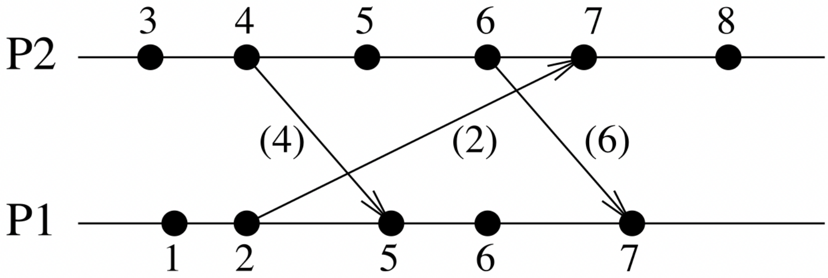
Dependable Distributed Systems – 5880V/UE

Part 1b: Distributed Systems: Models, Time – 2021-10-04

Prof. Dr. Hans P. Reiser | WS 2021/22

UNIVERSITÄT PASSAU



This week's lecture

1. Models of distributed systems
   - Modelling processes
   - Modelling communication
   - Modelling time
2. Relevance of time in distributed systems
   - Time and the order of events
   - Problem of clock synchronization
3. Logical clocks
   - Lamport clocks
   - Vector clocks

This week's lecture

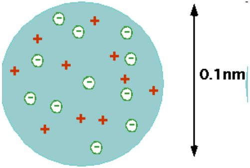
1. Models of distributed systems
   - Modelling processes
   - Modelling communication
   - Modelling time
2. Relevance of time in distributed systems
   - Time and the order of events
   - Problem of clock synchronization
3. Logical clocks
   - Lamport clocks
   - Vector clocks

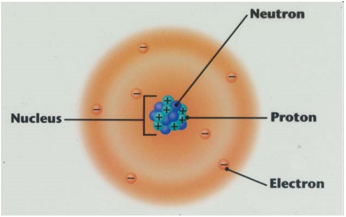
Models

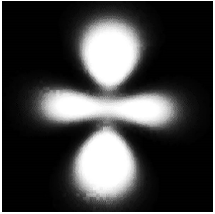
- Basic idea: a *model* is a simplification of an object (= system)
  - Allows us to reason about it
- A *model* for an object is a collection of attributes and a set of rules that govern how these attributes interact
  - Also called a theory
- Two important facts:
  - There is no single correct model
  - Answering different types of questions usually requires different models

Models of the atom

- Billiard Ball Model (1803) – John Dalton: atom: small solid sphere
- Plumb Pudding Model (1897) – Joseph John Thomson
- Solar System Model – Ernest Rutherford (1911), Neils Bohr (1913)
- Electron Cloud Model (1920's)







Model $\Rightarrow$ Assumptions

- Two processes, A and B, communicate by sending and receiving messages on a bidirectional channel. Neither process can fail. However, the channel can experience transient failures, resulting in the loss of a subset of the messages that have been sent.

Model $\Rightarrow$ Assumptions

- Two processes, A and B, communicate by sending and receiving messages on a bidirectional channel. Neither process can fail. However, the channel can experience transient failures, resulting in the loss of a subset of the messages that have been sent.

What are the **assumptions** above?

Good models

- A model is *accurate* to the extent that analyzing it yields truths about the object of interest
- A model is *tractable* if such an analysis is actually possible

Good models

- A model is *accurate* to the extent that analyzing it yields truths about the object of interest
- A model is *tractable* if such an analysis is actually possible
- Defining an accurate model is not difficult;
  defining an accurate+tractable model is
  - An accurate+tractable model will include exactly those attributes that affect the phenomena of interest
  - Level of detail is a key issue

Good models

- In building models for distributed systems, we typically seek answers to two fundamental questions:
- **Feasibility.** What classes of problems can be solved?
  - Can head-off wasted effort in design, implementation, and testing
- **Cost.** For those classes that can be solved, how expensive must the solution be?
  - Support evaluating any solution we devise
  - Avoid designs requiring protocols that are inherently slow or expensive

