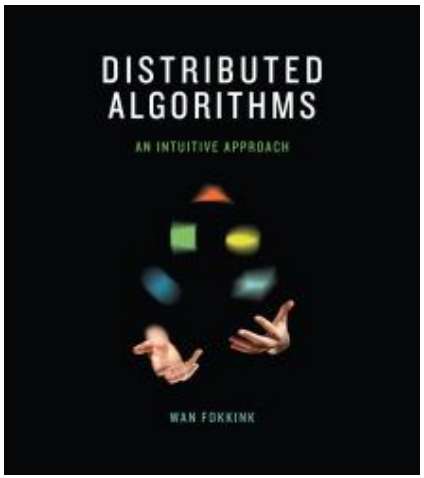
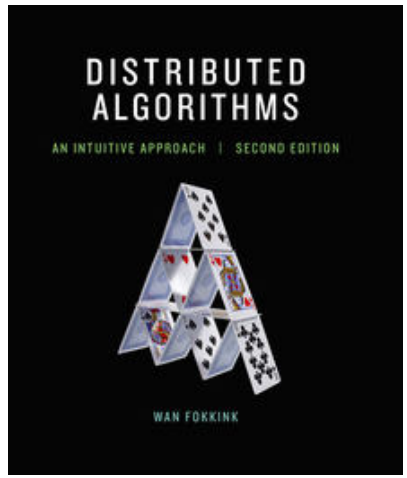
Distributed Algorithms





Paul Anderson

Based on slides and content associated with the following book:

Distributed Algorithms: An Intuitive Approach (2nd edition)

MIT Press, 2018

Algorithms

A skilled programmer must have good insight into algorithms.

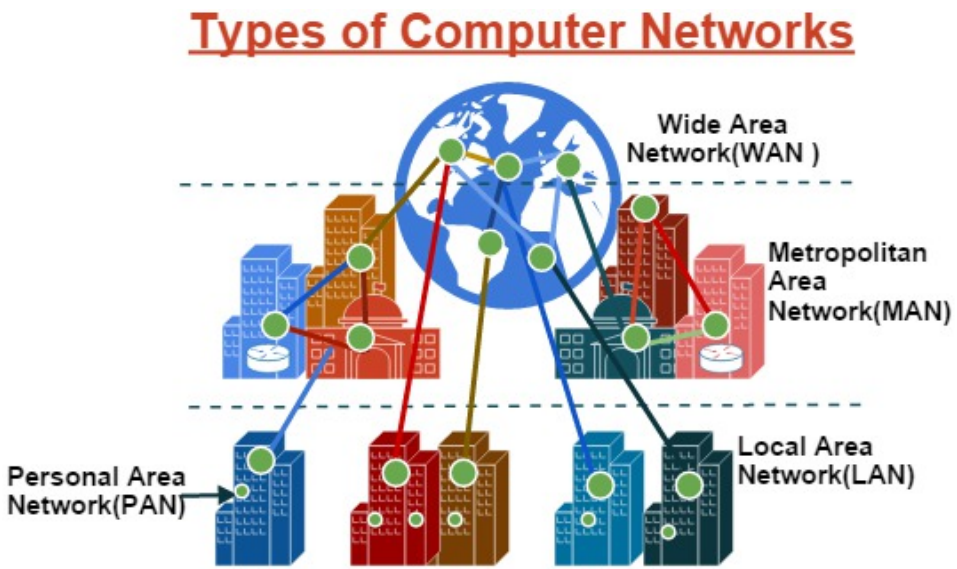
At bachelor level you were offered courses on basic algorithms:
searching, sorting, pattern recognition, graph problems, ...

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

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

Distributed systems

A distributed system is an interconnected collection of autonomous processes.



Motivation:

- 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 by their temporal occurrence.

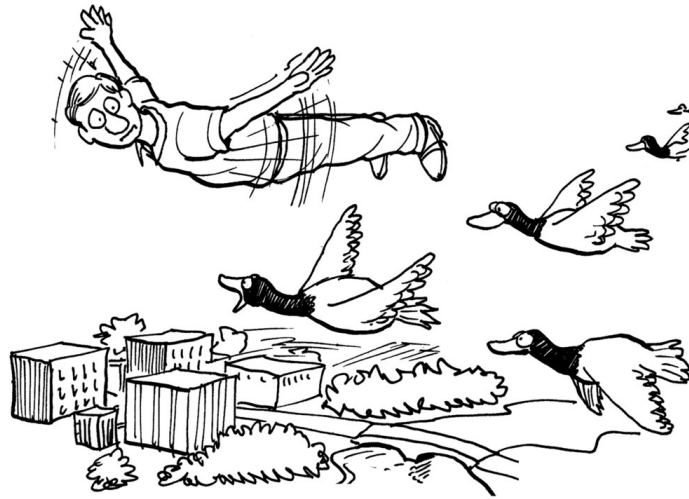
  Example: Mutual exclusion becomes an issue.

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

  Example: Race conditions.

Aim of this course

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



"Sometimes it's good to get a different perspective."

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

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

Message passing

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

We'll focus mainly on message passing when discussing algorithms.

