Distributed Algorithms

Wan Fokkink

Distributed Algorithms: An Intuitive Approach (2nd edition)
MIT Press, 2018
A skilled programmer must have good insight into algorithms.

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

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

You’re trained in algorithmic thought for uniprocessor programs (e.g. divide-and-conquer, greedy, memoization).
A distributed system is an interconnected collection of autonomous processes.

Motivation:

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

Distributed systems differ from uniprocessor systems in three aspects.

- **Lack of knowledge on the global state**: A process has no up-to-date knowledge on the local states of other processes.
  
  **Example**: Termination and deadlock detection become issues.

- **Lack of a global time frame**: No total order on events in an execution by their temporal occurrence.
  
  **Example**: Mutual exclusion becomes an issue.

- **Nondeterminism**: Different executions of a system can give different results.
  
  **Example**: Race conditions.
Aim of this course

This course offers a *bird’s-eye view* on a wide range of algorithms for basic and important challenges in distributed systems. It provides an algorithmic frame of mind for solving fundamental problems in distributed computing.

- Handwaving correctness arguments.
- Back-of-the-envelope complexity calculations.
- Carefully developed exercises to acquaint you with intricacies of distributed algorithms.
The two main paradigms to capture communication in a distributed system are message passing and shared memory. 

We’ll focus mainly on message passing.

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

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

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

We’ll mainly consider asynchronous communication.
In a computer network, messages are transported through a medium, which may lose, duplicate or garble these messages.

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

**Example:** Sliding window protocols.
Assumptions

Unless stated otherwise, we assume:

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

Channels can be \textit{directed} or \textit{bidirectional}.

\textbf{Question}: What is more general, an algorithm for a \textit{directed} or for an \textit{undirected} network?

\textbf{Remarks}:

\begin{itemize}
\item Algorithms for undirected networks often include ack’s.
\item Acyclic networks must always be undirected (else the network wouldn’t be strongly connected).
\end{itemize}
Complexity measures

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

**Message complexity:** Total number of messages exchanged.

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

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

**Space complexity:** Amount of memory needed for the processes.

Different executions require different consumption of resources.

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

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

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

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

$f = O(g)$ 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 $g = O(f)$. 
Now follows a formal framework for describing distributed systems, mainly to fix terminology.

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

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

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

The configuration evolves in discrete steps, called transitions.

A transition system consists of:

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

$\gamma \in \mathcal{C}$ is terminal if $\gamma \rightarrow \delta$ for no $\delta \in \mathcal{C}$. 
An execution is a sequence \( \gamma_0 \gamma_1 \gamma_2 \cdots \) of configurations that either is infinite or ends in a terminal configuration, such that:

- \( \gamma_0 \in \mathcal{I} \), and

- \( \gamma_i \rightarrow \gamma_{i+1} \) for all \( i \geq 0 \)

(except, for finite executions, the terminal \( \gamma_i \) at the end).

A configuration \( \delta \) 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.
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”
Assertion $P$ on configurations is an invariant if:

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

Each invariant is a safety property.

**Question**: Give a transition system $S$ and an assertion $P$ such that $P$ is a safety property but not an invariant for $S$. 
In each configuration of an asynchronous system, applicable events at different processes are independent.

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

- if $a$ and $b$ are events at the same process and $a$ occurs before $b$, then $a \prec b$; and

- if $a$ is a send and $b$ the corresponding receive event, then $a \prec b$.

This relation is irreflexive.

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

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

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

These permutations together form a computation.

All executions of a computation start in the same initial configuration. And if they are finite, they all end in the same terminal configuration.
Consider the finite execution $abc$.

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

Which executions are in the same computation?
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 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 $LC(a) = \max\{k, LC(b)\} + 1$. 
Consider the following sequences of events at processes $p_0, p_1, p_2$:

\[
\begin{align*}
  p_0 &: a \ s_1 \ r_3 \ b \\
  p_1 &: c \ r_2 \ s_3 \\
  p_2 &: r_1 \ d \ s_2 \ e
\end{align*}
\]

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

Provide all events with Lamport’s clock values.

**Answer:** 1 2 8 9

1 6 7

3 4 5 6
Vector clock

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

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

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

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

$VC(a) = (k_0, \ldots, k_{N-1})$ where each $k_i$ is the length of a longest causality chain $a_{i1}^j \prec \cdots \prec a_{ik_i}^j$ of events at process $p_j$ with $a_{ik_i}^j \preceq a$.

$VC$ can also be computed at run-time.
Consider the same sequences of events at processes $p_0, p_1, p_2$:

$p_0 : \quad a \quad s_1 \quad r_3 \quad b$

$p_1 : \quad c \quad r_2 \quad s_3$

$p_2 : \quad r_1 \quad d \quad s_2 \quad e$

Provide all events with vector clock values.

Answer: $(1 \ 0 \ 0)$ $(2 \ 0 \ 0)$ $(3 \ 3 \ 3)$ $(4 \ 3 \ 3)$

$(0 \ 1 \ 0)$ $(2 \ 2 \ 3)$ $(2 \ 3 \ 3)$

$(2 \ 0 \ 1)$ $(2 \ 0 \ 2)$ $(2 \ 0 \ 3)$ $(2 \ 0 \ 4)$
Vector clock - Correctness

Let $a \prec b$.

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

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

Let $\text{VC}(a) < \text{VC}(b)$.

Consider the longest causality chain $a^1_1 \prec \cdots \prec a^1_k = a$ of events at the process $p_i$ where $a$ occurs.

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

So $a \leq b$.

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

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

Snapshots can be used for:

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

Examples: deadlock, garbage.

**Challenge**: Take a snapshot without (temporarily) freezing the execution.
We distinguish basic messages of the underlying distributed algorithm and control messages of the snapshot algorithm.

A snapshot of a (basic) execution consists of:

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

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

We need to avoid the following situations.

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

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

Consider a directed network with *FIFO* channels.

**Initiators** take a local snapshot of their state, and send a control message \( \langle \text{marker} \rangle \) to their neighbors.

When a process that hasn’t yet taken a snapshot receives \( \langle \text{marker} \rangle \), it
  - takes a local snapshot of its state, and
  - sends \( \langle \text{marker} \rangle \) to all its neighbors.

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

If channels are FIFO, this produces a meaningful snapshot.

**Message complexity:** \( \Theta(E) \) (with \( E \) the number of edges)

**Worst-case time complexity:** \( O(D) \) (with \( D \) the diameter)
Chandy-Lamport algorithm - Example

The snapshot (processes red/blue/green, channels \(\emptyset\), \(\emptyset\), \(\emptyset\), \(\{m_2\}\)) isn't a configuration in the actual execution. The send of \(m_1\) isn't causally before the send of \(m_2\). So the snapshot is a configuration of an execution that is in the same computation as the actual execution.
The snapshot (processes red/blue/green, channels $\emptyset$, $\emptyset$, $\emptyset$, $\{m_2\}$) isn't a configuration in the actual execution. The send of $m_1$ isn't causally before the send of $m_2$. So the snapshot is a configuration of an execution that is in the same computation as the actual execution.
The snapshot (processes red/blue/green, channels $\emptyset$, $\emptyset$, $\emptyset$, $\{m_2\}$) isn't a configuration in the actual execution. The send of $m_1$ isn't causally before the send of $m_2$. So the snapshot is a configuration of an execution that is in the same computation as the actual execution.
The snapshot \((\emptyset, \emptyset, \emptyset, \{m_2\})\) isn’t a configuration in the actual execution. The send of \(m_1\) isn’t causally before the send of \(m_2\). So the snapshot is a configuration of an execution that is in the same computation as the actual execution.
The snapshot (processes red/blue/green, channels $\emptyset, \emptyset, \emptyset, \{m_2\}$) isn’t a configuration in the actual execution.

The send of $m_1$ isn’t causally before the send of $m_2$.

So the snapshot is a configuration of an execution that is in the same computation as the actual execution.
Claim: If a post-snapshot event \(e\) is causally before an event \(f\), then \(f\) is also post-snapshot.

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

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

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

Let \(e\) occur at \(p\) and \(f\) at \(q\).

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

Channels are FIFO, so \(q\) receives this \(\langle\text{marker}\rangle\) before \(f\).

Hence \(f\) is post-snapshot at \(q\).
Suppose channels are \textit{non-FIFO}. We use piggybacking.

\textbf{Initiators} take a local snapshot of their state.

When a process has taken its local snapshot, it appends \textit{true} to each outgoing basic message.

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

Process \textit{q} computes as channel state of \textit{pq} the basic messages without the tag \textit{true} that it receives via \textit{pq} after its local snapshot.
Question: How does $q$ know when it can determine the channel state of $pq$?

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

These control messages also ensure that all processes eventually take a local snapshot.
Lai-Yang algorithm - Multiple snapshots

**Question:** How can multiple subsequent snapshots be supported?

**Answer:** Each snapshot is provided with a sequence number.

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

Control messages carry the sequence number of their snapshot.
What we need from last lecture

fully asynchronous message passing framework
channels are non-FIFO, and can be directed or bidirectional

configurations and transitions at global level
states and events (internal/send/receive) at local level

(non)initiator
(de)centralized algorithm

causal order $\prec$ on events in an execution
computation of executions, by reordering concurrent events
snapshot algorithm to compute a configuration of a computation
basic/control algorithm
Wave algorithms

A **decide event** is a special internal event.

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

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

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

The initiator decides after the token has returned.

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

The ring algorithm is an example of a *traversal* algorithm.
A traversal algorithm is a centralized wave algorithm. The initiator sends around a token.

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

Traversal algorithms build a spanning tree:

- the initiator is the root; and
- each noninitiator has as parent the neighbor from which it received the token first.
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:** $2 \cdot E$ messages

**Time complexity:** $\leq 2 \cdot E$ 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 - Spanning tree

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

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

Frond edges, which aren’t part of the spanning tree, are dashed.
Claim: The token $\theta$ travels through each channel in either direction, and ends up at the initiator.

Proof: A noninitiator holding $\theta$, received $\theta$ once more than it sent $\theta$. So by R1, this process can send $\theta$ into a channel. Hence $\theta$ ends at the initiator, after traversing all its channels both ways.

Assume some channel isn’t traversed by $\theta$ both ways. Let $q$ be the earliest visited process with such a channel. $q$ has a parent $p$.

Since $\theta$ visits $p$ before $q$, it traverses the channel $pq$ both ways. So by R2, $q$ sends $\theta$ into all its channels. Since $q$ sends and receives $\theta$ an equal number of times, it also receives $\theta$ through all its channels.

So $\theta$ travels through all channels of $q$ both ways; contradiction.
Could this spanning tree have been produced by a depth-first search starting at $p$?
Depth-first search is obtained by adding to Tarry’s algorithm:

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

Example:

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

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

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

**Message complexity:** $2 \cdot N - 2$ messages (if there are $N$ processes)

Each tree edge carries 2 tokens.

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

**Bit complexity:** Up to $k \cdot N$ bits per message (where $k$ bits are needed to represent one process).
Awerbuch’s algorithm

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

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

The token is only forwarded to processes that weren’t yet visited by the token (except when a process sends the token to its parent).
Awerbuch’s algorithm - Complexity

Message complexity: \( \leq 4 \cdot E \) 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: \( \leq 4 \cdot N - 2 \) time units

Each tree edge carries 2 tokens.

Each process waits at most 2 time units for ack’s to return.
Abolish ack’s from Awerbuch’s algorithm.

The token is forwarded without delay.

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

Suppose process $p$ receives the token from a process $q \neq fw_p$.
Then $p$ marks $pq$ as frond edge and *dismisses* the token.

Suppose process $q$ receives an info message from $fw_q$.
Then $q$ marks $pq$ as frond edge and continues forwarding the token.
Cidon’s algorithm - Complexity

Message complexity: \( \leq 4 \cdot E \) messages

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

Time complexity: \( \leq 2 \cdot N - 2 \) time units

Each tree edge carries 2 tokens.

At least once per time unit, a token is forwarded through a tree edge.
Cidon’s algorithm - Example
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, who becomes its parent.

Then it sends a message to its parent.

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

It sends the decision to all neighbors except its parent.

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

Always two (neighboring) processes decide.
Tree algorithm - Example
Questions

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

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

Proof: Suppose some process $p$ never sends a message. $p$ doesn’t receive a message through two of its channels, $qp$ and $rp$. $q$ doesn’t receive a message through two of its channels, $pq$ and $sq$. Continuing this argument, we get a cycle of processes that don’t receive a message through two of their channels. Since the network topology is a tree, there is no cycle; contradiction. So each process eventually sends a message. Clearly each channel carries at least one message. There are $N – 1$ channels, so one channel carries two messages. Only the two processes connected by this channel decide.
The echo algorithm is a centralized wave algorithm for undirected networks.

- The initiator sends a message to all neighbors.
- When a noninitiator receives a message for the first time, it makes the sender its parent. Then it sends a message to all neighbors except its parent.
- When a noninitiator has received a message from all neighbors, it sends a message to its parent.
- When the initiator has received a message from all neighbors, it decides.

**Message complexity:** \(2 \cdot E\) messages
Echo algorithm - Example

decide

decide
Use the echo algorithm to determine the largest process id in a network.

Let each process initiate a run of the echo algorithm, tagged by its id. Processes only participate in the “largest” wave they have seen so far. Which of these concurrent waves perform a decide event?
A **deadlock** occurs if there is a cycle of processes waiting until:

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

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

**Examples:**

- A process is waiting for one message from a group of processes:  
  *N = 1*
- A database transaction first needs to lock several files:  
  *N = M*.
A (non-blocked) process can issue a request to $M$ other processes, and becomes blocked until $N$ of these requests have been granted.

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

Only non-blocked processes can grant a request.

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

There is an edge from node $p$ to node $q$ if $p$ sent a request to $q$ that wasn’t yet dismissed by $p$ or granted by $q$. 
Suppose process $p$ must wait for a message from process $q$.

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

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

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

Then edge $pq$ is removed from the wait-for graph, and $p$ becomes unblocked.
Suppose two processes $p$ and $q$ want to claim a resource.