Distributed systems

- Distributed systems: models about processes
- Multiple processes communicating over narrow bandwidth, high-latency channels, with some processes and channels faulty

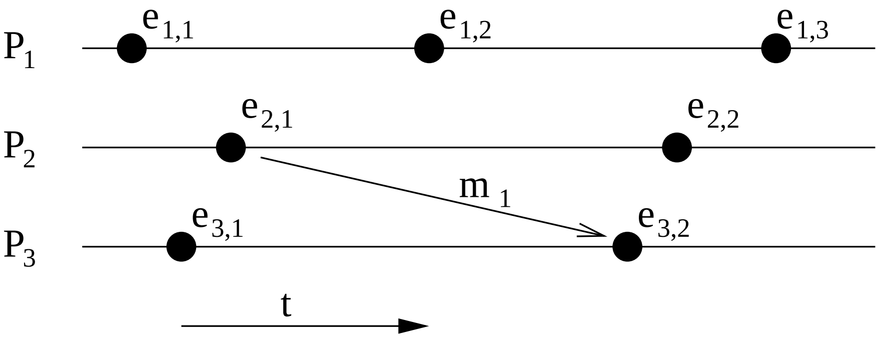
This week's lecture

1. Models of distributed systems
   - Modelling processes
   - Modelling communication
   - Modelling time
2. Relevance of time in distributed systems
   - Time and the order of events
   - Problem of clock synchronization
3. Logical clocks
   - Lamport clocks
   - Vector clocks

Modelling processes

- Node, computer, process ($A, B, C, \ldots$ or $P_1, P_2, P_3, \ldots$):
  - Used as synonyms
  - Independent, active instances in the distributed systems
  - Interaction with other participants with messages (usually)
- Global state
  - Vector $S = [S_1, S_2, \ldots, S_n]$ of local states S_i of all nodes
  - (sometimes: state of communication channels also relevant)
  - We can easily use S in formal arguments, but it is distributed over all nodes, and as such it is not easily observable
- *Step:* atomic local transition of a process P_i from state S_i to S_i'
  - Designated with s, σ (step) or e (event)
  - Spontaneous or triggered by message

Time-space diagram



Modelling faulty processes

- There are many failure models for distributed systems; all based on assigning responsibility for faulty behaviour to system components:
  - processors
  - communications channels

Modelling faulty processes

- There are many failure models for distributed systems; all based on assigning responsibility for faulty behaviour to system components:
  - processors
  - communications channels
- We count *faulty components*, not occurrences of faulty behavior
  - In classical work on fault-tolerant computing systems: occurrences of faulty behaviour are counted
- *t*-fault tolerant
  - system satisfies its **specification**, provided that **no more than *t*** of its components are **faulty**

    (we also use *f* or *k* instead of *t*, and *n* for the total number of nodes)

Faulty processes

- **Crash**
  - A process fails by *stopping* (does not execute any further steps). The process will remain in the crash state forever.
  - **Fail-stop**
    - All other operational processes reliably detect the crash
  - **Fail-silent**
    - Other processes are not reliably notified about the crash
    - If the communication system is not synchronous (see later), it is hard to distinguish a crahsed from a slow process

Faulty processes

- **Crash**
  - A process fails by *stopping* (does not execute any further steps). The process will remain in the crash state forever.
  - **Fail-stop**
    - All other operational processes reliably detect the crash
  - **Fail-silent**
    - Other processes are not reliably notified about the crash
    - If the communication system is not synchronous (see later), it is hard to distinguish a crahsed from a slow process
- **Crash-Recovery**
  - A correct process can fail and recover again (for a finite number of times)
  - A process is faulty if it stops forever, or if it infinitely often fails and recovers.
  - Special case "**omission**" (fault causes only loss of messages, not loss of local state)

Faulty processes

- **Eavesdropping faults**
  - Faulty processes behave as in the crash-recovery model (i.e., interactions with other processes occur according to specification or not at all)
  - In addition: Faulty processes may pass internal information to an external third party