(The technical systems component of this course discusses shared memory.)

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

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

We'll mainly consider asynchronous communication.

Communication protocols

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

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

Example: Sliding window protocols.

"Conceptually, each portion of the transmission (packets in most data link layers, but bytes in TCP) is assigned a unique consecutive sequence number, and the receiver uses the numbers to place received packets in the correct order, discarding duplicate packets and identifying missing ones. The problem with this is that there is no limit on the size of the sequence number that can be required."

Assumptions

Unless stated otherwise, we assume:

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

Directed versus bidirectional channels

Channels can be *directed* or *bidirectional*.

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

Remarks:

- Algorithms for undirected networks often include ack's.
- Acyclic networks must always be undirected

  (else the network wouldn't be strongly connected - every vertex is reachable).

Complexity measures

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

Message complexity: Total number of messages exchanged.

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

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

Space complexity: Amount of memory needed for the processes.

Different executions require different consumption of resources.

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

Big O notation

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

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

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

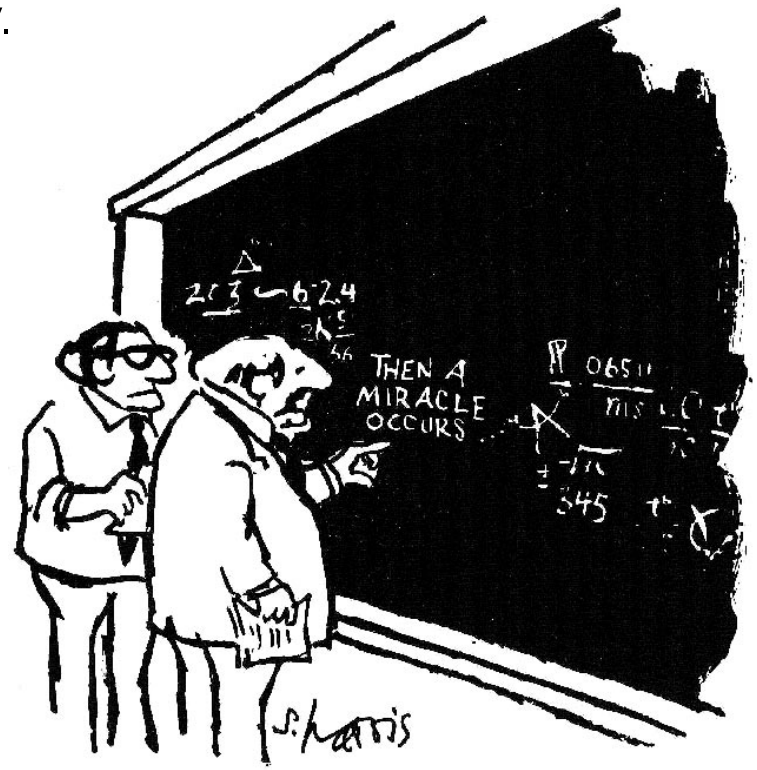
$f = O(g)$ if, for some $C > 0$, $f(n) \leq C \cdot g(n)$ for all $n \in \mathbb{N}$.

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

Formal framework

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

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



"I think you should be more explicit here in step two."

Transition systems

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

The configuration evolves in discrete steps, called transitions.

A transition system consists of:

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

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

Executions

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

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

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

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

States and events

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

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

A process can perform internal, send and receive events.

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

An algorithm is centralized if there is exactly one initiator.

A decentralized algorithm can have multiple initiators.

Assertions

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

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

"*something bad will never happen*"

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

"*something good will eventually happen*"

Invariants

Assertion P on configurations is an invariant if:

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

Each invariant is a safety property.

Causal order

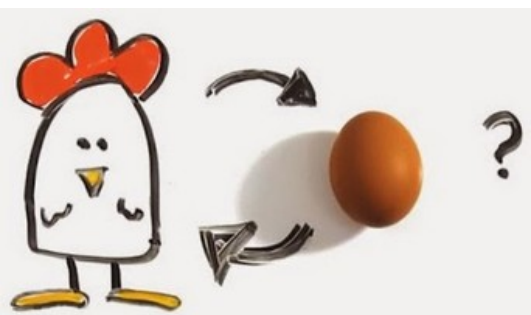
In each configuration of an asynchronous system, applicable events at different processes are independent.

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

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

This relation is *irreflexive*

(never holds between a term and itself).



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

Computations

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

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

These permutations together form a computation.

All executions of a computation start in the same initial configuration.

And if they are finite, they all end in the same terminal configuration.

Question

Consider the finite execution abc.

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

Which executions are in the same computation ?

Lamport's clock

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

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

LC can be computed at run-time:

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

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

Question

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

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

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

Provide all events with Lamport's clock values.

Answer: 1 2 8 9

1 6 7

3 4 5 6

Vector clock

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

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

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

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 $a_1^i \prec \cdots \prec a_{k_i}^i$ of events at process p_i with $a_{k_i}^i \preceq a$.

VC can also be computed at run-time.

Question

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

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

Provide all events with vector clock values.

Answer:

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

Vector clock - Correctness

Let $a \prec b$.

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

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

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

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

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

So $a \preceq b$.

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