

**B.TECH.**  
Revised Syllabus  
RTU & BTU SERIES

**Ashirwad's**



# **Distributed** *Systems*

*SS Manaktala | Tarun Goyal | Deepika Sainani*



# General Information

---

- 9 CFU all in the first semester
  - September 27<sup>th</sup>– December 23<sup>rd</sup>

- Instructors
  - Silvia Bonomi
  - Giovanni Farina



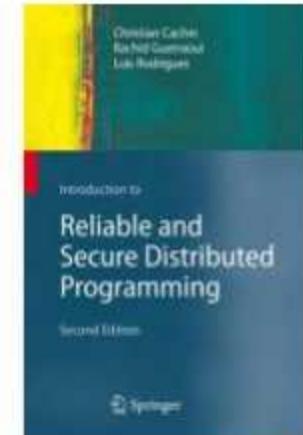
- Tutor
  - Alessandro Palma

# General Information

---

## Material

- Main Text books
  - C. Cachin, R. Guerraoui and L. Rodrigues. Introduction to Reliable and Secure Distributed Programming, Springer, 2011
- Scientific papers
- Supporting Slides



# General Information

---

## ➤ Exam

- Mandatory part
  - The exam is made of a written test
  - Questions may cover any topic contained in the final syllabus

## ➤ Optional part

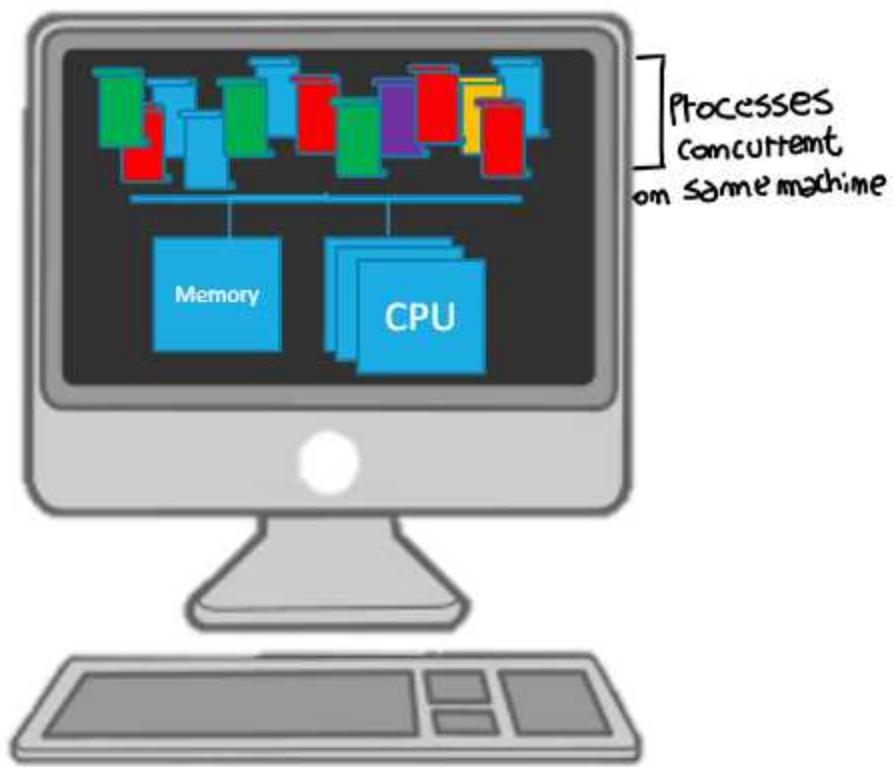
- Project assignment to develop in groups of 2-3 persons

## ➤ Rules

- All projects must be requested and assigned around mid November
- The exact delivery date will be established later in the course but, in any case, it will be no later than the first exam session
- The project will be evaluated with a score between 0 and 3 that will be added to the mark of your written test
- Further details will be available later in the course

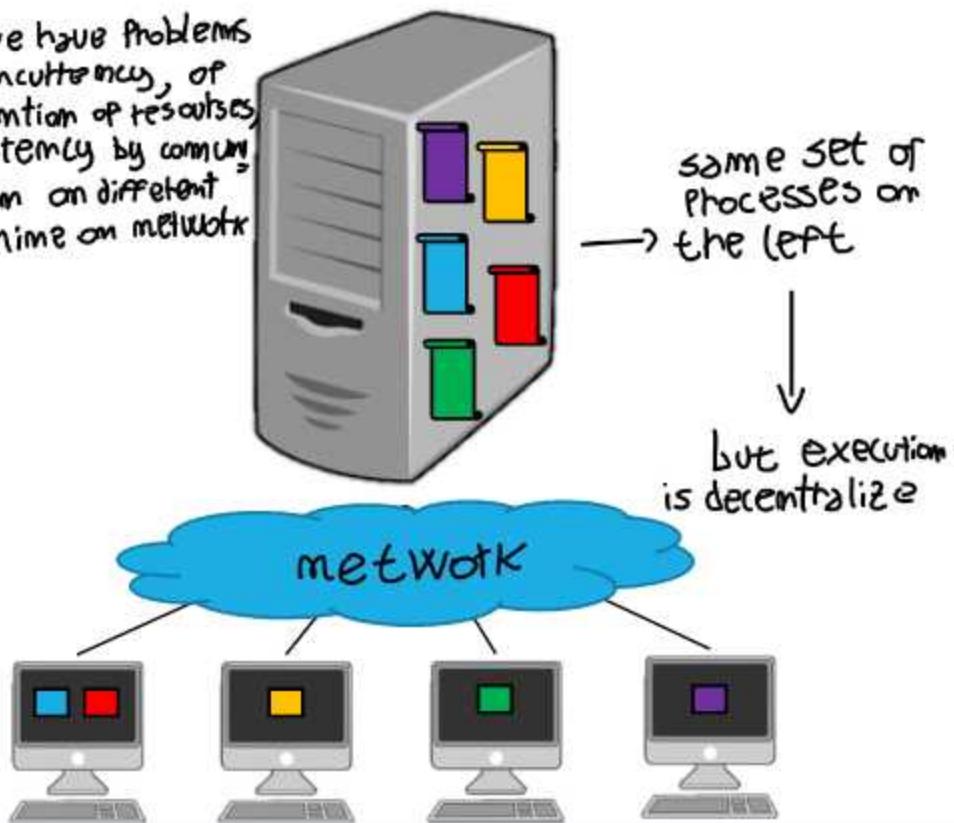
# From concurrent to distributed systems

