upper bound on network latency*.

In a partially synchronous systems,

- the bounds on the inaccuracy of local clocks and network latency are unknown, or
- these bounds are known, but only valid from some unknown point in time.

Dwork, Lynch and Stockmeyer showed that, for $k < \frac{N}{3}$, there is a k-Byzantine broadcast algorithm for partially synchronous systems.

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

$$S_q(P_q(m)) = P_q(S_q(m)) = m \qquad \text{for all } m \in \mathcal{M}.$$

S_q is kept *secret*, P_q is made *public*.

Underlying assumption: Computing S_q from P_q is expensive.

p sends a *secret* message m to q: $P_q(m)$

p sends a *signed* message m to q: $\langle m, S_p(m) \rangle$

We use a public-key cryptosystem so that the (at most k) Byzantine processes can't lie about the messages they have received.

Lamport-Shostak-Pease Authentication Algorithm

Pulse 1: The general broadcasts $\langle x_g, (S_g(x_g), g)\rangle$, and decides x_g.

Pulse i: If a lieutenant q receives a message $\langle v, (\sigma_1, p_1) : \cdots : (\sigma_i, p_i)\rangle$ that is valid, i.e.:

- $p_1 = g$
- $p_1, \ldots, p_i, q$ are distinct
- $P_{p_k}(\sigma_k) = v$ for $k = 1, \ldots, i$

then q includes v in the set W_q.

If $i \leq k$, then in pulse $i+1$, q sends to all other lieutenants

$$\langle v, (\sigma_1, p_1) : \cdots : (\sigma_i, p_i) : (S_q(v), q)\rangle$$

After pulse $k+1$, each correct lieutenant p decides

v	if W_p is a singleton $\{v\}$, or
0	otherwise (the general is Byzantine)

Lamport-Shostak-Pease Authentication Alg. - Correctness

Theorem: The Lamport-Shostak-Pease authentication algorithm is a k-Byzantine broadcast algorithm, for any k.

Proof: If the general is correct, then owing to authentication, correct lieutenants q only add x_g to W_q. So they all decide x_g.

Suppose a correct lieutenant receives a valid message $\langle v, \ell \rangle$ in a pulse $\leq k$. Then in the next pulse,
it makes all correct lieutenants p add v to W_p.

Suppose a correct lieutenant receives a valid message $\langle v, \ell \rangle$ in pulse $k{+}1$. Since ℓ has length $k{+}1$, it contains a correct q.
Then q received a valid message $\langle v, \ell' \rangle$ in a pulse $\leq k$.
In the next pulse, q made all correct lieutenants p add v to W_p.

So after pulse $k{+}1$, W_p is the same for all correct lieutenants p.

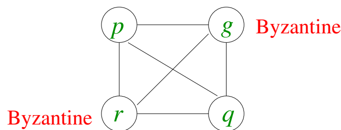
Lamport-Shostak-Pease Authentication Alg. - Optimization

Dolev-Strong optimization: A correct lieutenant broadcasts at most two messages, with different values.

Because when it has broadcast two different values,
all correct lieutenants are certain to decide 0.

Lamport-Shostak-Pease Authentication Alg. - Example

$N = 4$ and $k = 2$.



pulse 1: g sends $\langle 1, (S_g(1), g)\rangle$ to p and q

g sends $\langle 0, (S_g(0), g)\rangle$ to r

$W_p = W_q = \{1\}$

pulse 2: p broadcasts $\langle 1, (S_g(1), g) : (S_p(1), p)\rangle$

q broadcasts $\langle 1, (S_g(1), g) : (S_q(1), q)\rangle$

r sends $\langle 0, (S_g(0), g) : (S_r(0), r)\rangle$ to q

$W_p = \{1\}$ and $W_q = \{0, 1\}$

pulse 3: q broadcasts $\langle 0, (S_g(0), g) : (S_r(0), r) : (S_q(0), q)\rangle$

$W_p = W_q = \{0, 1\}$

p and q decide 0

Mutual Exclusion

Processes contend to enter their critical section.

A process that enters its critical section is called privileged.

For each execution, we require mutual exclusion and starvation-freeness:

- In every configuration, at most one process is privileged.
- If a process p_i tries to enter its critical section, and no process remains privileged forever, then p_i will eventually become privileged.

Mutual Exclusion with Message Passing

Mutual exclusion algorithms with *message passing* are generally based on one of the following three paradigms.

- Logical clock: Requests for entering a critical section are prioritized by means of logical time stamps.
- Token passing: The process holding the token is privileged.
- Quorums: To become privileged, a process needs the permission of a quorum of processes.

  Each pair of quorums should have a non-empty intersection.

Ricart-Agrawala Algorithm

When a process p_i wants to access its critical section, it sends $request(ts_i, i)$ to all other processes, with ts_i its logical time stamp.

When p_j receives this request, it sends permission to p_i as soon as:

- p_j isn't in its critical section, and
- p_j doesn't have a pending request with time stamp ts_j and $(ts_j, j) < (ts_i, i)$ (lexicographical order).

p_i enters its critical section when it has received permission from all other processes.

Ricart-Agrawala Algorithm - Correctness

Mutual exclusion: When p sends permission to q:

- p isn't in its critical section; and
- p won't get permission from q to enter its critical section until q has entered and left its critical section.

Starvation-freeness: Each request will eventually become the smallest request 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 sends permission to p_1, and sets its logical time to 1.

p_0 sends *request*$(2, 0)$ to p_1. When p_1 receives this message, it doesn't send permission to p_0, because $(1, 1) < (2, 0)$.

p_1 receives permission from p_0, and enters its critical section.

Ricart-Agrawala Algorithm - Example 2

$N = 2$, and p_0 and p_1 both are at logical time 0.

p_1 sends $request(1, 1)$ to p_0, and p_0 sends $request(1, 0)$ to p_1.