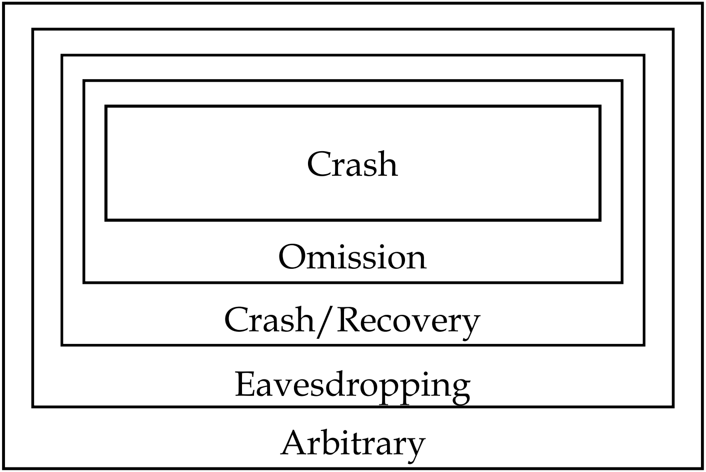
Faulty processes

- **Eavesdropping faults**
  - Faulty processes behave as in the crash-recovery model (i.e., interactions with other processes occur according to specification or not at all)
  - In addition: Faulty processes may pass internal information to an external third party
- **Byzantine faults**
  - Faulty processes can exhibit arbitrary behaviour
  - Usually: *"arbitrary"* restricted to *computationally feasible operations*
    - Example: cannot break strong cryptography

Important: these are models $\Rightarrow$ assumptions!

Hierarchy of fault classes

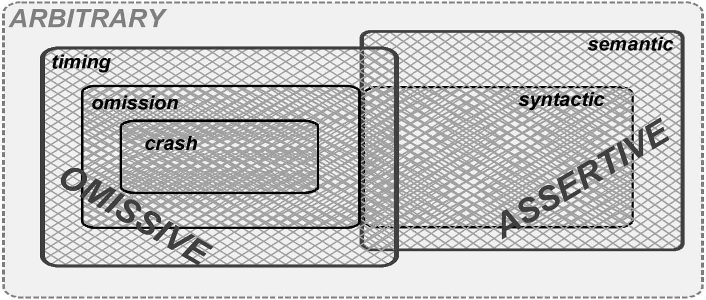
Hierarchical relations (according to RSDP):



Severity of faults

Classification according to DSSA:

- More detailed differentiation of some faults:
- *Assertive faults* / Value faults
  - Data in protocol-conforming interactions modified syntactically or semantically
- *Omissive faults*
  - Actions do not take place, or too late/too early



Overview

Modelling point-to-point communication

Point-to-point communication

- Sending message *m*
  - Process *p* sends a message *m* to *B* by executing *send(m)*
  - *send(m)* adds message *m* to *p*'s send queue
- Channel transports *m*
  - from *p*'s send queue
  - to *q*'s receive queue
- Reception of message *m*
  - q's runtime system calls *deliver(m)*, which
  - delivers the message *m* to process *q*, and
  - removes *m* from the receive queue.

Communication faults

Modelling message loss

- “Fair-loss links”: weak but useful model
  - No infinite message losses (if periodically repeating message, it eventually will arrive)
  - No infinite number of message duplications
  - No spurious messages (delivery of messages that have not been sent)
- “Stubborn links”:
  - Message is repeated infinitely often over fair-loss link
  - Results in reliable delivery without spurious messages
- “Perfect links” / “reliable links”
  - No loss of messages
  - No duplication
  - No spurious messages
- “Authenticated perfect links”
  - Perfect link with correct sender identification

This week's lecture

1. Models of distributed systems
   - Modelling processes
   - Modelling communication
   - Modelling time
2. Relevance of time in distributed systems
   - Time and the order of events
   - Problem of clock synchronization
3. Logical clocks
   - Lamport clocks
   - Vector clocks

Synchronous versus asynchronous systems

- *Asynchronous* system: we make no assumptions about process execution speeds and/or message delivery delays
- *Synchronous* system: we do make assumptions about these parameters
  - The relative speeds of processes are assumed to be bounded
  - The delays associated with communications channels also are

Synchronous versus asynchronous systems

- *Asynchronous* system: we make no assumptions about process execution speeds and/or message delivery delays
- *Synchronous* system: we do make assumptions about these parameters
  - The relative speeds of processes are assumed to be bounded
  - The delays associated with communications channels also are
- Postulating that a system is asynchronous is a non-assumption
  - **Every** system “is” asynchronous
    (i.e. satisfies assumptions that are made)
  - Algorithm for asynchronous system
    . . . can be used on every system!