## CONCURRENT SYSTEMS



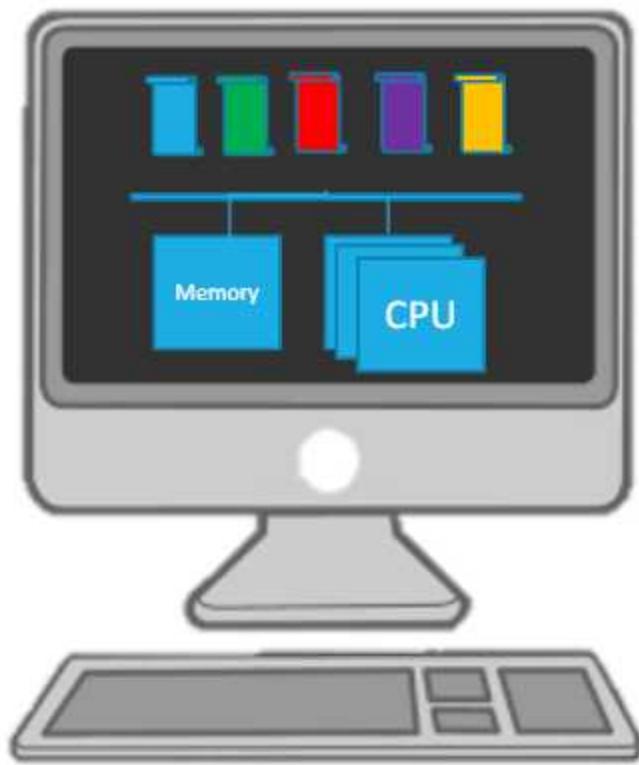
## DISTRIBUTED SYSTEMS

↳ we have problems  
of concurrency, of  
contention of resources,  
of latency by commun-  
ication on different  
machines on network

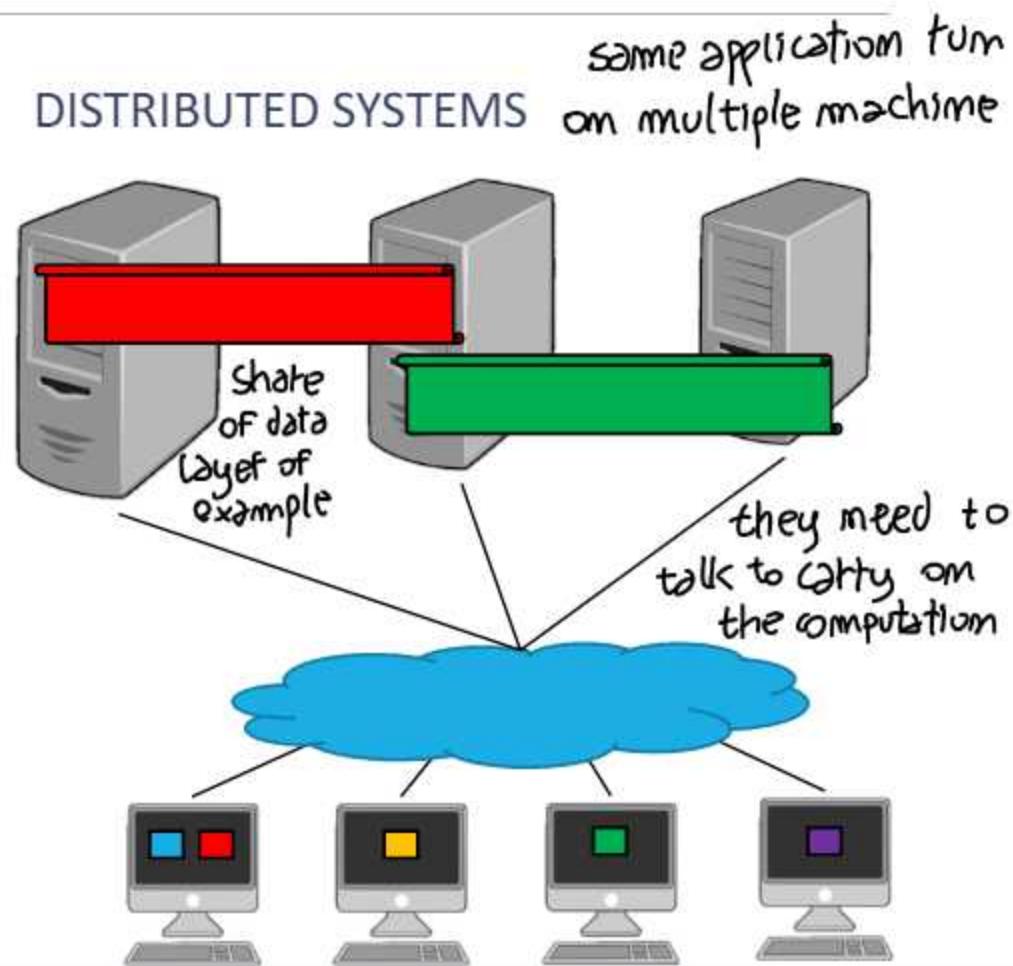


# From concurrent to distributed systems

CONCURRENT SYSTEMS



DISTRIBUTED SYSTEMS



# Definitions

what is happening on a machine is not related (in principles) to what is happening on another machine

1. *A distributed system is a set of spatially separate entities, each of these with a certain computational power that are able to communicate and to coordinate among themselves for reaching a common goal*

each machine is independent from other

2. *A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility*

3.

*A distributed system is a piece of software that ensures that a collection of independent computers appear to its users as a single coherent system*

(Maarten van Steen)

↳ Transparency: as a user, i don't care about how the application is implemented, the application should be one from my perspective. (Wolfgang Emmerich)

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

distribute system need to mask error (Leslie Lamport)

# Common points across definitions

---

- Set of entities/computes/machines *in different location*
- Communication, coordination, resource sharing
- Common Goal
- Appear as a single computing system *to user*



↳ single black-box that covers the complex distributed System

# Why Distributed Systems?

---

## 1. To Increase Performance

- To cope with the increasing demand of users in both processing power and data storage
- To reduce latency (users are spread all over the world and you want to provide them best user experience by reducing the response time)

## 2. To Build Dependable Services

- To cope with failures

↳ Ex. if i have a database deployed on a single machine and the machine fail, then the data is not more available. we want an application that has as much as possible the availability of the data, for this we distribute the data on different machine. to mask the failure

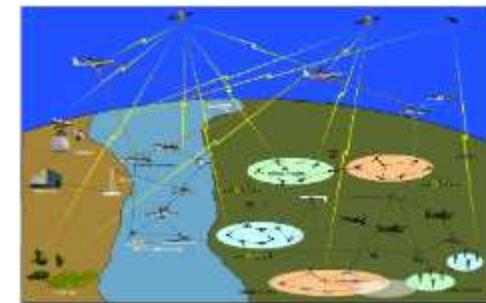
N.B.: the machine may fail at certain point.

