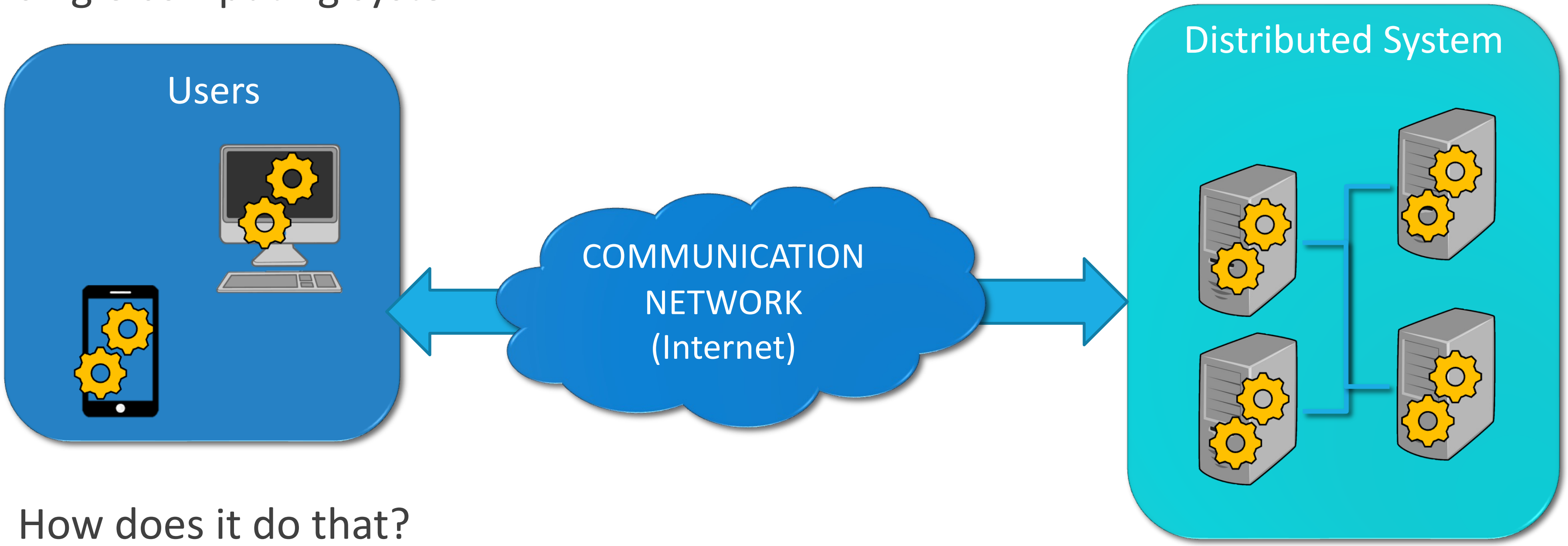
Dependable Distributed Systems Master of Science in Engineering in Computer Science

AA 2023/2024

LECTURE 2A: MODELLING DISTRIBUTED SYSTEMS

Recap

A distributed system is a set of entities/computes/machines communicating, coordinating and sharing resources to reach a common goal, appearing as a single computing system



How does it do that?

- Running distributed algorithm (i.e., a piece of software) that takes care of the the mentioned issues.

System deployment

The actual realization of a distributed system requires that we instantiate and place software components on real machines

There are many different choices that can be made in doing so

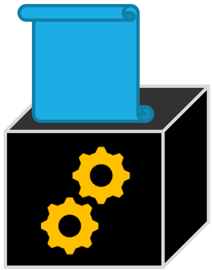
The final instantiation of a software architecture is also referred to as a system architecture

Architectural Style

More about this in Software Engineering

A style is formulated in terms of

- components
  - i.e, a modular unit with well-defined required and provided interfaces that is replaceable within its environment
- the way in which components are connected to each other
- the data exchanged between components
- how these elements are jointly configured into a system



Components can be **abstracted** by their specification and their interfaces

We will focus in this course on DISTRIBUTED abstractions and their algorithmic aspects

Why abstractions are so important?

1. capture properties that are common to a large and significant range of systems

2. help distinguish the fundamental from the accessory

3. prevent system designers and engineers from reinventing, over and over, the same solutions for slight variants of the very same problems.

The road to build a distributed abstraction

Step 1: definition of the system model

- A system model must:
  - describe the relevant elements in an abstract way
  - identify their intrinsic properties
  - characterize their interactions

Step 2: build a distributed abstraction

- understand how to design a protocol that capture recurring interaction patterns in distributed applications

Step 3: implement and deploy the distributed algorithm (potentially as part of your middleware)

- This last step is system dependent
  - you should choose the language, the architectural pattern, etc…

More about this in Software Engineering and Lab

Composition Model

The protocols we will consider in this course are presented in pseudo-code

The pseudo code reflects a reactive computing model where

- components of the same process communicate by exchanging events
- the algorithm is described as a set of event handlers
- handlers react to incoming events and possibly trigger new events.

