Technische Universität Berlin

Distributed Algorithms 2015/16
Introduction, Models, and More

Reinhardt Karnapke | Communication and Operating Systems Group

Overview

Introduction

- Definition of a distributed system
- Motivation for the use of distributed systems
- Characteristics of distributed systems and their consequences
- Examples of conceptual problems in distributed systems

Basic models of distributed systems

- Abstract view on a distributed system
- Exemplary network topologies
- Characteristics of communication channels
- Definition and state of a distributed algorithm
- System model and processing model
- Characteristics of (distributed) algorithms (e.g. time and message complexity)

Distributed Systems

Some Definitions

"A distributed computing system consists of multiple autonomous processors that do not share primary memory, but cooperate by sending messages over a communication network."
-- *Henri Bal*

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."
-- *Leslie Lamport*

"A distributed system is a collection of independent computers that appears to its users as a single coherent system"
-- *Andrew S. Tanenbaum*

Definition and Demarcation

Distributed systems

- Consist of autonomous computers, that are connected loosely by a network and communicate through message exchange to achieve a common functionality.

Computer Networks

- The networking of the computers is in the foreground not the cooperation of the computers
- Computer networks allow, e.g., the access to a remote computer via Telnet or SSH

Parallel Computers

- Have, in contrast to distributed systems, a common physical memory

Goals (i)

Capacity gain through real concurrent processes

- If application is well parallelizable, an otherwise unreachable capacity is possible
  - Common decoding of a cryptographic key
  - Parallel search for the prime factor of a large number
- cf. also Grid Computing or Cloud Computing
- If applicable, consider (dynamic) load sharing

Flexible, incremental expandability

- Including new functionality or scaling of the system by adding more computers

Goals (ii)

Supply of remote data and services

- Access on several databases (e.g. library catalogue)
- Usage of unused computer capacity (e.g. SETI@home)

Profitability

- Connected PCs usually offer better cost-benefit ratio than a supercomputer
- ⇒ Distribute application on many small computers if possible

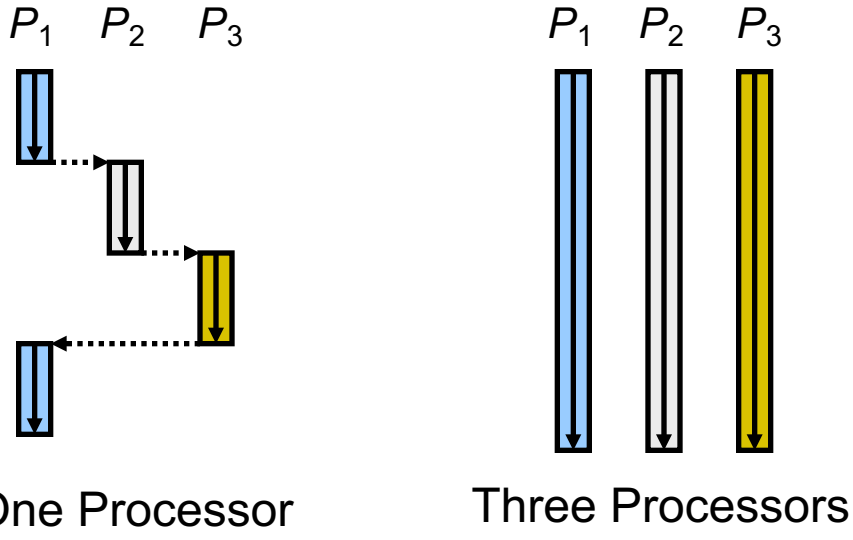
Fault tolerance through redundancy

- Redundant storage of data
- Primary/Backup Server
  (Backup takes over if Primary breaks down)

Characteristics & Consequences (i)

Concurrence

- Many processes with different execution speed
- Many events happen at the same time
- Actions are not reproducible → Non-determinism
- Mechanisms for coordination and synchronization of activities (e.g., for exclusive access on resources)



Characteristics & Consequences (ii)

No shared primary memory

- No node has global view on the whole state of the system → distribution of state

⇒ Mechanisms for a consistent view on the state necessary
(e.g., for the secure detection of termination)

Slow processes or connections cannot be distinguished from those that are broken down

⇒ No adequate detection of failures possible

Characteristics & Consequences (iii)

Communication only through message exchange

- Message transmission delay is indefinite, mostly not limited and varies unpredictably (e.g., the message delay in an Ethernet depends on the current load of the network)

Communication is error-prone

- Message loss, -duplication, -corruption is possible

⇒ Mechanisms for detection and handling of communication errors

Communication is insecure

- There is the danger that messages are intercepted, deliberately adulterated, added or peculated

⇒ Security mechanisms

Characteristics & Consequences (iv)

Computers and network connections can fail independent from each other (partial break down)

- In large systems, single failures of computers or network connections are likely
- A failure should not halt the whole system, but failures of single computers or network connections should be coped with

⇒ Fault tolerance mechanisms

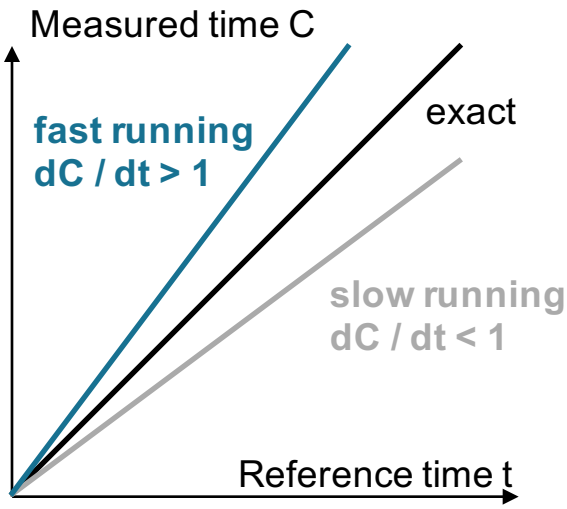
Characteristics & Consequences (v)

Independent physical system clocks with different speeds → clock drift

⇒ Mechanisms for balancing of different velocities necessary (clock synchronization)

Accuracy of clock synchronization and resolution of clocks is restricted

– Relative location of events in reference to each other is often more important than the exact time of their appearance
→ Logical clocks



Characteristics & Consequences (vi)

Different administrative domains

- For each domain, a different administrator is responsible
- Centralized administration in large systems impossible

⇒ Allowing decentralized administration and automating of administration (if possible and secure)

Components are heterogeneous → Heterogeneity

- Interoperability between components is difficult to guarantee

⇒ Interface standardization

Rules of Thumb

Only distribute system if this is either necessary or profitable; otherwise implement centralized solution

Avoid „Central point of failure”

- A partial failure should not paralyze the whole system
- Therefore, redundancy is necessary

Consider scalability

- The performance should not decrease disproportionately for an increasing number of participants → consider bottlenecks
- Mini-Prototypes do not allow conclusions concerning the performance of a large system