# Distributed Systems: examples

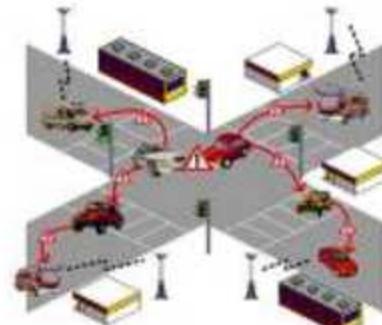
---



Internet



smart home



**OVERLAY NETWORK** is a logical network, not physical, that use to let processes communicate!!

# Distributed Systems

## CHARACTERISTIC 1: Collection of autonomous computing elements

### CHALLENGES

- absence of a global clock → how we coordinates machine?
- group membership
  - open groups vs closed groups
- overlay network
  - structured vs unstructured

→ in some application the set of machine is decided a priori, for example in closed environment like the cloud, or open like in block-chain, everybody can decide to enter in the network helping managing the chain, but exit improvise.

↳ we need to design the way, at network level, the processes communicate. You can use different type of graph (structured or unstructured)

# Distributed Systems

---

## **CHARACTERISTIC 2: Single coherent system**

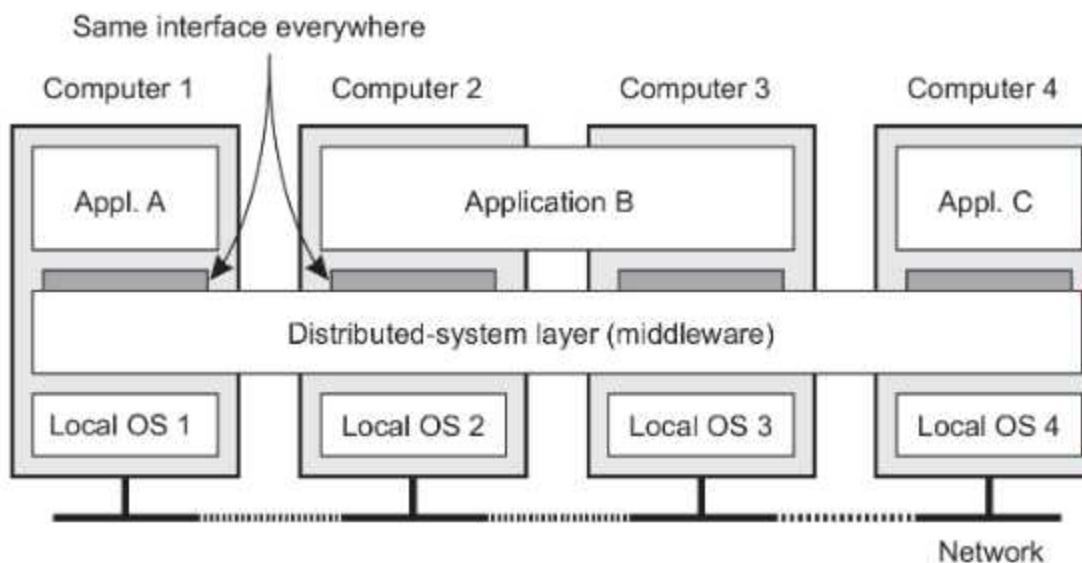
---

- a distributed system is coherent if it behaves according to the expectations of its users
  - the collection of nodes as a whole operates the same, no matter where, when, and how interaction between a user and the system takes place

## **CHALLENGE**

- Failures : We can always guarantee the access to the data

# Middleware and distributed systems

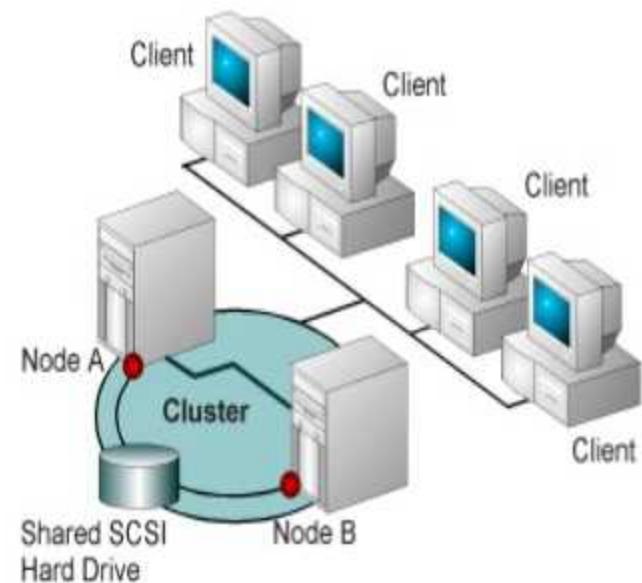
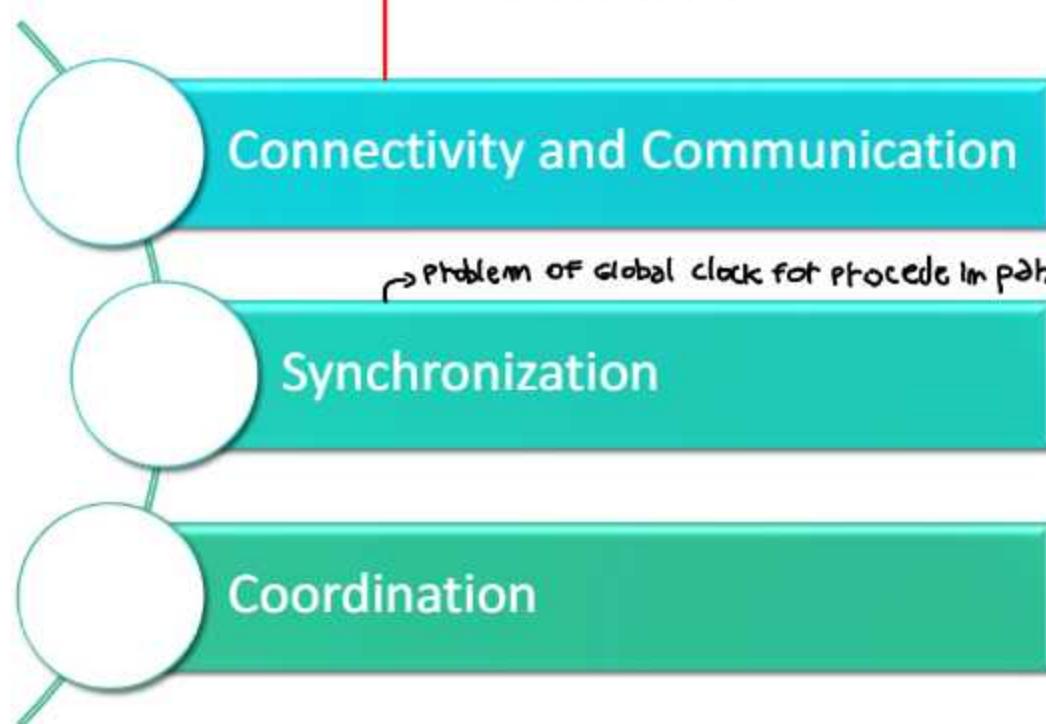


## The distributed system

- Allows inter-process communication
  - to components of a single distributed application to communicate with each other
  - to different applications
- hides, as best and reasonably as possible, the differences in hardware and operating systems from each application