When p_0 receives the request from p_1, it doesn't send permission to p_1, because $(1, 0) < (1, 1)$.

When p_1 receives the request from p_0, it sends permission to p_0, because $(1, 0) < (1, 1)$.

p_0 receives permission from p_1, and enters its critical section.

Ricart-Agrawala Algorithm - Optimization

Drawback: High message overhead,
because requests must be sent to all other processes.

Carvalho-Roucairol optimization: After a process q has exited its critical section, q only needs to send requests to processes that q has sent permission to since this exit.

Because as long as a process p hasn't obtained q's permission, p can't enter its critical section.

If such a process p sends a request to q while q is waiting for permissions, and p's request is smaller than q's request, then q sends both permission and a request to p.

Raymond's Algorithm

Given an undirected network, with a sink tree.
At any time, the root, holding a token, is privileged.

Each process maintains a FIFO queue, which can contain id's of its children, and its own id. Initially, this queue is empty.

Queue maintenance:

- When a non-root wants to enter its critical section, it adds its id to its own queue.
- When a non-root gets a new head at its (non-empty) queue, it asks its parent for the token.
- When a process receives a request for the token from a child, it adds this child to its queue.

Raymond's Algorithm

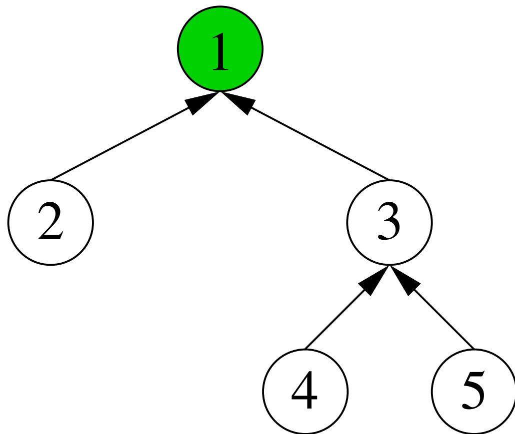
When the root exits its critical section (and its queue is non-empty),
it sends the token to the process q at the head of its queue,
makes q its parent, and removes q from the head of its queue.

Let a non-root p get the token from its parent,
with process q at the head of its queue.

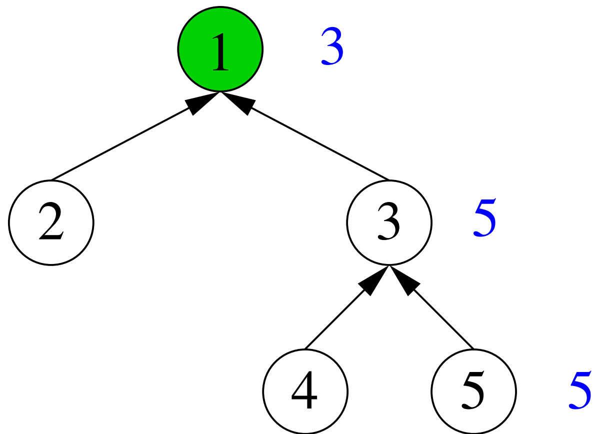
- if $p \neq q$, then p sends the token to q, and makes q its parent
- if $p = q$, then p becomes the root
  (i.e., it has no parent, and is privileged)

In both cases, p removes q from the head of its queue.