Only allow as much access as required for a task

Conclusions

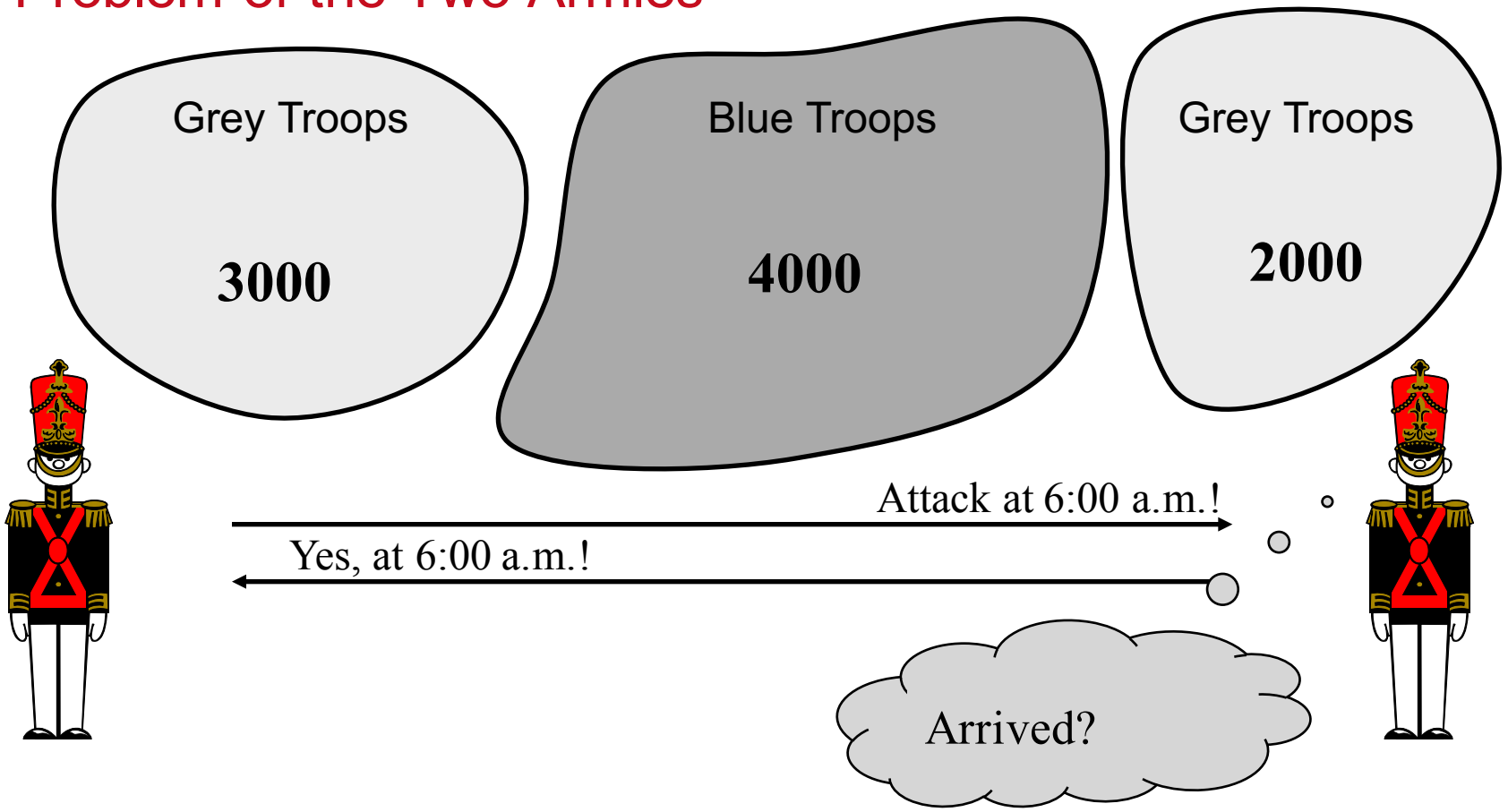
- Designing, implementing, testing and securing of a distributed system is much more complex than in the case of a centralized system
- Only if the distributed system is configured adequately, the distributed system is more powerful, more scalable and more fault tolerant, and probably also similarly secure as a centralized system
- Precondition for that is the understanding of occurring phenomena in distributed systems, the resulting problems and the knowledge about possible solution approaches
- Subject-matter of the lecture are models, concepts and algorithms contributing to the understanding and controlling of occurring phenomena in distributed systems