# Primary Goal: Overcome the limitation of centralized environment

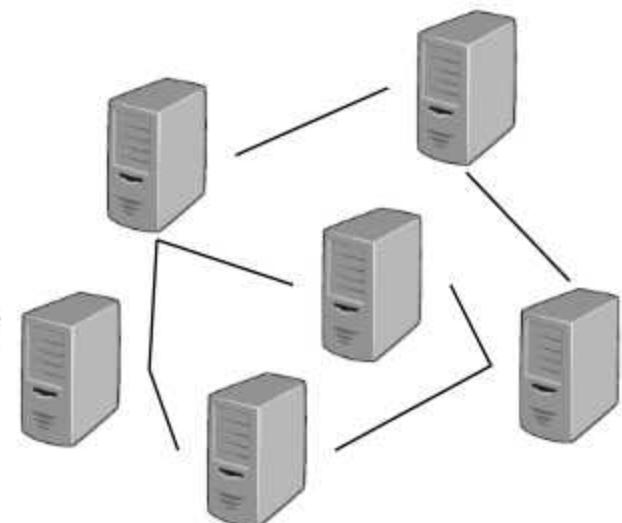
## PROBLEMS



# Primary Goal: Overcome the limitation of centralized environment

Coordination has to be implemented taking into account the following conditions that deviate from centralized systems:

1. Temporal and spatial concurrency
2. No global Clock ↗ don't exist in centralized systems
3. Failures
4. Unpredictable latencies ↗ in internet to seconds to hour.



These limitations restrict the set of coordination problems we can solve in a distributed setting

# Trends in Distributed Systems

---

Distributed systems are undergoing a period of significant change and this can be traced back to a number of influential trends:

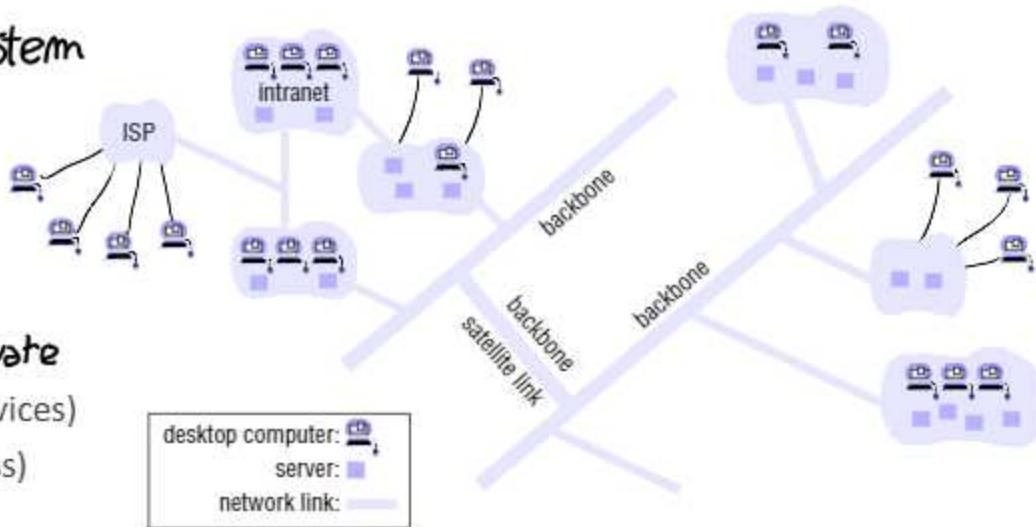
- the emergence of pervasive networking technology: *implementation of load on system, with the mobility*
- the emergence of ubiquitous computing coupled with the desire to support user mobility in distributed systems *→ application running whenever it is the user and its connectivity*
- the increasing demand for multimedia services: *Netflix, Dazn, ...*
- the view of distributed systems as a utility *, user don't see the complexity behind it, seem a real thing*

# Pervasive networking and the modern Internet

Figure 1.3 A typical portion of the Internet

## CHARACTERISTICS for design system

- Scale: number of estimated user.
- Heterogeneity in hardware and software
  - Devices (e.g. servers, workstations, tiny devices)
  - Communication Protocols (wired vs wireless)
  - Available services
- Absence of time and space limitation to connection requests : assumption on location and time



# Mobile and Ubiquitous Computing

---

Mobile computing is the performance of computing tasks while the user is on the move, or visiting places other than their usual environment

Ubiquitous computing is the harnessing of many small, cheap computational devices that are present in users' physical environments, including the home, office and even natural settings

## COMMON PROBLEMS

- N.B.:
- System scale
  - Dynamicity in the system
  - Heterogeneity of participants
  - Security Issues

# Distributed Multimedia Systems

---

Multimedia support is the ability of a system to support a range of media types in an integrated manner

- It should be able to perform the same functions for continuous media types such as audio and video

## CHALLENGES

- Temporal dimension
- Quality of Service is a strong requirement : *Like zoom for video quality*

# Distributed computing as a utility

## CLOUD COMPUTING

- IaaS
- PaaS
- SaaS

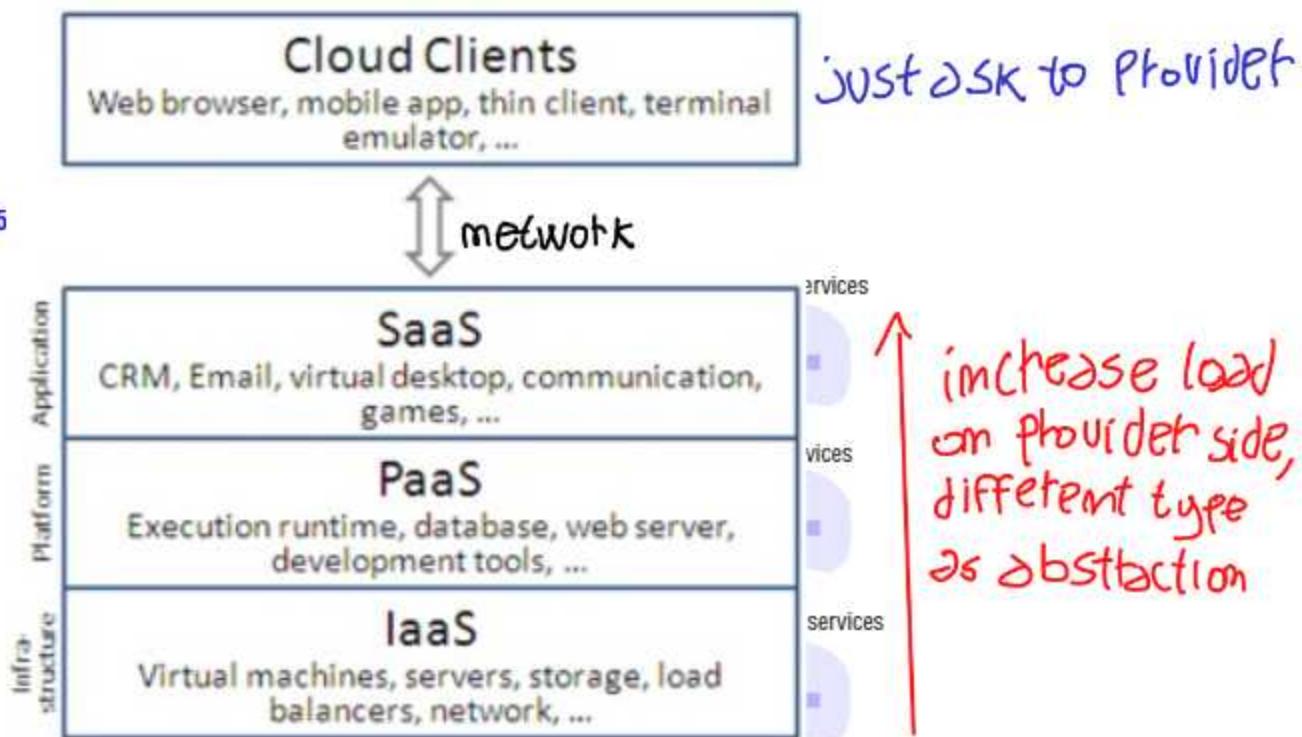
different level of services  
that you can ask to cloud  
provider

