# Large-scale Data Systems

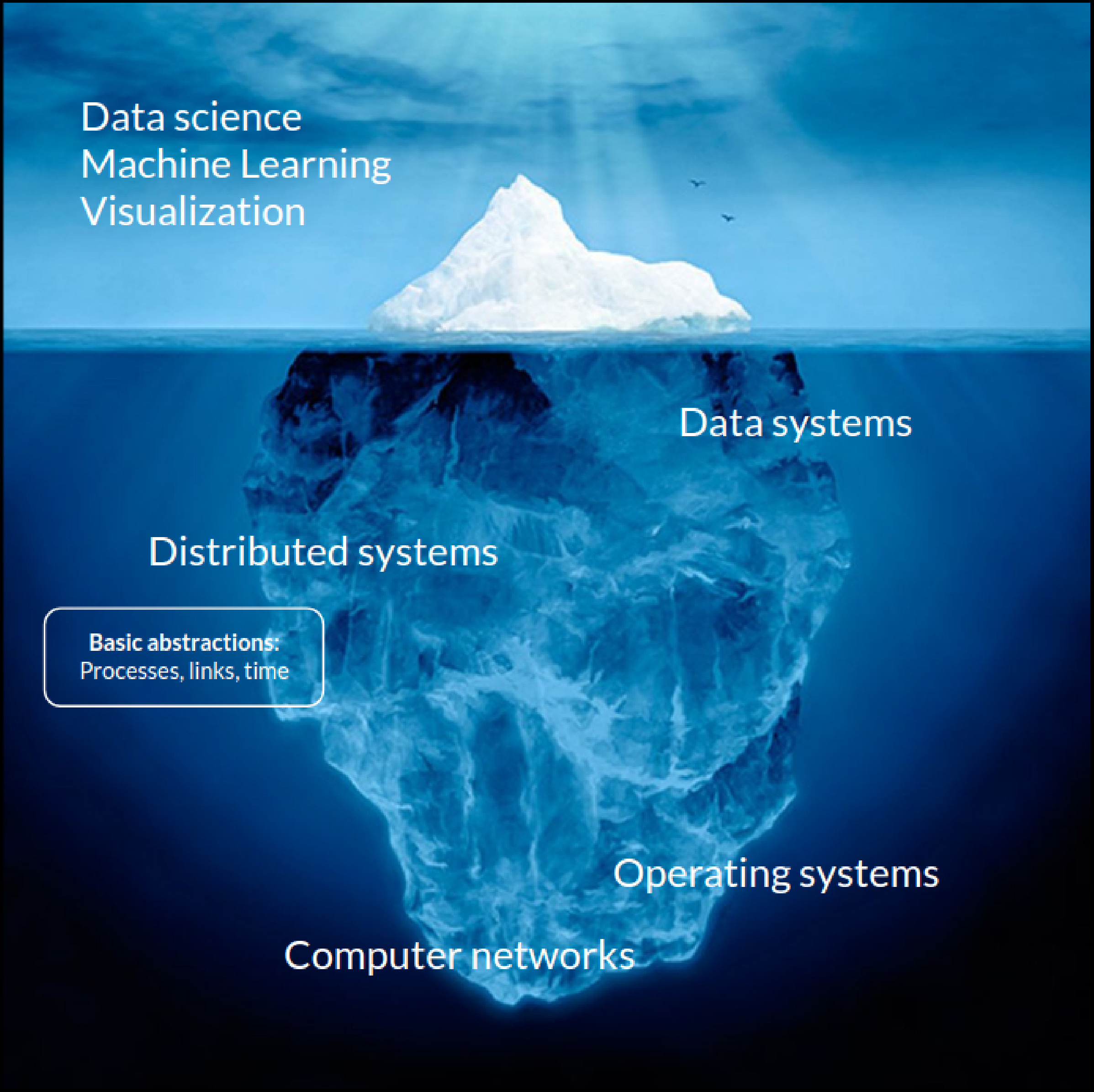
Lecture 2: Basic distributed abstractions

Prof. Gilles Louppe
g.louppe@uliege.be

LIÈGE université

# Today

- Define basic abstractions that capture the fundamental characteristics of distributed systems:
  - Process abstractions
  - Link abstractions
  - Timing abstractions
- A distributed system model = a combination of the three categories of abstractions.


Data science
Machine Learning
Visualization
Data systems
Distributed systems
Basic abstractions:
Processes, links, time
Operating systems
Computer networks

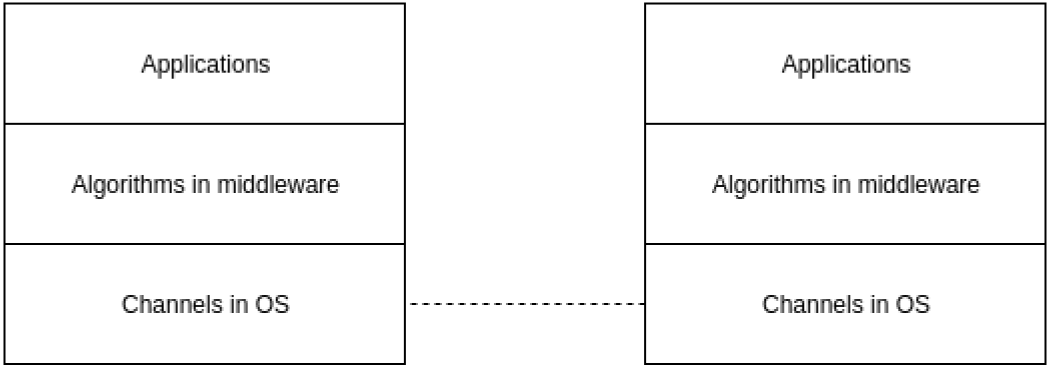
# Why distributed abstractions?

Reliable distributed applications need underlying services stronger than transport protocols (e.g., TCP or UDP).

*"All problems in computer science can be solved by another level of indirection" - David Wheeler.*

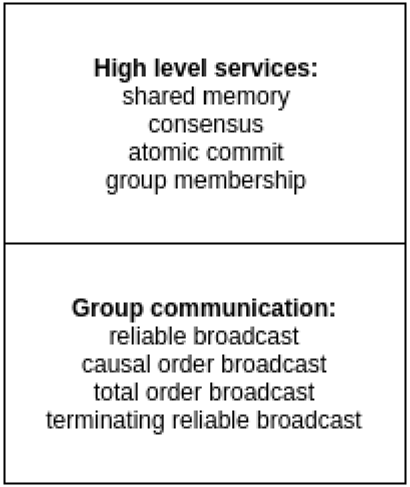
## Distributed abstractions

- Core of any distributed system is a set of distributed algorithms.
- Implemented as a middleware between network (OS) and the application.