Examples of Conceptual Problems in Distributed Systems

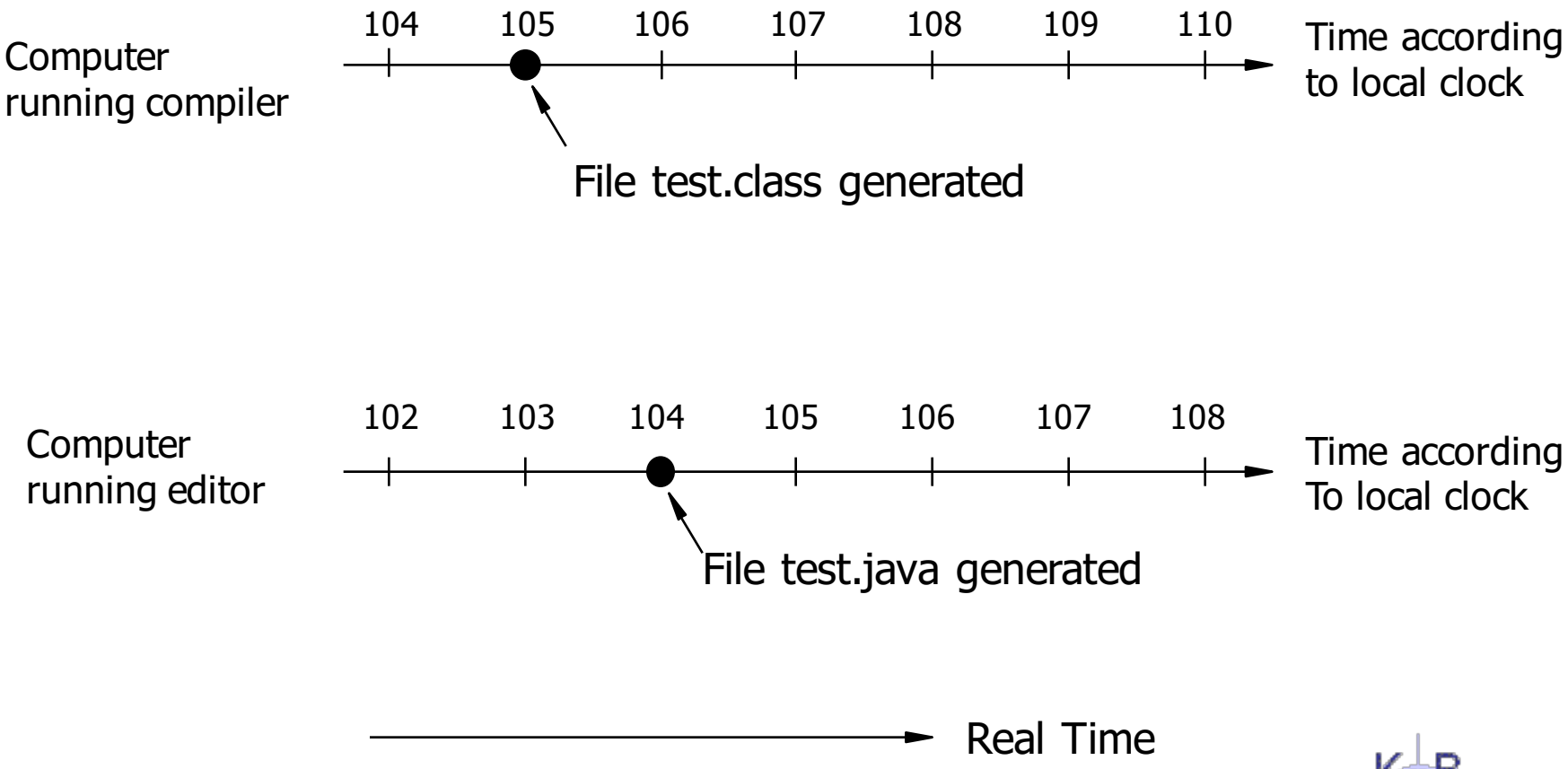
Conceptual Problems

Distributed Consensus
Clock synchronization
Detection and Dissolution or Avoiding of deadlocks
Causality preserving observations
Confidential data communication over insecure channels
Snapshot problem
Recognizing global termination
Distributed memory reassessment
Mutual exclusion
Assurance of consistency with replicated data
…

Problem of the Two Armies

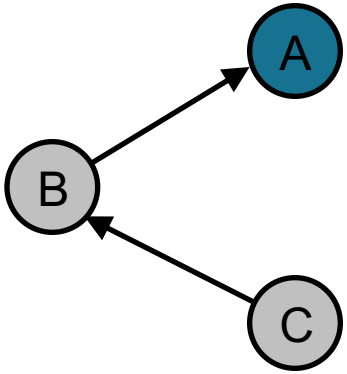


Distributed `make`

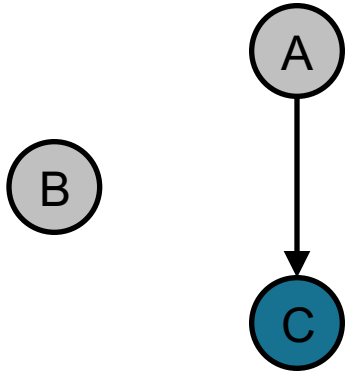


Phantom-Deadlocks

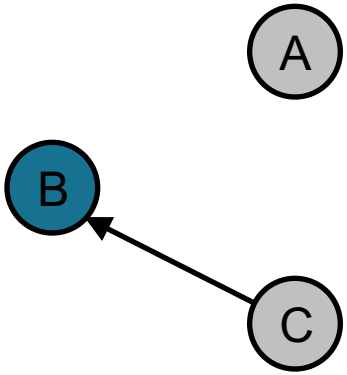
t = 1, observe B
⇒ B waits for A



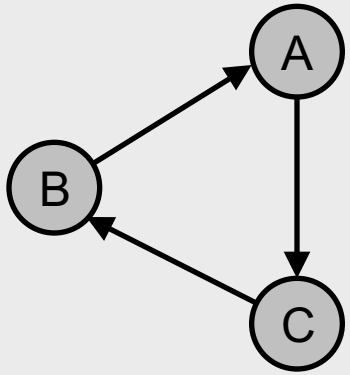
t = 3, observe A
⇒ A waits for C



t = 2, observe C
⇒ C waits for B

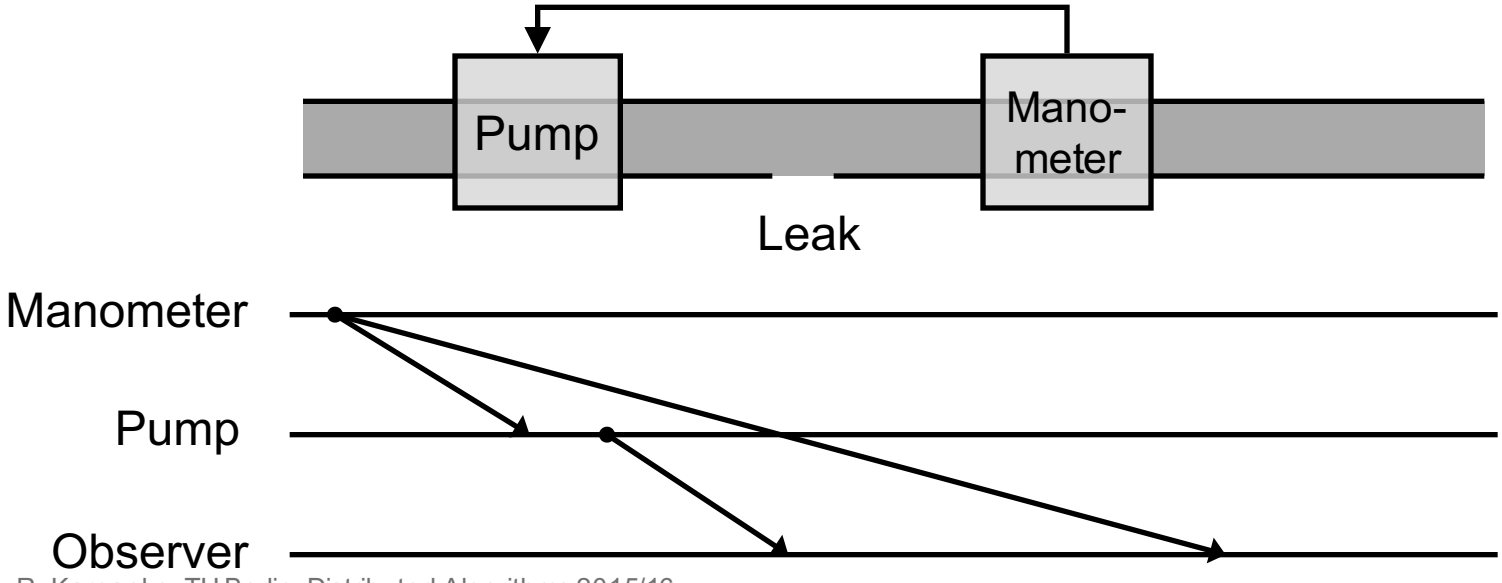


Wrong conclusion:
Cyclic waiting graph
→ deadlock



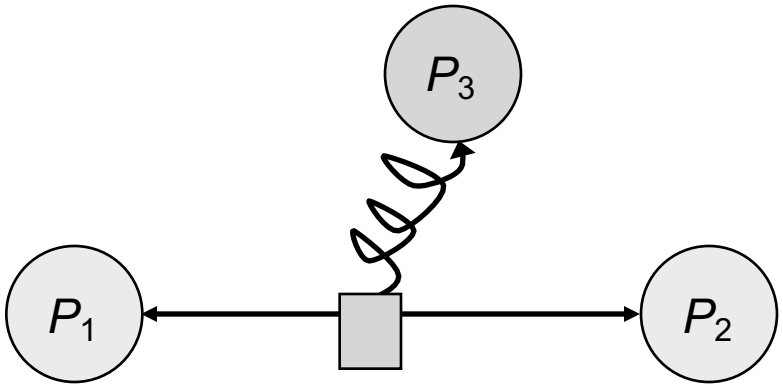
Causally Inconsistent Observations

- Problem: wrong conclusion of the observer caused by a causally inconsistent observation
- Real course of action: A leak occurred in the pipeline. To keep the flow constant, the pump increased the supply pressure.
- However, observer thinks, due to the order of the received messages:
  An unjustified activity of the pump increased the pressure until the pipeline burst, then oil leaked which was indicated by loss of pressure!