Software

Platform

infrastructure

Figure 1.5



# Dependability

---

# Dependability Definition

Such other systems are called  
Environment

A system is an entity that interacts with other  
entities, i.e., other systems, including  
hardware, software, humans, and the physical  
world with its natural phenomena.

1. **Dependability** is the ability of a system to deliver a service that can justifiably  
be trusted

A complex distributed system need to provide something to client that can trust, expected some level of quality and performance, tolerance of failure. System need to match user expectation.

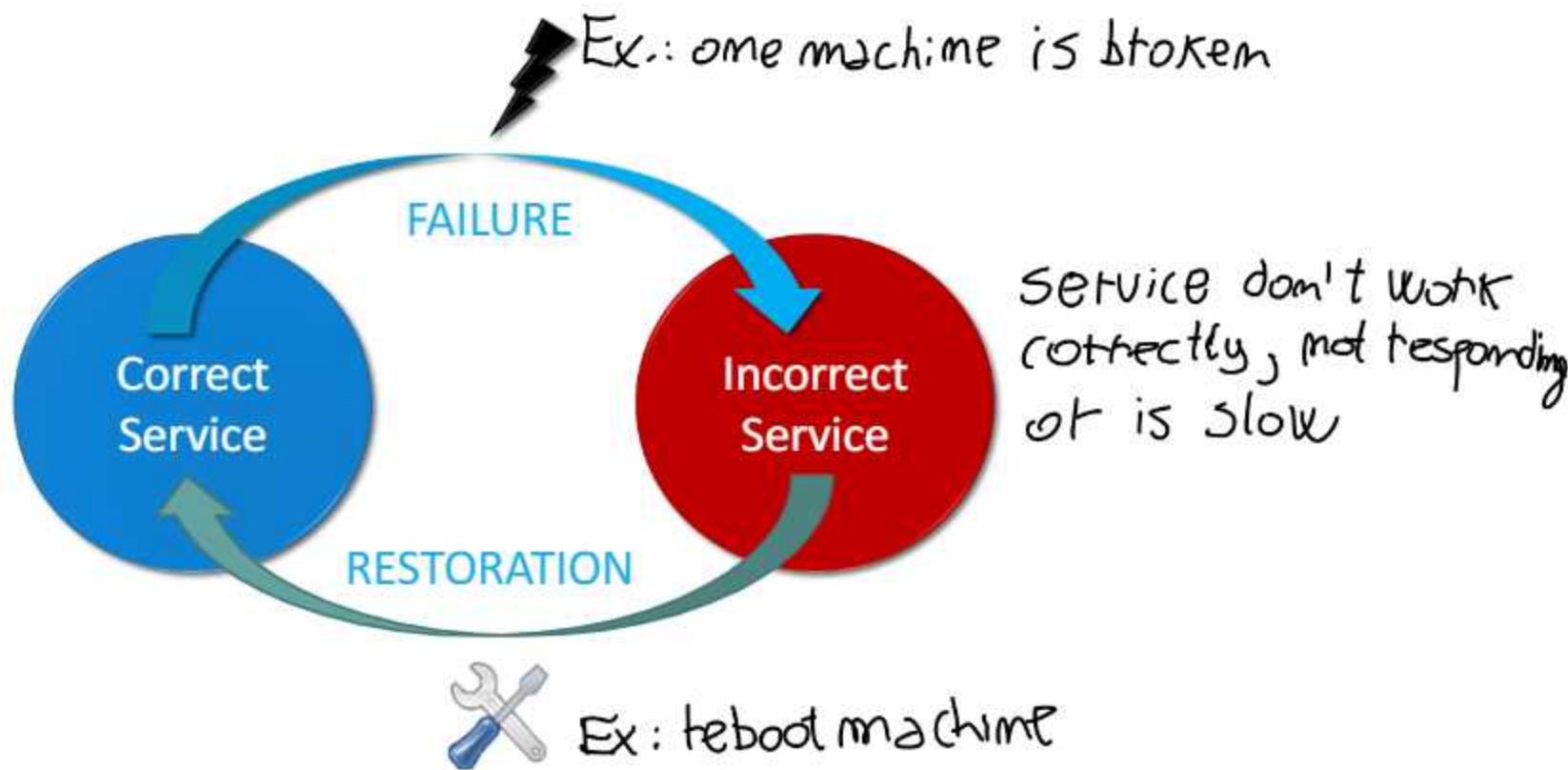
# An Alternative Dependability Definition

***Dependability is the ability of a system to avoid service failures that are more frequent and more severe than is acceptable***

A service failure (or simply failure) is an event that occurs when the delivered service deviates from correct service

- A correct service is delivered when the service implements its functional specification in terms of
  - functionality
  - performance

# Service Failure



# Service Failure



↳ scope of dependability is to ensure that this time is very short, close to zero.

## PROBLEM

? replication, ...

How to design, develop and deploy a system that is dependable and secure?

# Dependability Attributes (or requirements)

3 main attributes

Availability

Readiness of correct service

Reliability

Continuity of correct service

Safety

Absence of catastrophic consequences on the user(s) and the environment

Integrity

Absence of improper system alterations

Maintainability

Ability to undergo modifications and repairs, *continuing to work!*

# Secondary Dependability Attributes

---

## Robustness

Dependability with respect to external faults which characterizes a system reaction to a specific class of faults

↳ mehr Robuste, mehr Secure

# Failures, Errors and Faults *are different*

**OBSERVATION:** having a service failure means that there exists at least a deviation of the system behaviour from the correct service state.