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

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

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

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

The resource is given to $q$. Hence $w$ grants the request from $v$. Edge $vw$ is removed and edge $wv$ is created.
Drawing wait-for graphs

AND (3-out-of-3) request

OR (1-out-of-3) request
Draw the wait-for graph for the initial configuration of the tree algorithm, applied to the following network.

Is there a deadlock in this initial configuration?
Static analysis on a wait-for graph

During an execution of a basic algorithm, a snapshot is taken of the wait-for graph.

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

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

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

When no more grants are possible, nodes that remain blocked in the wait-for graph are deadlocked in the snapshot of the basic algorithm.
Is there a deadlock?
Static analysis - Example 1

Deadlock
Static analysis - Example 2

Diagram:

- Four nodes labeled 'b'
- Directed edges connect the nodes in a square pattern

No deadlock
Static analysis - Example 2

No deadlock
No deadlock
Given an undirected network, and a basic algorithm.

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

Each node $u$ takes a local snapshot of:

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

Then it computes:

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

$In_u$: the nodes it received a request from (not dismissed)
Bracha-Toueg deadlock detection algorithm

$\text{requests}_u$ is the number of grants $u$ requires to become unblocked.

When $u$ receives a grant message, $\text{requests}_u \leftarrow \text{requests}_u - 1$.

If $\text{requests}_u$ becomes 0, $u$ sends grant messages to all nodes in $I_n_u$.

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

**Challenge:** The initiator must detect termination of deadlock detection.
Initially $\text{notified}_u = false$ and $\text{free}_u = false$ at all nodes $u$.

The initiator starts a deadlock detection run by executing $\text{Notify}$.

\[ \text{Notify}_u: \quad \text{notified}_u \leftarrow true \]
for all $w \in \text{Out}_u$ send NOTIFY to $w$
if requests$_u = 0$ then $\text{Grant}_u$
for all $w \in \text{Out}_u$ await DONE from $w$

\[ \text{Grant}_u: \quad \text{free}_u \leftarrow true \]
for all $w \in \text{In}_u$ send GRANT to $w$
for all $w \in \text{In}_u$ 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 $\text{notified}_u = \text{false}$, then $u$ executes $\text{Notify}_u$.
$u$ sends back DONE.

Let $u$ receive GRANT.
If $\text{requests}_u > 0$, then $\text{requests}_u \leftarrow \text{requests}_u - 1$;
if $\text{requests}_u$ becomes 0, then $u$ executes $\text{Grant}_u$.
$u$ sends back ACK.

When the initiator has received DONE from all nodes in its $\text{Out}$ set, it checks the value of its $\text{free}$ field.

If it is still $\text{false}$, the initiator concludes it is deadlocked.
Bracha-Toueg deadlock detection algorithm - Example

$u$ is the initiator

requests$_u = 2$

requests$_x = 0$

requests$_v = 1$

requests$_w = 1$

wait DONE from $v, x$

NOTIFY

NOTIFY
Bracha-Toueg deadlock detection algorithm - Example

await DONE from $v, x$

$u$  

$\text{GRANT}$  

await DONE from $w$  
(for DONE to $u$)  

$\text{NOTIFY}$  

$w$  

$\text{GRANT}$  

await ACK from $u, w$  
(for DONE to $u$)  

$x$
Bracha-Toueg deadlock detection algorithm - Example

\[ \text{requests}_u = 1 \]

\[ \text{await DONE from } v, x \]

\[ \text{await ACK from } u, w \]
\[ \text{(for DONE to } u) \]

\[ \text{NOTIFY} \]

\[ \text{requests}_w = 0 \]

\[ \text{await DONE from } w \]
\[ \text{(for DONE to } u) \]

\[ \text{await DONE from } x \]
\[ \text{(for DONE to } v) \]

\[ \text{await ACK from } v \]
\[ \text{(for ACK to } x) \]

\[ \text{await ACK from } u, w \]
\[ \text{(for DONE to } u) \]
Bracha-Toueg deadlock detection algorithm - Example

\[ \text{await DONE from } \nu, \chi \]

\[ \text{await ACK from } \nu \quad (\text{for ACK to } \chi) \]

\[ \text{await ACK from } \nu \quad (\text{for ACK to } \chi) \]

\[ \text{GRANT} \]

\[ \text{requests}_\nu = 0 \]

\[ \text{await DONE from } \nu \quad (\text{for DONE to } \chi) \]

\[ \text{await DONE from } \nu \quad (\text{for DONE to } \chi) \]

\[ \text{await ACK from } \nu \quad (\text{for ACK to } \chi) \]

\[ \text{DONE} \]

\[ \text{await ACK from } \nu \quad (\text{for ACK to } \chi) \]

\[ \text{await ACK from } \nu \quad (\text{for ACK to } \chi) \]
Bracha-Toueg deadlock detection algorithm - Example

\[ \text{requests}_u = 0 \]

\begin{align*}
\text{await DONE from } v, x \\
\text{await ACK from } w \\
\text{(for DONE to } u) \\
\text{await DONE from } w \\
\text{(for DONE to } u) \\
\text{await ACK from } u \\
\text{(for ACK to } w) \\
\end{align*}

\begin{align*}
\text{await ACK from } w \\
\text{(for DONE to } u) \\
\text{await ACK from } v \\
\text{(for ACK to } x)
\end{align*}
Bracha-Toueg deadlock detection algorithm - Example

await DONE from \( v, x \)

await ACK from \( w \) (for DONE to \( u \))

DONE

await ACK from \( v \) (for ACK to \( x \))

ACK

\( u \)

\( x \)

\( v \)

\( w \)
Bracha-Toueg deadlock detection algorithm - Example

await DONE from $x$

await ACK from $w$
(for DONE to $u$)
Bracha-Toueg deadlock detection algorithm - Example

await DONE from x
free_u = true, so u concludes that it isn’t deadlocked.
The Bracha-Toueg algorithm is deadlock-free:

The initiator eventually receives DONE’s from all nodes in its Out set. At that moment the Bracha-Toueg algorithm has terminated.

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

1. NOTIFY/DONE’s construct a tree $T$ rooted in the initiator.

2. GRANT/ACK’s construct disjoint trees $T_v$, rooted in a node $v$ where from the start $requests_v = 0$.

The NOTIFY/DONE’s only complete when all GRANT/ACK’s have completed.
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 \textit{Out} set and its \textit{free} field is still \textit{false}, it is deadlocked.

Vice versa, if its \textit{free} field is \textit{true}, there is no deadlock yet, \textit{(if resource requests are granted nondeterministically)}. 
Could we apply the Bracha-Toueg algorithm to itself, to establish that it is a deadlock-free algorithm?

**Answer:** No.

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

Wave algorithm

Traversal algorithm
  ▶ ring algorithm
  ▶ Tarry’s algorithm
  ▶ depth-first search

Tree algorithm

Echo algorithm

Communication and resource deadlock

Wait-for graph

Bracha-Toueg deadlock detection algorithm
Termination detection

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

The *control* algorithm concerns termination detection and announcement. Announcement is simple; we focus on detection.

Termination detection shouldn’t influence basic computations.
Dijkstra-Scholten algorithm

Requires a centralized basic algorithm, and an undirected network.

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

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

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

Why doesn’t it immediately detect termination?
Shavit-Francez algorithm

Allows a decentralized basic algorithm; requires an undirected network.

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

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

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

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

Processes in a tree refuse to participate; decide calls Announce.
Why is the following termination detection algorithm not correct?

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

**Answer:** Let a process $p$ that wasn’t yet visited by the wave make a quiet process $q$ that was already visited active again. Next $p$ becomes quiet before the wave arrives. Now the wave can complete while $q$ is active.
Rana’s algorithm

Allows a decentralized basic algorithm; requires an undirected network.

Each basic message is acknowledged.

Lamport’s logical clock provides (basic and control) events with a time stamp.

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

Let process $p$ at time $t$ become quiet, i.e. (1) $p$ is passive, and (2) all basic messages it sent have been acknowledged.

Then $p$ starts a wave (of control messages), tagged with $t$ (and $p$).

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

If the wave completes, $p$ calls Announce.
Suppose a wave, tagged with some $t$, doesn’t complete.

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

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

When $p$ becomes quiet, it starts a new wave, tagged with a $t' > t$. 
Suppose a quiet process $q$ takes part in a wave, and is later on made active by a basic message from a process $p$ that wasn’t yet visited by this wave.

Then this wave won’t complete.

Namely, let the wave be tagged with $t$.

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

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

So $p$ won’t take part in the wave (because it is tagged with $t$).
What is a drawback of the Dijkstra-Scholten as well as Rana’s algorithm?

**Answer:** Requires one control message for every basic message.
Weight-throwing termination detection

Requires a **centralized** basic algorithm; allows a **directed** network.

The initiator has **weight** 1; noninitiators have weight 0.

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

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

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

When the **initiator** becomes **passive**, and has **regained weight** 1, it calls *Announce*.
Underflow: The weight of a process can become too small to be divided further.

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

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

Solution 2: The process initiates a weight-throwing termination detection sub-call, and only returns its weight to the initiator when (1) it has become passive, and (2) this sub-call has terminated.
The following centralized termination detection algorithm allows a decentralized basic algorithm and a directed network.

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

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

**Complication 2:** A traversal of only passive processes doesn’t guarantee termination (even if there are no basic messages in the channels).
The token is at $p_0$; only $s$ is active.

The token travels to $r$.

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

$s$ becomes passive.

The token travels on to $p_0$, which falsely calls Announce.
Safra’s algorithm

Allows a decentralized basic algorithm and a directed network.

Each process maintains a counter of type \( \mathbb{Z} \); initially it is 0. At each outgoing/incoming basic message, the counter is increased/decreased.

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

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

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

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

- When $p_0$ is passive, it sends a white token with counter 0.
- A noninitiator only forwards the token when it is passive.
- When a **black** process receives the token, the process becomes white and the token black.
  The token will stay black for the rest of the round trip.
- Eventually the token returns to $p_0$, who waits until it is passive:
  - If the **token is white** and the sum of all counters is zero, $p_0$ calls *Announce*.
  - Else, $p_0$ sends a white token again.
Safra’s algorithm - Example

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

$s$ sends a basic message $m$ to $q$, setting the counter of $s$ to 1. $s$ becomes passive.

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

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

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

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

$q$ becomes passive.

After two more round trips of the token, $p_0$ calls Announce.
When the system has terminated,
  ▶ the token will color all processes white, and
  ▶ the counters of the processes sum up to zero.

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

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

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

So, since the counter is 0:
  ▶ no process was made active after the token’s visit, and
  ▶ no messages are in transit.
Any suggestions for an optimization of Safra’s algorithm?

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

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

When $p$ becomes passive, it sends a fresh token, tagged with $p$. 
Garbage collection

Processes are provided with memory.

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

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

**Aim of garbage collection:** To reclaim inaccessible objects.

Three operations by processes to build or delete a reference:

- **Creation:** The object owner sends a pointer to another process.
- **Duplication:** A process that isn’t object owner sends a reference to another process.
- **Deletion:** The reference is deleted at its process.
Reference counting tracks the number of references to an object.

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

**Advantage:** Can be performed at run-time.

**Drawback:** Can’t reclaim *cyclic* garbage.
Indirect reference counting

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

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

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

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

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

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

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

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

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

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

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

If the total weight of the object becomes equal to its partial weight, and there are no pointers to the object, it can be reclaimed.
When the weight of a reference (or object) becomes too small to be divided further, no more duplication (or creation) is possible.

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

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

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

Duplicated references are then to the artificial object, so that references to the original object become *indirect*. 
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 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.
Indirect reference counting $\Rightarrow$ Dijkstra-Scholten termination detection.

Weighted reference counting $\Rightarrow$ weight-throwing termination detection.
**Mark-scan** garbage collection consists of two phases:

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

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

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

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

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

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

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

- Garbage in the young generation is collected frequently with mark-copy.
- Garbage in the old generation is collected sporadically with mark-compact.
This lecture in a nutshell

Termination detection
- Dijkstra-Scholten algorithm
- Shavit-Francez algorithm
- Rana’s algorithm
- weight throwing
- Safra’s algorithm

Garbage collection ⇒ termination detection
- indirect reference counting
- weighted reference counting
- mark-scan
- generational garbage collection
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 $u$:
Packets with destination $v$ that arrive at $u$ are passed on to $w$.

Criteria for good routing algorithms:
- use of optimal paths
- robust with respect to topology changes in the network
- cope with very large, dynamic networks
- table adaptation to avoid busy edges
Chandy-Misra shortest path algorithm

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

A centralized algorithm to compute shortest paths to initiator $u_0$.

Initially,
\begin{itemize}
  \item $dist_{u_0}(u_0) = 0$
  \item $dist_v(u_0) = \infty$ if $v \neq u_0$
  \item $parent_v(u_0) = \bot$ for all $v$
\end{itemize}

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

Let node $v$ receives $\langle d \rangle$ from neighbor $w$. If $d + \omega_{vw} < dist_v(u_0)$, then:
\begin{itemize}
  \item $dist_v(u_0) \leftarrow d + \omega_{vw}$ and $parent_v(u_0) \leftarrow w$
  \item $v$ sends $\langle dist_v(u_0) \rangle$ to its neighbors (except $w$)
\end{itemize}

Termination detection by e.g. the Dijkstra-Scholten algorithm.
Why is Rana’s algorithm not a good choice for detecting termination?

**Answer:** Nodes tend to become quiet, and start a wave, often.
Chandy-Misra algorithm - Example

\[
\begin{align*}
\text{dist}_{u_0} & \leftarrow 0 & \text{parent}_{u_0} & \leftarrow \bot \\
\text{dist}_w & \leftarrow 6 & \text{parent}_w & \leftarrow u_0 \\
\text{dist}_v & \leftarrow 7 & \text{parent}_v & \leftarrow w \\
\text{dist}_x & \leftarrow 8 & \text{parent}_x & \leftarrow v \\
\text{dist}_x & \leftarrow 7 & \text{parent}_x & \leftarrow w \\
\text{dist}_v & \leftarrow 4 & \text{parent}_v & \leftarrow u_0 \\
\text{dist}_w & \leftarrow 5 & \text{parent}_w & \leftarrow v \\
\text{dist}_x & \leftarrow 6 & \text{parent}_x & \leftarrow w \\
\text{dist}_x & \leftarrow 5 & \text{parent}_x & \leftarrow v \\
\text{dist}_x & \leftarrow 1 & \text{parent}_x & \leftarrow u_0 \\
\text{dist}_w & \leftarrow 2 & \text{parent}_w & \leftarrow x \\
\text{dist}_v & \leftarrow 3 & \text{parent}_v & \leftarrow w \\
\text{dist}_v & \leftarrow 2 & \text{parent}_v & \leftarrow x
\end{align*}
\]
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.