Distributed Secret Agreement

- Problem: P_1 and P_2 want to agree on a *secret key* over an insecure channel and encrypt their following communication



Basic Models for Distributed Systems

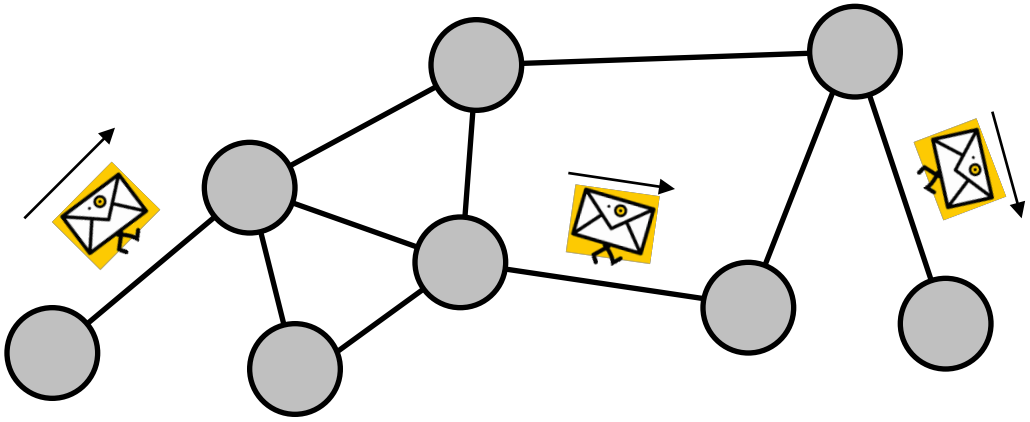
Distributed System – An abstract view

A distributed system is a connected graph consisting of a set of *nodes* and a set of *edges*

- nodes are also denoted as processes, computers etc.
- edges are often denoted links, channels etc.

Nodes can exchange messages with their respective direct neighbors over channels

Arbitrary nodes can communicate through routing



Network-Topologies

Default for the lecture

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

- Node set $V = \{v_0, \ldots, v_{n-1}\}$
- Edge set $E = \{e_0, \ldots, e_{m-1}\}$, $e_i = (v_{i\_1}, v_{i\_2})$

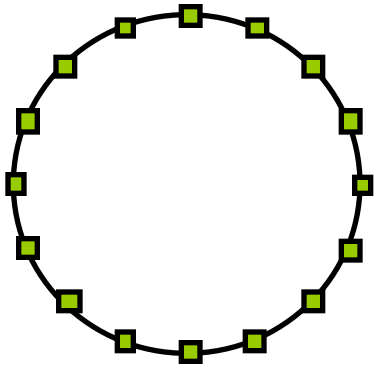
Static vs. dynamic topology

- Static: node- and edge set do not change
- Dynamic: node- and edge set can change over time

Special Topologies

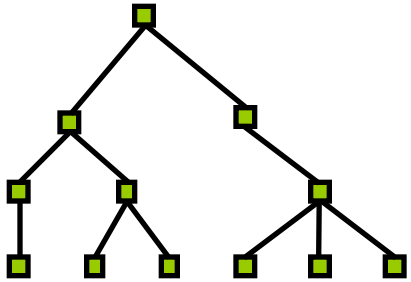
Ring

- Number of nodes = number of edges ($n = m$)
- $E = \{(v_i, v_j) \mid j = (i + 1) \bmod n\}$
- Node degree (Neighbors per nodes): 2
- Node degree constant with scaling
- Diameter (longest shortest path) $\lfloor n / 2 \rfloor$



Tree

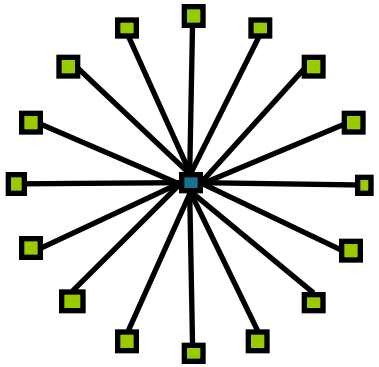
- Connected graph without cycles
- $m = n - 1$



Special Topologies

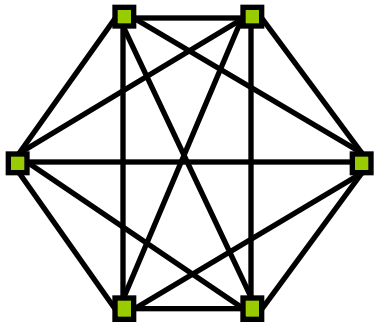
Star

- Special tree with central node v_0
- $E = \{(v_0, v_i) \mid i \neq 0\}$
- Neighbors per node: 1 or $n - 1$
- Diameter 2
- Not (infinitely) scalable

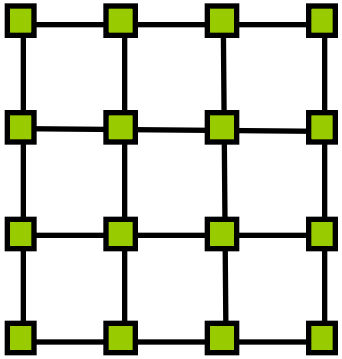


Complete Graph

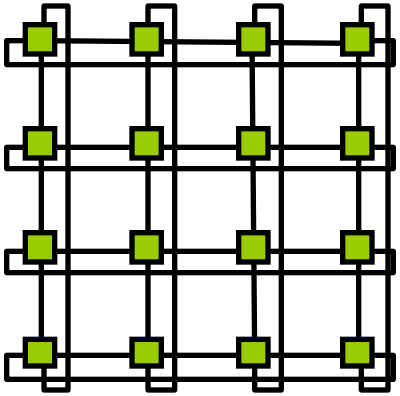
- $m = n\,(n - 1) / 2$
- $E = \{(e_{i\_1}, e_{i\_2}) \mid i\_1 < i\_2\}$
- Neighbors per node: $n - 1$
- Diameter 1
- Not (infinitely) scalable



Special Topologies



4x4-Mesh

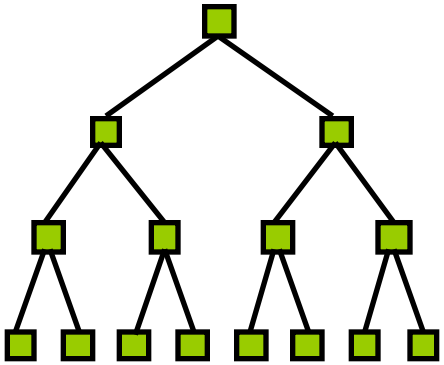


4x4-Torus

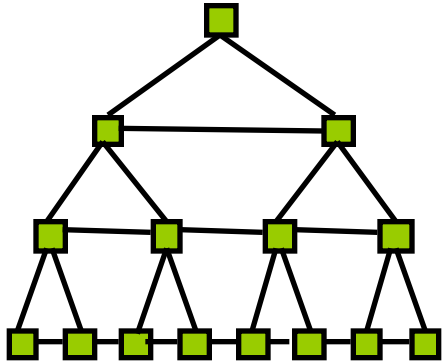
Mesh structures

- Constant node degree with scaling
- Diameter increases with root of node number
- Expandable in small increments
- Good support of algorithms with local communication structure (e.g., modeling of physical processes)