The deviation from the correct state is called an **error**

The adjudged or hypothesized cause of an error is called a **fault**

- A Fault can be internal or external of a system

what goes on system,  
machine is broken

application is  
not answering

failure of service

Fault

Causes

Error

Generates

Failure

# The Means to Attain Dependability

## Fault Prevention

Prevent the occurrence or introduction of faults

## Fault Tolerance

Avoid service failures in the presence of faults

## Fault Removal

Reduce the number and severity of faults

## Fault Forecasting

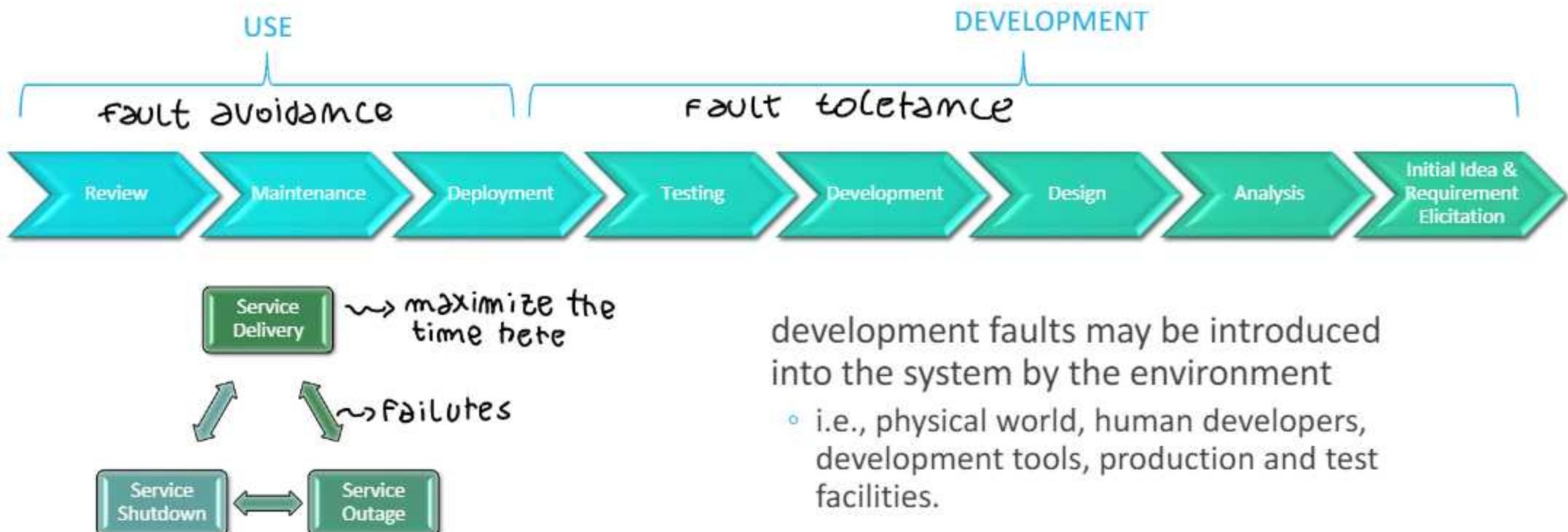
Estimate the present number, the future incidence and the likely consequences of faults

Aim to provide the ability to deliver a service that can be trusted

Aim to justify that the system is likely to meet its functional, dependability and security requirements

# Threats to Dependability during its life cycle

THE LIFE CYCLE OF A SYSTEM CONSISTS OF TWO MAIN PHASES



# Dependable Distributed Systems

## Characteristics and Challenges



# Heterogeneity

problem of a system to be design, developed and deployed in different platform. This problem is masked to user.

Heterogeneity impacts at different layers

- Networks
- Hardware
- Operating Systems
- Programming Language
- Implementations from different Developers

## SOLUTIONS

→ abstraction from the detail of single machine

- Middleware (from RPC to Service oriented Architectures)
- Mobile code and Virtual Machine

Many  
technological  
aspects will be  
Studied in  
Software  
Engineering

*specification* is what you want the system is able to do and how do you want to interact with the system is *interface*.

↳ openness because we have two independent box.

# Openness

*Openness is the capability of a system to be extended and re-implemented*



A specification of a service/component is well-formed if it is :

- Complete i.e., every thing related to the behaviour has been specified
- Neutral i.e., it does not offer any detail on a possible implementation

How to work with specifications will be part of DDS

An Interface describes:

- the syntax and the semantic of a service/component
  - available functions/services
  - input parameters, exceptions
- It can be realized through an IDL

Interface definition language (IDL): XML, ...

High-level interfaces will be part of DDS

Technological aspects will be Studied in Software Engineering

# Openness guarantee:

key requirement

Interoperability	Capability of two systems to cooperate by using services/components specified by a common standard
Portability	Capability of a service/component implemented on a distributed system A to work on a system B without doing any modification <i>(cross platform)</i>
Flexibility	Capability of a system to configure/ orchestrate components developed by various programmers, <i>modular approach</i>
Add-on Features	Capability of a distributed system of adding components/services and be integrating in a running system
Evolvability	Capability of a system to evolve in time (e.g., leaving active two different version of the same service)
Self-*	Capability of a system to reconfigure, to manage itself without human intervention

# Security → CIA Requirement



DDS will look at these attributes from the design point of view

You will study how to technically manage security in Cybersecurity course

# Secondary Security Attributes

~~NO~~

## Accountability

Availability and integrity of the identity of the person who performed an operation

## Authenticity

Integrity of a message content and origin, and possibly of some other information, such as the time of emission

## Non-repudiability

availability and integrity of the identity of the sender of a message or of the receiver

