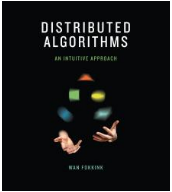
Distributed Algorithms



Wan Fokkink

Distributed Algorithms: An Intuitive Approach

MIT Press, 2013

Distributed versus uniprocessor

Distributed systems differ from uniprocessor systems in three aspects.

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

  Example: termination and deadlock detection become an issue.

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

  Example: mutual exclusion becomes an issue.

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

  Example: race conditions.

Message passing

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

We will only consider message passing.

(The course *Concurrency & Multithreading* treats shared memory.)

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

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

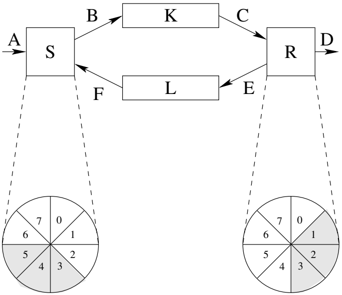
We will mainly consider asynchronous communication.

Communication protocols

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

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

Example: Sliding window protocols.



Assumptions

Unless stated otherwise, we assume:

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

Directed versus undirected channels

Channels can be *directed* or *undirected*.

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

Remarks:

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

Complexity measures

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

Message complexity: Total number of messages exchanged.

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

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

Space complexity: Amount of space needed for the processes.

Different executions may give rise to 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 $g = O(f)$.

Formal framework

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

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

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

Transition systems

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

The configuration evolves in discrete steps, called transitions.

A transition system consists of:

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

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

Executions

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

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

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 is an invariant if:

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

Each invariant is a safety property.

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

Causal order

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

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

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

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

Computations

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