Synchronous versus asynchronous systems

- Postulating that a system is synchronous constrains how processes and communications channels are implemented
  - Scheduler that multiplexes processors must not violate the constraints on process execution speeds
  - This implies that all processors in the system have access to approximately rate-synchronized real-time clocks
  - Queuing delays, unpredictable routings, and retransmission due to errors must not violate the constraints on channel delays

Synchronous versus asynchronous systems

- Postulating that a system is synchronous constrains how processes and communications channels are implemented
  - Scheduler that multiplexes processors must not violate the constraints on process execution speeds
  - This implies that all processors in the system have access to approximately rate-synchronized real-time clocks
  - Queuing delays, unpredictable routings, and retransmission due to errors must not violate the constraints on channel delays

- In asserting that a system is synchronous, we rule out certain system behaviors
  - This enables us to employ
    - simpler protocols (complexity)
    - cheaper protocols (overhead)

    than required in an asynchronous system

Synchronous versus asynchronous systems

There are other models in the spectrum:

- Partial synchrony
- Timed-asynchronous
- Wormholes
- . . .

More details see later lectures. . .

Wrap-up of this part

What you should remember about models and distributed systems:
What you should remember about models:

- How to model processes and process failures
- How to model communication in a distributed system
- How to model time in a distributed system

Time in distributed systems

1. Models of distributed systems
   - Modelling processes
   - Modelling communication
   - Modelling time
2. Relevance of time in distributed systems
   - Time and the order of events
   - Problem of clock synchronization
3. Logical clocks
   - Lamport clocks
   - Vector clocks

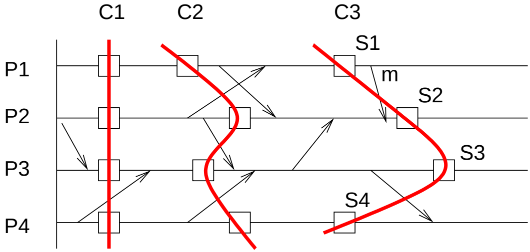
Time and the order of events: `make` example

```
% vi test.c
[...]

% make
make: Warning: File ‘test.c’ has modification time 94 s in the future
cc test.c -o test
make: warning: Clock skew detected. Your build may be incomplete.

% make
make: Warning: File ‘test.c’ has modification time 93 s in the future
cc test.c -o test
make: warning: Clock skew detected. Your build may be incomplete.
```

- Make tool does not recognize that *test.c was not modified*. . .
- Or worse: modification is not detected, *file will not get compiled!*

Time and the order of events: Checkpoints

- Given: A distributed system of interacting nodes
- For a consistent checkpoint, the local state of a set of nodes shall be saved at a fixed time.
- Problem: inconsistency may occur if the local clocks differ (C3)



C3: Effect of *m* is included in S_2, but not in S_1!

Time and the order of events

Examples: monitoring, debugging, determining the root causes of errors

- Distributed data acquisition for logging or debugging purposes
- Real order cannot be reconstructed if locally generated timestamps are not synchronized

Problem of clock synchronization

- There are no completely identical physical clocks
  - Different initialization (constant offset)
  - Different speed (frequency error)
  - Subject to environmental conditions (e.g., component ageing, temperature dependency)

$\Rightarrow$ Without synchronization errors may continuously grow!