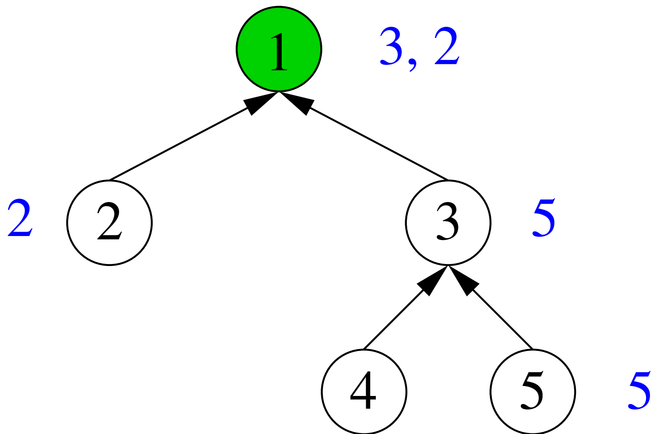
Raymond's Algorithm - Example



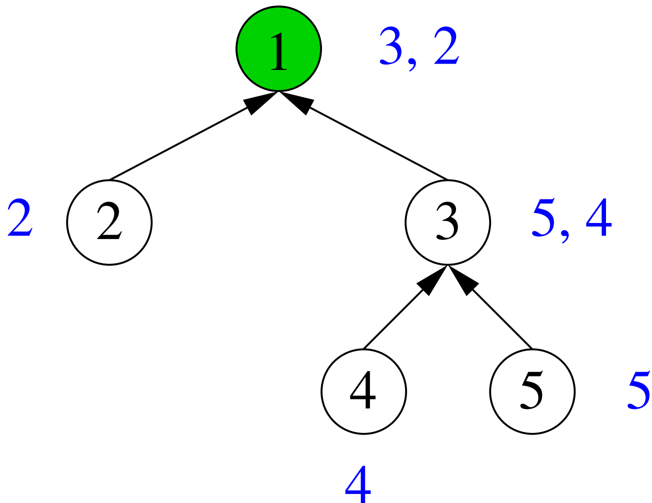
Raymond's Algorithm - Example

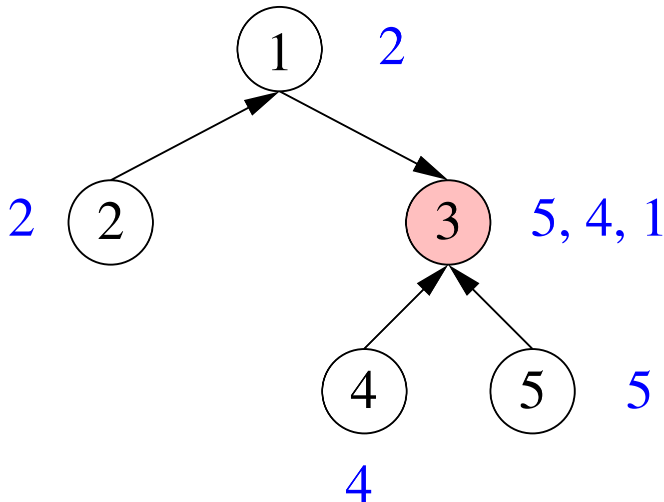


Raymond's Algorithm - Example

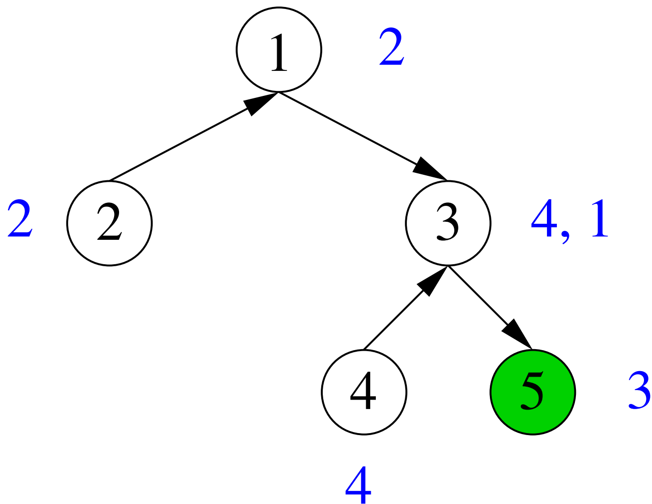


Raymond's Algorithm - Example

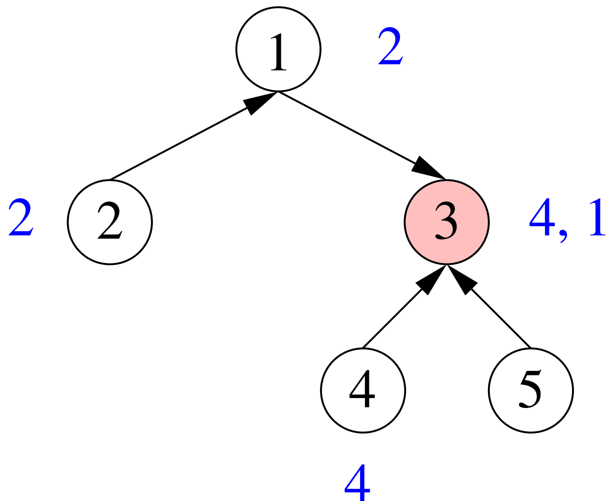



1
2
2
2
3
5, 4, 1
4
5
5
4

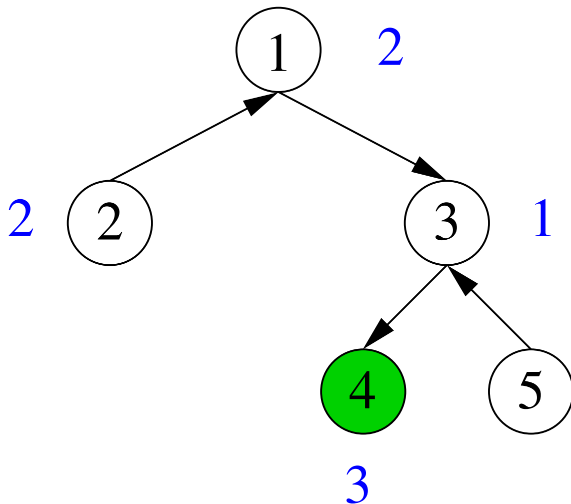
Raymond's Algorithm - Example



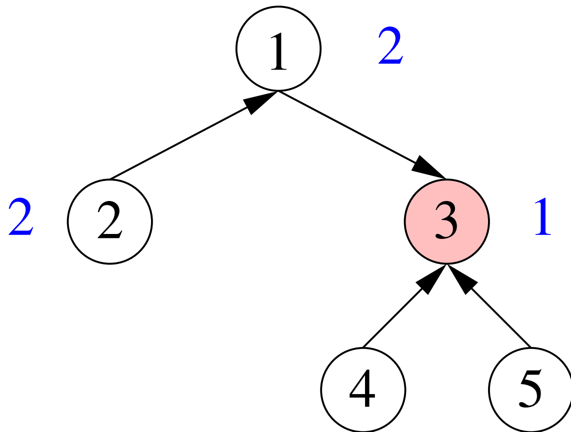
Raymond's Algorithm - Example



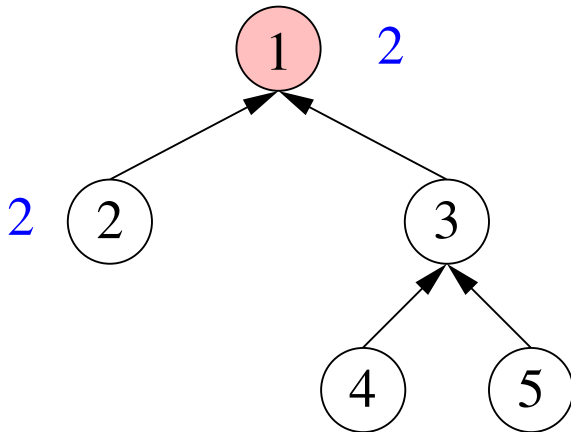
Raymond's Algorithm - Example



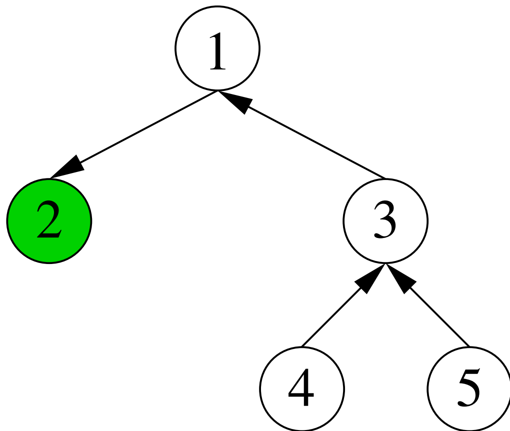
Raymond's Algorithm - Example



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 is starvation-free,
because eventually each request in a queue moves to the head of this queue, and a chain of requests never contains a cycle.

Drawback: Sensitive to failures.

Agrawal-El Abbadi Algorithm

For simplicity we assume that $N = 2^k - 1$, for some $k > 1$.

The processes are structured in a *binary tree* of depth $k-1$.

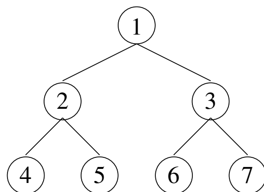
A *quorum* consists of all processes on a path from the root to a leaf.

If a *non-leaf* p has crashed (or is non-responsive), permission can be asked from all processes on two paths instead: from each child of p to some leaf.

To enter a critical section, permission from a quorum is required.

Agrawal-El Abbadi Algorithm - Example

Example: Let $N = 7$.



Possible quorums are:

- $\{1, 2, 4\}$, $\{1, 2, 5\}$, $\{1, 3, 6\}$, $\{1, 3, 7\}$
- if 1 crashed: $\{2, 4, 3, 6\}$, $\{2, 5, 3, 6\}$, $\{2, 4, 3, 7\}$, $\{2, 5, 3, 7\}$
- if 2 crashed: $\{1, 4, 5\}$
- if 3 crashed: $\{1, 6, 7\}$

Question: What are the quorums if 1,2 crashed? And if 1,2,3 crashed? And if 1,2,4 crashed?

Agrawal-El Abbadi Algorithm - Mutual Exclusion

We prove, by induction on depth k, that each pair of quorums has a non-empty intersection, and so mutual exclusion is guaranteed.

A quorum with 1 contains a quorum in one of the subtrees below 1, while a quorum without 1 contains a quorum in both subtrees below 1.

- If two quorums both contain 1, we are done.
- If two quorums both don't contain 1, then by induction they have elements in common in the two subtrees below process 1.
- Suppose quorum Q contains 1, while quorum Q' doesn't. Then Q contains a quorum in one of the subtrees below 1, and Q' also contains a quorum in this subtree. So by induction, Q and Q' have an element in common in this subtree.