Composition Model and its code

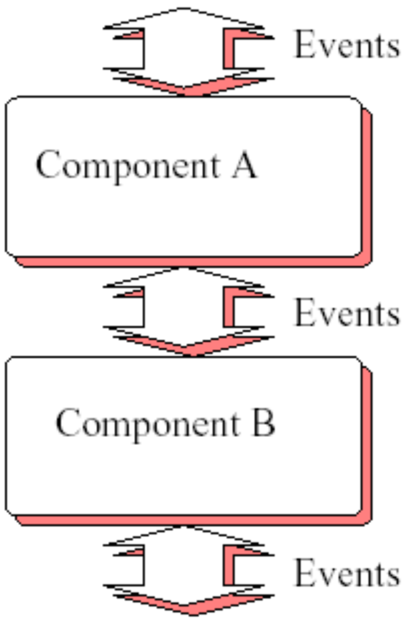


Figure 1.1. Composition model

upon event $\langle\, co_1, \mathit{Event}_1 \mid att_1^1, att_1^2, \ldots \,\rangle$ **do**
    do something;
    trigger $\langle\, co_2, \mathit{Event}_2 \mid att_2^1, att_2^2, \ldots \,\rangle$;   *// send some event*

upon event $\langle\, co_1, \mathit{Event}_3 \mid att_3^1, att_3^2, \ldots \,\rangle$ **do**
    do something else;
    trigger $\langle\, co_2, \mathit{Event}_4 \mid att_4^1, att_4^2, \ldots \,\rangle$;   *// send some other event*

Composition Model and its code

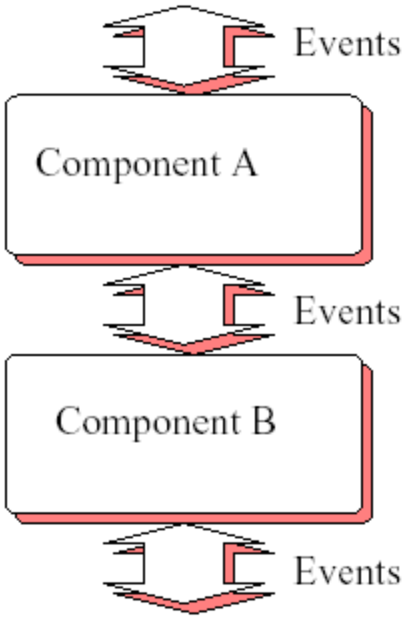
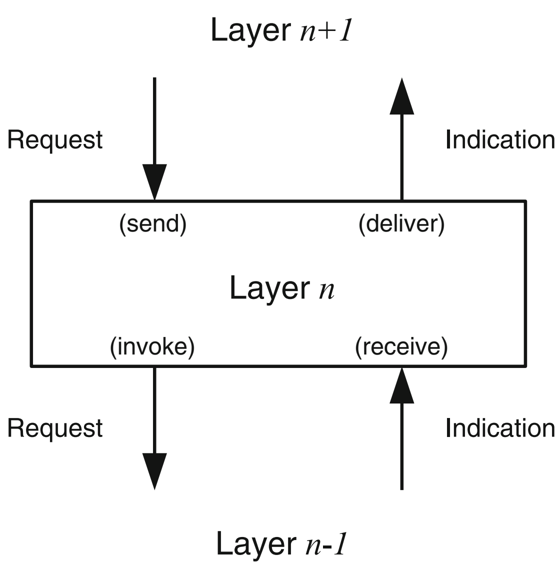


Figure 1.1. Composition model

upon *condition* **do** // an internal event
do something;

upon event $\langle$ *co*, *Event* $|$ $att_1^1, att_1^2, \ldots \rangle$ **such that** *condition* **do**
do something;

Programming Interface



Request events

- used by a component c1 to request a service to component c2
- e.g., c1 may ask to c2 to disseminate a message in a group

Confirmation events

- used by a component to confirm the completion of a request

Indication events

- used by a component c1 to deliver information to a component c2
- e.g., the notification of a message delivery

Example - Job handler

Module 1.1: Interface and properties of a job handler

Module:

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

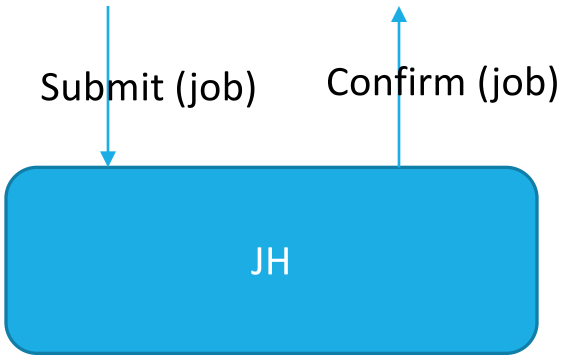
Events:

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

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

Properties:

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

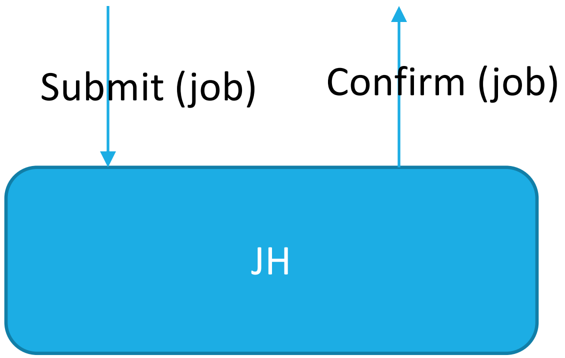


Example – Job handler (synchronous implementation)

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* $\rangle$;



Example – Job handler (asynchronous implementation)

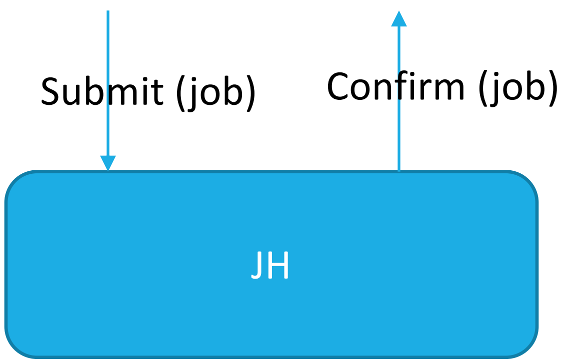
Algorithm 1.2: Asynchronous Job Handler

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);
$buffer := buffer \setminus \{job\}$;



Example - Layering

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.