Special Topologies



Binary Tree



Binary X-Tree

Complete *k*-nary tree or *X*-Tree

- $h = \lceil \log_k n \rceil$ (logarithmic height)
- Expandable in potencies of *k*
- Constant node degree with scaling
- Node degree maximal $k + 1$ (or maximal $k + 3$)

Hypercube

Hypercube = Cube of dimension *d*

Recursive construction principle

- Hypercube of dimension 0: single node
- Hypercube of dimension *d* + 1: *„Take two cubes of dimension d and connect the corresponding vertices“*

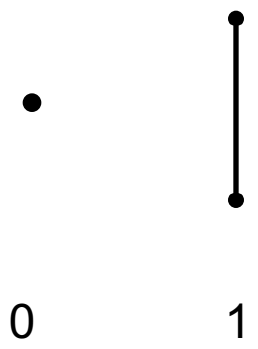
0

Hypercube

Hypercube = Cube of dimension *d*

Recursive construction principle

- Hypercube of dimension 0: single node
- Hypercube of dimension *d* + 1: *„Take two cubes of dimension d and connect the corresponding vertices“*

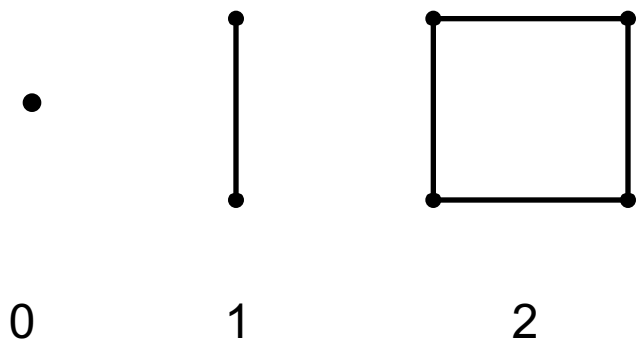


Hypercube

Hypercube = Cube of dimension *d*

Recursive construction principle

- Hypercube of dimension 0: single node
- Hypercube of dimension *d* + 1: *„Take two cubes of dimension d and connect the corresponding vertices“*

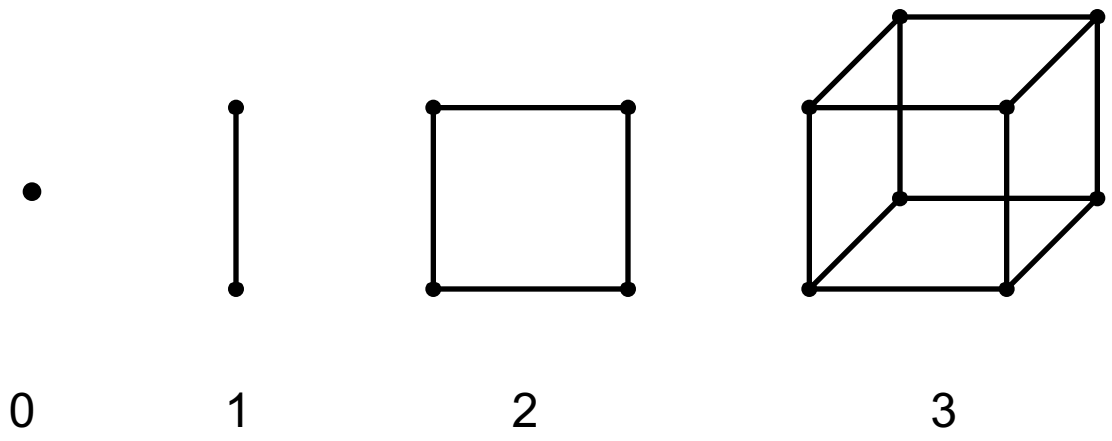


Hypercube

Hypercube = Cube of dimension *d*

Recursive construction principle

- Hypercube of dimension 0: single node
- Hypercube of dimension *d* + 1: *„Take two cubes of dimension d and connect the corresponding vertices“*

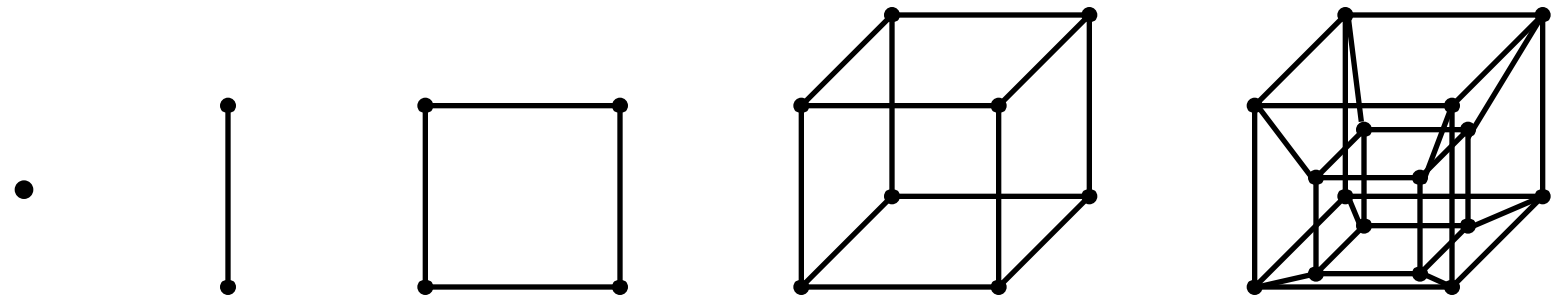


Hypercube

Hypercube = Cube of dimension *d*

Recursive construction principle

- Hypercube of dimension 0: single node
- Hypercube of dimension *d* + 1: *„Take two cubes of dimension d and connect the corresponding vertices“*



Characteristics of Hypercubes

Number of nodes $n = 2^d$

Number of edges $m = d\ 2^{d-1}$

Longest shortest path length between two nodes
(occurs with diagonal opposite nodes)

$d = \log n$

Many path alternatives → Fault tolerance

Node degree = d (not constant with scaling!)

Average path length = $d / 2$

Simple routing of single messages

- XOR of send and destination address
  (one bit vector with d bits each)
- Dimensions whose bits equals 1 in the result are traversed successively. Does the order matter at all?