(At most $2 \cdot (N - 1)$ messages per edge.)
Merlin-Segall shortest path algorithm

A centralized algorithm to compute shortest paths to initiator $u_0$.

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

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

1. Let node $v$ get $\langle d \rangle$ from neighbor $w$.
   
   If $d + \omega_{vw} < dist_v(u_0)$, then $dist_v(u_0) \leftarrow d + \omega_{vw}$
   
   (and $v$ stores $w$ as future value for $parent_v(u_0)$).

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

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

$u_0$ starts a new round after receiving a message from all neighbors.
After $i$ rounds, all shortest paths of $\leq i$ hops have been computed.

So the algorithm can \textit{terminate} after $N - 1$ rounds.

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

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

Because all $N - 1$ rounds take $2 \cdot E$ messages.

No separate termination detection is needed.
Merlin-Segall algorithm - Example (round 1)
Merlin-Segall algorithm - Example (round 2)
Merlin-Segall algorithm - Example (round 3)
A number is attached to distance messages.

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

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

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

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

Example:

$x$ informs $u_0$ (via $v$) that an edge of the sink tree has failed.
Toueg’s all-pairs shortest path algorithm

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

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

$$
\begin{align*}
d^S(u, u) &= 0 \\
d^\emptyset(u, v) &= \omega_{uv} \text{ if } u \neq v \text{ and } uv \in E \\
d^\emptyset(u, v) &= \infty \text{ if } u \neq v \text{ and } uv \not\in E \\
d^{S\cup\{w\}}(u, v) &= \min\{ d^S(u, v), d^S(u, w) + d^S(w, v) \} \quad (w \not\in S)
\end{align*}
$$

When $S$ contains all nodes, $d^S$ is the standard distance function.
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 $w \not\in S$ 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:** $\Theta(N^3)$

For all $N$ pivots, $N^2$ equations need to be calculated.
Question

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

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

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

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

Initially, at each node $u$:

- $S_u = \emptyset$

- $\text{dist}_u(u) = 0$ and $\text{parent}_u(u) = \bot$

- For each $v \neq u$, either
  
  $\text{dist}_u(v) = \omega_{uv}$ and $\text{parent}_u(v) = v$ if there is an edge $uv$, or
  
  $\text{dist}_u(v) = \infty$ and $\text{parent}_u(v) = \bot$ otherwise
Toueg’s algorithm

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

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

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

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

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

If $u \neq w$, it waits for the values $\text{dist}_w(v)$ from $\text{parent}_u(w)$.

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

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

$$\text{dist}_u(w) + \text{dist}_w(v) < \text{dist}_u(v).$$

If so, $\text{dist}_u(v) \leftarrow \text{dist}_u(w) + \text{dist}_w(v)$ and $\text{parent}_u(v) \leftarrow \text{parent}_u(w)$.

Finally, $u$ proceeds to the next pivot round, with $S_u \leftarrow S_u \cup \{w\}$. 
<table>
<thead>
<tr>
<th>Pivot</th>
<th>$dist_x(v)$</th>
<th>$dist_v(x)$</th>
<th>Parent</th>
<th>$parent_x(v)$</th>
<th>$parent_v(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>5</td>
<td>5</td>
<td>$u$</td>
<td>$u$</td>
<td>$u$</td>
</tr>
<tr>
<td>$v$</td>
<td>5</td>
<td>5</td>
<td>$v$</td>
<td>$v$</td>
<td>$v$</td>
</tr>
<tr>
<td>$w$</td>
<td>2</td>
<td>2</td>
<td>$w$</td>
<td>$w$</td>
<td>$w$</td>
</tr>
<tr>
<td>$x$</td>
<td>2</td>
<td>2</td>
<td>$x$</td>
<td>$x$</td>
<td>$x$</td>
</tr>
</tbody>
</table>

Here, $dist(v)$ represents the distance from node $v$ to itself, and $parent(v)$ represents the parent of node $v$ in the tree structure.
Toueg’s algorithm - Complexity + drawbacks

Message complexity: \( O(N^2) \)

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

Drawbacks:

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

Which addition needs to be made to the algorithm to allow that nodes can discard the routing tables of pivots at some point?

Answer: Next to ⟨request, w⟩, u informs all its other neighbors that they do not need to forward w’s routing table to u.

Then the message complexity increases to $O(N \cdot E)$. 
Let $parent_u(w) = x$ with $x \neq w$ at the start of the $w$-pivot round.

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

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

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

Such a change is eventually detected by its neighbors.

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

Each node locally computes shortest paths.

(E.g. with the Bellman-Ford algorithm, if links can have negative weights.)
In link-state routing, nodes periodically sends a link-state packet, with
- the node’s edges and their weights (based on latency, bandwidth)
- a sequence number (which increases with each broadcast)

Link-state packets are flooded through the network.

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

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

Each node locally computes shortest paths (e.g. with Dijkstra’s alg.).
Flooding entire routing tables (instead of only edges and weights) tends to produce a less efficient algorithm.

Why is that?

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

And, of course, the bit complexity increases dramatically.
When a node recovers from a crash, its sequence number starts at 0. So its link-state packets may be ignored for a long time.

Therefore link-state packets carry a time-to-live (TTL) field. After this time the information from the packet may be discarded by a packet with a lower sequence number.

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

When it becomes 0, the packet is discarded.
The **RIP protocol** employs *distance-vector routing*.

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

Distance-vector and link-state routing don’t scale to the Internet, because of sending entire routing tables, or flooding.

Therefore the Internet is divided into **autonomous systems**.

Each autonomous system uses the RIP or OSPF protocol.
The **Border Gateway Protocol** routes between autonomous systems.

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

A router may update its routing table

- because it detects a topology change, or
- because of an update in the routing table of a neighbor.
Routing tables to guide a packet to its destination

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

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

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

Link-state / distance-vector routing and the border gateway protocol employ classical routing algorithms on the Internet
Consider an *undirected*, *unweighted* network.

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

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

The following centralized algorithm requires $O(N \cdot \sqrt{E})$ messages (for each root).
Breadth-first search - A “simple” algorithm

Initially (after round 0), the initiator is at distance 0, noninitiators are at distance $\infty$, and parents are undefined.

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

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

In round $n + 1$:

```
forward/reverse

explore/reverse

explore/reverse
```

0

$n$

$n + 1$
Breadth-first search - A “simple” algorithm

- Messages \(\langle \text{forward}, n \rangle\) travel down the tree, from the initiator to nodes at distance \(n\).

- When a node at distance \(n\) gets \(\langle \text{forward}, n \rangle\), it sends \(\langle \text{explore}, n + 1 \rangle\) to neighbors that aren’t at distance \(n - 1\).

Let a node \(v\) receive \(\langle \text{explore}, n + 1 \rangle\).

- If \(\text{dist}_v = \infty\), then \(\text{dist}_v \leftarrow n + 1\), the sender becomes \(v\)’s parent, and \(v\) sends back \(\langle \text{reverse}, \text{true} \rangle\).

- If \(\text{dist}_v = n + 1\), then \(v\) stores that the sender is at distance \(n\), and \(v\) sends back \(\langle \text{reverse}, \text{false} \rangle\).

- If \(\text{dist}_v = n\), then this is a negative ack for the \(\langle \text{explore}, n + 1 \rangle\) that \(v\) sends into this edge.
Breadth-first search - A “simple” algorithm

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

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

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

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

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

In round $n + 2$, nodes only send a forward to children that reported newly discovered nodes in round $n + 1$. 
Breadth-first search - Complexity

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

There are at most \( N \) rounds.

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

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

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

Round \( n \) is completed in at most \( 2 \cdot n \) time units, for \( n = 1, \ldots, N \).
Frederickson’s iterative deepening algorithm

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

Initially, the initiator is at distance 0, noninitiators are at distance $\infty$, and parents are undefined.

After round $n$, the tree has been constructed up to depth $\ell \cdot n$.

In round $n + 1$:

- $\langle \text{forward}, \ell \cdot n \rangle$ travels down the tree, from the initiator to nodes at distance $\ell \cdot n$.

- When a node at distance $\ell \cdot n$ gets $\langle \text{forward}, \ell \cdot n \rangle$, it sends $\langle \text{explore}, \ell \cdot n + 1 \rangle$ to neighbors that aren’t at distance $\ell \cdot n - 1$. 
Frederickson’s algorithm - Complications

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

How can this happen?

Which complication may arise as a result? How can it be resolved?

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

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

How can this happen? How can it be resolved?

Solution: $w$ can dismiss this forward.

In the previous round, $w$ informed $v$ that it is no longer a child.
Frederickson’s algorithm

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

- $k < dist_v$:
  
  $dist_v \leftarrow k$, and the sender becomes $v$’s parent.

  If $\ell$ doesn’t divide $k$, then $v$ sends $\langle \text{explore}, k + 1 \rangle$ to its other neighbors.

  If $\ell$ divides $k$, then $v$ sends back $\langle \text{reverse}, k, \text{true} \rangle$.

- $k \geq dist_v$:

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

  Else $v$ doesn’t send a reply (because it sent $\langle \text{explore}, dist_v + 1 \rangle$ into this edge).
Frederickson’s algorithm

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

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

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

Then it sends $\langle \text{reverse}, b \rangle$ to its parent, where $b = \text{true}$ if and only if it received $\langle \text{reverse}, \_, \text{true} \rangle$ from a child.
Frederickson’s algorithm

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

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

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

In round $n + 2$, nodes only send a *forward* to children that reported newly discovered nodes in round $n + 1$. 
Apply Frederickson’s algorithm to the network below, with initiator $u_0$ and $\ell = 2$:

Give a computation in which:

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

This network diagram shows the connections between nodes, which are necessary for understanding the algorithm's application and the specific conditions mentioned.
Frederickson’s algorithm - Complexity

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

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

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

In total, edges carry at most 2\( \ell \) explore’s and 2\( \ell \) replying reverse’s.
(In total, frond edges carry at most 1 spurious forward.)

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

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

If \( \ell = \lceil \frac{N}{\sqrt{E}} \rceil \), both message and time complexity are \( O(N \cdot \sqrt{E}) \).
What is the optimal value of $\ell$ in case the network is:

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

Hint: Consider the (bounded) memory at the processes.
A **store-and-forward deadlock** occurs when a group of packets are all waiting for the use of a buffer slot occupied by a packet in the group.

A **controller** avoids such deadlocks.

It prescribes whether a packet can be generated or forwarded, and in which buffer slot it is put next.
Consider an undirected network, supplied with routing tables.

Processes store data packets traveling to their destination in buffers.

Possible events:

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

At a node with an empty buffer, packet generation must be allowed.
For simplicity we assume *synchronous* communication.

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

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

**Answer:** A packet can only be eliminated by the sender when its reception has been acknowledged.
The network consists of nodes $u_0, \ldots, u_{N-1}$.

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

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

- When a packet with destination $u_i$ is generated at $v$, it is placed in the $i^{\text{th}}$ buffer slot of $v$.

- If $vw$ is an edge in $T_i$, then the $i^{\text{th}}$ buffer slot of $v$ is linked to the $i^{\text{th}}$ buffer slot of $w$. 
**Theorem:** The destination controller is deadlock-free.

**Proof:** Consider a reachable configuration $\gamma$.

Make forwarding and consumption transitions to a configuration $\delta$ where no forwarding or consumption is possible.

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

So in $\delta$, all buffers are empty.
Hops-so-far controller

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

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

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

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

- A generated packet is placed in the $0^{th}$ buffer slot.

- For each edge $vw$ and any $j < K$, the $j^{th}$ buffer slot of $v$ is linked to the $(j+1)^{th}$ buffer slot of $w$, and vice versa.
Theorem: The hops-so-far controller is deadlock-free.

Proof: Consider a reachable configuration $\gamma$.

Make forwarding and consumption transitions to a configuration $\delta$ where no forwarding or consumption is possible.

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

So in $\delta$, $K^{th}$ buffer slots are all empty.

Suppose all $i^{th}$ buffer slots are empty in $\delta$, for some $1 \leq i \leq K$.
Then all $(i-1)^{th}$ buffer slots must also be empty in $\delta$.
For else some packet in an $(i-1)^{th}$ buffer slot could be forwarded or consumed.

Concluding, in $\delta$ all buffers are empty.
Consider an undirected network.

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

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

An acyclic orientation cover of \( \mathcal{P} \) consists of acyclic orientations \( G_0, \ldots, G_{n-1} \) such that each path in \( \mathcal{P} \) is the concatenation of paths \( P_0, \ldots, P_{n-1} \) in \( G_0, \ldots, G_{n-1} \).
For each undirected ring there exists a cover, consisting of three acyclic orientations, of the collection of minimum-hop paths.

For instance, in case of a ring of size six:
Acyclic orientation cover controller

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

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

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

- A generated packet is placed in the $0^{th}$ buffer slot.

- Let $vw$ be an edge in $G_i$.

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

  If $i < n-1$, then the $i^{th}$ buffer slot of $w$ is linked to the $(i+1)^{th}$ buffer slot of $v$. 
Consider a packet $\pi$; it is routed via the sink tree of its destination.

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

While $\pi$ is in an $i^{th}$ slot with $i < n - 1$ and not yet at its destination, it can be forwarded to the $i^{th}$ or $(i + 1)^{th}$ slot at the next node.

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

Then $\pi$ can be routed to its destination via $(n - 1)^{th}$ buffer slots.
For each undirected ring there exists a deadlock-free controller that:

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

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

Show how the resulting acyclic orientation cover controller links buffer slots.
Theorem: Let all packets be routed via paths in $\mathcal{P}$. Then the acyclic orientation cover controller is deadlock-free.

Proof: Consider a reachable configuration $\gamma$. Make forwarding and consumption transitions to a configuration $\delta$ where no forwarding or consumption is possible.