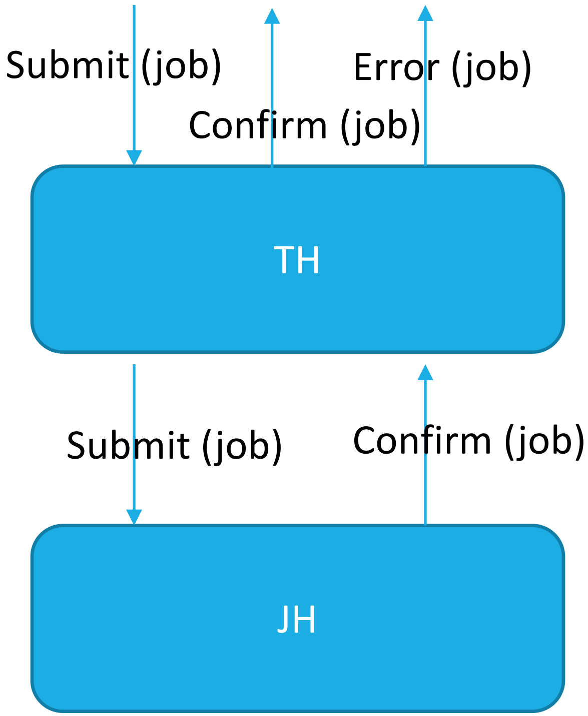
Indication: $\langle$ *th*, *Confirm* | *job* $\rangle$: Confirms that the given job has been (or will be) transformed and processed.

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.



Example - Layering

Algorithm 1.3: Job-Transformation by Buffering

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

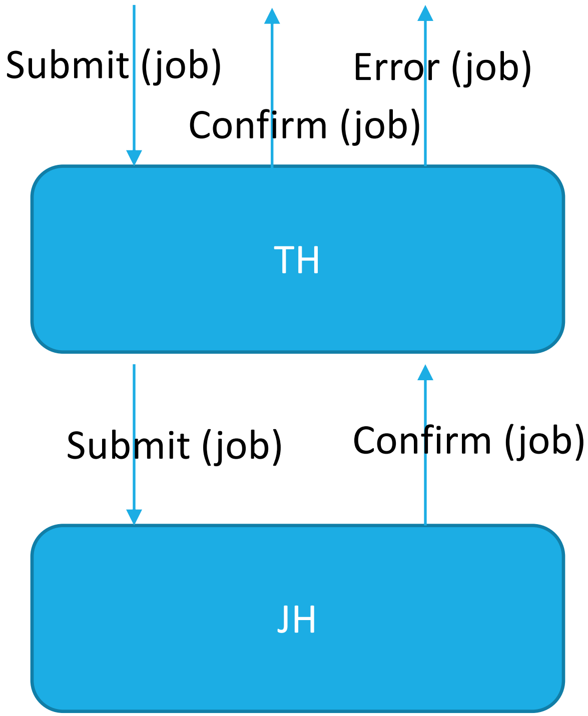
Uses:
JobHandler, **instance** *jh*.

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

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



Modelling Distributed Computations

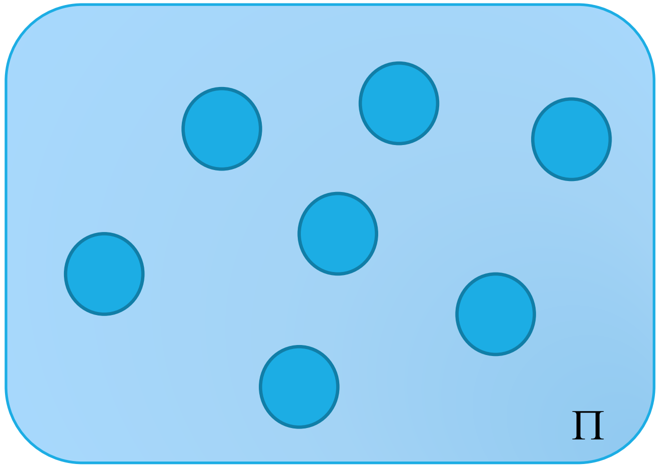
- Modelling Processes and their interactions
- Specification in terms of Safety and Liveness Property
- Modelling Failures
- Timing Assumptions

Processes

- Π denotes the set of processes

- Unless stated otherwise, this set is static and does not change, and every process knows the identities of all processes.
  - Sometimes, a function rank : $\Pi \rightarrow \{1, \ldots, N\}$ is used to associate every process with a unique index between 1 and N

- In the description of an algorithm, the special process name *self* denotes the name of the process that executes the code



Processes' interactions and Messages

- Processes in Π communicate by exchanging messages
- Messages are uniquely identified
  - e.g., using a sequence number or a local clock, together with the process identifier.

- All messages that are ever exchanged by some distributed algorithm are unique.

- Messages are exchanged by the processes through communication *links*.