Self-Stabilization

All configurations are initial configurations.

An algorithm is self-stabilizing if every execution eventually reaches a "correct" configuration.

Advantages:

- fault tolerance
- straightforward initialization

Self-Stabilization: Communication

In a *message-passing* setting, processes might all be initialized in a state in which they are waiting for a message.

Then the self-stabilizing algorithm wouldn't exhibit any behavior.

Therefore, in self-stabilizing algorithms, processes communicate via variables in shared memory.

A process can read the variables of its neighbors.

Dijkstra's Self-Stabilizing Token Ring

Let $p_0, \ldots, p_{N-1}$ form a directed ring.
Each p_i holds a value $x_i \in \{0, \ldots, K-1\}$ with $K \geq N$.

- p_i with $0 < i < N$ is privileged if $x_i \neq x_{i-1}$.
- p_0 is privileged if $x_0 = x_{N-1}$.

Each privileged process is allowed to change its value, causing the loss of its privilege:

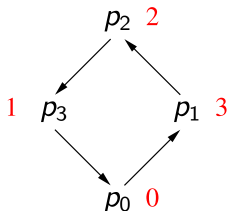
- $x_i := x_{i-1}$ when $x_i \neq x_{i-1}$, for $0 < i < N$
- $x_0 := (x_0+1) \bmod K$ when $x_0 = x_{N-1}$

If $K \geq N$, then Dijkstra's token ring *self-stabilizes*.
That is, each execution eventually satisfies *mutual exclusion*.

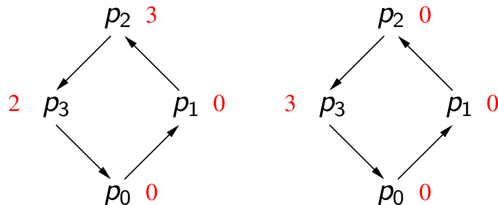
Moreover, Dijkstra's token ring is *starvation-free*.

Dijkstra's Token Ring - Example

Let $N = K = 4$. Consider the initial configuration



It 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.

An event never increases the number of privileged processes.

Consider an (infinite) 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, x_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, $x_0 \neq x_i$ for all $0 < i < N$.

The next time p_0 becomes privileged, clearly $x_i = x_0$ for all $0 < i < N$.
So then mutual exclusion has been achieved.

Arora-Gouda Self-Stabilizing Spanning Tree Algorithm

Given an undirected network.

An *upper bound* K on the *network size* is known to all processes.

The process with the *largest* id becomes the *root*.

Each process p maintains the following variables:

$parent_p$: its parent in the spanning tree

$root_p$: the root of the spanning tree

$dist_p$: its distance from the root via the spanning tree

Arora-Gouda Spanning Tree Algorithm - Complications

Due to arbitrary initialization, there are three complications.

Complication 1: Multiple processes may consider themselves root.