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

Suppose all $i^{th}$ buffer slots are empty in $\delta$, for some $i = 1, \ldots, n-1$. Then all $(i-1)^{th}$ buffer slots must also be empty in $\delta$.

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

Concluding, in $\delta$ all buffers are empty.
Slow-start algorithm in TCP

Back to the asynchronous, pragmatic world of the Internet.

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

It is the maximum number of unacknowledged packets on this edge.
The congestion window grows \textit{linearly} with each received ack, up to some threshold.

\textbf{Question}: Explain why the congestion window may \textit{double} with every \textquotedblleft round trip time\textquotedblright.

The congestion window is \textit{reset to the initial size} (in TCP Tahoe) or \textit{halved} (in TCP Reno) with each lost data packet.
Take-home messages of the current lecture

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

iterative deepening (a la Frederickson’s alg.)

optimization of a parameter ($\ell$) based on a complexity analysis

importance of deadlock-free packet switching

acyclic orientation cover controller

congestion window in TCP
Election algorithms

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

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

Assumptions:

- The algorithm is decentralized:
  The initiators can be any non-empty set of processes.

- All processes have the same local algorithm.

- Process id’s are unique, and from a totally ordered set.
Chang-Roberts algorithm

Consider a directed ring.

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

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

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

*Passive* processes (including all noninitiators) pass on messages.

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

**Average-case message complexity:** $O(N \cdot \log N)$
Chang-Roberts algorithm - Example

All processes are initiators.

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

clockwise: \( 2 \cdot N - 1 \) messages
Franklin’s algorithm

Consider an undirected ring.

Each active process \( p \) repeatedly compares its own id with the id’s of its nearest active neighbors on both sides.

If such a neighbor has a larger id, then \( p \) becomes passive.

Initially, initiators are active, and noninitiators are passive.

Each round, an active process \( p \):

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

Passive processes pass on incoming messages.
Franklin’s algorithm - Complexity

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

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

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

Each round takes \( 2 \cdot N \) messages.

**Question:** Give an example with \( N = 4 \) that takes three rounds.

**Question:** Show that for any \( N \) there is a ring that takes two rounds.
Franklin’s algorithm - Example

Suppose this ring is *directed*, in the clockwise direction.

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

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

\[ \text{--->s--->q--->p--->r--->t--->} \]

- 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:** $O(N \cdot \log N)$
Consider the following clockwise oriented ring.
Question: How can the tree algorithm be used to make the process with the largest id in an undirected, acyclic network the leader?

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

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

- Initially, initiators send a wake-up message to all neighbors.
- When a noninitiator receives a first wake-up message, it wakes up and sends a wake-up message to all neighbors.
- When a process has received a wake-up message from all its neighbors, it starts the election phase.
The local election algorithm at a process $p$:

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

Message complexity: $2 \cdot N - 2$ messages (without the wake-up phase)
In case a process $p$ receives a parent request from its parent, why does it need to recompute $\max_p$?
The wake-up phase is omitted.

Tree election algorithm - Example
Echo algorithm with extinction

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

Non-initiators join the first wave that hits them.

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

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

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

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

Worst-case message complexity: $O(N \cdot E)$
Consider an **undirected, weighted** network.

We assume that different edges have different weights.

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

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

**Example:**

![Diagram of a minimum spanning tree](image)
Lemma: Let $F$ be a fragment (i.e., a connected subgraph of the minimum spanning tree $M$).

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

Then $e$ is in $M$.

Proof: Suppose not.

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

Replacing $f$ by $e$ in $M$ gives a spanning tree with a smaller sum of weights of edges.
Kruskal’s algorithm

A *uniprocessor* algorithm for computing minimum spanning trees.

- Initially, each node forms a separate fragment.

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

This algorithm also works when edges have the same weight.

Then the minimum spanning tree may not be unique.
Gallager-Humblet-Spira algorithm

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

Distributed computation of a minimum spanning tree:

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

Complications: Is an edge lowest-weight? Is it outgoing?
Each fragment carries a (unique) name \( fn : \mathbb{R} \) and a level \( \ell : \mathbb{N} \).

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

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

\[
\begin{align*}
\ell < \ell' & \land F \xrightarrow{e_F} F' : \quad F \cup F' = (fn', \ell') \\
\ell = \ell' & \land e_F = e_{F'} : \quad F \cup F' = \text{(weight } e_F, \ell + 1) 
\end{align*}
\]

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**.
What fragment results if two processes at level 0 connect to each other?

**Answer:** A single core edge, with its weight as name, and level 1.
Parameters of a process

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

The *status* of its *channels*:
- **basic edge** (undecided)
- **branch edge** (in the spanning tree)
- **rejected** (not in the spanning tree)

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

Its *parent* (toward the core edge).
Non-initiators wake up when they receive a (connect or test) message.

Each initiator, and noninitiator after it has woken up:

- sets its level to 0
- sets its lowest-weight edge to branch
- sends $\langle$connect, 0$\rangle$ into this channel
- sets its other channels to basic
- sets its state to found
Joining two fragments

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

- If $\ell < \ell'$, then $p$ sent $\langle \text{connect}, \ell \rangle$ to $q$.
  
  $q$ sends $\langle \text{initiate}, fn', \ell', \text{find} \rangle$ to $p$.
  
  $F \cup F'$ inherits the core edge of $F'$.

- If $\ell = \ell'$, then $p$ and $q$ sent $\langle \text{connect}, \ell \rangle$ to each other.
  
  They send $\langle \text{initiate}, \text{weight } pq, \ell + 1, \text{find} \rangle$ to each other.
  
  $F \cup F'$ gets core edge $pq$.

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

It passes on the message through its other branch edges.
Computing the lowest-weight outgoing edge

In case of \texttt{initiate, fn, \ell, find}, \( p \) checks in increasing order of weight if one of its \textit{basic} edges \( pq \) is \textit{outgoing}, by sending \( \texttt{test, fn, \ell} \) to \( q \).

While \( \ell > level_q \), \( q \) \textit{postpones} processing the incoming \texttt{test} message.

Let \( \ell \leq level_q \).

- If \( q \) is in fragment \( fn \), then \( q \) replies \texttt{reject}.
  
  In this case \( p \) and \( q \) will set \( pq \) to \texttt{rejected}.

- Else, \( q \) replies \texttt{accept}.

When a basic edge is accepted, or there are no basic edges left, \( p \) \textit{stops} the search.
Why does $q$ postpone processing the incoming $\langle \text{test}, _, \ell \rangle$ message from $p$ while $\ell > \text{level}_q$?

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

Why does this postponement not lead to a deadlock?

**Answer:** There is always a fragment with a smallest level.
Reporting to the core nodes

- $p$ waits for all its branch edges, except its parent, to report.

- $p$ sets its state to *found*.

- $p$ computes the minimum $\lambda$ of (1) these reports, and (2) the weight of its lowest-weight outgoing basic edge (or $\infty$, if no such channel was found).

- If $\lambda < \infty$, $p$ stores either the branch edge that sent $\lambda$, or its basic edge of weight $\lambda$.

- $p$ sends $\langle \text{report}, \lambda \rangle$ 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 $\mu$ first sends changeroot toward the lowest-weight outgoing basic edge.

  (The core edge becomes a regular tree edge.)

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

$p$ sets this channel to branch, and sends $\langle \text{connect}, \text{level}_p \rangle$ into it.
When \( q \) receives \( \langle \text{connect}, \text{level}_p \rangle \) from \( p \), \( \text{level}_q \geq \text{level}_p \).

Namely, either \( \text{level}_p = 0 \), or \( q \) earlier sent **accept** to \( p \).

- If \( \text{level}_q > \text{level}_p \), then \( q \) sets \( \text{qp} \) to *branch* and sends \( \langle \text{initiate}, \text{name}_q, \text{level}_q, \frac{\text{find}}{\text{found}} \rangle \) to \( p \).

- As long as \( \text{level}_q = \text{level}_p \) and \( \text{qp} \) isn’t a branch edge, \( q \) postpones processing the **connect** message.

- If \( \text{level}_q = \text{level}_p \) and \( \text{qp} \) is a branch edge (meaning that \( q \) sent \( \langle \text{connect}, \text{level}_q \rangle \) to \( p \)), then \( q \) sends \( \langle \text{initiate}, \text{weight} \text{qp}, \text{level}_q + 1, \text{find} \rangle \) to \( p \) (and vice versa).

In this case \( \text{pq} \) becomes the core edge.
If $level_q = level_p$ and $qp$ isn’t a branch edge, why does $q$ postpone processing the incoming `connect` message from $p$?

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

Then the fragment of $p$ should subsume the name and level of that joint fragment, instead of weight $pq$ and $level_p + 1$.

Why does this postponement not give rise to a deadlock?

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

**Answer:** Because different channels have different weights.
Suppose a process reported a lowest-weight outgoing basic edge, and in return receives $\langle$\textit{initiate}, $fn$, $\ell$, $find\rangle$.

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

\textbf{Answer:} Its fragment may in the meantime have joined the fragment at the other side of this basic edge.
Gallager-Humblet-Spira algorithm - Example

$pq \quad qp \quad \langle \text{connect}, 0 \rangle$

$pq \quad qp \quad \langle \text{initiate}, 5, 1, \text{find} \rangle$

$ps \quad qr \quad \langle \text{test}, 5, 1 \rangle$

$tq \quad \langle \text{connect}, 0 \rangle$

$qt \quad \langle \text{initiate}, 5, 1, \text{find} \rangle$

$tq \quad \langle \text{report}, \infty \rangle$

$rs \quad sr \quad \langle \text{connect}, 0 \rangle$

$rs \quad sr \quad \langle \text{initiate}, 3, 1, \text{find} \rangle$

$sp \quad rq \quad \text{accept}$

$pq \quad \langle \text{report}, 9 \rangle$

$qp \quad \langle \text{report}, 7 \rangle$

$qr \quad \langle \text{connect}, 1 \rangle$

$sp \quad rq \quad \langle \text{test}, 3, 1 \rangle$

$ps \quad qr \quad \text{accept}$
Worst-case message complexity:  \( O(E + N \cdot \log N) \)

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

Between two subsequent joins, a process:

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

A fragment at level \( \ell \) contains \( \geq 2^\ell \) processes.

So each process experiences at most \( \lfloor \log_2 N \rfloor \) joins.
By two extra messages at the very end, the core node with the largest id becomes the leader.


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

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

\[ \Omega(E + N \cdot \log N) \]
Leader election

Decentralized / uniform local algorithm / unique process id’s

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

Tree election algorithm

Echo algorithm with extinction

Gallager-Humblet-Spira minimum spanning tree algorithm
In an anonymous network, processes (and channels) have no unique id.

Processes may be anonymous for several reasons:

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

**Question**: Suppose there is one leader. How can each process be provided with a unique id?
**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 \( \gamma_0 \) is symmetric and \( \gamma_0 \rightarrow \gamma_1 \), then there are transitions \( \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_N \) with \( \gamma_N \) symmetric.

In a symmetric configuration there isn’t one leader.

So there is an infinite computation in which no leader is elected.
An execution is **fair** if each event that is applicable in infinitely many configurations, occurs infinitely often in the computation.

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

(Basically because in the proof, $\gamma_0 \to \gamma_1$ can be chosen freely.)
In a probabilistic algorithm, a process may flip a coin, and perform an event based on the outcome of this coin flip.

Probabilistic algorithms where all computations terminate in a correct configuration aren’t so interesting.

Because letting the coin e.g. always flip heads yields a correct non-probabilistic algorithm.
Las Vegas and Monte Carlo algorithms

A probabilistic algorithm is Las Vegas if:

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

It is Monte Carlo if:

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

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

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

Give a Las Vegas algorithm that terminates with probability 1.
Given an anonymous, directed ring. All processes know the ring size $N$.

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

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

Complication: Different processes may select the same id.

Solution: Each message is supplied with a hop count. A message that arrives at its source has hop count $N$.

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

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

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

The 3rd value is the hop count. The 4th value signals if another process with the same id was encountered during the round trip. If $p$ receives:

- $(n', i, h, b)$ with $n' > n$, or $n' = n$ and $i > id_p$:
  
  $p$ becomes passive and sends $(n', i, h + 1, b)$.

- $(n', i, h, b)$ with $n' < n$, or $n' = n$ and $i < id_p$:
  
  $p$ dismisses the message.

- $(n, id_p, h, b)$ with $h < N$:
  
  $p$ sends $(n, id_p, h + 1, true)$.

- $(n, id_p, N, true)$:
  
  $p$ proceeds to round $n + 1$.

- $(n, id_p, N, false)$:
  
  $p$ becomes the leader.

*Passive* processes pass on messages, increasing their hop count by one.
Question: How can an infinite computation occur?

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

Without round numbers, the algorithm would be flawed.

Example:

\[
\begin{array}{c}
\text{i} \\
\text{j} \\
\text{k} \\
\end{array} \quad \langle j, 1, false \rangle \quad \begin{array}{c}
\text{i} \\
\text{j} \\
\text{k} \\
\end{array} \quad \langle j, 1, false \rangle
\]

\[i > j\quad j > k, \ell\]

Average-case message complexity: \[O(N \cdot \log N)\]
Election in arbitrary anonymous networks

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

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

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

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

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

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

All other messages report 0.

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

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

Else, it selects a new id, and initiates a new wave, in the next round.
Election in arbitrary anonymous networks - Example

\[ i > j > k > \ell > m. \]

Only waves that complete are shown.

The process at the left computes size 6, and becomes the leader.
Is there another scenario in which the right-hand side node progresses to round 2?
Theorem: There is no Las Vegas algorithm to compute the size of an anonymous ring.

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

Because when a leader is known, the network size can be computed using a centralized wave algorithm with the leader as initiator.
Theorem: There is no Las Vegas algorithm to compute the size of an anonymous ring.

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

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

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

Let each event at a $p_i$ in $C$ be executed concurrently at $p_i$ and $p_{i+N}$. This computation terminates with the incorrect outcome $N$. 
Itai-Rodeh ring size algorithm

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

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

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

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

\[(i, 2) \rightarrow (j, 2) \rightarrow (i, 2) \rightarrow (k, 2) \rightarrow (i, 2) \rightarrow (\ell, 3) \rightarrow (m, 3) \rightarrow (\ell, 3) \rightarrow (\ell, 3) \rightarrow (i, 2) \rightarrow (i, 2) \rightarrow (\ell, 3) \rightarrow (j, 4) \rightarrow (j, 4)\]
Upon message-termination, is \( est_p \) always the same at all \( p \)?
The probability of computing an incorrect ring size tends to zero when $R$ tends to infinity.