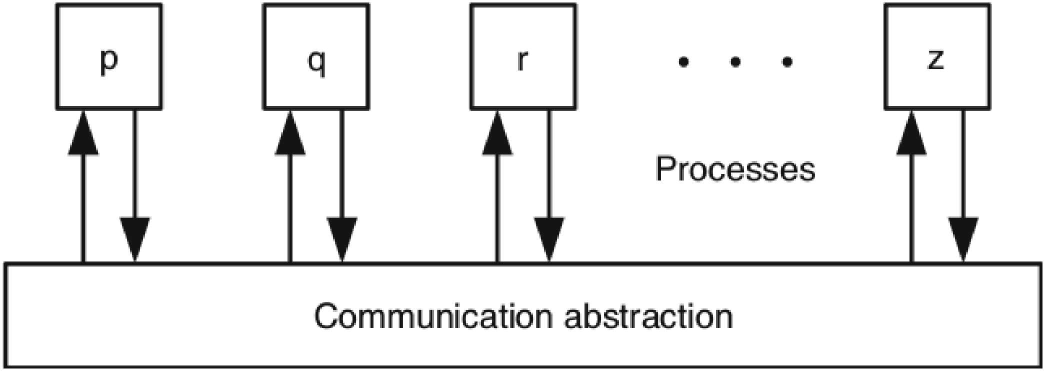
## Network protocols are not enough

- Communication
  - Reliability guarantees (e.g. with TCP) are only offered for one-to-one communication (client-server).
  - How to do group communication?
- High-level services
  - Sometimes one-to-many communication is not enough.
  - Need reliable higher-level services.
- Strategy: build complex distributed systems in a bottom-up fashion, from simpler ones.



# Distributed computation

# Distributed algorithms




- A distributed algorithm is a distributed collection $\Pi = \{p, q, r, ...\}$ of $N$ processes implemented by identical automata.
- The automaton at a process regulates the way the process executes its computation steps.
- Processes jointly implement the application.
    - Need for coordination.

# Event-driven programming

- Every process consists of modules or components.
  - Modules may exist in multiple instances.
  - Every instance has a unique identifier and is characterized by a set of properties.
- Asynchronous events represent communication or control flow between components.
  - Each component is constructed as a state-machine whose transitions are triggered by the reception of events.
  - Events carry information (sender, message, etc)

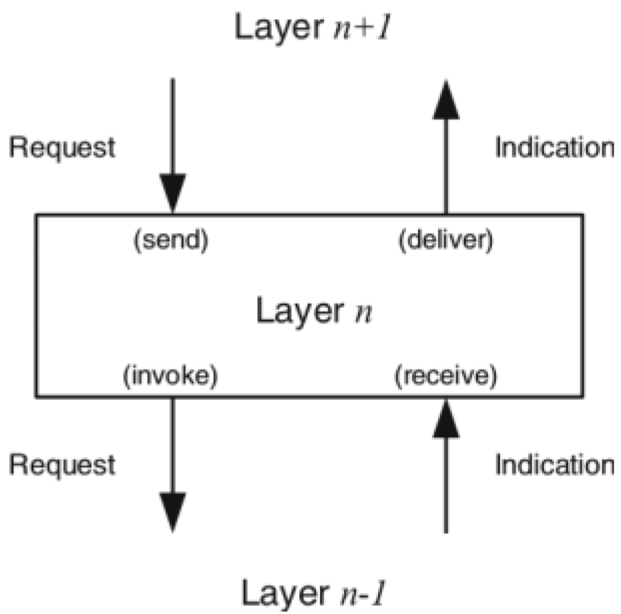
## Reactive programming model

**upon event** $\langle\, co_1, Event_1 \mid att_1^1, att_1^2, \ldots \rangle$ **do**
    do something;
    **trigger** $\langle\, co_2, Event_2 \mid att_2^1, att_2^2, \ldots \rangle$;    *// send some event*

**upon event** $\langle\, co_1, Event_3 \mid att_3^1, att_3^2, \ldots \rangle$ **do**
    do something else;
    **trigger** $\langle\, co_2, Event_4 \mid att_4^1, att_4^2, \ldots \rangle$;    *// send some other event*

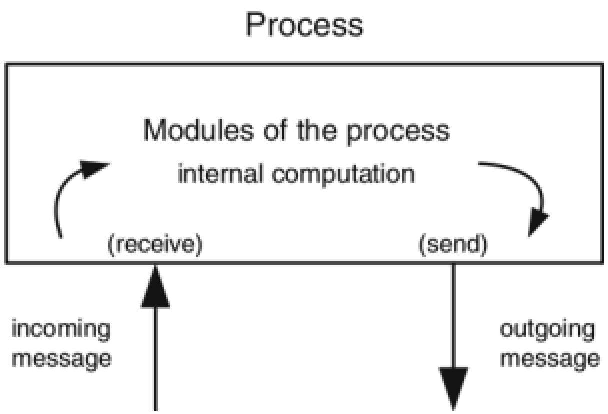
Effectively, a distributed algorithm is described by a set of event handlers.

## Layered modular architecture




- Components can be composed locally to build software stacks.
  - The top of the stack is the application layer.
  - The bottom of the stack the transport or network layer.
- Distributed programming abstraction layers are typically in the middle.
- We assume that every process executes the code triggered by events in a mutually exclusive way, without concurrently processing $\geq 2$ events.

## Execution




- The execution of a distributed algorithm is a sequence of steps executed by its processes.
- A process step consists in
  - receiving a message from another process,
  - executing a local computation,
  - sending a message to some process.
- Local messages between components are treated as local computation.
- We assume deterministic process steps (with respect to the message received and the local state prior to executing a step).

## Example: Job handler

**Module 1.1:** Interface and properties of a job handler

**Module:**

**Name:** JobHandler, **instance** *jh*.

**Events:**

**Request:** $\langle$ *jh*, *Submit* | *job* $\rangle$: Requests a job to be processed.

**Indication:** $\langle$ *jh*, *Confirm* | *job* $\rangle$: Confirms that the given job has been (or will be) processed.