Complication 2: There may be a cycle in the spanning tree.

Complication 3: $root_p$ may not be the id of any process in the network.

Arora-Gouda Spanning Tree Algorithm

A process p declares itself *root*, i.e.

$$parent_p := \bot \qquad root_p := p \qquad dist_p := 0$$

if it detects an inconsistency in its local variables:

- $root_p < p$, or
- $parent_p = \bot$, and $root_p \neq p$ or $dist_p > 0$, or
- $parent_p$ isn't a neighbor of p and $parent_p \neq \bot$, or
- $dist_p \geq K$.

Arora-Gouda Spanning Tree Algorithm

Let there be no inconsistency in the local variables of p.

Let q be a neighbor of p with $dist_q < K$.

Suppose $q = parent_p$.

If $root_p \neq root_q$, then $root_p := root_q$.

If $dist_p \neq dist_q + 1$, then $dist_p := dist_q + 1$.

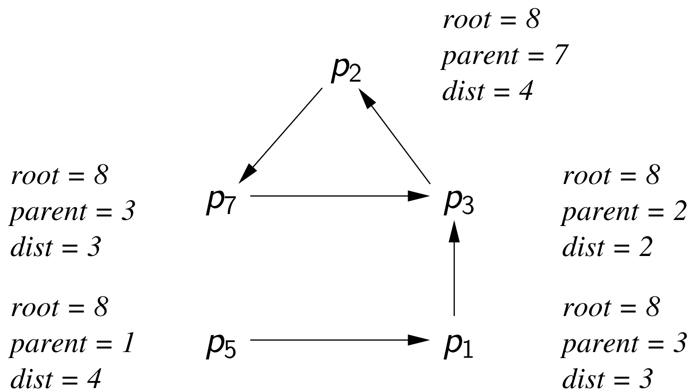
Suppose $q \neq parent_p$.

If $root_p < root_q$, then

$$parent_p := q \qquad root_p := root_q \qquad dist_p := dist_q + 1$$

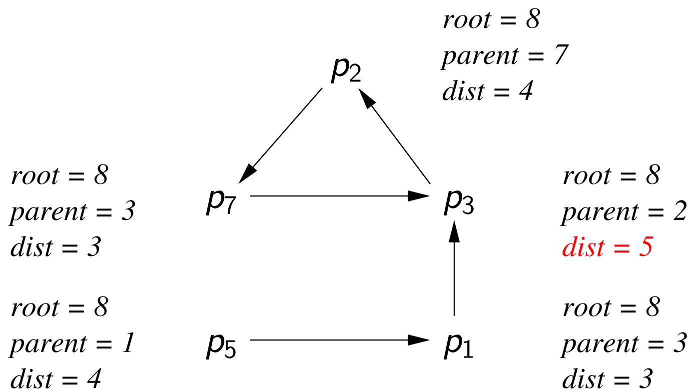
Arora-Gouda Spanning Tree Algorithm - Example

$K = 5$



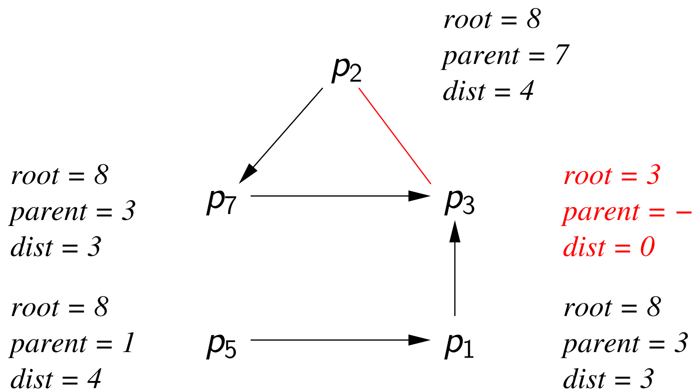
Arora-Gouda Spanning Tree Algorithm - Example

$K = 5$



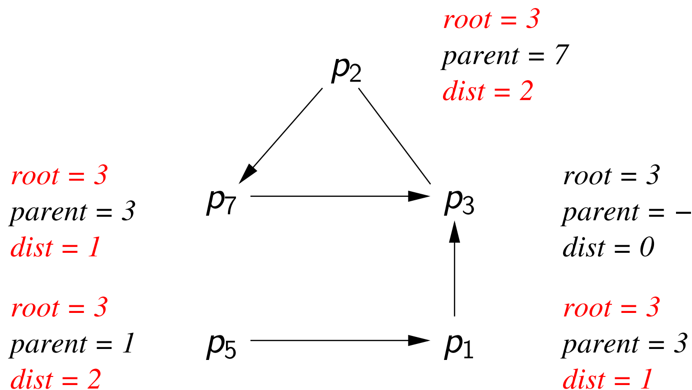
Arora-Gouda Spanning Tree Algorithm - Example

$K = 5$



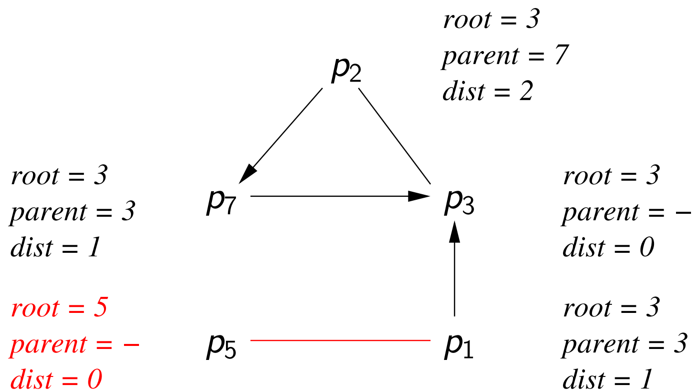
Arora-Gouda Spanning Tree Algorithm - Example