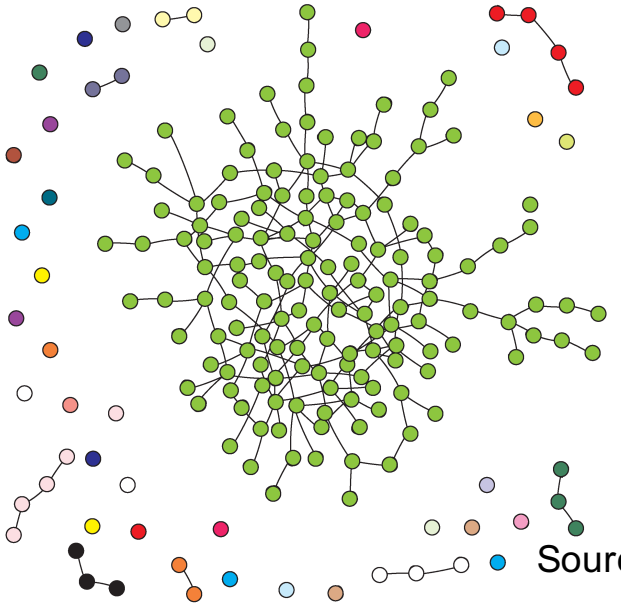
Random Networks

Generation of a random connected network with n nodes and m edges

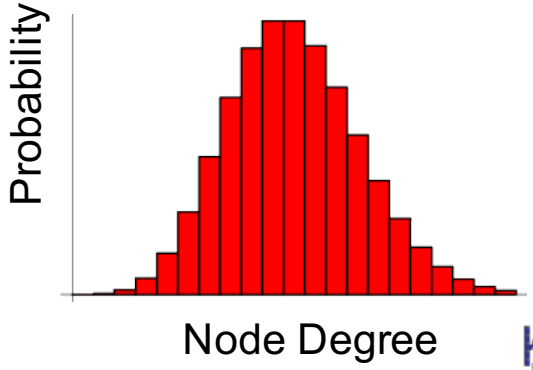
- Choose m-times two nodes randomly and connect them with an edge

Characteristics

- For $m > n / 2$, a large component with short paths between the contained nodes is formed
- Node degree is approximately Poisson-distributed



Source: [1]



Small World-Networks

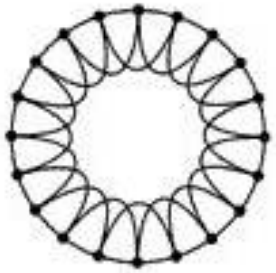
Characteristics

- High clustering-coefficient
- Short paths

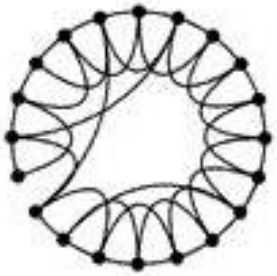
Generation according to

Watts and Strogatz [2]

- Starting point is a ring in which each node is connected with his k next nodes
- Choose a new node randomly for each edge with probability p
- Here: Average node degree is k



regular
(p = 0)

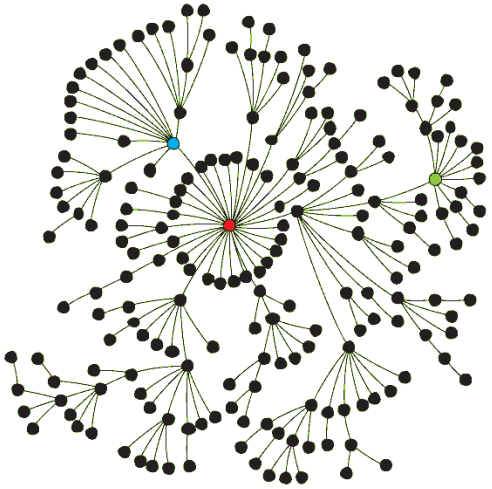


Small World

random
(p = 1)

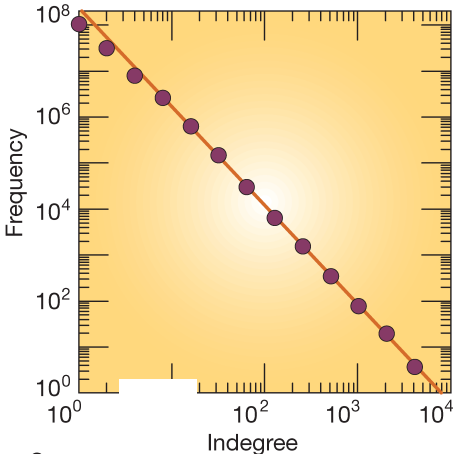
Scale-Free Topologies

- Additional characteristics to those of Small World-topologies
  - Distribution of the number of neighbors follows a power law, i.e., $P(k) \sim k^{-\gamma}$
  - ⇒ Many nodes with few neighbors
  - ⇒ Few nodes (*hubs*) with many neighbors
- Close to many real networks (e.g., Internet, WWW, social networks)



Source: [1]

Ingoing degree of web-pages in the WWW



Source: [1]

Scale-Free Topologies

Generation through *Preferential Attachment*

- Start with a small number of nodes n_0
- With every step, a node is added and connected with $n_1 \leq n_0$ other nodes
- Hereby, a node i with degree k_i is chosen with probability $P(k_i) = k_i / \sum_j k_j$

⇒ „The rich get richer“

- For one topology generated with those rules: $\gamma = 3$

Channel Characteristics (from the processes‘ view)

Default for the lecture

Reliability

- Reliable: Every sent message arrives once and unchanged
- Unreliable: Errors may occur
  - Loss : Sent message is not received
  - Duplication: Sent message is received several times
  - Corruption: Sent message is received corrupted
  - Adding: A message that was not sent is received

Order

- unordered: Messages can overtake each other
- FIFO: Messages cannot overtake each other

Channel Characteristics (from the processes' view)

Default for the lecture

Capacity

- Unrestricted: An arbitrary number of messages within the channel is possible
- Restricted: A maximum of *n* messages in the channel at the same time
  - When overflow occurs, either messages are rejected or sender is blocked until there is enough space in the channel

Direction

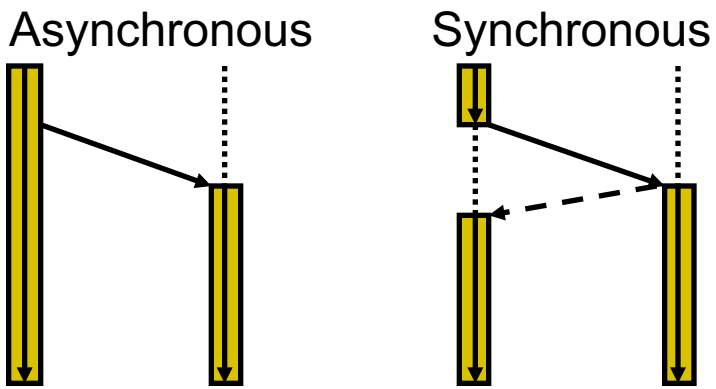
- Bidirectional: Send and receive possible in both directions
- Unidirectional: Send and receive only possible in one direction

Channel Characteristics (from the processes‘ view)

Default for the lecture

Synchronous vs. Asynchronous

- Asynchronous: The sending of a message is not blocking
- Synchronous: The sending of a message is blocking
  - Sender is blocked until the receiver took the massage; implementation through implicit acknowledgement



Distributed Algorithm

- Is executed on the nodes of the system in form of processes
- We consider only one process per node
- Thus, we often use both terms synonymously
- Different parts of the algorithm can run on different nodes
- The nodes communicate by exchanging messages over the channels

