Distributed Systems
Master of Science in Engineering in Computer Science

AA 2018/2019

LECTURE 1: 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

Why distributed 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.

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

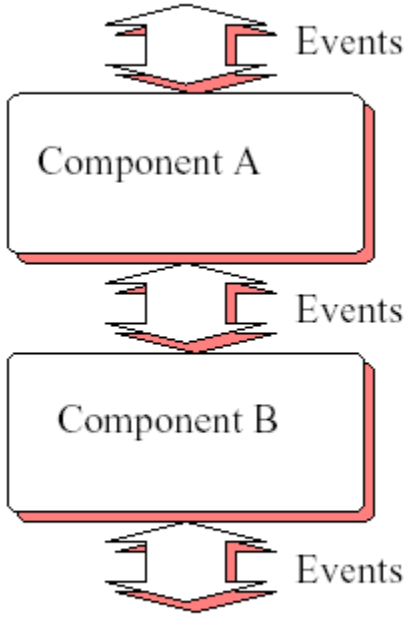


Figure 1.1. Composition model

---

upon event $\langle\, co_1, Event_1 \mid att_1^1, att_1^2, \ldots \,\rangle$ **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*

Composition Model and its code

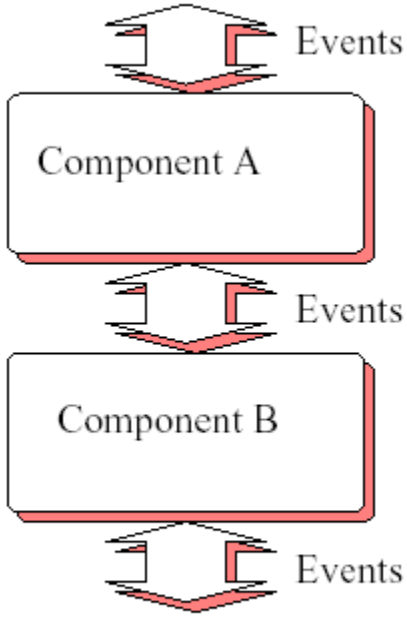
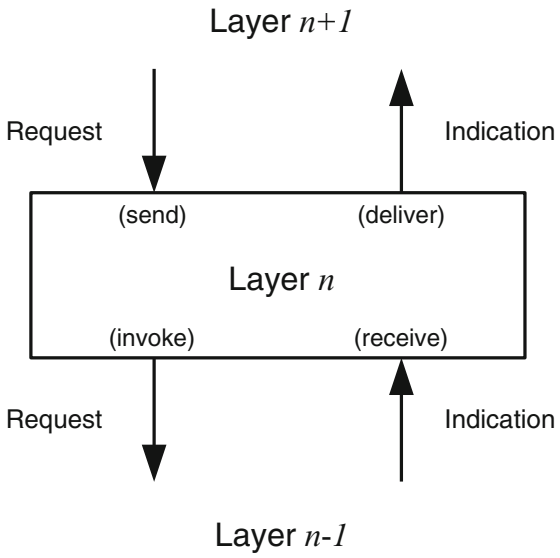


Figure 1.1. Composition model

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 are used by a component to request a service from another component: for instance, the application layer might trigger a *request* event at a component in charge of broadcasting a message to a set of processes in a group with some reliability guarantee, or proposing a value to be decided on by the group.
- *Confirmation* events are used by a component to confirm the completion of a request. Typically, the component in charge of implementing a broadcast will confirm to the application layer that the message was indeed broadcast or that the value suggested has indeed been proposed to the group: the component uses here a *confirmation* event.

- *Indication* events are used by a given component to *deliver* information to another component. Considering the broadcast example above, at every process that is a destination of the message, the component in charge of implementing the actual broadcast primitive will typically perform some processing to ensure the corresponding reliability guarantee, and then use an *indication* event to deliver the message to the application layer. Similarly, the decision on a value will be indicated with such an event.