$K = 5$



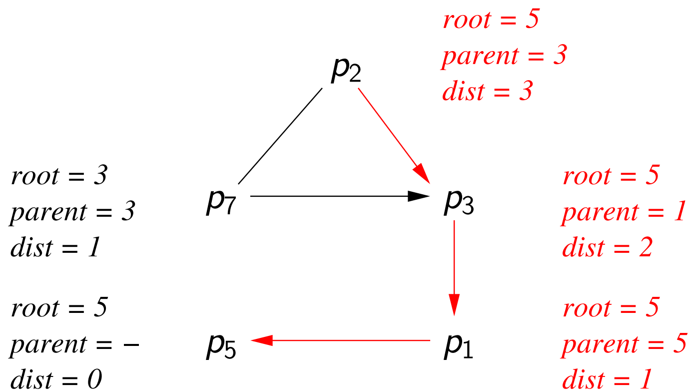
Arora-Gouda Spanning Tree Algorithm - Example

$K = 5$



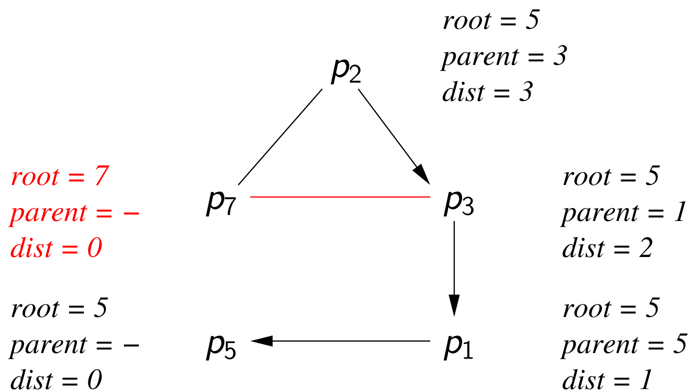
Arora-Gouda Spanning Tree Algorithm - Example

$K = 5$



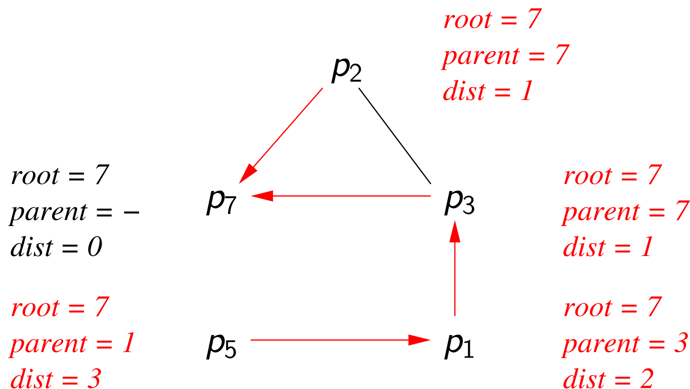
Arora-Gouda Spanning Tree Algorithm - Example

$K = 5$



Arora-Gouda Spanning Tree Algorithm - Example

$K = 5$



Arora-Gouda Spanning Tree Algorithm - Correctness

"False" root values, which aren't an id of any process in the network, will eventually disappear.

Namely, such false roots can only "survive" if there is a cycle of processes that all have this root value.

Distance values of processes in such a cycle will keep on increasing, until one of them gets distance K and declares itself root. Then the cycle is "broken".

Therefore the process with the largest id will eventually declare itself root.

Then the network will converge to a spanning tree with this process as root.

Afek-Kutten-Yung Self-Stabilizing Spanning Tree Algorithm

No upper bound on the network size needs to be known.
Again the process with the *largest* id becomes the *root*.

A process p can declare itself *root*, i.e.

$$parent_p := \perp \qquad root_p := p \qquad dist_p := 0$$

if it detects an inconsistency in its local variables,
or with the local variables of its parent (and p isn't yet a root):

- $root_p \leq p$, or
- $parent_p$ isn't a neighbor of p and $parent_p \neq \perp$, or
- $parent_p \neq \perp$, and $root_p \neq root_{parent_p}$ or $dist_p \neq dist_{parent_p} + 1$.

Question

Suppose that during an application of the Afek-Kutten-Yung algorithm, the created directed network contains a cycle.

Why can always some process on such a cycle declare itself root ?

Answer: Because at some p on this cycle, $dist_p \neq dist_{parent_p} + 1$.

Afek-Kutten-Yung Spanning Tree Algorithm

A root p can make a neighbor q its *parent* if $p < root_q$ and no neighbor of p has a *root* value larger than $root_q$:

$$parent_p := q \qquad root_p := root_q \qquad dist_p := dist_q + 1$$

Complication: Processes can infinitely often join a component with a "false" root.

Afek-Kutten-Yung Spanning Tree Algorithm - Example

Given two processes 0 and 1.

$parent_0 = 1$ and $parent_1 = 0$; $root_0 = root_1 = 2$; $dist_0 = dist_1 = 0$.

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:
$parent_0 := \perp$, $root_0 := 0$ and $dist_0 := 0$.

Since $root_0 < root_1$, 0 makes 1 its *parent*:
$parent_0 := 1$, $root_0 := 2$ and $dist_0 := 2$.

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:
$parent_1 := \perp$, $root_1 := 1$ and $dist_1 := 0$.

Since $root_1 < root_0$, 1 makes 0 its *parent*:
$parent_1 := 0$, $root_1 := 2$ and $dist_1 := 3$.   Et cetera

Afek-Kutten-Yung Spanning Tree Algorithm - Join Requests

Let $root_q$ be greater than p, and the maximum of the *root* values of the neighbors of p.

Before p makes q its parent, it must wait until q's component has a proper root. Therefore p first sends a *join request* to q.

This request is forwarded through q's component, toward the root of this component.