State of a Distributed Algorithm

Each node has a local state; it consists of the local variables of the algorithm

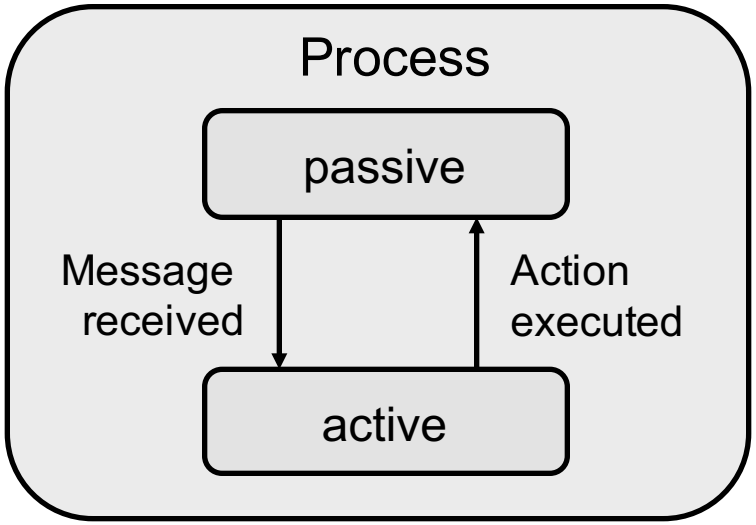
The state of a channel consists of the messages in it

The state of a distributed algorithm consists of

- the state of the nodes and
- the state of the channels

Processing Model

- Actions are triggered from outside
- Initial action is carried out when starting the algorithm
- Then, each process waits passively for the arrival of a message
- If a message arrives, a respective atomic action is executed
- Messages arriving in the meantime are buffered
- An action can change the local state of the process and send messages to other processes



P_1:
{Init}
SEND(Ping) TO P_2;

P_2:
{RECEIVE (Ping) from P_1}
SEND(Pong) TO P_1;

Synchronous System model

- The maximal duration of actions and the maximal delay of messages are assumed to be restricted and known
  - Example: Actions do not need time, messages need exactly one time unit. Then, distributed algorithms can run in synchronized rounds.

- In the synchronous model, decisions can be made due to the course of time
  - Example: If a message is not received within the maximal delay, one can conclude securely that an error occurred (node crash, message loss etc.)

Asynchronous System Model

- Actions and messages can last arbitrarily long
  or the limits are unknown

- From the course of time (e.g., timeouts) no (secure) information can be concluded
  - Example: If a message is not received before a timeout occurs, this can have several reasons
    - The message is delayed longer than usual
    - The message got lost
    - The sending process has send the message later than usual
    - The sending process has crashed before it could send the message

Synchronous vs. Asynchronous Model

Algorithms for problems in *synchronous* systems

- Often simple algorithms exist. However, they are mostly not directly or only restricted applicable because they make unrealistic assumptions

Algorithms for problems in *asynchronous* systems

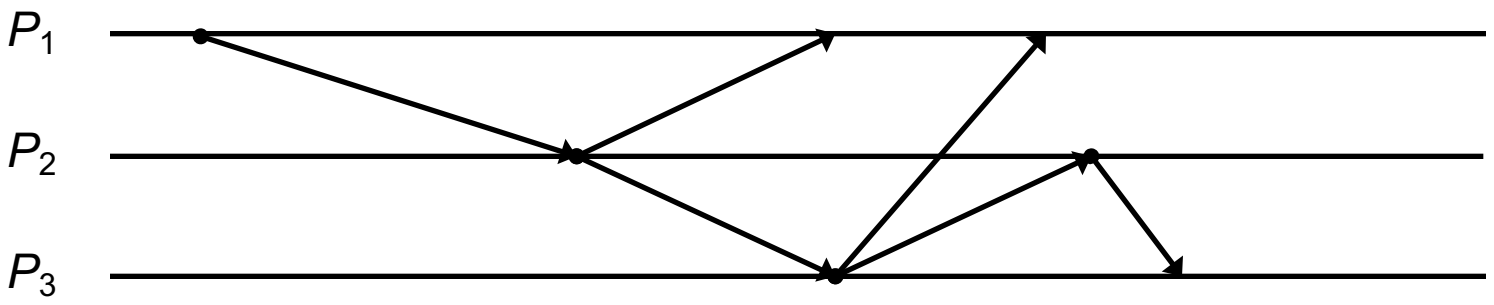
- For many problems, there are no algorithms
- If they exist, they are often complex and inefficient
- If there is an algorithm for a problem, it is usually well applicable

Atom Model

Default for the lecture

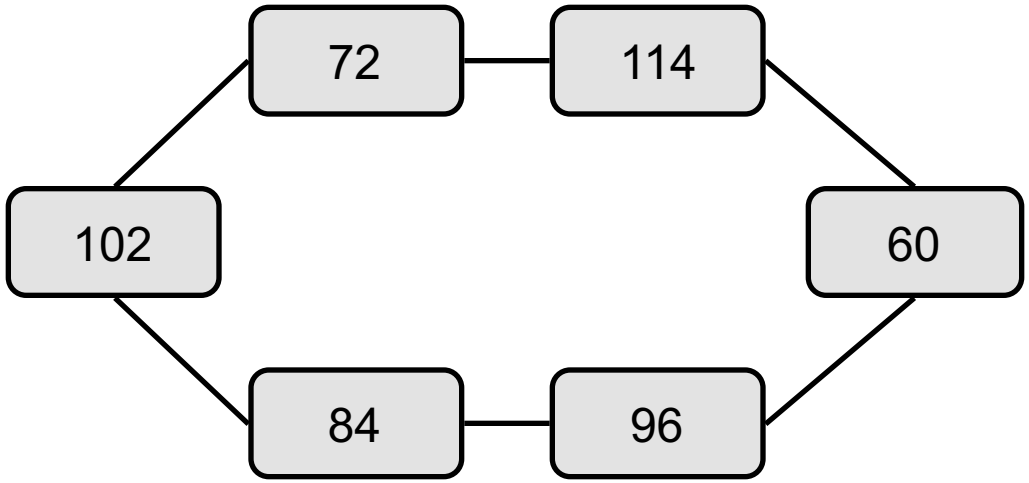
- Is a partly synchronized model
- Difference to asynchronous model: actions are timeless
- If a process receives a message, its local state changes according to the action and it can send messages

- Time-Space-Diagram: Graphical representation of (local events and) interactions of all processes



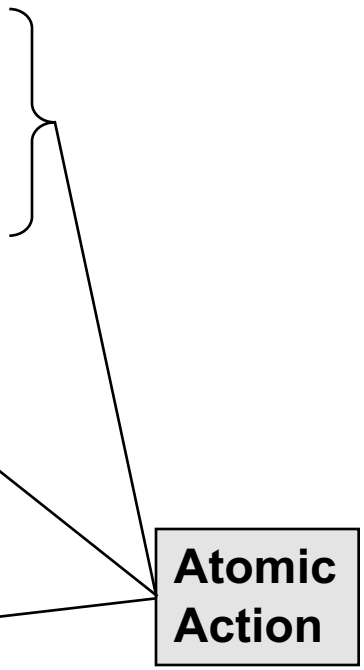
Example: Distributed Maximum-Algorithm

- A group of philosophers with different ages is sitting around a table and wants to determine the oldest among them
- Every philosopher can only communicate with both his two neighbors