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

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

Each round they send a message, which takes at most $N$ steps.
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.
The IEEE 1394 standard is a serial multimedia bus.

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

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

The network size is unknown to the processes.

The tree algorithm for undirected, acyclic networks is used.

Networks that contain a cycle give a time-out.
IEEE 1394 election algorithm

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

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

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

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

Impossibility of election in anonymous networks

Las Vegas / Monte Carlo algorithms

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

Echo election algorithm for anonymous networks (Las Vegas)

No Las Vegas algorithm for computing anonymous network size

Itai-Rodeh ring size algorithm (Monte Carlo)

IEEE 1394 election algorithm
A process may \textit{crash}, i.e., unexpectedly stop executing events.

\textbf{Assumption:} The network is \textit{complete}, i.e., there is a bidirectional channel between each pair of different processes.

So process crashes never make the remaining network disconnected.

\textbf{Assumption:} Crashing of processes can’t be observed.
**Binary consensus**: Initially, all processes randomly select 0 or 1.

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

Consensus underlies many important problems in distributed computing: termination detection, mutual exclusion, leader election, ...
Validity: If all processes randomly select the same initial value $b$, then all alive processes decide $b$.

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

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

Each 1-crash consensus algorithm has a bivalent initial configuration, due to validity.

From such a configuration one reach terminal configurations with a decision 0 as well as with a decision 1.
Theorem: No algorithm for 1-crash consensus always terminates.

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

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

---

Impossibility of 1-crash consensus
A set $S$ of processes is called $b$-potent, in a configuration, if by only executing events at processes in $S$, some process in $S$ can decide $b$.

**Question**: Consider any $k$-crash consensus algorithm. Why should each set of $N - k$ processes be $b$-potent for some $b$?
Impossibility of 1-crash consensus

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

**Proof:** Consider a 1-crash consensus algorithm.

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

- Let the transitions correspond to events at *different* processes. Then $\gamma_0 \rightarrow \delta \leftarrow \gamma_1$ for some $\delta$. So $\gamma_0$ or $\gamma_1$ is bivalent.

- Let the transitions correspond to events at *one* process $p$. In $\gamma$, $p$ can crash, so the other processes are $b$-potent for some $b$. Likewise for $\gamma_0$ and $\gamma_1$. It follows that $\gamma_{1-b}$ is bivalent.

So each bivalent configuration has a transition to a bivalent configuration. Hence each bivalent initial configuration yields an infinite computation.

There exist *fair* infinite computations.
Let $N = 4$. At most one process can crash.

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

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

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

(Admittedly, this computation isn’t fair.)
Why is there no Las Vegas algorithm for $(N - 1)$-crash consensus?

**Answer:** Each process must then be able to decide by itself.

So if communication is very slow, eventually each alive process may decide for its own value.
Theorem: Let $k \geq \frac{N}{2}$. There is no Las Vegas algorithm for $k$-crash consensus.

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

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

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

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

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

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

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

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

With a (very small) positive probability all alive processes decide for the same value.
Bracha-Toueg crash consensus algorithm

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

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

If $p$ decides $b$, it broadcasts $\langle n + 1, b, N - k \rangle$ and $\langle n + 2, b, N - k \rangle$, and terminates.
$N = 3$ and $k = 1$. Each round, alive processes require two incoming messages, and two $b$-votes with weight 2 to decide $b$. 

(Messages of a process to itself aren’t depicted.)
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 process receives a $\langle q, b, w \rangle$ with $w > \frac{N}{2}$.

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

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

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

Concluding, all alive processes decide for the same value.
Proof (part II): Assumption: Scheduling of messages is fair.

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

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

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

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

Concluding, the algorithm terminates with probability 1.
A failure detector at a process keeps track which processes have (or may have) crashed.

Given an upper bound on network latency, and heartbeat messages, one can implement a failure detector.

With a failure detector, the proof of impossibility of 1-crash consensus no longer applies.

For this setting, terminating crash consensus algorithms exist.
Aim: To detect *crashed* processes.

We assume a time domain, with a total order. 

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

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

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

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

Each computation is decorated with:

- a failure pattern \( F \)
- a failure detector history \( H \)
Complete failure detector

We require that failure detectors are complete:

From some time onward, each crashed process is suspected by each alive process.
A failure detector is **strongly accurate** if only crashed processes are ever suspected.

**Assumptions:**

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

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

This failure detector is **complete** and **strongly accurate**.
A failure detector is **weakly accurate** if *some* process is never suspected by any process.

Assume a complete and **weakly accurate** failure detector.

We give a **rotating coordinator** algorithm for \((N - 1)\)-crash consensus.
Processes are numbered: \( p_0, \ldots, p_{N-1} \).

Initially, each process randomly selects 0 or 1. In round \( n \):

- \( p_n \) (if alive) broadcasts its value.

- Each process waits:
  - either for an incoming message from \( p_n \), in which case it adopts the value of \( p_n \);
  - or until it suspects that \( p_n \) has crashed.

After round \( N - 1 \), each alive process decides for its value.

**Correctness:** Let \( p_j \) never be suspected.

After round \( j \), all alive processes have the same value \( b \).

Hence, after round \( N - 1 \), all alive 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 alive process broadcasts alive every $\nu$ time units.
- There is an unknown upper bound on network latency.

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

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

If $q$ receives a message from a suspected process $p$, then $p$ is no longer suspected and $d_q \leftarrow d_q + 1$.

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

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

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

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

Suppose that for a long time the processes in $S$ suspect the processes in $T$ crashed, and the processes in $T$ suspect the processes in $S$ crashed.

The processes in $S$ can then decide 0, while the process in $T$ can decide 1.
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 is used for $k$-crash consensus.

Each process $q$ records the last round $lu_q$ in which it updated $value_q$.

Initially, $value_q \in \{0, 1\}$ and $lu_q = -1$.

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

Round $n$ is coordinated by $p_c$ with $c = n \mod N$. 
Chandra-Toueg $k$-crash consensus algorithm

- In round $n$, each alive process $q$ sends $\langle \text{vote}, n, value_q, lu_q \rangle$ to $p_c$.

- $p_c$ (if alive) waits until $N - k$ such messages arrived, and selects one, say $\langle \text{vote}, n, b, \ell \rangle$, with $\ell$ as large as possible.

  $value_{p_c} \leftarrow b$, $lu_{p_c} \leftarrow n$, and $p_c$ broadcasts $\langle \text{value}, n, b \rangle$.

- Each alive process $q$ waits:
  - either until $\langle \text{value}, n, b \rangle$ arrives: then $value_q \leftarrow b$, $lu_q \leftarrow n$, and $q$ sends $\langle \text{ack}, n \rangle$ to $p_c$;
  - or until it suspects $p_c$ crashed: then $q$ sends $\langle \text{nack}, n \rangle$ to $p_c$.

- If $p_c$ receives $> k$ messages $\langle \text{ack}, n \rangle$, then $p_c$ decides $b$, and broadcasts $\langle \text{decide}, b \rangle$.

An undecided process that receives $\langle \text{decide}, b \rangle$, decides $b$. 
Theorem: Let $k < \frac{N}{2}$. The Chandra-Toueg algorithm is an (always terminating) $k$-crash consensus algorithm.

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

1. there are $> k$ processes $q$ with $lu_q \geq n$, and
2. $lu_q \geq n$ implies $value_q = b$.

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

This follows by induction on $m - n$.

By (1), in round $m$ the coordinator receives a vote with $lu \geq n$.

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

So from round $n$ onward, processes can only decide $b$. 
Proof (part II):

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

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

Since $N - k > k$, it decides.

All alive processes eventually receive the `decide` message of $p$, and also decide.
Chandra-Toueg algorithm - Example

$\langle \text{vote}, 0, 0, -1 \rangle \quad \text{lu} = -1$

$\langle \text{value}, 0, 0 \rangle \quad \text{lu} = -1$

$\langle \text{ack}, 0 \rangle \quad \text{lu} = 0$

$\langle \text{vote}, 1, 1, -1 \rangle$

$\langle \text{value}, 1, 0 \rangle \quad \text{lu} = 1$

$\langle \text{ack}, 1 \rangle$

$\langle \text{decide}, 0 \rangle \quad \text{lu} = 1$

$\langle \text{decide}, 0 \rangle \quad \text{lu} = 1$

$\langle \text{decide}, 0 \rangle \quad \text{lu} = 1$

$N = 3$ and $k = 1$

Messages and ack’s that a process sends to itself and 'irrelevant' messages are omitted.
Crashed processes

Complete network / crashes can’t be observed

(Binary) consensus

No algorithm for 1-crash consensus always terminates

If \( k \geq \frac{N}{2} \), there is no Las Vegas algorithm for \( k \)-crash consensus

Bracha-Toueg \( k \)-crash consensus algorithm for \( k < \frac{N}{2} \)
Complete failure detector

Strongly accurate failure detector

Rotating coordinator crash consensus algorithm with a weakly accurate failure detector

Eventually strongly accurate failure detector

$k$-crash consensus for $k \geq \frac{N}{2}$ remains impossible with an eventually strongly accurate failure detection

Chandra-Toueg crash consensus algorithm with an eventually weakly accurate failure detector
Processes contend to enter their *critical section*. A process (allowed to be) in its critical section is called *privileged*.

For each computation we require:

*Mutual exclusion:* Always at most one process is privileged.

*Starvation-freeness:* If a process $p$ tries to enter its critical section, and no process stays privileged forever, then $p$ eventually becomes privileged.

*Applications:* Distributed shared memory. Avoidance of race conditions.
Mutual exclusion with message passing

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

- **Logical clock:** Requests to enter a critical section are prioritized by means of logical time stamps.

- **Token passing:** The process holding the token is privileged.

- **Leader election:** A process that wants to become privileged sends a request to the leader.

- **Quorum:** To become privileged, a process needs permission from a quorum of processes.

Each pair of quorums has a non-empty intersection.
A process $p_i$ that wants to access its critical section sends $\text{request}(ts_i, i)$ to all other processes, with $ts_i$ its \textit{logical time stamp} with regard to Lamport’s clock.

When $p_j$ receives this request, it sends \textit{permission} to $p_i$ as soon as:

- $p_j$ isn’t privileged, and

- $p_j$ doesn’t have a pending request with time stamp $ts_j$ where $(ts_j, j) < (ts_i, i)$ (lexicographical order).

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

When $p_i$ exits its critical section, it sends permission to all pending requests.
$N = 2$, and $p_0$ and $p_1$ both are at logical time 0.

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

When $p_0$ receives this message, it sends permission to $p_1$, setting the time at $p_0$ to 2.

$p_0$ sends $request(2, 0)$ to $p_1$.

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

$p_1$ receives permission from $p_0$, and enters its critical section.
Ricart-Agrawala algorithm - Example 2

\[ N = 2, \text{ and } p_0 \text{ and } p_1 \text{ both are at logical time } 0. \]

\[ p_1 \text{ sends } request(0, 1) \text{ to } p_0, \text{ and } p_0 \text{ sends } request(0, 0) \text{ to } p_1. \]

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

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

\( p_0 \) receives permission from \( p_1 \), and enters its critical section.
Mutual exclusion: When $p$ sends permission to $q$:

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

(Because $p$’s pending or future request is larger than $q$’s current request.)

Starvation-freeness: Each request will eventually become the smallest request in the network.
**Drawback:** High message overhead, because requests must be sent to all other processes.

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

Suppose $q$ is waiting for permissions and didn’t send a request to $p$.

If $p$ sends a request to $q$ that is smaller than $q$’s request, then $q$ sends both permission and a request to $p$.

This optimization is correct since for each pair of distinct processes, at least one must ask permission from the other.
Let processes $p_0$, $p_1$, $p_2$ become privileged, in this order.

Next $p_0$ and $p_1$ concurrently want to become privileged again, with the same logical time stamp.

Give two scenario’s with the Carvalho-Roucairol optimization: one where $p_0$ and one where $p_1$ becomes privileged.

**Answer:** $p_0$ needs permission from $p_1$ and $p_2$; $p_1$ needs permission only from $p_2$.

If $p_0$’s request reaches $p_1$ before permission from $p_2$, then $p_1$ sends permission and a request to $p_0$.

Else $p_1$ enters its critical section, and answers $p_0$’s request after exiting its critical section.
Given an **undirected** network, with a **sink tree**.

At any time, the **root**, holding a **token**, is privileged.

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

**Queue maintenance:**

- When a non-root wants to enter its critical section, it adds its id to its own queue.
- When a non-root gets a new head at its (non-empty) queue, it asks its parent for the token.
- When a process receives a request for the token from a child, it adds this child to its queue.
Raymond’s algorithm

When the root exits its critical section (and its queue is non-empty),

- it sends the token to the process \( q \) at the head of its queue,
- makes \( q \) its parent, and
- removes \( q \) from the head of its queue.

Let \( p \) get the token from its parent, with \( q \) at the head of its queue:

- If \( q \neq p \), then \( p \) sends the token to \( q \), and makes \( q \) its parent.
- If \( q = p \), then \( p \) becomes the root (i.e., it has no parent, and is privileged).

In both cases, \( p \) removes \( q \) from the head of its queue.
Raymond’s algorithm - Example
Raymond’s algorithm - Example
Raymond’s algorithm - Example
Raymond’s algorithm - Example
Raymond’s algorithm - Example
Raymond’s algorithm - Example
Raymond’s algorithm - Example
Raymond’s algorithm - Example
Raymond’s algorithm - Example
Raymond’s algorithm provides **mutual exclusion**, because at all times there is at most one root.

Raymond’s algorithm is **starvation-free**, because eventually each request in a queue moves to the head of this queue, and a chain of requests never contains a cycle.

**Drawback**: Sensitive to failures.
What is the Achilles’ heel of a mutual exclusion algorithm based on a leader?

**Answer:** The leader is a single point of failure.
To enter a critical section, permission from a quorum is required.

For simplicity we assume that $N = 2^k - 1$, for some $k > 1$.