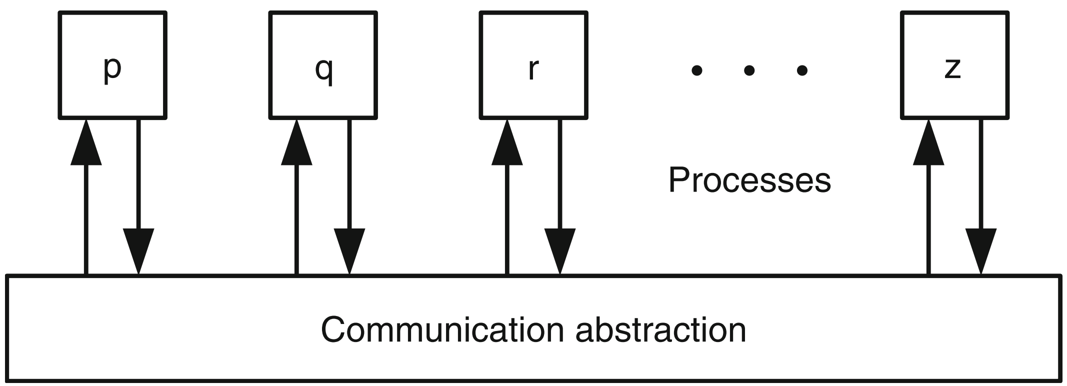
Distributed Algorithms

A *distributed algorithm* consists of a distributed collection of automata, one per process.

The automaton at a process regulates the way the process executes its computation steps, i.e., how it reacts to a message.

Every process is implemented by the same automaton

The *execution* of a distributed algorithm is represented by a sequence of steps executed by the processes.



Safety and Liveness

Safety properties state that the algorithm should not do anything wrong

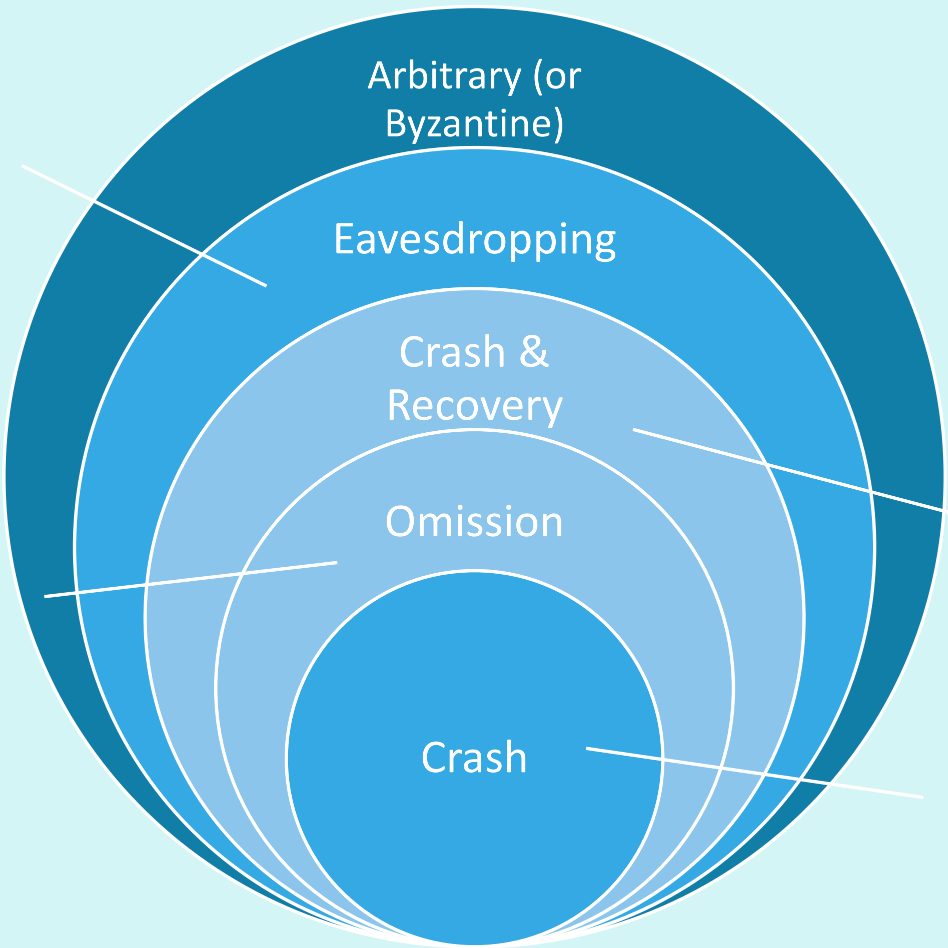
- a *safety property* is a property of a distributed algorithm that can be violated at some time t and never be satisfied again after that time
- a safety property is a property such that, whenever it is violated in some execution E of an algorithm, there is a partial execution E′ of E such that the property will be violated in any extension of E′

Liveness properties ensure that eventually something good happens

- a liveness property is a property of a distributed system execution such that, for any time t, there is some hope that the property can be satisfied at some time t′ ≥ t.

Failure Models

a process leaks information obtained in an algorithm to an outside entity



a process can crash and stop to send messages, but might recover later

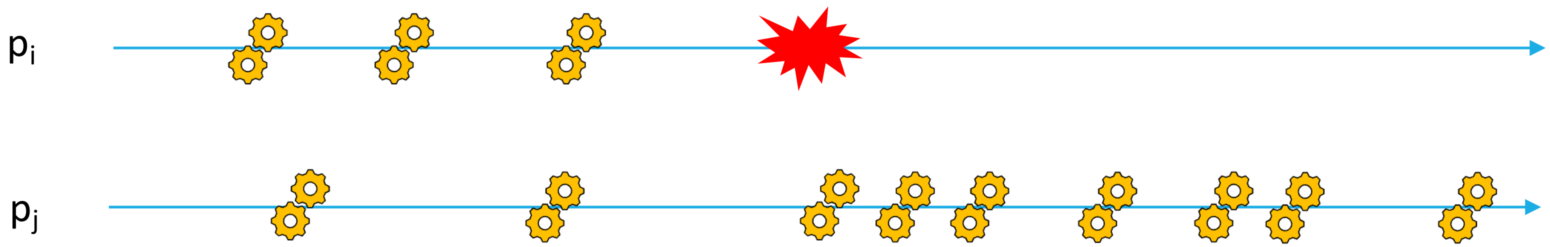
a process does not send (or receive) a message that it is supposed to send (or receive) according to its algorithm

the process stops its execution

Crash Fault

The crash-stop process abstraction model a process that *crashes* at time *t* and never recovers after that time