Example: Distributed Maximum-Algorithm

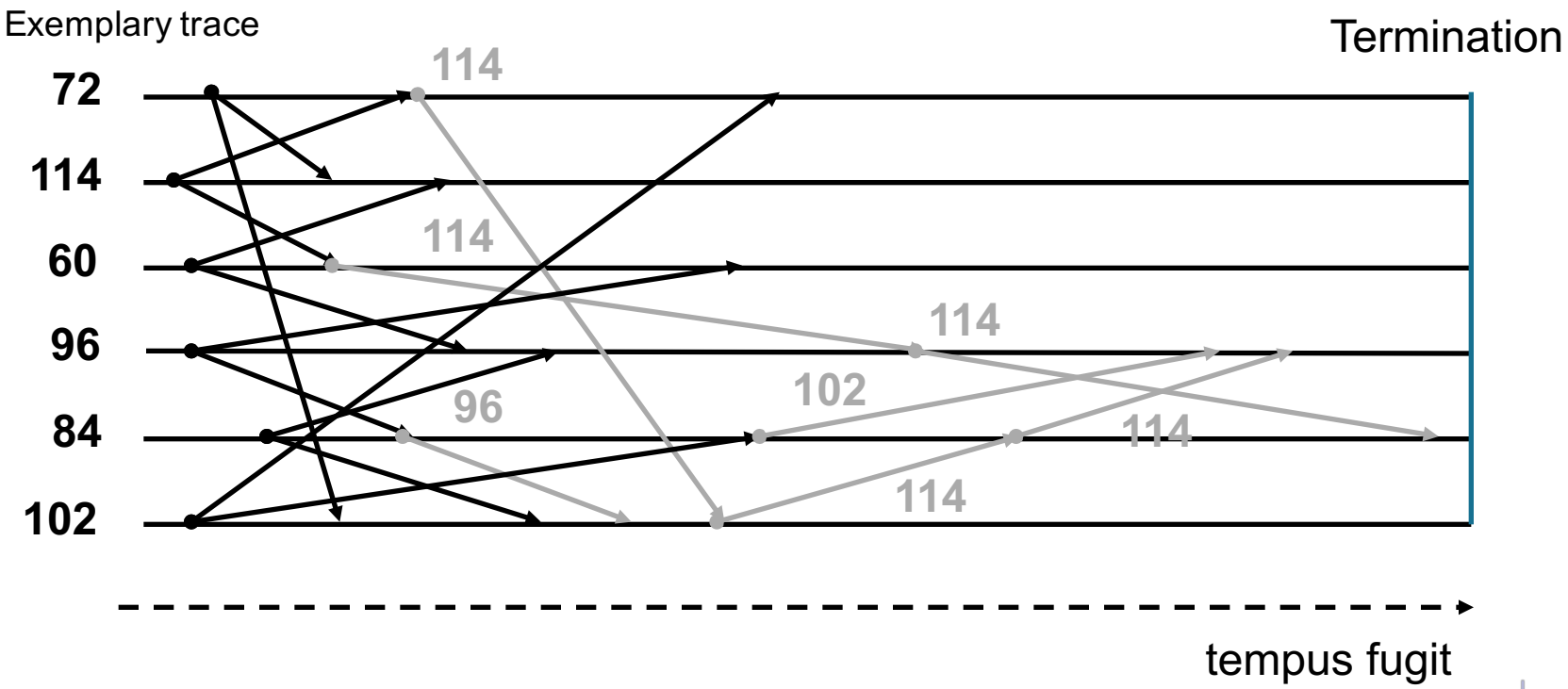
```
{INIT}: // is executed by every philosopher
   myMax := <own age>;
   SEND(<left neighbor>, myMax);
   SEND(<right neighbor>, myMax);

{RECEIVE (<left neighbor>, neighborMax)}
   IF (neighborMax > myMax) THEN
      myMax := neighborMax;
      SEND(<right neighbor>, myMax);
   ENDIF

{RECEIVE (<right neighbor, myMax)}
   IF (neighborMax > myMax) THEN
      myMax := neighborMax;
      SEND(<left neighbor>, myMax);
   ENDIF
```



A process can only receive messages after it was initialized.
Before that, arriving messages are stored and, after the initialization, delivered in the order they arrived.

Example: Distributed Maximum-Algorithm

Due to different message latencies, traces can vary



Example: Distributed Maximum-Algorithm

Are there several possible traces for one input?

Does every trace lead to the correct result? (here 114)?

How many messages are needed at least and when does this case occur?

How many messages are needed at most or on average? What does that depend on?

How should the philosophers know to start the algorithm?

Is it sufficient if an arbitrary philosopher starts the algorithm?

Example: Distributed Maximum-Algorithm

Does the algorithm always terminate?

How long does it take at least or at most or on average until the algorithm terminates?

How does a single philosopher recognize termination?

What happens if several philosophers have the same age?

Characteristics of Algorithms

Deterministic vs. Non-deterministic

- Deterministic:
  Always the same *trace* with the same input
- Non-deterministic:
  Different *traces* with the same input possible

Determined vs. Not determined

- Determined: Always the same *result* with the same input
  - Deterministic algorithms are always determined
  - Non-deterministic algorithms can be determined
    (e.g. Quicksort with randomly chosen Pivot-element)
- Not determined:
  Different *results* with the same input possible

Characteristics of Algorithms

Terminating: Terminate for each (valid) input after a limited number of steps

Partially correct: If they terminate, they always deliver a correct result

Totally correct: Terminating and partially correct

Characteristics of Distributed Algorithms

Are (apart from few exceptions) non deterministic. Reasons are, e.g., unpredictable message delay and varying processor speed

- Non determinism can be counteracted with synchronization mechanisms
- But that decreases the degree of parallelism

There are both determined and not determined distributed algorithms

Characteristics of Distributed Algorithms

- Distributed algorithms shall usually be totally correct
- But not for every problem to be solved there is a totally correct algorithm
- Often, an algorithm with weakened requirements
  (e.g., with respect to termination) can help

- Some algorithms shall not terminate
  (e.g., continuous clock synchronization)
- Then, often the fulfillment of other conditions is required
  (e.g., of an invariant)

Complexity of Distributed Algorithms

Message Complexity

Variable Time Complexity

- Actions timeless
- Messages need at most one time unit

Unit Time Complexity

- Actions timeless
- Messages need exactly one time unit

Attention: Not always

variable time complexity ≤ unit time *complexity*

Reason: The variable time complexity allows traces which are not allowed by the unit time complexity

Literature

1. S. Strogatz. Exploring complex networks. Nature, 410:268--276, 2001.
2. D. Watts and S. Strogatz. Collective Dynamics of 'Small-World' Networks. Nature, 393:440--442, 1998.
3. Akkoyunlu, E. A., Ekanadham, K., and Huber, R. V. Some constraints and tradeoffs in the design of network communications. In Proceedings of the Fifth ACM Symposium on Operating Systems Principles. ACM, New York, NY, 67-74, 1975.