Example - Job handler

Module 1.1: Interface and properties of a job handler

Module:

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

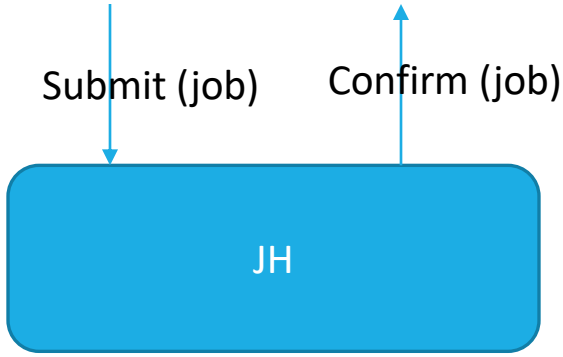
Events:

Request: ⟨ *jh*, *Submit* | *job* ⟩: Requests a job to be processed.

Indication: ⟨ *jh*, *Confirm* | *job* ⟩: Confirms that the given job has been (or will be) processed.

Properties:

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

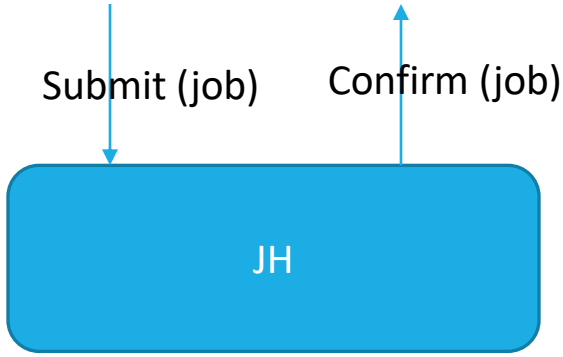


Example – Job handler (synchronous implementation

Algorithm 1.1: Synchronous Job Handler

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

upon event ⟨ *jh*, *Submit* | *job* ⟩ **do**
process(*job*);
trigger ⟨ *jh*, *Confirm* | *job* ⟩;



Example – Job handler (asynchronous implementation

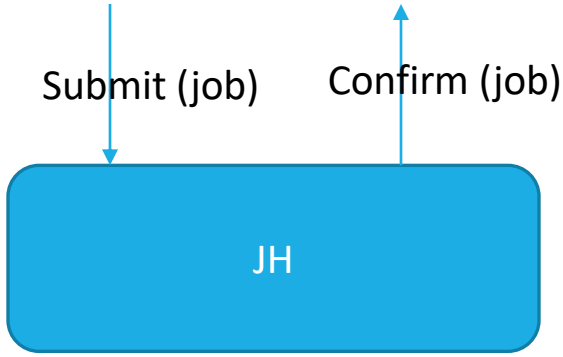
Algorithm 1.2: Asynchronous Job Handler

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

upon event ⟨ *jh*, *Init* ⟩ **do**
buffer := ∅;

upon event ⟨ *jh*, *Submit* | *job* ⟩ **do**
buffer := *buffer* ∪ {*job*};
trigger ⟨ *jh*, *Confirm* | *job* ⟩;

upon *buffer* ≠ ∅ **do**
job := *selectjob*(*buffer*);
process(*job*);
buffer := *buffer* \ {*job*};



Example - Layering

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

Module:

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

Events:

Request: ⟨ *th*, *Submit* | *job* ⟩: Submits a job for transformation and for processing.

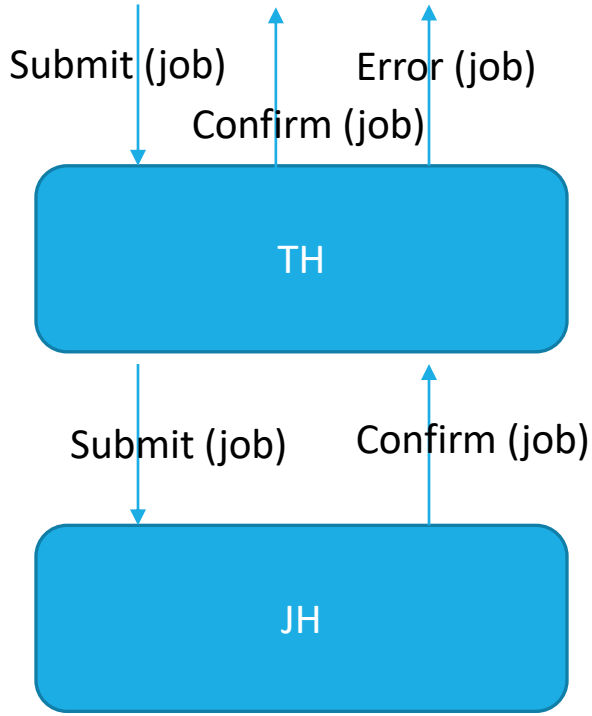
Indication:⟨ *th*, *Confirm* | *job* ⟩: Confirms that the given job has been (or will be) transformed and processed.