**Properties:**

**JH1:** *Guaranteed response:* Every submitted job is eventually confirmed.

**Algorithm 1.1:** Synchronous Job Handler

**Implements:**

JobHandler, **instance** *jh*.

**upon event** $\langle$ *jh*, *Submit* | *job* $\rangle$ **do**

process(*job*);

**trigger** $\langle$ *jh*, *Confirm* | *job* $\rangle$;

**Algorithm 1.2:** Asynchronous Job Handler

**Implements:**
JobHandler, **instance** *jh*.

**upon event** $\langle$ *jh*, *Init* $\rangle$ **do**
*buffer* := $\emptyset$;

**upon event** $\langle$ *jh*, *Submit* | *job* $\rangle$ **do**
*buffer* := *buffer* $\cup$ $\{$*job*$\}$;
**trigger** $\langle$ *jh*, *Confirm* | *job* $\rangle$;

**upon** *buffer* $\neq \emptyset$ **do**
*job* := selectjob(*buffer*);
process(*job*);
*buffer* := *buffer* $\setminus$ $\{$*job*$\}$;

**Module 1.2:** Interface and properties of a job transformation and processing abstraction

**Module:**

**Name:** TransformationHandler, **instance** *th*.

**Events:**

**Request:** $\langle$ *th*, *Submit* | *job* $\rangle$: Submits a job for transformation and for processing.

**Indication:** $\langle$ *th*, *Confirm* | *job* $\rangle$: Confirms that the given job has been (or will be) transformed and processed.

**Indication:** $\langle$ *th*, *Error* | *job* $\rangle$: Indicates that the transformation of the given job failed.

**Properties:**

**TH1:** *Guaranteed response:* Every submitted job is eventually confirmed or its transformation fails.

**TH2:** *Soundness:* A submitted job whose transformation fails is not processed.

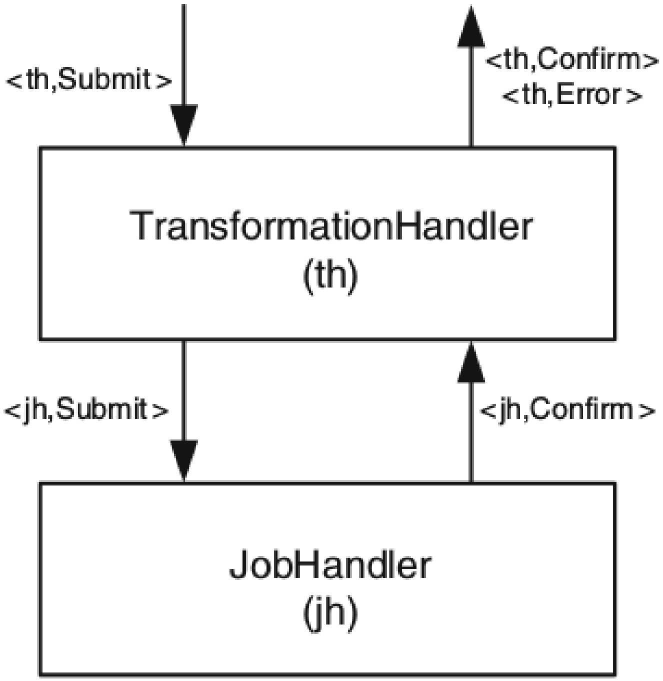



**Figure 1.3:** A stack of job-transformation and job-handler modules

**Algorithm 1.3:** Job-Transformation by Buffering

**Implements:**
  TransformationHandler, **instance** *th*.

**Uses:**
  JobHandler, **instance** *jh*.

**upon event** $\langle$ *th*, *Init* $\rangle$ **do**
  $top := 1;$
  $bottom := 1;$
  $handling := \text{FALSE};$
  $buffer := [\perp]^M;$

**upon event** $\langle$ *th*, *Submit* | *job* $\rangle$ **do**
  **if** $bottom + M = top$ **then**
    **trigger** $\langle$ *th*, *Error* | *job* $\rangle$;
  **else**
    $buffer[top \bmod M + 1] := job;$
    $top := top + 1;$
    **trigger** $\langle$ *th*, *Confirm* | *job* $\rangle$;

**upon** $bottom < top \wedge handling = \text{FALSE}$ **do**
  $job := buffer[bottom \bmod M + 1];$
  $bottom := bottom + 1;$
  $handling := \text{TRUE};$
  **trigger** $\langle$ *jh*, *Submit* | *job* $\rangle$;

**upon event** $\langle$ *jh*, *Confirm* | *job* $\rangle$ **do**
  $handling := \text{FALSE};$

# Liveness and safety

- Implementing a distributed programming abstraction requires satisfying its correctness in all possible executions of the algorithm.
    - i.e., in all possible interleaving of steps.
- Correctness of an abstraction is expressed in terms of liveness and safety properties.
    - Safety: properties that state that nothing bad ever happens.
        - A safety property is a property such that, whenever it is violated in some execution $E$ of an algorithm, there is a prefix $E'$ of $E$ such that the property will be violated in any extension of $E'$.
    - Liveness: properties that state something good eventually happens.
        - A liveness property is a property such that for any prefix $E'$ of $E$, there exists an extension of $E'$ for which the property is satisfied.
- Any property can be expressed as the conjunction of safety property and a liveness property.

## Example 1: Traffic lights at an intersection

- Safety: only one direction should have a green light.
- Liveness: every direction should eventually get a green light.

## Example 2: TCP

- Safety: messages are not duplicated and received in the order they were sent.
- Liveness: messages are not lost.
  - i.e., messages are eventually delivered.

# Assumptions

- In our abstraction of a distributed system, we need to specify the assumptions needed for the algorithm to be correct.