The root sends back an *ack* toward p, which travels the reverse path of the request.

When p receives this ack, it makes q its *parent*:
$parent_p := q$, $root_p := root_q$ and $dist_p := dist_q + 1$.

Afek-Kutten-Yung Spanning Tree Algorithm - Example

Given two processes 0 and 1.

$parent_0 = 1$ and $parent_1 = 0$; $root_0 = root_1 = 2$; $dist_0 = dist_1 = 0$.

Since $dist_0 \neq dist_1 + 1$, 0 declares itself *root*:
$parent_0 := \perp$, $root_0 := 0$ and $dist_0 := 0$.

Since $root_0 < root_1$, 0 sends a *join request* to 1.

This join request doesn't immediately trigger an ack.

Since $dist_1 \neq dist_0 + 1$, 1 declares itself *root*:
$parent_1 := \perp$, $root_1 := 1$ and $dist_1 := 0$.

Since 1 is now a proper root, it replies to the join request of 0 with an ack, and 0 makes 1 its *parent*:
$parent_0 := 1$, $root_0 := 1$ and $dist_0 := 1$.

Afek-Kutten-Yung Spanning Tree Algorithm - Communicatio

Communication is performed using *shared memory*,
so join requests and ack's are encoded in shared variables.

The path of a join request is remembered in local variables.

A process can be forwarding and awaiting an ack for at most one join request at a time. (That's why in the previous example 1 can't send 0's join request on to 0.)

Join requests are only forwarded between "consistent" processes.

Afek-Kutten-Yung Spanning Tree Algorithm - Example

Given a ring with processes p, q, r, and $s > p, q, r$.

Initially, p and q consider themselves root,
while r has p as parent and considers s the root.

Since $root_r > q$ and $root_r > root_p$, q sends a join request to r.

Without the consistency check, r would forward this join request to p.
Since p considers itself root, it would send back an ack to q (via r),
and q would make r its parent and consider s the root.

Since $root_r \neq root_p$, r makes itself root.

Now we would have a symmetrical configuration to the initial one.

Afek-Kutten-Yung Spanning Tree Algorithm - Correctness

Each component in the network with a false root has an inconsistency, so a process in this component will declare itself root.

Join requests, which are only passed on between consistent processes, guarantee that processes only finitely often join a component with a false root, (each time due to improper initial values of local variables).

Since processes can only be involved in one join request at a time, there can't be a cycle of join requests.

These observations together imply that eventually false roots will disappear, the process with the largest id in the network will declare itself root, and the network will converge to a spanning tree with this process as the root.

Jobs

A job is a unit of work, scheduled and executed by the system.

Parameters of jobs are:

- functional behavior
- time constraints
- resource requirements

Jobs 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 time needed to perform the job (assuming it executes alone and all resources are available)

absolute deadline: when a job is required to have been completed

relative deadline: maximum allowed time between arrival and completion of a job

hard deadline: late completion not allowed

soft deadline: late completion allowed

Types of Tasks

A task is a set of related jobs.

We distinguishes three types of tasks:

- periodic: known input at 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

We focus on aperiodic and sporadic *jobs*.

Periodic Tasks

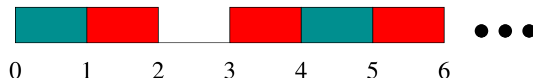
A periodic task is defined by:

- release time r of the first periodic job
- period p (periodic time interval, at the start of which a periodic job is released)
- execution time e of each periodic job

We assume that the relative deadline of each periodic job equals its period.

Periodic Tasks - Example

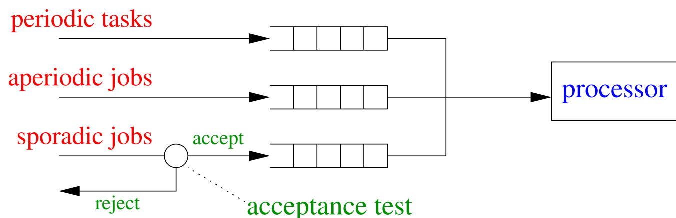
$T_1 = (1, 2, 1)$ and $T_2 = (0, 3, 1)$.



The *utilization* is $\frac{5}{6}$.

The conflict at time 3 must be resolved by some scheduler.

Job Queues



Sporadic jobs are only accepted if they can be completed in time.

Aperiodic jobs are always accepted, and performed such that periodic and accepted sporadic jobs don't 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 at a *processor* is the sum of utilizations of its periodic tasks.

Assumptions:

- Jobs are preemptive: they can be suspended at any time in their execution
- No resource competition

Theorem: Scheduling of periodic tasks on a processor is feasible if and only if its utilization is ≤ 1.

Scheduler

The scheduler of a processor schedules and allocates resources to jobs (according to some scheduling and resource access control algorithms).

- Jobs can't be scheduled before their release times.
- The total amount of time assigned to a job must equal its (maximum) execution time.

A schedule is feasible if all hard deadlines are met.

An optimal scheduler produces a feasible schedule whenever possible.

Off-Line Scheduling

An off-line schedule 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.

Off-line scheduling is conceptually simple, but can't cope well with:

- jitter: imprecise release and execution times
- extra workload
- nondeterminism
- system modifications

On-Line Scheduling

An on-line 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): available idle time of a job until the next deadline

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).

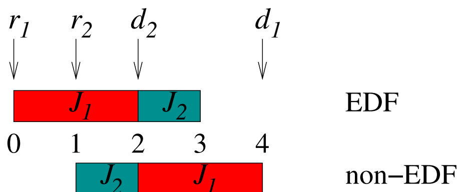
Let $T_1 = (0, 4, 2)$ and $T_2 = (0, 6, 3)$.

EDF Scheduler

Earliest Deadline First: The earlier the deadline, the higher the priority.