You will study how to technically manage security in Cybersecurity course

# Dependability & Security



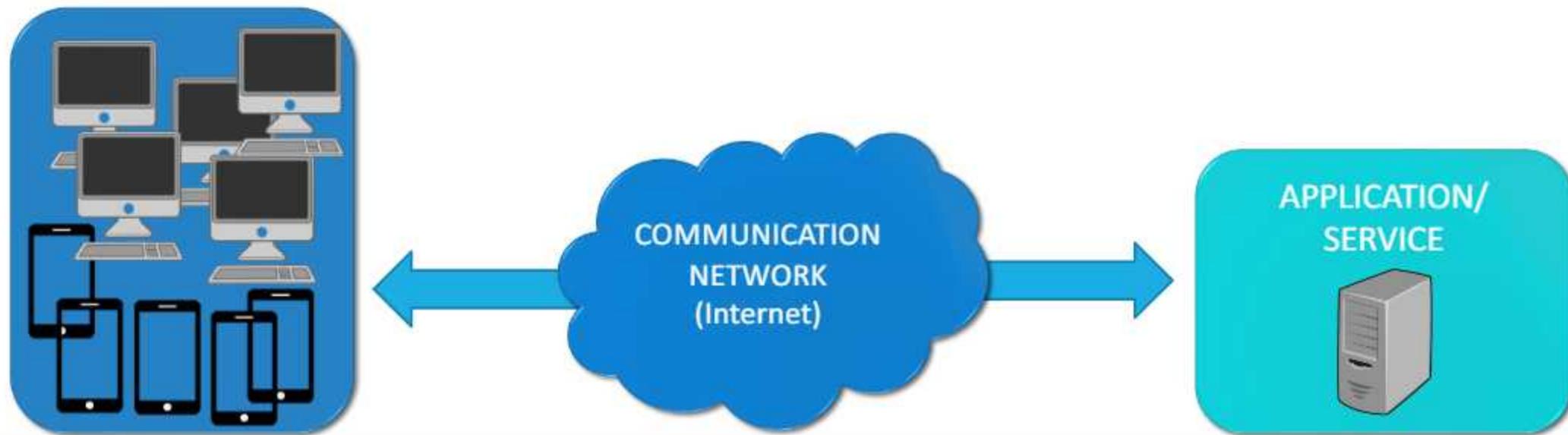
# Scalability

capability of the system to continue working even if the composition of system change, the load

*A system is scalable if it remains running with adequate performance even if the number of resources and/or the number of users grow up of orders of magnitude*



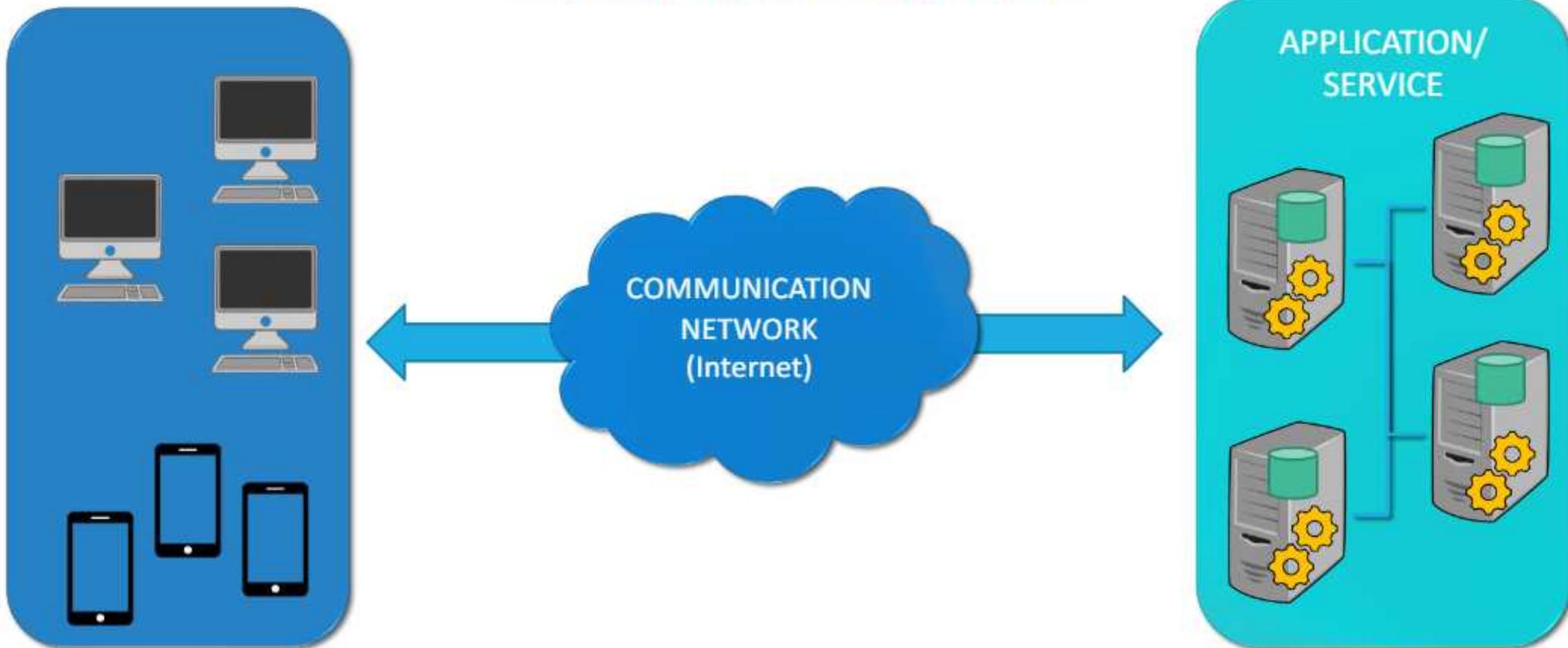
Centralization is against scalability



# Scalability

THE SOLUTION IS IN REPLICATION

tradeoff:  
When introduce multiple machine, they need to communicate on the network, require time.  
distribute load on different machine



# Scalability

---

## REPLICATION ISSUES



- Service
  - Coordination Problems



- Data
  - Consistency Problems



- Computation
  - No node has the current state of the whole system
  - Nodes base their decisions on data they own
  - A failure of a node should not compromise the goal of the algorithm

if service is **statefull** (need to maintain the state) the cost is high and is complex, **stateless** is more easy.

# Scalability

---

The project of a scalable system shows four main problems

**System Dynamicity**

- Adding/removing servers/processes on-the-fly

**Check performance metrics**

- Servers/processes have not to interact with all application' users
- Employ algorithms that do not require to use the entire set of data

**Using carefully scarce resource**

- e.g., battery drain in embedded systems

**Avoiding bottlenecks**

- e.g., Centralized vs distributed DNS



Note that deployments can impose (for security or business requirements) centralized solutions under several conditions