- A distributed system model includes assumptions on:
    - failure behavior of processes and channels
    - timing behavior of processes and channels

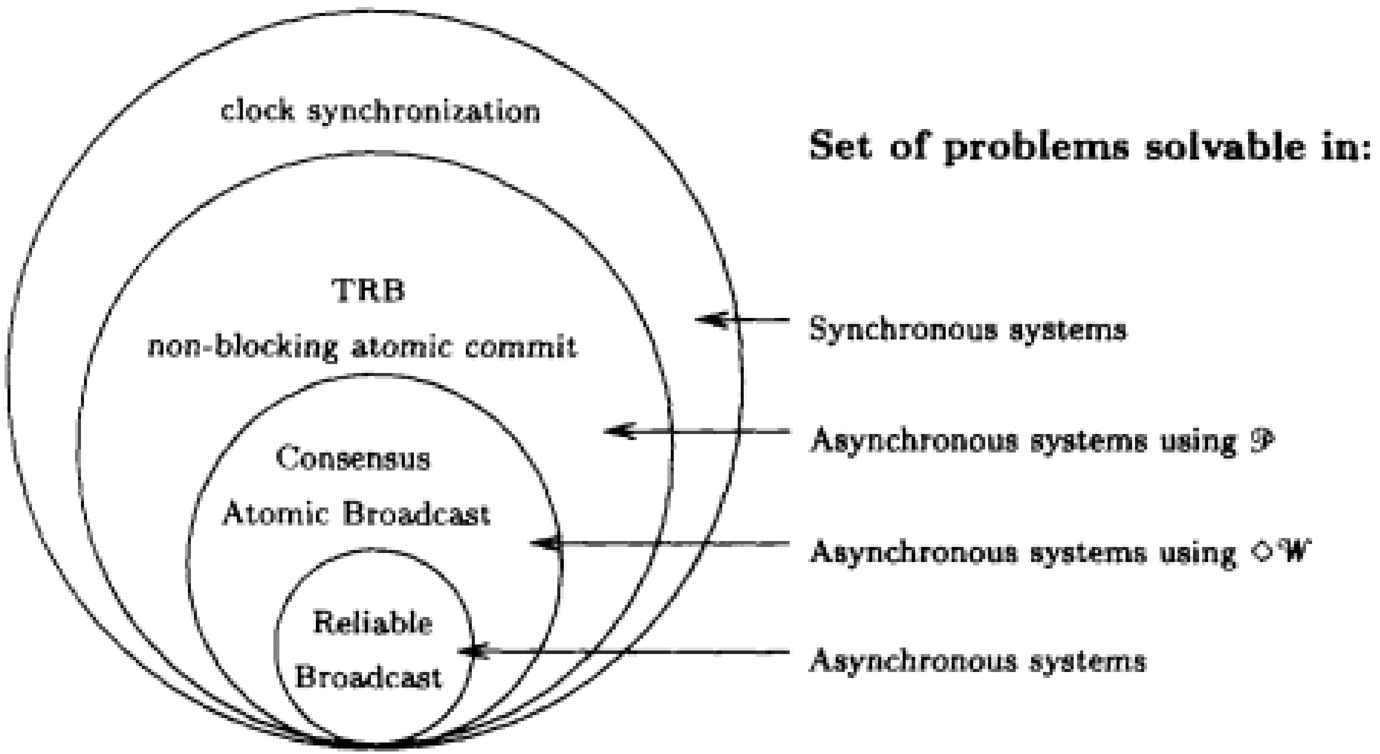



FIG. 9. Problem solvability in different distributed computing models.

*Together, these assumptions define sets of solvable problems.*

# Process abstractions

# Process failures

- Processes may fail in four different ways:
  - Crash-stop
  - Omissions
  - Crash-recovery
  - Byzantine / arbitrary
- Processes that do not fail in an execution are correct.

## Crash-stop failures

- A process stops taking steps.
  - Not sending messages.
  - Not receiving messages.
- We assume the crash-stop process abstraction by default.
  - Hence, do not recover.
  - [Q] Does this mean that processes are not allowed to recover?

## Omission failures

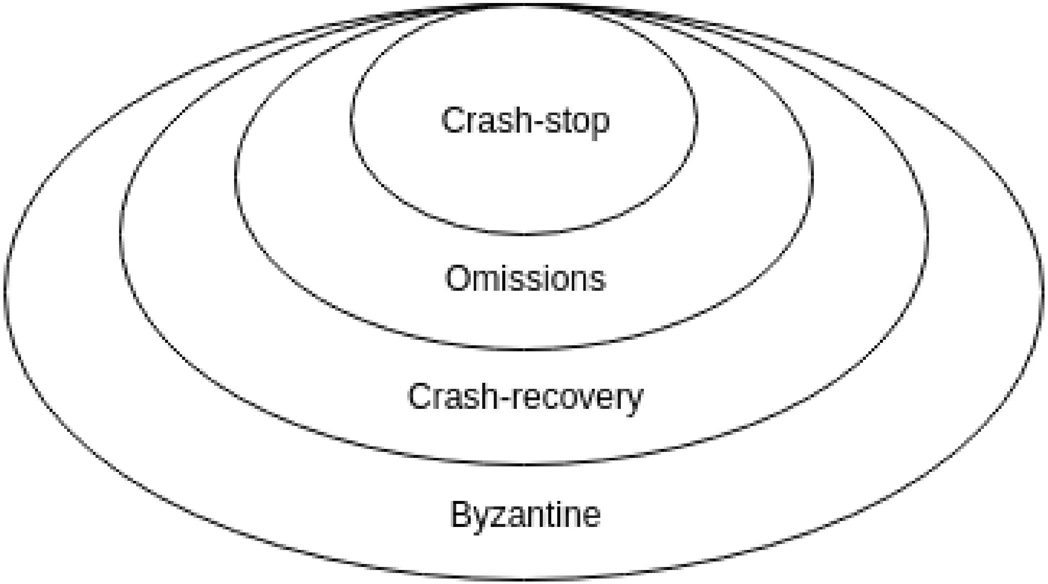
- Process omits sending or receiving messages.
  - Send omission: A process omits to send a message it has to send according to its algorithm.
  - Receive omission: A process fails to receive a message that was sent to it.
- Often, omission failures are due to buffer overflows.
- With omission failures, a process deviates from its algorithm by dropping messages that should have been exchanged with other processes.

## Crash-recovery failures

- A process might crash.
  - It stops taking steps, not receiving and sending messages.
- It may recover after crashing.
  - The process emits a `<Recovery>` event upon recovery.
- Access to stable storage:
  - May read/write (expensive) to permanent storage device.
  - Storage survives crashes.
  - E.g., save state to storage, crash, recover, read saved state, ...
- A failure is different in the crash-recovery abstraction:
  - A process is faulty in an execution if
    - It crashes and never recovers, or
    - It crashes and recovers infinitely often.
  - Hence, a correct process may crash and recover.