The processes are structured in a binary tree of depth $k - 1$.

A quorum consists of all processes on a path from the root to a leaf.

If a non-leaf $p$ has crashed (or is unresponsive), permission is asked from all processes on two paths instead: from each child of $p$ to a leaf.
Example: Let $N = 7$.

Possible quorums are:

- $\{1, 2, 4\}$, $\{1, 2, 5\}$, $\{1, 3, 6\}$, $\{1, 3, 7\}$
- if 1 crashed: $\{2, 4, 3, 6\}$, $\{2, 5, 3, 6\}$, $\{2, 4, 3, 7\}$, $\{2, 5, 3, 7\}$
- if 2 crashed: $\{1, 4, 5\}$ (and $\{1, 3, 6\}$, $\{1, 3, 7\}$)
- if 3 crashed: $\{1, 6, 7\}$ (and $\{1, 2, 4\}$, $\{1, 2, 5\}$)

Question: What are the quorums if 1,2 crashed? And if 1,2,3 crashed? And if 1,2,4 crashed?
A process $p$ that wants to enter its critical section, places the root of the tree in a queue.

$p$ repeatedly tries to get permission from the head $r$ of its queue.

If successful, $r$ is removed from $p$’s queue.

If $r$ is a non-leaf, one of $r$’s children is appended to $p$’s queue.

If non-leaf $r$ has crashed, it is removed from $p$’s queue, and both of $r$’s children are appended at the end of the queue (in a fixed order, to avoid deadlocks).

If leaf $r$ has crashed, $p$ aborts its attempt to become privileged.

When $p$’s queue becomes empty, it enters its critical section.

After exiting its critical section, $p$ informs all processes in the quorum that their permission to $p$ can be withdrawn.
$p$ and $q$ concurrently want to enter their critical section.

$p$ gets permission from 1, and wants permission from 3.

1 crashes, and $q$ now wants permission from 2 and 3.

$q$ gets permission from 2, and appends 4 to its queue.

$q$ obtains permission from 3, and appends 7 to its queue.

3 crashes, and $p$ now wants permission from 6 and 7.

$q$ gets permission from 4, and now wants permission from 7.

$p$ gets permission from both 6 and 7, and enters its critical section.
Agrawal-El Abbadi algorithm - Mutual exclusion

We prove, by induction on depth $k$, that each pair of quorums has a non-empty intersection, and so mutual exclusion is guaranteed.

A quorum with 1 contains a quorum in one of the subtrees below 1, while a quorum without 1 contains a quorum in both subtrees below 1.

- If two quorums both contain 1, we are done.
- If two quorums both don’t contain 1, then by induction they have elements in common in the two subtrees below process 1.
- Suppose quorum $Q$ contains 1, while quorum $Q'$ doesn’t. Then $Q$ contains a quorum in one of the subtrees below 1, and $Q'$ also contains a quorum in this subtree. By induction, they have an element in common in this subtree.
Agrawal-El Abbadi algorithm - Deadlock-freeness

In case of a crashed process, let its left child be put before its right child in the queue of a process that wants to become privileged.

Let a process $p$ at depth $d$ in the binary tree be greater than any process

- at a depth $> d$ in the binary tree, or
- at depth $d$ and more to the right than $p$ in the binary tree.

A process with permission from $r$, never needs permission from a $q < r$.

This guarantees that, in case some leave is responsive, eventually some process will become privileged.

Starvation can happen, if a process waits for a permission infinitely long. (This can be easily resolved.)
Self-stabilization

Let all configurations be initial configurations.

An algorithm is self-stabilizing if every computation eventually reaches a correct configuration.

Advantages:

▶ fault tolerance
▶ straightforward initialization

Self-stabilizing operating systems and databases have been developed.
In a *message-passing* setting, processes might all be initialized in a state where they are waiting for a message.

Then the self-stabilizing algorithm wouldn’t exhibit any behavior.

Therefore, in self-stabilizing algorithms, processes communicate via variables in *shared memory*.

We assume that a process can read the variables of its neighbors.
Processes $p_0, \ldots, p_{N-1}$ form a directed ring.

Each $p_i$ holds a value $x_i \in \{0, \ldots, K - 1\}$ with $K \geq N$.

- $p_i$ for $i = 1, \ldots, N - 1$ is privileged if $x_i \neq x_{i-1}$.
- $p_0$ is privileged if $x_0 = x_{N-1}$.

Each privileged process is allowed to change its value, causing the loss of its privilege:

- $x_i \leftarrow x_{i-1}$ when $x_i \neq x_{i-1}$, for $i = 1, \ldots, N - 1$
- $x_0 \leftarrow (x_0 + 1) \mod K$ when $x_0 = x_{N-1}$

If $K \geq N$, then Dijkstra’s token ring self-stabilizes. That is, each computation eventually satisfies mutual exclusion. Moreover, Dijkstra’s token ring is starvation-free.
Dijkstra’s token ring - Example

Let \( N = K = 4 \). Consider the initial configuration

It isn’t hard to see that the ring self-stabilizes. For instance,
**Theorem:** If \( K \geq N \), then Dijkstra’s token ring self-stabilizes.

**Proof:** In each configuration at least one process is privileged. An event never increases the number of privileged processes.

Consider an (infinite) computation. After at most \( \frac{1}{2}(N - 1)N \) events at \( p_1, \ldots, p_{N-1} \), an event must happen at \( p_0 \).

So during the computation, \( x_0 \) ranges over all values in \( \{0, \ldots, K - 1\} \).

Since \( p_1, \ldots, p_{N-1} \) only copy values, they stick to their \( \leq N - 1 \) values as long as \( x_0 \) equals \( x_i \) for some \( i = 1, \ldots, N - 1 \).

Since \( K \geq N \), at some point, \( x_0 \neq x_i \) for all \( i = 1, \ldots, N - 1 \).

The next time \( p_0 \) becomes privileged, clearly \( x_i = x_0 \) for all \( i \).

So then mutual exclusion has been achieved.
Let $N \geq 3$. Argue that Dijkstra’s token ring self-stabilizes if $K = N - 1$.

This lower bound for $K$ is sharp! (See the next slide.)

**Answer:** Consider any computation.

At some moment, $p_{N-1}$ copies the value from $p_{N-2}$.

Then $p_1, \ldots, p_{N-1}$ hold $\leq N - 2$ different values (because $N \geq 3$).

Since $p_1, \ldots, p_{N-1}$ only copy values, they stick to these $\leq N - 2$ values as long as $x_0$ equals $x_i$ for some $i = 1, \ldots, N - 1$.

Since $K \geq N - 1$, at some point, $x_0 \neq x_i$ for all $i = 1, \ldots, N - 1$. 
Example: \( N \geq 4 \) and \( K = N - 2 \). Consider the initial configuration

It doesn’t always self-stabilize.
We compute a spanning tree in an undirected network.

As always, each process is supposed to have a unique id.

The process with the largest id becomes the *root*.

Each process $p$ maintains the following variables:

- $parent_p$: its parent in the spanning tree
- $root_p$: the root of the spanning tree
- $dist_p$: its distance from the root via the spanning tree
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.
A non-root $p$ declares itself root, i.e.

$$\begin{align*}
\text{parent}_p & \leftarrow \bot \\
\text{root}_p & \leftarrow p \\
\text{dist}_p & \leftarrow 0
\end{align*}$$

if it detects an inconsistency in its root or parent value, or with the root or dist value of its parent:

- $\text{root}_p \leq p$, or
- $\text{parent}_p = \bot$, or
- $\text{parent}_p \neq \bot$, and $\text{parent}_p$ isn’t a neighbor of $p$
  or $\text{root}_p \neq \text{root}_{\text{parent}_p}$ or $\text{dist}_p \neq \text{dist}_{\text{parent}_p} + 1$. 
Question

Suppose that during an application of the Afek-Kutten-Yung algorithm, the created directed network contains a cycle with a “false” root.

Why is such a cycle always broken?

**Answer:** At some $p$ on this cycle, $\text{dist}_p \neq \text{dist}_{\text{parent}_p} + 1$.

So $p$ declares itself root.
Afek-Kutten-Yung spanning tree algorithm

A root $p$ makes a neighbor $q$ its parent if $p < root_q$:

$$
\begin{align*}
parent_p &\leftarrow q \\
root_p &\leftarrow root_q \\
dist_p &\leftarrow dist_q + 1
\end{align*}
$$

**Complication:** Processes can infinitely often rejoin a component with a false root.
Given two processes 0 and 1.

\[\text{parent}_0 = 1 \quad \text{parent}_1 = 0 \quad \text{root}_0 = \text{root}_1 = 2 \quad \text{dist}_0 = 0 \quad \text{dist}_1 = 1\]

Since \(\text{dist}_0 \neq \text{dist}_1 + 1\), process 0 declares itself \textit{root}:

\[\text{parent}_0 \leftarrow \perp \quad \text{root}_0 \leftarrow 0 \quad \text{dist}_0 \leftarrow 0\]

Since \(\text{root}_0 < \text{root}_1\), process 0 makes process 1 its \textit{parent}:

\[\text{parent}_0 \leftarrow 1 \quad \text{root}_0 \leftarrow 2 \quad \text{dist}_0 \leftarrow 2\]

Since \(\text{dist}_1 \neq \text{dist}_0 + 1\), process 1 declares itself \textit{root}:

\[\text{parent}_1 \leftarrow \perp \quad \text{root}_1 \leftarrow 1 \quad \text{dist}_1 \leftarrow 0\]

Since \(\text{root}_1 < \text{root}_0\), process 1 makes process 0 its \textit{parent}:

\[\text{parent}_1 \leftarrow 0 \quad \text{root}_1 \leftarrow 2 \quad \text{dist}_1 \leftarrow 3 \quad \text{et cetera}\]
Before $p$ makes $q$ its parent, it must wait until $q$’s component has a proper root. Therefore $p$ first sends a *join request* to $q$.

This request is forwarded through $q$’s component, toward the root of this component.

The root sends back an *ack* toward $p$, which retraces the path of the request.

Only when $p$ receives this ack, it makes $q$ its parent:

\[
\text{parent}_p \leftarrow q \quad \text{root}_p \leftarrow \text{root}_q \quad \text{dist}_p \leftarrow \text{dist}_q + 1
\]

Join requests are only forwarded between “consistent” processes.
Afek-Kutten-Yung spanning tree alg. - Example

Given two processes 0 and 1.

\[ \text{parent}_0 = 1 \quad \text{parent}_1 = 0 \quad \text{root}_0 = \text{root}_1 = 2 \quad \text{dist}_0 = \text{dist}_1 = 0 \]

Since \( \text{dist}_0 \neq \text{dist}_1 + 1 \), process 0 declares itself \( \text{root} \):

\[ \text{parent}_0 \leftarrow \bot \quad \text{root}_0 \leftarrow 0 \quad \text{dist}_0 \leftarrow 0 \]

Since \( \text{root}_0 < \text{root}_1 \), process 0 sends a join request to process 1.

This join request doesn’t immediately trigger an ack.

Since \( \text{dist}_1 \neq \text{dist}_0 + 1 \), 1 declares itself \( \text{root} \):

\[ \text{parent}_1 \leftarrow \bot \quad \text{root}_1 \leftarrow 1 \quad \text{dist}_1 \leftarrow 0 \]

Since process 1 is now a proper root, it replies to the join request of process 0 with an ack, and process 0 makes process 1 its \( \text{parent} \):

\[ \text{parent}_0 \leftarrow 1 \quad \text{root}_0 \leftarrow 1 \quad \text{dist}_0 \leftarrow 1 \]
A process can only be forwarding and awaiting an ack for at most one join request at a time.

(That’s why in the previous example, process 1 can’t forward process 0’s join request on to process 0.)

Communication is performed using shared memory, so join requests and ack’s are encoded in shared variables.

The path of a join request is remembered in local variables.

For simplicity, join requests are here presented in a message passing framework with synchronous communication.
Given a ring with processes $p, q, r$, and $s > p, q, r$.

Initially, $p$ and $q$ consider themselves root; $r$ has $p$ as parent and considers $s$ the root.

Since $\text{root}_r > q$, $q$ sends a join request to $r$.

*Without the consistency check*, $r$ would forward this join request to $p$. Since $p$ considers itself root, it would send back an ack to $q$ (via $r$), and $q$ would make $r$ its parent and consider $s$ the root.

Since $\text{root}_r \neq \text{root}_p$, $r$ makes itself root.

Now we would have a symmetrical configuration to the initial one.
Afek-Kutten-Yung spanning tree alg. - Correctness

Each component in the network with a false root has an inconsistency, so a process in this component will declare itself root.

Since processes can only be involved in one join request at a time, each join request is eventually acknowledged.

Since join requests are only passed on between consistent processes, processes can only finitely often join a component with a false root (each time due to improper initial values of local variables).

These observations imply that eventually false roots will disappear, the process with the largest id in the network will declare itself root, and the network converges to a spanning tree with this process as root.
Mutual exclusion

Ricart-Agrawala algorithm with a logical clock

Raymond’s algorithm with token passing

Agrawal-El Abbadi algorithm with quorums

Self-stabilization

Dijkstra’s self-stabilizing mutual exclusion algorithm

Afek-Kutten-Yung self-stabilizing spanning tree algorithm
Checkpointing and rollback recovery cope with crash failures.

Assumptions:

- A failure detector that is complete and strongly accurate.
- If a process crashes, another process continues its execution.
- Each process $p$ has stable storage, which remains accessible to the other processes in a consistent state after $p$ has crashed.

Stable storage can be implemented using two disks, where updates in memory on the first disk are faithfully copied to the second disk.
Checkpoints

Each process periodically saves in stable storage its state (*checkpointing*) and received basic messages (*message logging*).

Processes don’t need to coordinate their checkpoints and logs.

Checkpointing and message logging are performed sporadically because they require stable storage, which makes these operations expensive.

If a process crashes, the alive processes perform a *rollback* toward a consistent configuration, from which the execution is restarted.

An event is rolled back if

- it happened after the last checkpoint at the crashed process and can’t be replayed using its message log,
- or is causally after such an event at the crashed process.
When process $p_0$ recovers from its crash, its state is restored to its last checkpoint, and the receipt of message $m_1$ is replayed.