Indication:⟨ *th*, *Error* | *job* ⟩: 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*.

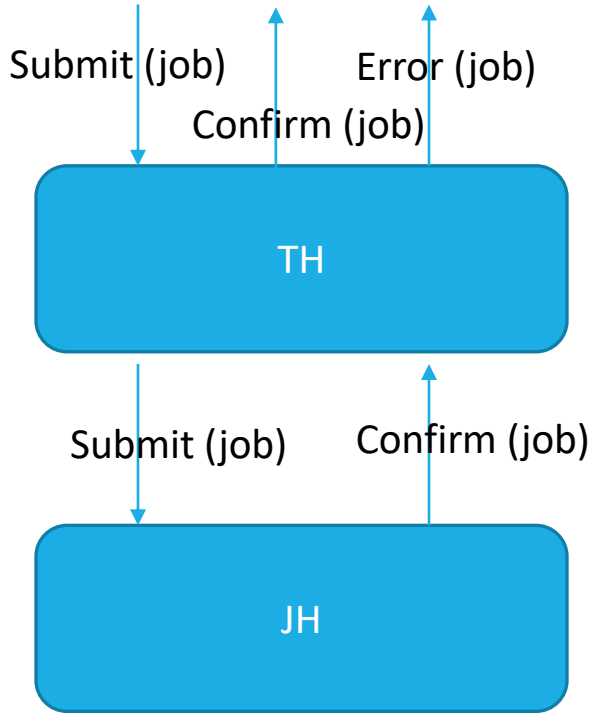
Uses:
JobHandler, **instance** *jh*.

upon event ⟨ *th*, *Init* ⟩ **do**
top := 1;
bottom := 1;
handling := FALSE;
buffer := $[\perp]^M$;

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

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

upon event ⟨ *jh*, *Confirm* | *job* ⟩ **do**
handling := FALSE;



Exercises

Module 1.1 Interface of a printing module

Module:

Name: Print.

Events:

Request: ⟨ *PrintRequest* | rqid, str ⟩: Requests a string to be printed. The token rqid is an identifier of the request.

Confirmation:⟨ *PrintConfirm* | rqid ⟩: Used to confirm that the printing request with identifier rqid succeeded.

Module 1.2 Interface of a bounded printing module

Module:

Name: BoundedPrint.

Events:

Request: ⟨ *BoundedPrintRequest* | rqid, str ⟩: Request a string to be printed. The token rqid is an identifier of the request.

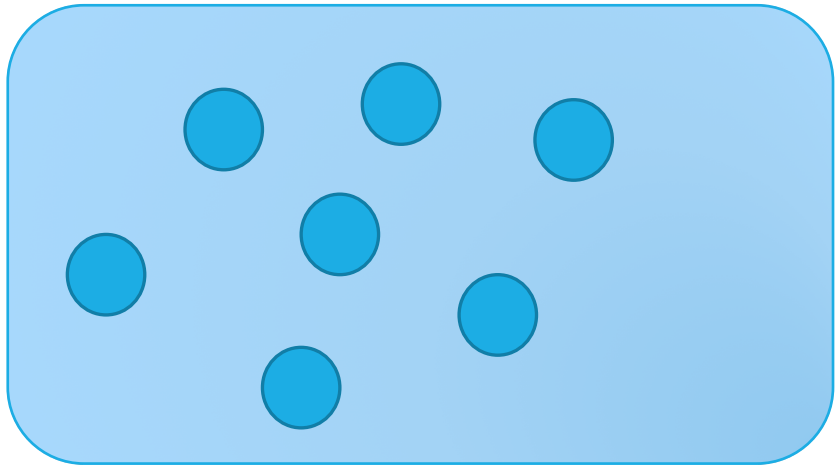
Confirmation:⟨ *PrintStatus* | rqid, status ⟩: Used to return the outcome of the printing request: Ok or Nok.

Indication:⟨ *PrintAlarm* ⟩: Used to indicate that the threshold was reached.