## Byzantine failures

- A process may behave arbitrarily.
  - Sending messages not specified by its algorithm.
  - Updating its state as not specified by its algorithm.
- Might behave maliciously, attacking the system.
  - Several malicious nodes might collude.

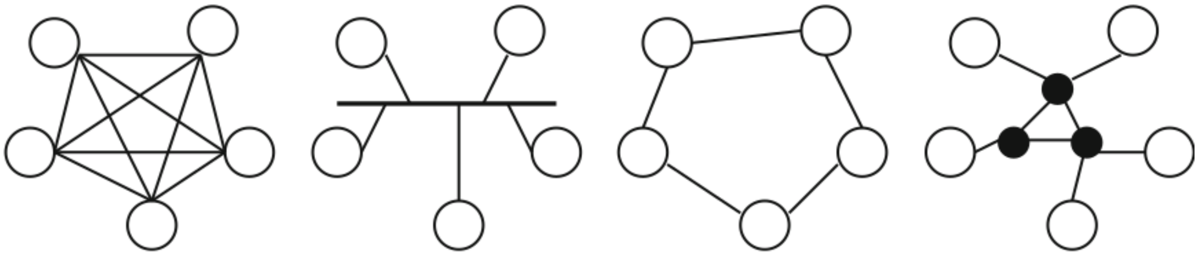



*Fault-tolerance hierarchy*

# Communication abstractions

# Links

- Every process may logically communicate with every other process (a).
- The physical implementation may differ (b-d).



(a) (b) (c) (d)

# Link failures

- Fair-loss links
  - Channel delivers any message sent, with non-zero probability.
- Stubborn links
  - Channel delivers any message sent infinitely many times.
  - Can be implemented using fair-loss links.
- Perfect links (reliable)
  - Channel delivers any message sent exactly once.
  - Can be implemented using stubborn links.
  - By default, we assume the perfect links abstraction.

**Exercice**
What abstraction do UDP and TCP implement?

## Stubborn links ($sl$)

**Module:**

**Name:** StubbornPointToPointLinks, **instance** *sl*.

**Events:**

**Request:** $\langle$ *sl*, *Send* | $q$, $m$ $\rangle$: Requests to send message $m$ to process $q$.

**Indication:** $\langle$ *sl*, *Deliver* | $p$, $m$ $\rangle$: Delivers message $m$ sent by process $p$.

**Properties:**

**SL1:** *Stubborn delivery:* If a correct process $p$ sends a message $m$ once to a correct process $q$, then $q$ delivers $m$ an infinite number of times.

**SL2:** *No creation:* If some process $q$ delivers a message $m$ with sender $p$, then $m$ was previously sent to $q$ by process $p$.

**Exercice**
Which property is safety/liveness/neither?

## Perfect links ($pl$)

**Module:**

**Name:** PerfectPointToPointLinks, **instance** *pl*.

**Events:**

**Request:** ⟨ *pl*, *Send* | $q$, $m$ ⟩: Requests to send message $m$ to process $q$.

**Indication:** ⟨ *pl*, *Deliver* | $p$, $m$ ⟩: Delivers message $m$ sent by process $p$.

**Properties:**

**PL1:** *Reliable delivery:* If a correct process $p$ sends a message $m$ to a correct process $q$, then $q$ eventually delivers $m$.

**PL2:** *No duplication:* No message is delivered by a process more than once.

**PL3:** *No creation:* If some process $q$ delivers a message $m$ with sender $p$, then $m$ was previously sent to $q$ by process $p$.

**Exercice**
Which property is safety/liveness/neither?

**Implements:**
PerfectPointToPointLinks, **instance** *pl*.

**Uses:**
StubbornPointToPointLinks, **instance** *sl*.

**upon event** $\langle$ *pl*, *Init* $\rangle$ **do**
*delivered* := $\emptyset$;

**upon event** $\langle$ *pl*, *Send* | $q, m$ $\rangle$ **do**
**trigger** $\langle$ *sl*, *Send* | $q, m$ $\rangle$;

**upon event** $\langle$ *sl*, *Deliver* | $p, m$ $\rangle$ **do**
**if** $m \notin$ *delivered* **then**
*delivered* := *delivered* $\cup \{m\}$;
**trigger** $\langle$ *pl*, *Deliver* | $p, m$ $\rangle$;

**Exercice**
How does TCP efficiently maintain its `delivered` log?

## Correctness of $pl$

- PL1. Reliable delivery
  - Guaranteed by the Stubborn link abstraction. (The Stubborn link will deliver the message an infinite number of times.)
- PL2. No duplication
  - Guaranteed by the log mechanism.
- PL3. No creation
  - Guaranteed by the Stubborn link abstraction.

# Timing abstractions

# Timing assumptions

- Timing assumptions correspond to the behavior of processes and links with respect to the passage of time. They relate to
  - different processing speeds of processes;
  - different speeds of messages (channels).
- Three basic types of system:
  - Asynchronous system
  - Synchronous system
  - Partially synchronous system