A process is said to be
- *faulty* if it crashes at some time during the execution
- *correct* if it never crashes and executes an infinite number of steps



Dependability and crash fault

RECALL: fault-tolerance is one of the main technique used to achieve dependability

To achieve fault-tolerance we need to design a distributed algorithm that is working despite the presence of faults

This is done by assuming that only a limited number f of processes are faulty

The relation between the number f of potentially faulty processes and the total number N of processes in the system is generally called *resilience*.

⚠ Assuming an upper bound on the number of faulty processes f means that any number of processes *up to f* may fail

Crash-stop vs crash-recovery process abstraction

OBSERVATION: processes that crash can be restarted and hence may recover

With the crash-stop abstraction, a recovered process is no longer part of the system

However, the crash-stop process abstraction
- does not preclude the possibility of recovery
- does not imply that recovery should be prevented for a given algorithm to behave correctly

It simply means that the algorithm should not rely on some of the processes to recover in order to pursue its execution

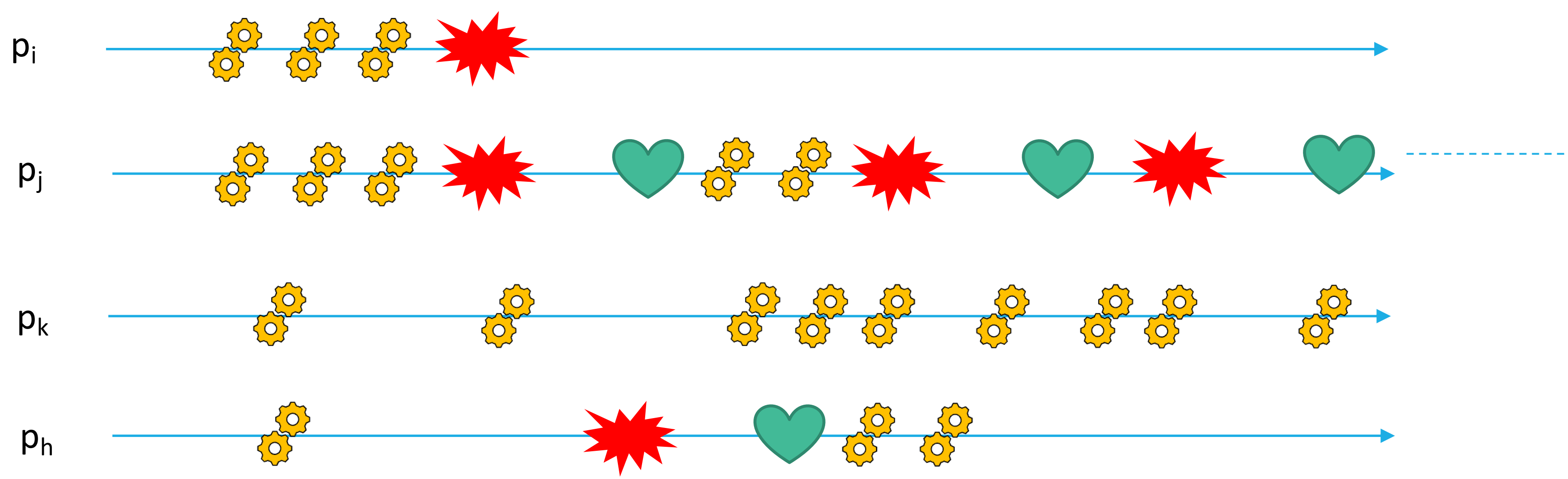
Crash-recovery fault

A process is *faulty* if

- the process crashes and never recovers or
- the process keeps infinitely often crashing and recovering

A process that is not faulty is said to be correct

- Note that a process that crashes and recovers a finite number of time is considered correct in this model



Crash-recovery fault

A characteristic of this model is that a process might suffer *amnesia*
- i.e., it crashes and lose its internal state

This significantly complicates the design of algorithms
- upon recovery, the process might send new messages that contradict messages that it might have sent prior to the crash

To cope with this issue, we may assume that every process has a stable storage (also called a log) which can be accessed through `store()` and `retrieve()` operations

Upon recovery, we assume that a process is aware that it has crashed and recovered

Timing Assumptions

- Synchronous
- Partially Synchronous
- Asynchronous

Synchronous System

Characterized by three properties

1. Synchronous processing
   - Known Upper Bound on the time taken by a process to execute a basic step
2. Synchronous Communication
   - Known Upper Bound on the time taken by a message to reach a destination
3. Synchronous physical clocks
   - Known Upper Bound on drift of a local clock wrt real time

Services provided in Synchronous systems

- Timed failure detection
- Measure of transit delay
- Coordination based on time
- Worst case performance (e.g., response time of a service in case of failures)
- Synchronized clocks

A Major problem is the coverage of the synchrony assumption!!!!