Modelling Distributed Computations

1. Specification in terms of Safety and Liveness Property
2. Processes, Messages, Automata and Steps
3. Failure Model
4. Timing Assumptions
5. Communication Model

Processes and Massages



N processes each with its own identifier

Processes communicate by exchanging messages

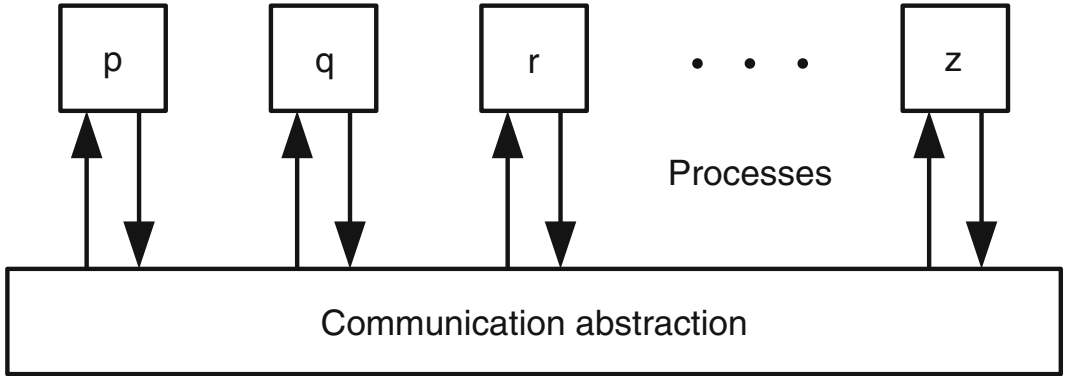
Distributed Algorithms

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

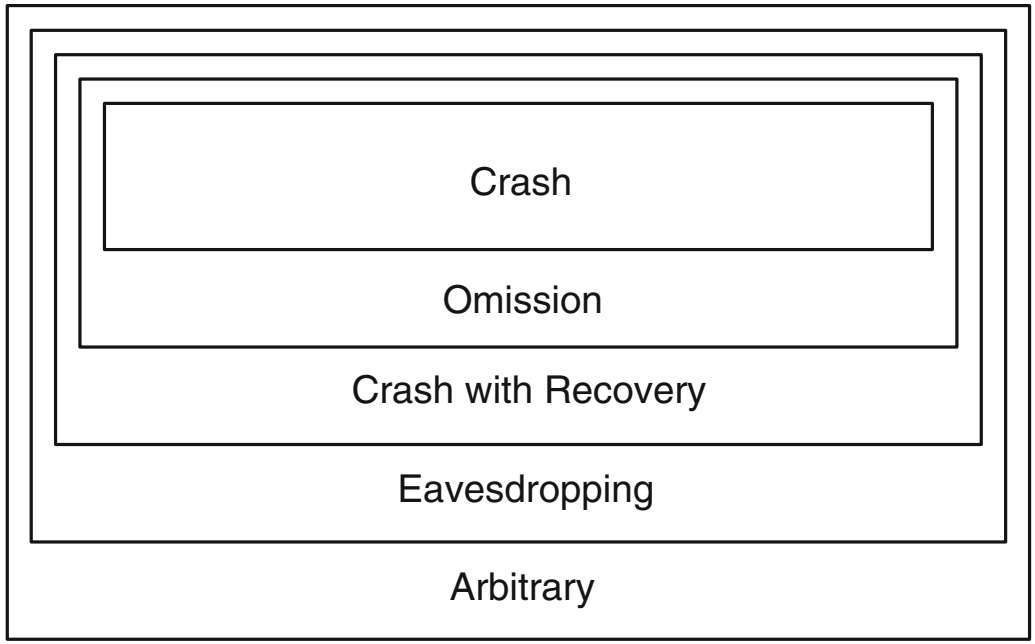


Figure 2.3: Types of process failures

Timing Assumptions

1. Synchronous
2. Asynchronous
3. Partially synchronous

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

Major problem the coverage of the synchrony assumption!!!!
This turns out in difficulty of building a system where timing
Assumptions hold with high probability

Asynchronous Systems

Assuming an asynchronous distributed system comes down to not making 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, such that 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

It 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

Communication Model

Communication Primitives

Reliable vs Unreliable