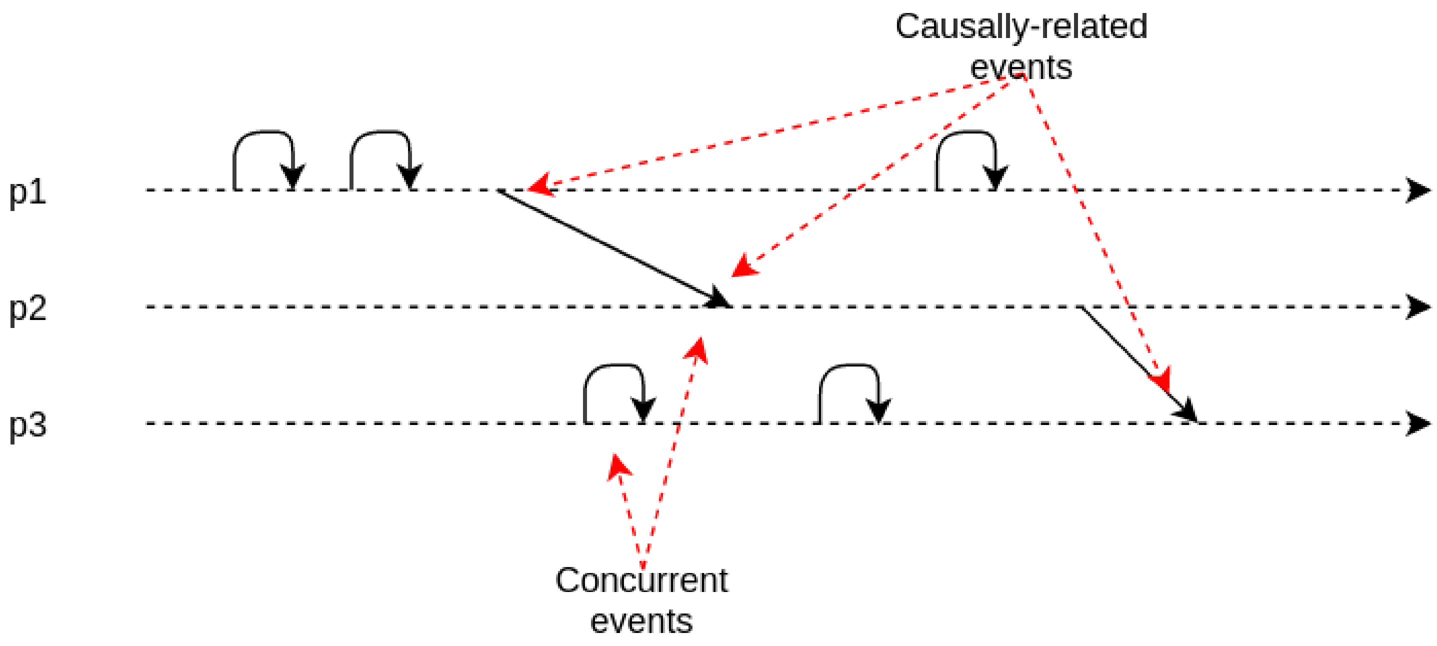
# Asynchronous systems

- No timing assumptions on processes and links.
    - Processes do not have access to any sort of physical clock.
    - Processing time may vary arbitrarily.
    - No bound on transmission time.
- But causality between events can still be determined.
    - How?

## Causal order

The happened-before relation $e_1 \to e_2$ denotes that $e_1$ may have caused $e_2$. It is true in the following cases:

- FIFO order: $e_1$ and $e_2$ occurred at the same process $p$ and $e_1$ occurred $e_2$;
- Network order: $e_1$ corresponds to the transmission of $m$ at a process $p$ and $e_2$ corresponds to its reception at a process $q$;
- Transitivity: if $e_1 \to e'$ and $e' \to e_2$, then $e_1 \to e_2$.


Causally-related
events
p1
p2
p3
Concurrent
events

## Similarity of executions

- The view of $p$ in $E$, denoted $E|p$ is the subsequence of process steps in $E$ restricted to those of $p$
- Two executions $E$ and $F$ are similar w.r.t. to $p$ if $E|p = F|p$.
- Two executions $E$ and $F$ are similar if $E|p = F|p$ for all processes $p$.

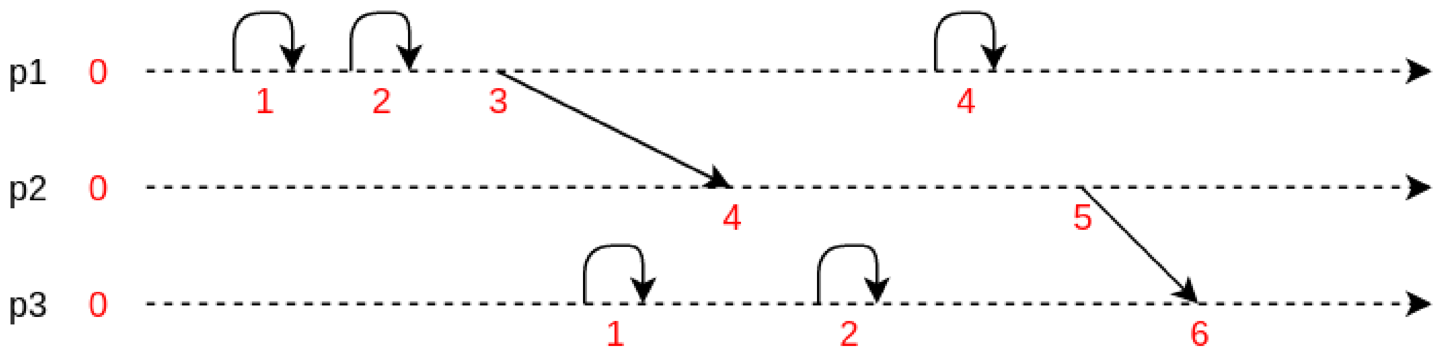
## Computation theorem

If two executions $E$ and $F$ have the same collection of events and their causal order is preserved, then $E$ and $F$ are similar executions.

## Logical clocks

In an asynchronous distributed system, the passage of time can be measured with logical clocks:

- Each process has a local logical clock $l_p$, initially set a $0$.
- Whenever an event occurs locally at $p$ or when a process sends a message, $p$ increments its logical clock.
    - $l_p := l_p + 1$
- When $p$ sends a message event $m$, it timestamps the message with its current logical time, $t(m) := l_p$.
- When $p$ receives a message event $m$ with timestamp $t(m)$, $p$ updates its logical clock.
    - $l_p := \max(l_p, t(m)) + 1$


p1
0
1
2
3
4
p2
0
4
5
p3
0
1
2
6

## Clock consistency condition

Logical clocks capture cause-effect relations:

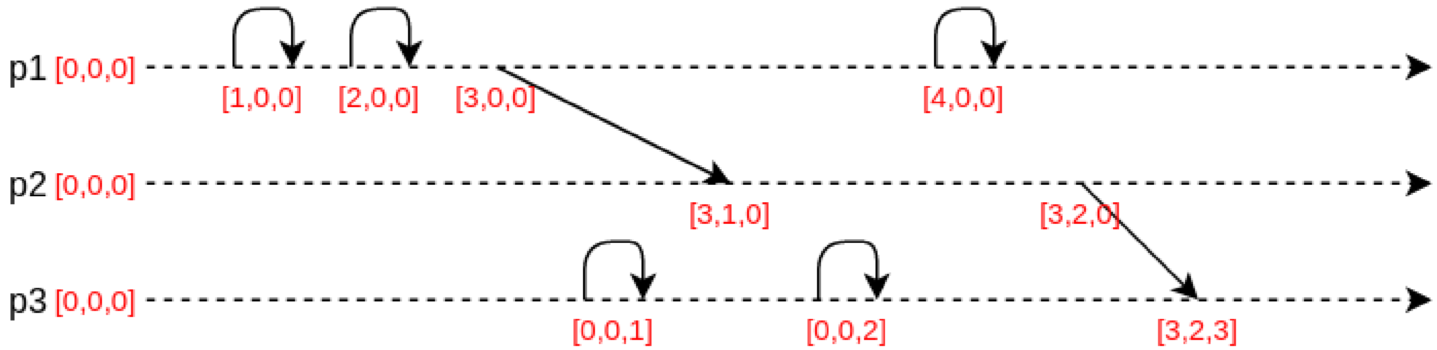
$$e_1 \rightarrow e_2 \Rightarrow t(e_1) < t(e_2)$$

- If $e_1$ is the cause of $e_2$, then $t(e_1) < t(e_2)$.
  - Can you prove it?
- But not necessarily the opposite:
  - $t(e_1) < t(e_2)$ does not imply $e_1 \rightarrow e_2$.
  - $e_1$ and $e_2$ may be logically concurrent.

## Vector clocks

Vector clocks fix this issue by making it possible to tell when two events cannot be causally related, i.e. when they are concurrent.

- Each process $p$ maintains a vector $V_p$ of $N$ clocks, initially set at $V_p[i] = 0 \, \forall i$.
- Whenever an event occurs locally at $p$ or when a process sends a message, $p$ increments the $p$-th element of its vector clock.
  - $V_p[p] := V_p[p] + 1$
- When $p$ sends a message event $m$, it piggybacks its vector clock as $V_m := V_p$.
- When $p$ receives a message event $m$ with the vector clock $V_m$, $p$ updates its vector clock.
  - $V_p[p] := V_p[p] + 1$
  - $V_p[i] := \max(V_p[i], V_m[i])$, for $i \neq p$.


p1 [0,0,0]
[1,0,0]
[2,0,0]
[3,0,0]
[4,0,0]
p2 [0,0,0]
[3,1,0]
[3,2,0]
p3 [0,0,0]
[0,0,1]
[0,0,2]
[3,2,3]

## Comparing vector clocks

- $V_p = V_q$
  - iff $\forall i \; V_p[i] = V_q[i]$.
- $V_p \leq V_q$
  - iff $\forall i \; V_p[i] \leq V_q[i]$.
- $V_p < V_q$
  - iff $V_p \leq V_q$ AND $\exists j \; V_p[j] < V_q[j]$
- $V_p$ and $V_q$ are logically concurrent.
  - iff NOT $V_p \leq V_q$ AND NOT $V_q \leq V_p$

# Synchronous systems

Assumption of three properties:

- Synchronous computation
  - Known upper bound on the process computation delay.
- Synchronous communication
  - Known upper bound on message transmission delay.
- Synchronous physical clocks
  - Processes have access to a local physical clock;
  - Known upper bound on clock drift and clock skew.

**Exercice**

Why studying synchronous systems? What services can be provided?

# Partially synchronous systems

A partially synchronous system is a system that is synchronous most of the time.

- There are periods where the timing assumptions of a synchronous system do not hold.
- But the distributed algorithm will have a long enough time window where everything behaves nicely, so that it can achieve its goal.

**Exercice**

Are there such systems?

# Failure detection

- It is tedious to model (partial) synchrony.
- Timing assumptions are mostly needed to detect failures.
  - Heartbeats, timeouts, etc.
- We define failure detector abstractions to encapsulate timing assumptions:
  - Black box giving suspicions regarding node failures;
  - Accuracy of suspicions depends on model strength.

## Implementation of failure detectors

A typical implementation is the following:

- Periodically exchange hearbeat messages;
- Timeout based on worst case message round trip;
- If timeout, then suspect node;
- If reception of a message from a suspected node, revise suspicion and increase timeout.

# Perfect detector ($\mathcal{P}$)

Assuming a crash-stop process abstraction, the perfect detector encapsulates the timing assumptions of a synchronous system.

**Module:**

**Name:** PerfectFailureDetector, **instance** $\mathcal{P}$.

**Events:**

**Indication:** $\langle\, \mathcal{P}, \mathit{Crash} \mid p \,\rangle$: Detects that process $p$ has crashed.

**Properties:**

**PFD1:** *Strong completeness:* Eventually, every process that crashes is permanently detected by every correct process.

**PFD2:** *Strong accuracy:* If a process $p$ is detected by any process, then $p$ has crashed.

**Exercice**
Which property is safety/liveness/neither?

**Implements:**

PerfectFailureDetector, **instance** $\mathcal{P}$.

**Uses:**

PerfectPointToPointLinks, **instance** *pl*.

**upon event** $\langle\, \mathcal{P}, \mathit{Init} \,\rangle$ **do**
    *alive* $:= \Pi$;
    *detected* $:= \emptyset$;
    *starttimer*$(\Delta)$;

**upon event** $\langle\, \mathit{Timeout} \,\rangle$ **do**
    **forall** $p \in \Pi$ **do**
        **if** $(p \notin \mathit{alive}) \wedge (p \notin \mathit{detected})$ **then**
            $\mathit{detected} := \mathit{detected} \cup \{p\}$;
            **trigger** $\langle\, \mathcal{P}, \mathit{Crash} \mid p \,\rangle$;
        **trigger** $\langle\, pl, \mathit{Send} \mid p$, [HEARTBEATREQUEST] $\rangle$;
    *alive* $:= \emptyset$;
    *starttimer*$(\Delta)$;

**upon event** $\langle\, pl, \mathit{Deliver} \mid q$, [HEARTBEATREQUEST] $\rangle$ **do**
    **trigger** $\langle\, pl, \mathit{Send} \mid q$, [HEARTBEATREPLY] $\rangle$;

**upon event** $\langle\, pl, \mathit{Deliver} \mid p$, [HEARTBEATREPLY] $\rangle$ **do**
    $\mathit{alive} := \mathit{alive} \cup \{p\}$;

## Correctness

We assume a synchronous system:

- The transmission delay is bounded by some known constant.
- Local processing is negligible.
- The timeout delay $\Delta$ is chosen to be large enough such that
  - every process has enough time to send a heartbeat message to all,
  - every heartbeat message has enough time to be delivered,
  - the correct destination processes have enough time to process the heartbeat and to send a reply,
  - the replies have enough time to reach the original sender and to be processed.