Theorem: Given one processor, and preemptive jobs.
When jobs don't compete for resources, the EDF scheduler is optimal.

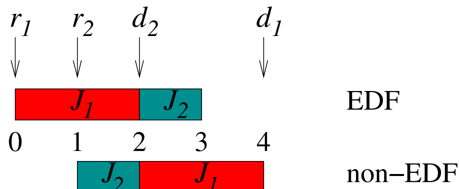
Example: Non-optimality in case of non-preemptive jobs.



EDF Scheduler

Example: Non-optimality in case of resource competition.

Let J_1 and J_2 both require resource R.



LST Scheduler

Slack is the available idle time of a job until the next deadline.

Least Slack-Time first: less slack gives a job higher priority.

Theorem: Given one processor, and preemptive jobs.
When jobs don't compete for resources, the LST scheduler is optimal.

Slack gives precise information how much time can be spent on other jobs, but is computationally expensive.

With the LST scheduler, priorities of jobs change dynamically.

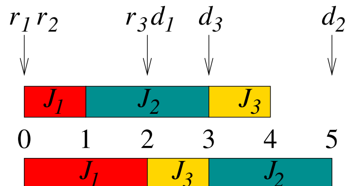
Continuous scheduling decisions would lead to context switch overhead in case of two jobs with the same slack.

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.

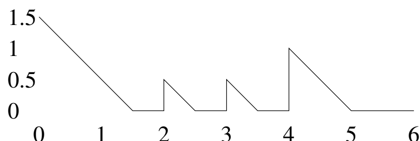
Scheduling Aperiodic Jobs

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 when the processor has 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 (especially 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.

$\sum_{k=1}^{n} \frac{e_k}{p_k} + \frac{e_s}{p_s} \leq 1$ doesn't 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.

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

$$\sum_{k=1}^{n} \frac{e_k}{p_k} + \tilde{u}_s \leq 1$$

If the aperiodic queue is non-empty, a deadline d is determined for the head of the queue, according to the rules below.

If, 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*, then

$$d := \max(d, t) + \frac{e}{\tilde{u}_s}$$

with e the execution time of the new head of the aperiodic queue.

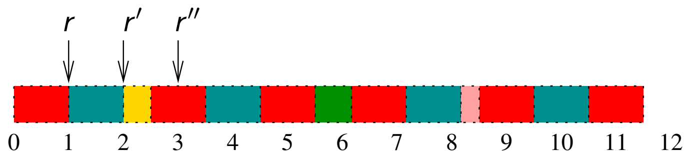
Initially, $d = 0$.

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 $6\frac{1}{6}$) 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

Periodic and accepted sporadic jobs are scheduled according to EDF.

The acceptance test may reject schedulable sporadic jobs.

The total bandwidth server can be integrated with an acceptance test for sporadic jobs (e.g. make the allowed utilization rate $\tilde{u}_s$ dynamic).

Remote Access Control Algorithms

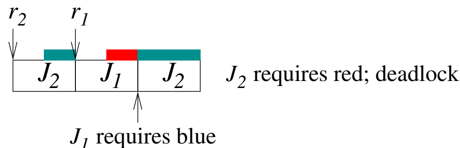
Resource units can be requested by jobs during their execution, and are allocated to jobs in a mutually exclusive fashion.

When a requested resource is refused, the job is preempted.

Dangers of resource sharing:

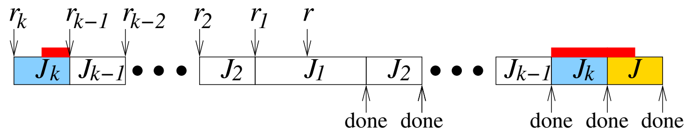
(1) Deadlock can occur.

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.

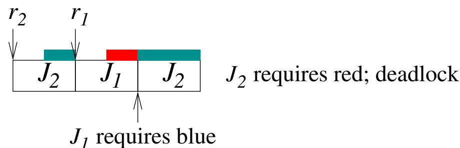


Priority Inheritance

When a job J requires a resource R and is preempted because a lower-priority job J' holds R, then J' inherits the priority of J until it releases R.

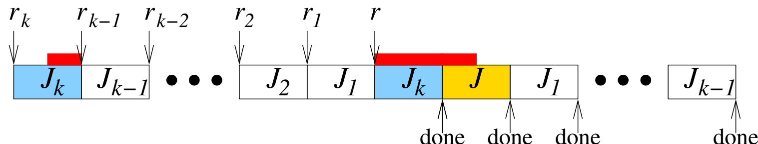
(1) Deadlock can still occur.

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 the red resource.



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 a *processor* 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 isn't released until its priority is higher than
the priority ceiling of the processor.

Priority Ceiling

When no resources are in use, all arrived jobs are released immediately.

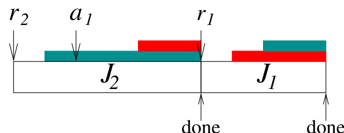
Assumption: The resources required by a job are known beforehand.

Note: In the pictures to follow, *a* denotes the *arrival* of a job (different from its release).

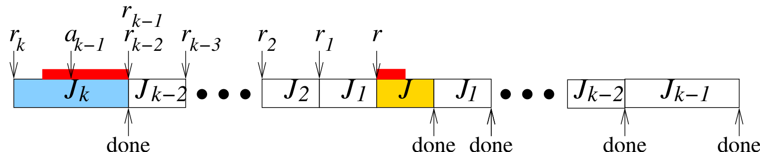
Priority Ceiling

(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.

Priority Inheritance

Priority ceiling has no effect on jobs that have been released.

Therefore we also impose priority inheritance: A job inherits the priority of a higher-priority job that it is blocking.

Question: What would happen if J were only known at its arrival ?

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 $\geq t$.