- Common clock for all the nodes of a distributed system (usually) not feasible
- Central reference clocks
  - e.g., radio transmission (WWV, DCF77, GPS)

  with technically limited accuracy

Time in distributed systems

Logical clocks

The basic idea (Lamport's logical clock):

- Order of events on different components is relevant
  - only if one event could influence another, i.e.,
  - only if the components interact

Logical clocks

The basic idea (Lamport's logical clock):

- Order of events on different components is relevant
  - only if one event could influence another, i.e.,
  - only if the components interact

- Synchronization upon interaction (i.e., communication), such that
  - timestamps respect real order of events

Logical clocks

Description of the algorithm

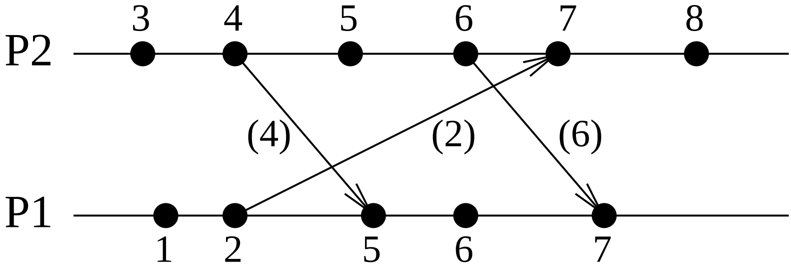
- Each process P_i has a local clock C_i that counts as follows:
  - Local action or send action by P_i: $C_i = C_i + 1$
    timestamp of the action: value after increasing
  - Message m carries a timestamp t_m
    t_m equals the timestamp of the send action
  - P_i with local clock value C_i
    receives a message m with timestamp t_m:

    $C_i = max(C_i, t_m) + 1$

    Resulting C_i: timestamp of the reception event

Logical clocks

Example:



Logical clocks

Properties

- Causal dependency of an event e_2 on event e_1 (i.e., e_1 may have influenced e_2, shorthand $e_1 \rightarrow e_2$): $e_1 \rightarrow e_2 \Rightarrow t(e_1) < t(e_2)$
- The other direction $t(e_1) < t(e_2) \Rightarrow e_1 \rightarrow e_2$ does not hold!
- The logical timestamps create a partial order on the set of all events!

Logical clocks

Properties

- Causal dependency of an event e_2 on event e_1 (i.e., e_1 may have influenced e_2, shorthand $e_1 \rightarrow e_2$): $e_1 \rightarrow e_2 \Rightarrow t(e_1) < t(e_2)$
- The other direction $t(e_1) < t(e_2) \Rightarrow e_1 \rightarrow e_2$ does not hold!
- The logical timestamps create a partial order on the set of all events!

Extentions to obtain a total order

- Timestamp of event at process i: (local time C_i, process number i)
- Order relation: $(C_i, i) < (C_k, k) \Leftrightarrow C_i < C_k \vee (C_i = C_k \wedge i < k)$

Logical clocks: vector clocks

Vector clocks:

- Each process has a local clock, which consists of a vector of N values (N = number of existing nodes)
- Implementation:
  - Initialization of each vector with the zero vector
  - Between two local events on a process P_i, the corresponding component of the vector is incremented: $C_i[i] = C_i[i] + 1$
  - Messages carry a timestamp t equal to the vector C_i of the sender
  - A process P_i that receives a message
    - incrementes own component in time vector
    - combines it with the received time vector t

    $C_i[i] := C_i[i] + 1$
    For $k = 1 \ldots N : C_i[k] := max(C_i[k], t[k])$

Logical clocks: vector clocks

Properties:

- Creates a causal order: $t(e_1) < t(e_2) \Leftrightarrow e_1 \rightarrow e_2$
- Definition of "<":
  $t(e_1) = (a_1, a_2, \ldots, a_n)$ and $t(e_2) := (b_1, b_2, \ldots, b_n)$:
  $t(e_1) < t(e_2) \Leftrightarrow (\forall i : a_i \leq b_i) \wedge (\exists i : a_i < b_i)$

Advantage:

- Exact statement about causal relations of events

Disadvantage:

- High communication overhead
  - vector of N elements in each message
  - scales poorly for large N

Summary

Modelling distributed systems

- Processes (including failures), communication, time

Time

- Relevance of time
- Logical clocks: Lamport clocks, vector clocks