- PFD1. Strong completeness
    - A crashed process $p$ stops replying to heartbeat messages, and no process will deliver its messages. Every correct process will thus eventually detect the crash of $p$.
- PFD2. Strong accuracy
    - The crash of $p$ is detected by some other process $q$ only if $q$ does not deliver a message from $p$ before the timeout period.
    - This happens only if $p$ has indeed crashed, because the algorithm makes sure $p$ must have sent a message otherwise and the synchrony assumptions imply that the message should have been delivered before the timeout period.

# Eventually perfect detector ($\diamond\mathcal{P}$)

The eventually perfect detector encapsulates the timing assumptions of a partially synchronous system.

**Module:**

**Name:** EventuallyPerfectFailureDetector, **instance** $\diamond\mathcal{P}$.

**Events:**

**Indication:** $\langle\, \diamond\mathcal{P}, \mathit{Suspect} \mid p \,\rangle$: Notifies that process $p$ is suspected to have crashed.

**Indication:** $\langle\, \diamond\mathcal{P}, \mathit{Restore} \mid p \,\rangle$: Notifies that process $p$ is not suspected anymore.

**Properties:**

**EPFD1:** *Strong completeness:* Eventually, every process that crashes is permanently suspected by every correct process.

**EPFD2:** *Eventual strong accuracy:* Eventually, no correct process is suspected by any correct process.

**Implements:**
EventuallyPerfectFailureDetector, **instance** $\diamond\mathcal{P}$.

**Uses:**
PerfectPointToPointLinks, **instance** *pl*.

**upon event** $\langle\, \diamond\mathcal{P}, \mathit{Init} \,\rangle$ **do**
$\mathit{alive} := \Pi$;
$\mathit{suspected} := \emptyset$;
$\mathit{delay} := \Delta$;
$\mathit{starttimer}(\mathit{delay})$;

**upon event** $\langle\, \mathit{Timeout} \,\rangle$ **do**
**if** $\mathit{alive} \cap \mathit{suspected} \neq \emptyset$ **then**
$\mathit{delay} := \mathit{delay} + \Delta$;
**forall** $p \in \Pi$ **do**
**if** $(p \notin \mathit{alive}) \wedge (p \notin \mathit{suspected})$ **then**
$\mathit{suspected} := \mathit{suspected} \cup \{p\}$;
**trigger** $\langle\, \diamond\mathcal{P}, \mathit{Suspect} \mid p \,\rangle$;
**else if** $(p \in \mathit{alive}) \wedge (p \in \mathit{suspected})$ **then**
$\mathit{suspected} := \mathit{suspected} \setminus \{p\}$;
**trigger** $\langle\, \diamond\mathcal{P}, \mathit{Restore} \mid p \,\rangle$;
**trigger** $\langle\, \mathit{pl}, \mathit{Send} \mid p, [\text{HEARTBEATREQUEST}] \,\rangle$;
$\mathit{alive} := \emptyset$;
$\mathit{starttimer}(\mathit{delay})$;

**upon event** $\langle\, \mathit{pl}, \mathit{Deliver} \mid q, [\text{HEARTBEATREQUEST}] \,\rangle$ **do**
**trigger** $\langle\, \mathit{pl}, \mathit{Send} \mid q, [\text{HEARTBEATREPLY}] \,\rangle$;

**upon event** $\langle\, \mathit{pl}, \mathit{Deliver} \mid p, [\text{HEARTBEATREPLY}] \,\rangle$ **do**
$\mathit{alive} := \mathit{alive} \cup \{p\}$;

**Exercice**

Show that this implementation is correct.

# Leader election ($le$)

- Failure detection captures failure behavior.
  - Detects failed processes.
- Leader election is an abstraction that also captures failure behavior.
  - Detects correct nodes.
  - But a single and same for all, called the leader.
- If the current leader crashes, a new leader should be elected.

**Module:**

**Name:** LeaderElection, **instance** *le*.

**Events:**

**Indication:** $\langle\ le, Leader \mid p\ \rangle$: Indicates that process $p$ is elected as leader.

**Properties:**

**LE1:** *Eventual detection:* Either there is no correct process, or some correct process is eventually elected as the leader.

**LE2:** *Accuracy:* If a process is leader, then all previously elected leaders have crashed.

**Implements:**

LeaderElection, **instance** *le*.

**Uses:**

PerfectFailureDetector, **instance** $\mathcal{P}$.

**upon event** $\langle$ *le*, *Init* $\rangle$ **do**

*suspected* $:= \emptyset$;

*leader* $:= \perp$;

**upon event** $\langle$ $\mathcal{P}$, *Crash* $\mid p$ $\rangle$ **do**

*suspected* $:=$ *suspected* $\cup \{p\}$;

**upon** *leader* $\neq$ maxrank($\Pi \setminus$ *suspected*) **do**

*leader* $:=$ maxrank($\Pi \setminus$ *suspected*);

**trigger** $\langle$ *le*, Leader $\mid$ *leader* $\rangle$;

**Exercice**

- Show that this implementation is correct.
- Is $le$ a failure detector?

# Distributed system models

# Distributed system models

We define a distributed system model as the combination of (i) a process abstraction, (ii) a link abstraction, and (iii) a failure detector abstraction.

- Fail-stop (synchronous)
    - Crash-stop process abstraction
    - Perfect links
    - Perfect failure detector
- Fail-silent (asynchronous)
    - Crash-stop process abstraction
    - Perfect links

- Fail-noisy (partially synchronous)
  - Crash-stop process abstraction
  - Perfect links
  - Eventually perfect failure detector
- Fail-recovery
  - Crash-stop process abstraction
  - Stubborn links

The fail-stop distributed system model substantially simplifies the design of distributed algorithms.

The end.

# References


- Alpern, Bowen, and Fred B. Schneider. "Recognizing safety and liveness." Distributed computing 2.3 (1987): 117-126.
- Lamport, Leslie. "Time, clocks, and the ordering of events in a distributed system." Communications of the ACM 21.7 (1978): 558-565.
- Fidge, Colin J. "Timestamps in message-passing systems that preserve the partial ordering." (1987): 56-66.