This turns out in the difficulty of building a system where timing assumptions hold with high probability

Asynchronous Systems

Assuming an asynchronous distributed system means to not make any timing assumption about processes and links

Even without access to physical clocks, it is still possible to measure the passage of time based on the transmission and delivery of messages

- time is defined with respect to communication

Time measured in this way is called *logical time*, and the resulting notion of a clock is called a *logical clock*.

Partial (eventual) synchrony

Generally distributed systems are synchronous most of the time and then they experience bounded asynchrony periods

One way to capture partial synchrony is "eventual synchrony"

- i.e., there is an unknown time t after which the system becomes synchronous

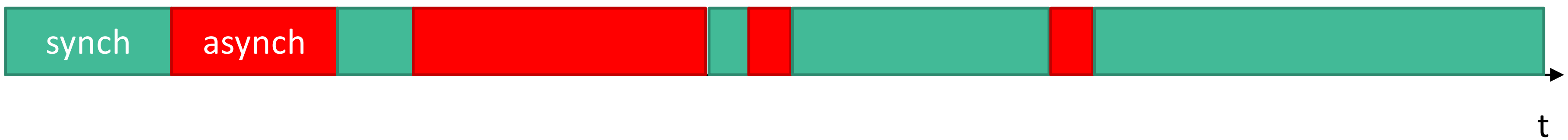
This assumption captures the fact that the system does not behave always as synchronous

WARNING: Assuming Partial synchrony does not mean that

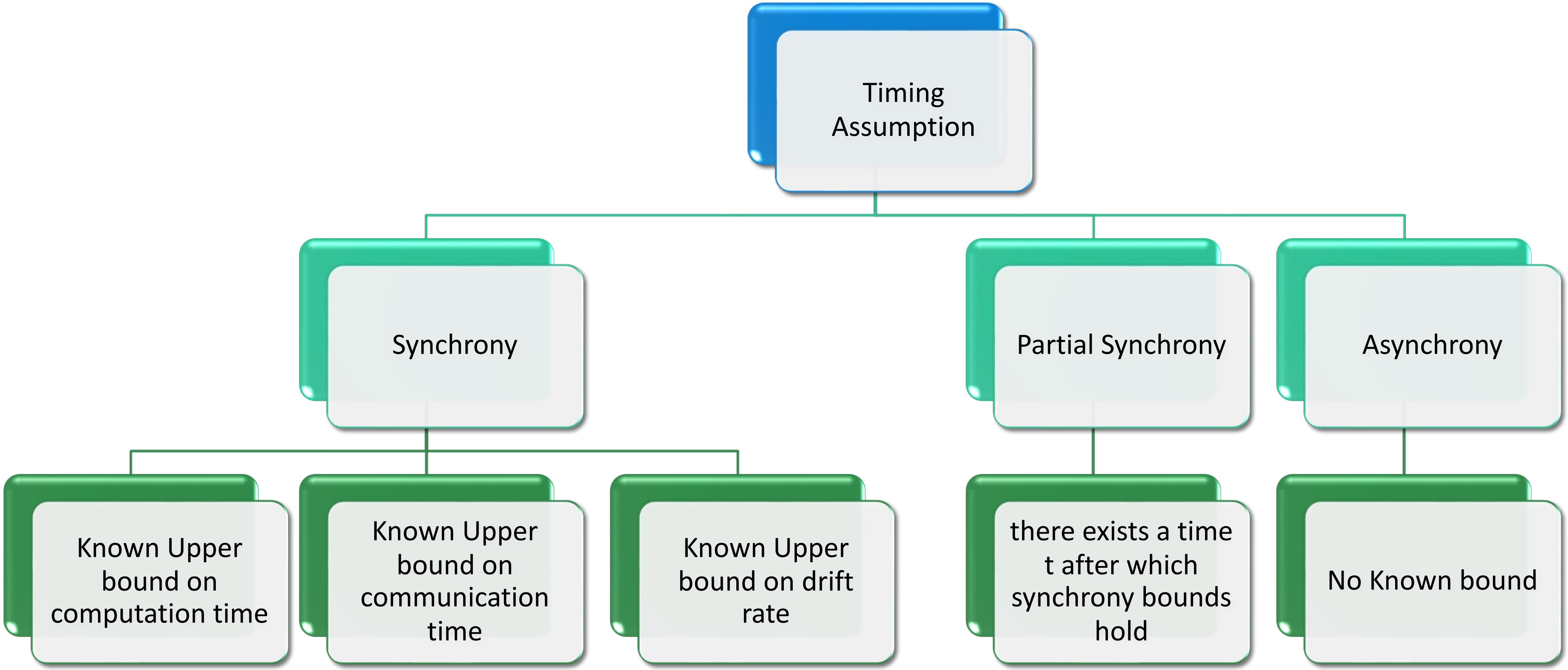
- After t all the system (including hardware, software and network components) becomes synchronous forever
- The system starts asynchronous and then after some (may be long) time it becomes synchronous

What do we expect from partial synchrony

There is a period of synchrony long enough to terminate the distributed algorithm



Summary on Timing Assumptions



References

C. Cachin, R. Guerraoui and L. Rodrigues. Introduction to Reliable and Secure Distributed Programming, Springer, 2011

- Chapter 2, Sections 1, 2, 5

Dependable Distributed Systems
Master of Science in Engineering in Computer Science