The fact that $p_0$ received $m_2$ is lost.

$p_1$ is rolled back to before the receipt of $m_3$.

$p_2$ is in turn rolled back to before the receipt of $m_4$.

Furthermore, $p_2$ needs to resend $m_2$.

When $m_5$ arrives, $p_0$ needs to discard it.
The logical vector clock is used to determine which basic events should be discarded in the rollback.

(Control events are disregarded by the vector clock.)

During the rollback procedure, the basic computation is stalled.

The algorithm can’t cope with multiple concurrent crashes.

If crashes are rare and recovery phases take little time, it is reasonable to neglect the possibility of a crash during a recovery phase.
Each basic message contains the logical time of its send event, so that the logical time of the corresponding receive event can be determined.

The (logical) time of a process is the time of its last basic event.
Initially it is \((0, \ldots, 0)\).

At checkpoints, the time of the process is saved in stable storage.
Vector times of receive events are kept in the message log.
When a crashed process $p_i$ restarts, it retrieves its last checkpoint and message log from stable storage.

From the checkpoint, $p_i$ replays basic events until its message log is empty. $p_i$ knows when to perform receive events in the replay by their vector times.

Let $p_i$’s last reconstructed basic event have vector time $(k_0, \ldots, k_{N-1})$. Then $p_i$ broadcasts a control message $(k_0, \ldots, k_{N-1})$, tagged with $i$.

This initiates a rollback procedure at the other processes, in which events with the vector time’s $i$th coordinate $> k_i$ are discarded.
When a process $q$ receives $p_i$’s control message, it checks whether the $i$th coordinate of its vector time is $> k_i$.

If so, $q$ restarts at its last checkpoint for which the vector time’s $i$th coordinate is $\leq k_i$.

It replays events up to (but not including) the first basic event for which the vector time’s $i$th coordinate is $> k_i$.

Basic messages received by $q$ beyond this point are kept only if the $i$th coordinate of the vector time of their send event is $\leq k_i$.

These are clustered after the point where the replay at $q$ halted.
An “orphan” message, for which the corresponding send event was rolled back, may arrive after completion of the recovery phase.

Each process $p$ has a sequence number $seq_p$, which initially is 0, and is increased by 1 at each new recovery phase.

$seq_p$, paired with the time stamp $(k_i, i)$ of the corresponding recovery phase, is kept in stable storage.

$seq_p$ is attached to each basic message sent by $p$.

Orphan messages carry an old sequence number, while the $i$th coordinate of the vector time of their send event is greater than $k_i$.

They are discarded by the receiver.
If a basic send event to the *crashed* process $p_i$ isn’t rolled back and its vector time isn’t smaller than $(k_0, \ldots, k_{N-1})$, then it is sent again.

Because the corresponding receive event may have been irrecoverably lost in the crash.

If the message was in transit during the crash, $p_i$ recognizes from the vector time of the send event that it receives this message twice.

The second instance of the message is discarded.

To support resending basic messages, message logs can also contain send events.
The sequence number initially is 0.

All basic messages carry this number.

\( p_0 \) restarts from its last checkpoint with \( \text{seq}_{p_0} = 1 \) and replays the receipt of \( m_1 \) from its message log, with time \( (k_0, k_1, k_2) \).

\( p_0 \) sends control messages to \( p_1 \) and \( p_2 \). They are received, and \( p_1 \) and \( p_2 \) start the rollback procedure with sequence number 1.

By \( m_3 \) and \( m_4 \), the vector times at \( p_1 \) and \( p_2 \) are \( > k_0 \) at index 0. So they restart at their last checkpoint (not shown in the picture) and replay events, until right before the receipt of \( m_3 \) and \( m_4 \).
$m_1$ and $m_2$ are resent by $p_1$ and $p_2$.

At $p_0$, $m_1$ is discarded because it is in $p_0$’s message log.

$m_2$ is treated as a new message.

When $m_5$ reaches $p_0$, it is discarded, because its sequence number is 0, while the vector time of its send event carries a value $> k_0$ at index 0.
Distributed transactions

A (distributed) transaction is a sequence of events, performed as one indivisible unit (by multiple processes).

It either
- **commits**: all its events are performed at once; or
- **aborts**: none of its events take effect.

**Example**: A customer buys shoes via the Internet.

If the customer’s credit is sufficient and the shoes are in stock, then placing the order, transferring the money and shipping the shoes are performed.

Otherwise all these events are canceled.
ACID properties

A *transaction* must satisfy the following properties:

- **Atomicity**: Either all or none of its events take effect.
- **Consistency**: A commit results in a valid configuration.
- **Isolation**: Intermediate effects of a transaction remain invisible while it hasn’t committed.
- **Durability**: The effects of a committed transaction are permanent.

These properties (especially atomicity and durability) must also hold when a process crashes.

Processes have *stable storage*. They employ *rollback recovery*.
Serialization

Each execution of concurrent transactions must be serializable:
Transactions take effect in the order by which they commit.

Different transactions may *share the same memory*.

A *synchronization conflict* may occur if one transaction writes to a variable $x$ while concurrently another transaction reads or writes to $x$.

Transactions must read values of variables in line with the order in which transactions commit.

Two problems can occur if a transaction determines the value of a write operation based on a stale value.
A **lost update** occurs if a write by a committed transaction is ignored by a transaction that commits later.

**Example:** Transactions $T_1$ and $T_2$ want to add €10 and €20, respectively, to bank account $A$.

Both transactions read the value €50 of $A$.

$T_1$ writes €60 as value of $A$ and commits.

$T_2$ writes €70 as value of $A$ and commits.

Only €20 has been added to $A$, instead of €30.
An inconsistent retrieval occurs if a transaction reads inconsistent values due to writes by a concurrent transaction.

**Example:** $T_1$ wants to move €10 from bank account $A_1$ to bank account $A_2$. $T_2$ wants to compute the sum of $A_1$ and $A_2$.

$T_1$ reads the value of $A_1$ and subtracts €10 from it.

$T_2$ reads the values of $A_1$ and $A_2$ and commits.

$T_1$ reads the value of $A_2$, adds €10 to it, and commits.

The sum of $A_1$ and $A_2$ computed by $T_2$ misses out on €10.
Locks

Serialization can be achieved with locks on variables.

Locks must be obtained in line with the serialization order.

A distinction is made between read and write locks:

- Multiple transactions may concurrently get the same read lock.
- While a write lock is held, no other transaction holds this write or the corresponding read lock.

Question: After a transaction released a lock, why can it no longer claim a lock?
Two-phase locking consists of two subsequent phases:

- a growing phase, in which locks are accumulated; and
- a shrinking phase, in which the acquired locks are released.

Read locks can be released early on in the shrinking phase.

Write locks can only be released after committing or aborting, when it is clear whether written values take effect.

Corresponding variables then have the written value in case of a commit, or the original value in case of an abort.
Two-phase locking - Example

Again, $T_1$ wants to add €10 and $T_2$ €20 to bank account $A$.

Prudent lock management should disallow that both transactions concurrently obtain the read lock of $A$, as this leads to a deadlock.

One transaction obtains the read lock, promotes it to a write lock, adds its amount to $A$, commits, and releases the lock.

Next, the other transaction performs this same sequence of events.

The value of $A$ has increased with €30 by adding €10 and €20 in some sequential order.
A **deadlock** can occur if e.g. two concurrent transactions request the same two locks in opposite orders.

Locking leads to **decreased performance** due to lock management and the reduction in concurrency.

If (the process running) a transaction **crashes**, the locks it holds can only be released after the transaction aborts or its execution is resumed.

In the absence of locks, two more problems need to be avoided, due to writes by aborted transactions.
A premature write by a transaction to a variable $x$ is obliterated by a concurrent transaction that wrote to $x$ earlier, aborted, and reset the value of $x$.

**Example:** Again $T_1$ and $T_2$ want to add €10 and €20, respectively, to bank account $A$.

Both transactions read the value €50 of $A$.

$T_1$ writes €60 as new value of $A$.

$T_2$ writes €70 as new value of $A$ and commits.

$T_1$ aborts and resets the value of $A$ to €50.

No money has been added to the account, instead of €20.
A **dirty read** returns a value written by a concurrent transaction that eventually aborts.

**Example:** Again $T_1$ and $T_2$ want to add €10 and €20, respectively, to bank account $A$.

$T_1$ reads the value €50 of $A$, and writes €60 as new value of $A$.

$T_2$ reads €60 as the value of $A$.

$T_1$ aborts and resets the value of $A$ to €50.

$T_2$ writes €80 as new value of $A$ and commits.

€30 has been added to the value of $A$, instead of €20.
Avoiding the four problems

To avoid premature writes and dirty reads, transactions perform tentative writes on local copies of variables.

These local values become definite when the transaction commits.

We discuss two approaches (without locks) to avoid that transactions read stale values:

1. time stamps
2. optimistic concurrency control
Time stamps

Each transaction carries a unique time stamp. (Typically the time of a global clock at the moment of instantiation.)

Transactions are serialized according to their time stamp.

A transaction can only commit when no other ongoing transaction has a smaller time stamp.
Consider a (tentative) read of $x$ by transaction $T$ with time stamp $t$.

The read operation is delayed while there are ongoing transactions with time stamps $< t$ that wrote to $x$.

The read operation returns the last value written to $x$ by a committed transaction with a time stamp $\leq t$. 
Consider a (tentative) write to $x$ by transaction $T$ with time stamp $t$.

The write operation is performed only if no transaction with a time stamp $\geq t$ read $x$.

The write is therefore first checked with all those transactions.

Before a transaction starts, it is announced at all ongoing transactions.

If a transaction with a time stamp $\geq t$ read $x$, then $T$ aborts.
Again $T_1$ and $T_2$, with time stamps $t_1$ and $t_2$, want to add €10 and €20, respectively, to bank account $A$.

Let $t_1 < t_2$.

$T_1$ and $T_2$ both read the value €50 of $A$.

$T_1$ is disallowed to add €10 to the value of $A$ and aborts, in view of the read by $T_2$ and $t_1 < t_2$.

$T_2$ changes the value of $A$ to €70 and commits.
Premature writes can’t occur because transactions perform tentative writes on local copies of variables.

Dirty reads can’t occur because reads always return a value written by a committed or the same transaction.

Lost updates and inconsistent retrievals can’t occur because reads and writes are performed in accordance with the time stamp order.
Optimistic concurrency control

If synchronization conflicts are rare, a transaction may perform reads and tentative writes without worrying about such conflicts.

Transactions perform tentative reads and writes on private copies of variables.

A transaction only commits if a *validation* at the end concludes that no stale values were used.

Then all written values are copied from the private workspace to shared memory.
Each transaction $T$ gets a sequence number at the start of validation. These are issued in ascending order and represent the serialization order of transactions.

At the start of its execution, $T$ stores the sequence number $k$ of the last transaction to commit.

After its working phase, $T$ is assigned a sequence number $\ell$ and waits until all transactions with a sequence number $< \ell$ aborted or committed.

Then $T$ checks whether variables it read during its working phase were written to by committed transactions with a number $> k$ but $< \ell$.

If so, $T$ aborts. Else it commits.
Again $T_1$ and $T_2$ want to add €10 and €20 to bank account $A$.

Let both transactions store the same highest sequence number $k$ of any committed transaction so far.

Next, they copy the value €50 of $A$ to their private workspaces.

$T_1$ and $T_2$ read this value and write €60 and €70, respectively, in their private workspaces as new value of $A$.

$T_1$ and $T_2$ proceed to validation, and are assigned sequence numbers $k + 1$ and $k + 2$, respectively.

Validation of $T_1$ succeeds, so it commits and copies the value €60 to shared memory.

Validation of $T_2$ fails because $T_1$ has a sequence number between $k$ and $k + 2$ and it wrote a value to $A$, while $T_2$ read the value of $A$. 
Premature writes, dirty reads and inconsistent retrievals are precluded.

Because writes aren’t rolled back by aborted transactions and reads return a consistent set of committed values.

Lost updates can’t occur due to validation.
Two-phase commit protocol

A distributed transaction is initiated by a process called the coordinator. The other processes in the transaction are called cohorts.

At the start, the coordinator sends a request to all cohorts.

At the end, the coordinator decides if the transaction commits.

The two-phase commit protocol employs a unanimous voting scheme: The transaction commits only if all participants vote to do so.

Each participant locally keeps track of updates it makes to data values during the transaction. The old and new values are both kept.
Two-phase commit protocol

Voting phase: The coordinator asks each participant (including itself) whether it agrees to commit the transaction.

Each participant replies either **yes**, if all its events succeeded, or **no**, if it experienced a synchronization conflict.

To cope with crashes, before sending **yes**, a process first copies its tentative changes to stable storage.

Completion phase: If all participants vote **yes**, the coordinator broadcasts **commit**; else it broadcasts **abort**.

If **commit** is received, participants make the effects of their events during the transaction visible to others.

If **abort** is received, participants roll back to the old values.

Participants send **ack** to the coordinator. It considers the transaction finished when **ack**’s have been received from all participants.
To cope with crashes, we assume a *complete and strongly accurate failure detector*.

In the voting phase, if the coordinator detects that a cohort crashed, this counts as a **no** vote.

In the completion phase, if the coordinator broadcasts **commit** and then finds a cohort crashed, the written values at the crashed process must be made visible before the transaction can complete.

This is possible because the crashed process copied these values to stable storage before replying **yes**.
Suppose:

- The coordinator crashes at the start of the completion phase.
- A cohort crashes concurrently.
- The coordinator sent **commit** or **abort** only to the crashed cohort, which before crashing made the effects of the transaction visible or undid these effects irrevocably, respectively.

After a new coordinator has been put in place, the cohorts must still reach agreement on whether the transaction should be committed.

Until this agreement is reached, the cohorts are blocked.
Let the coordinator enter an intermediate **precommit phase** after receiving **yes** from all participants.

It sends **precommit** to all participants.

Upon receipt of **precommit**, each participant replies with **ack**.

After receiving **ack** from all participants, the coordinator broadcasts **commit** and proceeds as in the two-phase commit protocol.
If a cohort crashes before sending *ack*, the coordinator could still decide to commit the transaction.

The process replacing the crashed cohort then completes its part of the transaction.