AA 2022/2023

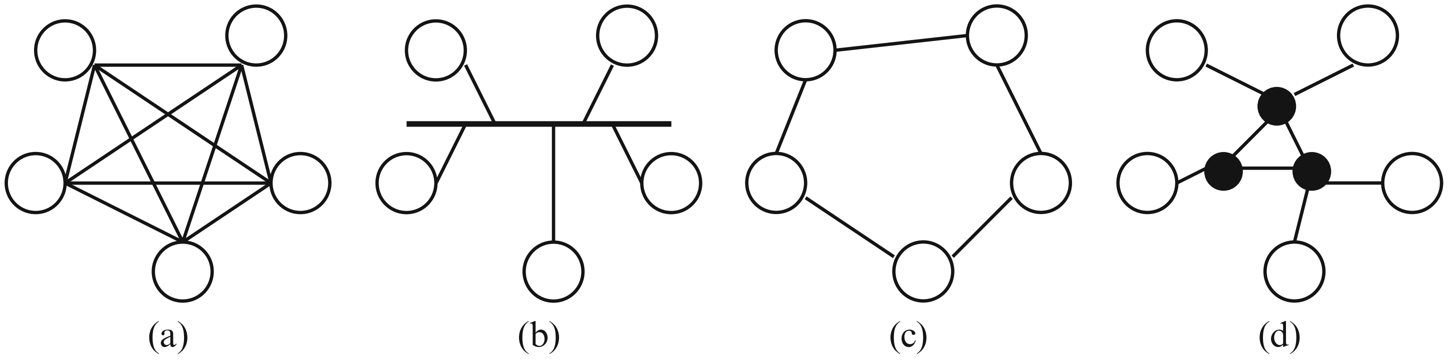
LECTURE 2B: ABSTRACTING COMMUNICATIONS

Link Abstraction

The abstraction of a link is used to represent the network components of the distributed system

Every pair of processes is connected by a bidirectional link

- it could be possible that processes are arranged in a complex topology, and you need to implement a routing algorithm to realize such abstraction



Link Abstractions under crash failure assumption

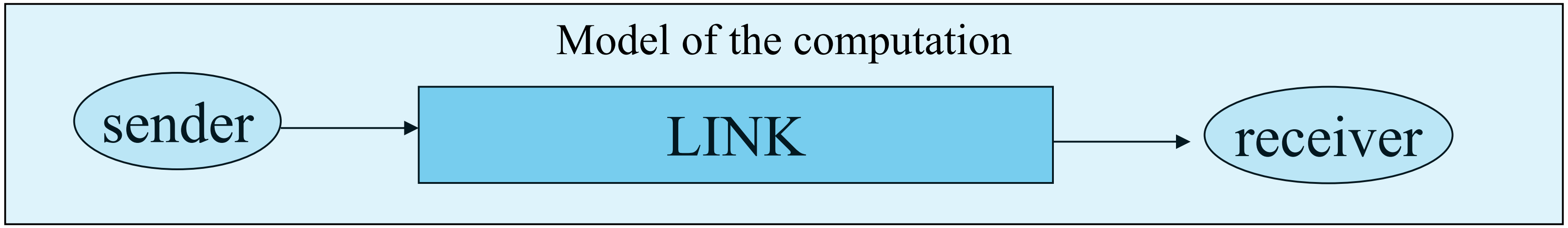
1. *fair-loss links* (captures the basic idea that messages might be lost but the probability for a message not to be lost is nonzero).

2. *Stubborn links*

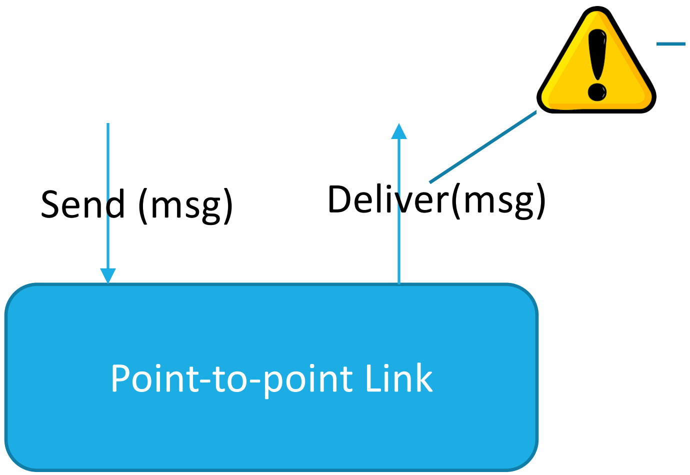
3. *Perfect links*

System Model

- Two processes (sender and receiver)
- Messages
  - can be lost
  - experience an unpredictable time to reach the destination
- Processes
  - can crash
  - The time taken by each process to execute an operation is bounded (such a bound can be unknown)



A generic link interface



Pay Attention!
Deliver is different from receive

A message is typically received at a given port of the network and stored within some buffer, and then some algorithm is executed to make sure the properties of the required link abstraction are satisfied, before the message is actually delivered.

Fair-loss Point-to-Point Link: Specification

Module 2.1: Interface and properties of fair-loss point-to-point links

Module:

Name: FairLossPointToPointLinks, **instance** *fll*.

Events:

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

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

Properties:

FLL1: *Fair-loss:* If a correct process p infinitely often sends a message m to a correct process q, then q delivers m an infinite number of times.

FLL2: *Finite duplication:* If a correct process p sends a message m a finite number of times to process q, then m cannot be delivered an infinite number of times by q.

FLL3: *No creation:* If some process q delivers a message m with sender p, then m was previously sent to q by process p.

Fair-loss Point-to-Point Link: Issues

The sender must take care of the retransmissions if it wants to be sure that a message m is delivered at its destination.

Stubborn Link

The specification does not guarantee that the sender can stop the retransmission of each message

Quiescient Implementation

Each message may be delivered more than once.

Perfect Link

Stubborn Point-to-Point Link: Specification

Module 2.2: Interface and properties of stubborn point-to-point links

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.

Stubborn Point-to-Point Link: Implementation

Algorithm 2.1: Retransmit Forever

Implements:
  StubbornPointToPointLinks, **instance** *sl*.

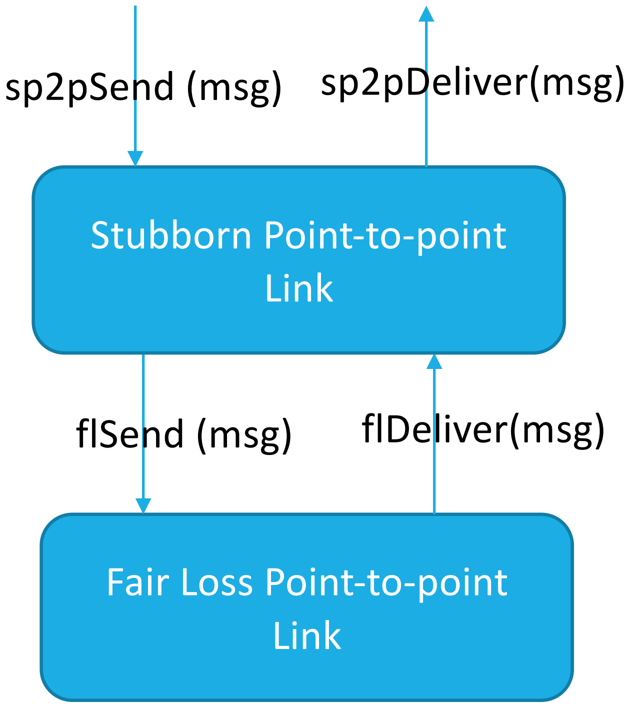
Uses:
  FairLossPointToPointLinks, **instance** *fll*.

upon event $\langle$ *sl*, *Init* $\rangle$ **do**
  sent := $\emptyset$;
  starttimer(Δ);

upon event $\langle$ *Timeout* $\rangle$ **do**
  forall $(q, m) \in$ *sent* **do**
    trigger $\langle$ *fll*, *Send* | q, m $\rangle$;
  starttimer(Δ);

upon event $\langle$ *sl*, *Send* | q, m $\rangle$ **do**
  trigger $\langle$ *fll*, *Send* | q, m $\rangle$;
  sent := *sent* $\cup \{(q, m)\}$;

upon event $\langle$ *fll*, *Deliver* | p, m $\rangle$ **do**
  trigger $\langle$ *sl*, *Deliver* | p, m $\rangle$;



Perfect Point-to-Point Link: Specification

Module 2.3: Interface and properties of perfect point-to-point links

Module:

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

Events:

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

Indication: $\langle$ *pl*, *Deliver* $|$ p, m $\rangle$: 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.

Perfect Point-to-Point Link: Implementation

Algorithm 2.2: Eliminate Duplicates

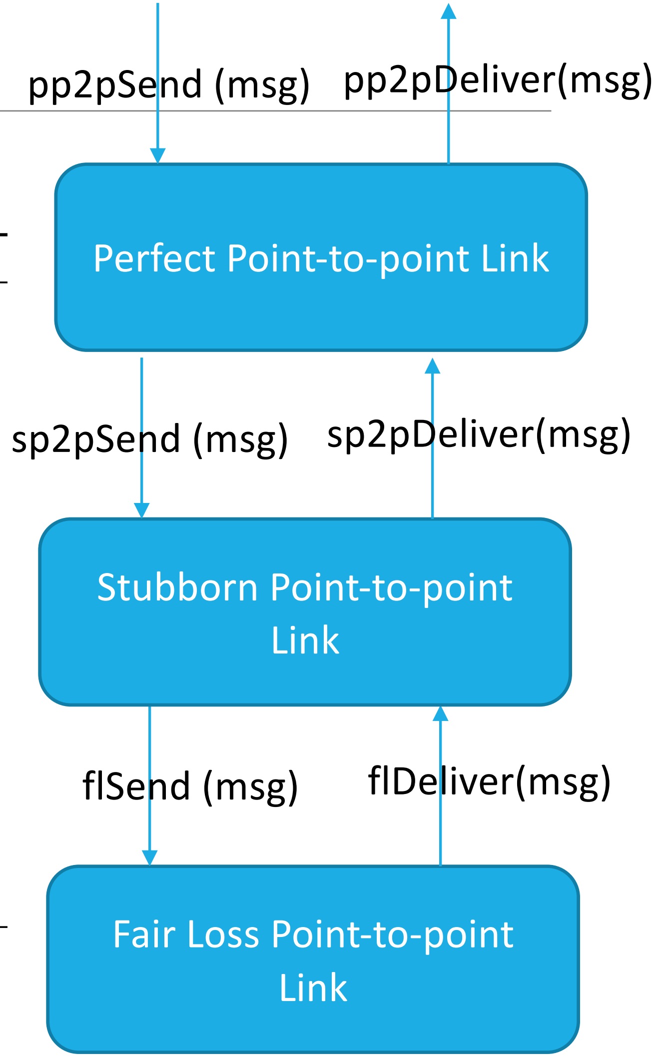
Implements:
PerfectPointToPointLinks, **instance** *pl*.

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

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

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

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



References

C. Cachin, R. Guerraoui and L. Rodrigues. Introduction to Reliable and Secure Distributed Programming, Springer, 2011

- Chapter 2, Sections 4 up to 2.4.4