If the coordinator crashes in the precommit phase, the cohorts can agree among themselves whether to commit or abort the transaction.

Because they didn’t roll back or make visible their written values.
Lecture in a nutshell

Peterson-Kearns rollback recovery algorithm
- checkpoints
- vector clock
- stable storage

Transactions
- ACID properties
- lost update, inconsistent retrieval, dirty read, premature write
- two-phase locking
- time stamps
- optimistic concurrency control

Two- and three-phase commit protocols
Distributed computer systems are vulnerable to hostile attacks.

Examples:

- An eavesdropper may secretly listen to private messages.
- An unauthorized intruder may modify parts of a database.

An information security system aims to provide:

- **Confidentiality**: Disclose information only to authorized parties.
- **Integrity**: Accuracy of information is preserved at all times.
- **Availability**: Information is accessible when needed.
Attacks

*Confidentiality* can be threatened by a *spoofing attack*, in which the attacker masquerades as another.

In a *man-in-the-middle attack*, the attacker relays messages between two parties who think they are communicating directly with each other.

*Integrity* may be violated by an *unauthorized intruder* who manages to obtain write access in a database.

*Availability* can be threatened by a *denial-of-service attack*: A flood of messages from many sources forces the target system to shut down.
A hash function $h : D \rightarrow E$ casts a set $D$ to a much smaller set $E$. Computing $h(d)$ is cheap for all $d \in D$.

A hash function allows to fit elements from a vast data domain into a table of fixed size and perform fast lookups.

A cryptographic hash function $h : D \rightarrow E$ is collision resistant:
It is very hard to find different $d, d' \in D$ with $h(d) = h(d')$.

Consequently, it is preimage resistant:
Given an $e \in E$, it is very hard to find a $d \in D$ with $h(d) = e$. 

Cryptographic hash function
Developing collision-resistant hash functions is a huge challenge. (E.g., widely used SHA-0 and MD5 turned out to be vulnerable.)

**Birthday attack:** A collision is on average found after generating $\sqrt{|E|}$ hash values.

$(\sqrt{365} \approx 19)$

SHA-256 hashes to bit strings of length 256.

So the birthday attack tends to require $2^{128}$ hash values before finding a collision.
Each *leaf* of a **Merkle tree** is a data block.

Each *non-leaf* carries the (cryptographic) hash value of its children.

The **Merkle root** is a fingerprint of the data blocks in the leaves.

Suppose data blocks are obtained from an untrusted source.

By obtaining the corresponding Merkle root from a trusted source, it can be checked if the data is uncorrupted.

An advantage of a tree, compared to a list, is that branches of the tree can be downloaded and verified individually.
Public-key cryptography

Given large finite message domains $\mathcal{M}$ and $\mathcal{E}$.

A public-key cryptosystem consists of functions $P_q : \mathcal{M} \to \mathcal{E}$ and $S_q : \mathcal{E} \to \mathcal{M}$ for each process $q$, with

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

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

Underlying assumption: Computing $S_q$ from $P_q$ is very expensive.

$p$ sends a secret message $m$ to $q$: $P_q(m)$
The **RSA cryptosystem** uses that for all prime numbers $p, q$ and positive numbers $m$ with $m = 1 \mod (p - 1)(q - 1)$,

$$k^m = k \mod pq$$

for all integers $k$.

**Elliptic curve cryptography**, based on elliptic curves over finite fields, requires smaller keys than RSA to reach the same level of security.

New entities in the network create a fresh secret/public key pair at a trusted **public key server**.

A trusted **certificate authority** upon request dispenses public keys of entities (signed with its own secret key, to prevent attackers from spreading bogus public keys).
A **proof-of-work** is a puzzle that takes much processing power to solve, while it is easy to verify that a proposed solution is correct.

The puzzle can dissuade attackers from performing a vast number of small tasks with bad intent.

**Example:** To deter email spammers, require a proof-of-work for each sent email, on the basis of the email header.
Sybil attack: Create many nodes, to dominate a majority vote.

A defense is to require a proof-of-work for each vote.

Then an attacker can only bully the network if it possesses a vast amount of processing power.

Ideally the hardness of a puzzle can be adapted easily, in case puzzles are solved at a faster rate.
A public blockchain is an openly accessible ledger, stored as a decentralized distributed database.

It is a (growing) linked list of blocks with validated transactions.

Each block (except the *Genesis block*) has a link to its predecessor.
The blockchain is supposed to contain only legitimate transactions, temporally ordered by their places in the chain.

Anybody can try to add a new block at the end of the list.

Attackers must be prevented from tampering with the blockchain or adding fraudulent blocks.
The database is distributed over a peer-to-peer network in which peers (i.e., processes) can freely join and leave.

Some peers hold the entire history of the blockchain in memory.

Many peers have only sufficient information to validate new blocks.

A peer that joins the network, downloads and verifies blocks from other peers.

The decentralized and replicated nature of a blockchain is important for accessibility as well as security.
The **Bitcoin** protocol employs a blockchain and cryptographic hash functions (SHA-256) for financial transactions on the Internet.

Each peer controls multiple addresses.

Each address has a unique public/secret key pair.

A **transaction** collects *cryptocurrency* called bitcoins from one set of addresses and attributes it to another set of addresses.

- Its inputs are outputs of earlier transactions.
- Each of its outputs is protected by means of the public key of the destination address.
A transaction is broadcast to all peers, to let it be included in a block and then added to the blockchain.

Upon receiving a new transaction, a peer validates that

- all financial contributors to the transaction agree, and
- their contributions have not been spent previously.

Miners collect validated transactions and bundle them in a block.

To add a block to the blockchain, a miner must complete a proof-of-work.
When a miner succeeds in adding a block, it broadcasts the block.

Peers accept the block as new head of the blockchain after checking
  ► the proof-of-work, and
  ► that all transactions in the block are valid.

The miner is rewarded with some newly created bitcoins, which are included in the block itself as a coinbase transaction.

The reward reduces over time and will be eventually become 0.

Moreover, a voluntary transaction fee is paid to a miner.

Ownership of bitcoins is defined by sequences of digitally signed transactions originating in coinbase transactions.
A fork in the blockchain occurs when two miners complete the proof-of-work and add a block at almost the same time.

Miners will continue to append new blocks to the longest chain they are aware of.

One of the forks will quickly prevail, and the orphan blocks in the shorter fork will be ignored.
A dishonest miner can create a long fork of rogue blocks that starts from an older block and turns genuine blocks into orphans.

Mining is therefore intentionally resource-intensive.

If a majority of the computing power is controlled by honest miners, a fork of rogue blocks will be outpaced by the fork of honest blocks.

Another important protective measure is that the proof-of-work chains a block to its predecessor through cryptographic means.
The *body* of a block consists of its transactions, stored in a Merkle tree.

Its *header* consists of:

- The hash value of the header of the previous block in the chain.
- The Merkle root of its own body.
- A time stamp based on real time.
- A *nonce*, being some 32-bit value.

The 1st part firmly chains the block to its predecessor.

The 2nd part protects the integrity of the body.

The 4th part is the basis of the proof-of-work required from miners.
Bitcoin - Content of a block

Block $n$
- Hash value
- Nonce
- Time stamp
- Merkle root

Block $n+1$
- Hash value
- Nonce
- Time stamp
- Merkle root

Block $n+2$
- Hash value
- Nonce
- Time stamp
- Merkle root

Orphan block
The *proof-or-work* is to find a nonce for the 4th part such that the hash value of the block’s header starts with many ($\pm 40$) zeros.

In this guessing game, one out of $2^{\pm 40}$ nonces yields a solution.

The nonce starts at 0 and is increased until a solution is found or it overflows.

If it overflows, an *extra nonce field* within the coinbase transaction of the block is incremented and the nonce restarts at 0.

The difficulty of the puzzle is adapted, based on how fast new blocks are being added.
Mining is a gamble and requires a lot of processing power.

Therefore many miners have formed pools that share resources and rewards, usually under some centralized form of coordination.

For the security of the Bitcoin protocol, it is essential that no pool controls a large part of the hashing power in the network.

Currently most blocks are mined by a small number of pools.

Mining rewards decrease over time, which increases the future threat of a mining monopoly.
Proof-of-stake

Proof-of-work is a waste of energy and money.

In **proof-of-stake**, a peer’s voting power in adding a new block is proportional to its amount of cryptocurrency.

The cryptocurrency added to the system with each new block is divided among all peers in the same proportional manner.

Attackers must own a lot of cryptocurrency, giving them a strong interest to maintain the integrity of the system.

**Drawbacks of proof-of-stake:**

- Little incentive to validate new blocks.
- A perverse incentive to stock up on cryptocurrency.

Incentives are needed to counterbalance these factors.
The application of blockchains goes beyond financial transactions.

A blockchain can be endowed with program instructions, to run a tamper-resistant computer program.

**Example:** A smart contract automatically enforces the terms of a legal agreement.

E.g., regulate car insurance, particularly in case of an accident.
Huygens’ principle

Each point on a wave front becomes a source of a spherical wave.
Christiaan Huygens predicted in 1678 that light behaves as a wave.

Thomas Young showed in 1805 that this is indeed the case.

But if light is measured at the slits, its particles (photons) travel in a straight line.
An elementary particle can behave as a wave or as a particle.

**Superposition**: A particle is simultaneously in a range of states, with some probability distribution.

Interaction with an observer causes a particle to assume a single (deterministic) state.

Superposition is represented using *complex* numbers.
Complex numbers

“God made the natural numbers; all else is the work of man”

(Leopold Kronecker, 1886)

\[
\begin{align*}
 x + 1 & = 0 & x & = -1 & \mathbb{Z} \\
 2 \cdot x & = 1 & x & = \frac{1}{2} & \mathbb{Q} \\
 x^2 & = 2 & x & = \sqrt{2} & \mathbb{R} \\
 x^2 & = -1 & x & = i & \mathbb{C}
\end{align*}
\]

A complex number in \( \mathbb{C} \) is of the form \( a + bi \) with \( a, b \in \mathbb{R} \).

Fundamental theorem of algebra: \( \mathbb{C} \) is algebraically closed!
A qubit (short for *quantum bit*) is in a superposition

\[ \alpha_0 |0\rangle + \alpha_1 |1\rangle \]

with \( \alpha_0, \alpha_1 \in \mathbb{C} \), where \( |\alpha_0|^2 + |\alpha_1|^2 = 1 \).

At interaction with an observer, the qubit takes on the value 0 with probability \( |\alpha_0|^2 \), and the value 1 with probability \( |\alpha_1|^2 \).

After such an interaction the qubit is no longer in superposition, but in a single state 0 or 1.
Quantum operations on qubits are linear and preserve probability mass 1.

The Hadamard transform brings $|0\rangle$ and $|1\rangle$ into a superposition where the outcomes 0 and 1 are equally likely:

- $|0\rangle$ maps to $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$
- $|1\rangle$ maps to $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$

Applying two Hadamard transforms in a row yields:

\[
|0\rangle \mapsto \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \\
\mapsto \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) + \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right) = |0\rangle
\]

**Question:** What happens to $|1\rangle$?
Alice chooses $n$ random classical bits, and sends them as qubits, each with 50% chance encoded with the Hadamard transform, over a public quantum channel to Bob.

On each received qubit, Bob applies the Hadamard transform with 50% chance.

Next Bob measures the qubit.

Bob is guaranteed to measure the correct value if either:

- Alice and Bob both didn’t apply the Hadamard transform to the qubit; or
- Alice and Bob both applied the Hadamard transform to the qubit.
Alice and Bob now communicate over a public classical channel.

Alice and Bob inform each other to which bits they applied the Hadamard transform.

Thus Alice and Bob can determine which qubits Bob has certainly measured correctly.

These (on average $\frac{n}{2}$) bit values form the secret key.
Suppose Eve listens in on the quantum channel, and tries to obtain the secret key.

Since qubits can’t be cloned, Eve needs to measure qubits, so that they are no longer in superposition.

Eve must (like Bob) gamble for each qubit whether to apply the Hadamard transform.

Each time Eve guesses wrong, she may introduce a mistake.
Alice sends *half* of her secret key to Bob, via the classical channel.

If a significant majority of these bits were measured correctly by Bob, then eavesdroppers (almost certainly) gained insufficient information to obtain the secret key.

Finally, Bob applies an error correction algorithm to repair possible mistakes in the secret key introduced by eavesdroppers.

Error correction algorithms add redundant information.

- **Hamming code**: can discover one mistake.  
  (Used in flash memory.)
- **Reed-Solomon code**: is based on polynomials.  
  (Used in cd’s, dvd’s, hard discs.)
Questions

Why are errors introduced by Eve outside the secret key irrelevant?

**Answer:** Alice and Bob ignore such values.

Eve can overhear on the public channel which bits form the secret key.

Suppose Eve performs a Hadamard transform on a qubit and then measures it. Why should she apply another Hadamard transform on the qubit before sending it to Bob?

**Answer:** If the qubit is part of the secret key and Eve guesses right (i.e., Alice and Bob apply a Hadamard transform on the qubit too), then Bob is guaranteed to read the correct result.
$n = 8$. Alice’s random initial values are $10010100$.

Alice randomly decides to apply the Hadamard transform on the first, third, fourth and eighth qubit.

Bob applies it on the first, second, third, seventh and eighth qubit.

The secret key computed by Alice and Bob consists of the first, third, fifth, sixth and eighth qubit: $10010$.

**Question**: Suppose Eve measures the third qubit and then sends it to Bob. How can this introduce errors at both Eve and Bob?

Likewise if Eve applies a Hadamard on the fifth qubit, measures it, applies another Hadamard transform, and sends it to Bob.
Security

- confidentiality, integrity, availability
- cryptographic hash function
- Merkle tree
- public-key cryptography
- proof-of-work

Blockchain

- Bitcoin
- miners
- forks
- proof-of-work
- proof-of-stake
- smart contract
Quantum cryptography

- superposition
- qubits
- Hadamard transform
- BB84 key exchange protocol
Edsger W. Dijkstra prize in distributed computing


**2010**: Chandra, Toueg, *Unreliable failure detectors for reliable distributed systems*, 1996

**2014**: Chandy, Lamport, *Distributed snapshots: determining global states of distributed systems*, 1985