# Concurrency

---



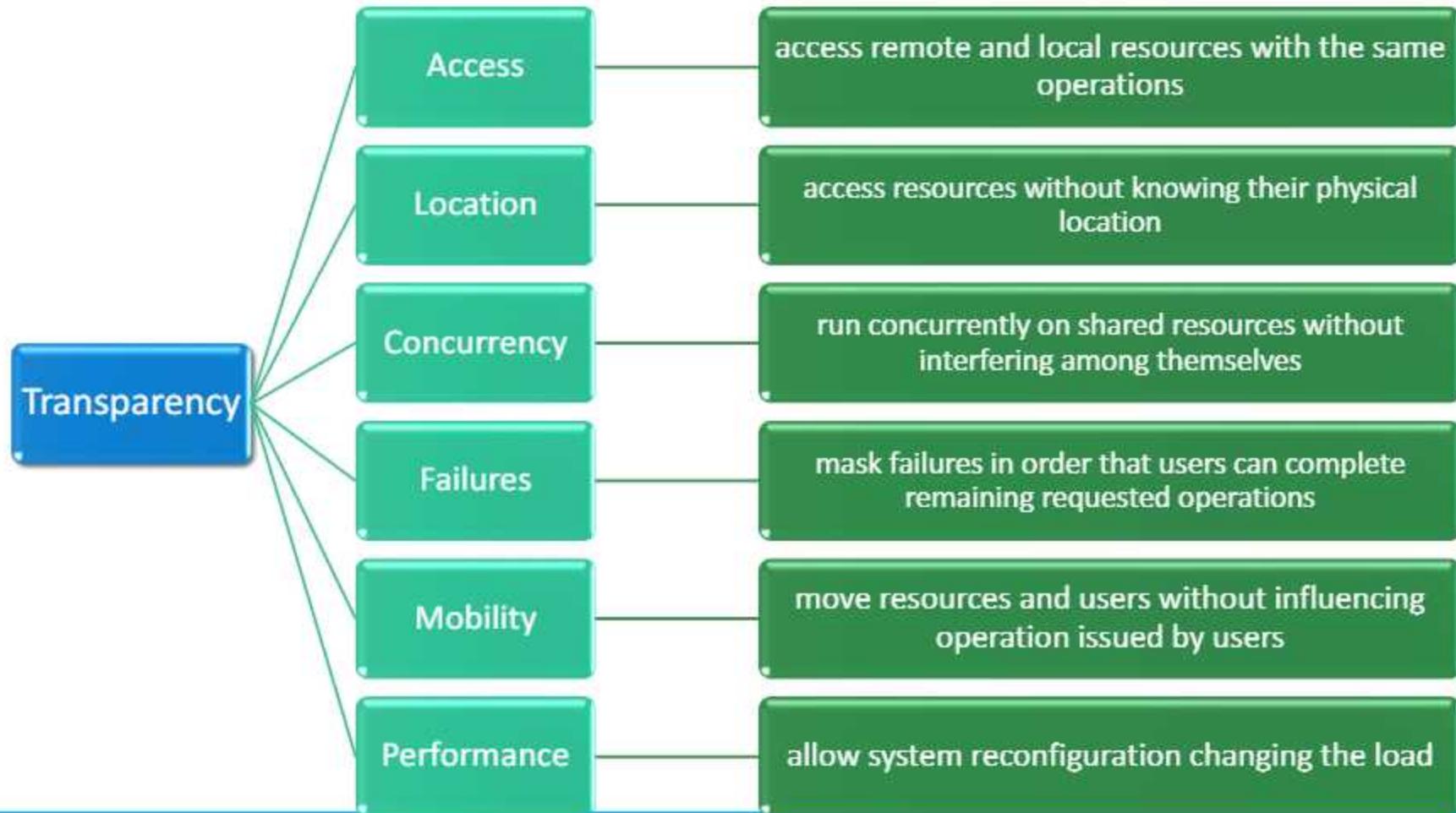
Multiple access to shared resources

- If clients invoke concurrently read and write methods on a shared variable, which value returns each read?

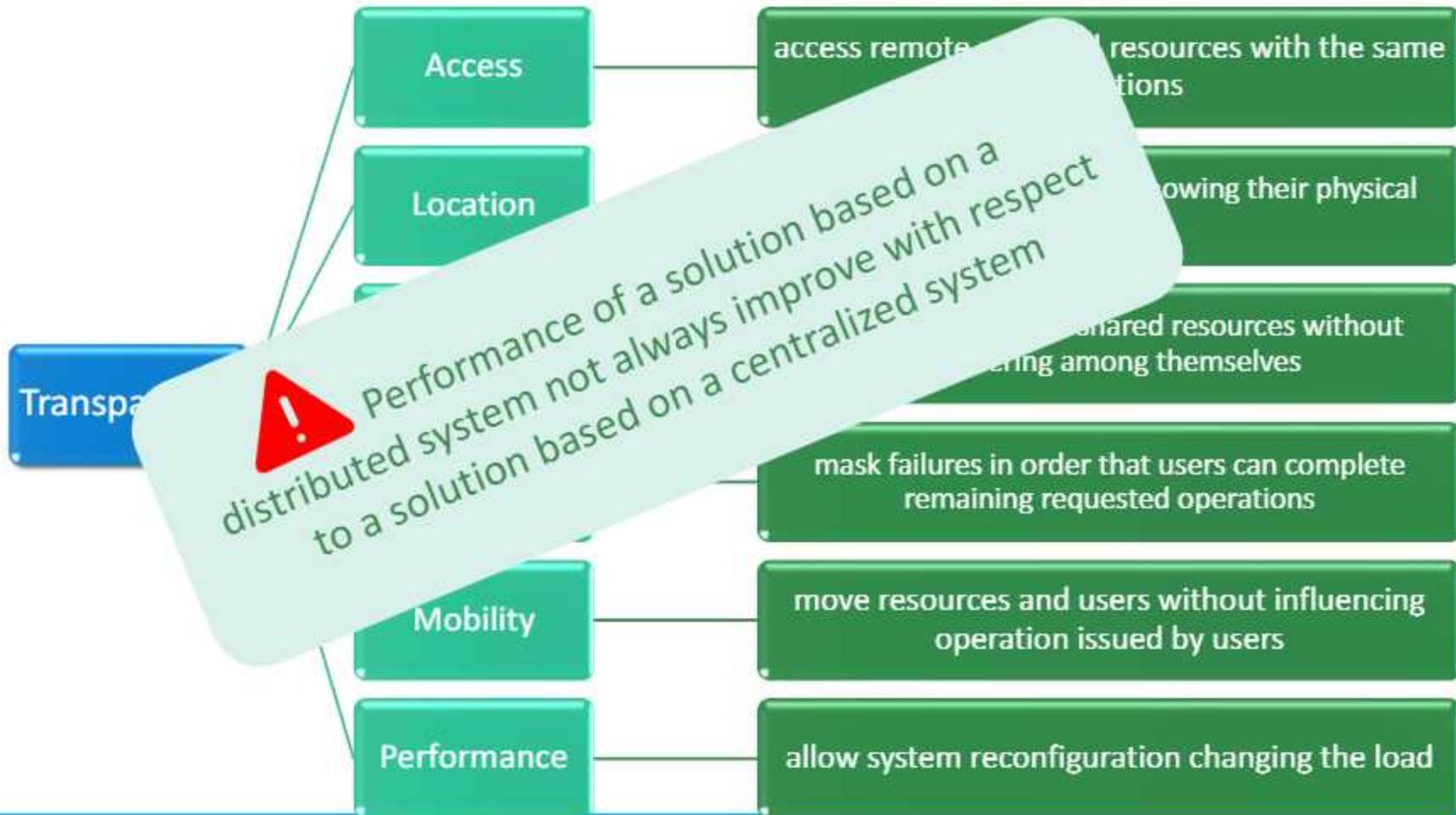
Coordination

Synchronization

# Transparency



# Transparency



# References

---

- Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, Carl E. Landwehr: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1(1): 11-33 (2004)  
<https://ieeexplore.ieee.org/document/1335465/>

**NOTE:** Use the Sapienza proxy to access this paper. Instruction on how to do it can be found here <https://login.ezproxy.uniroma1.it/login>

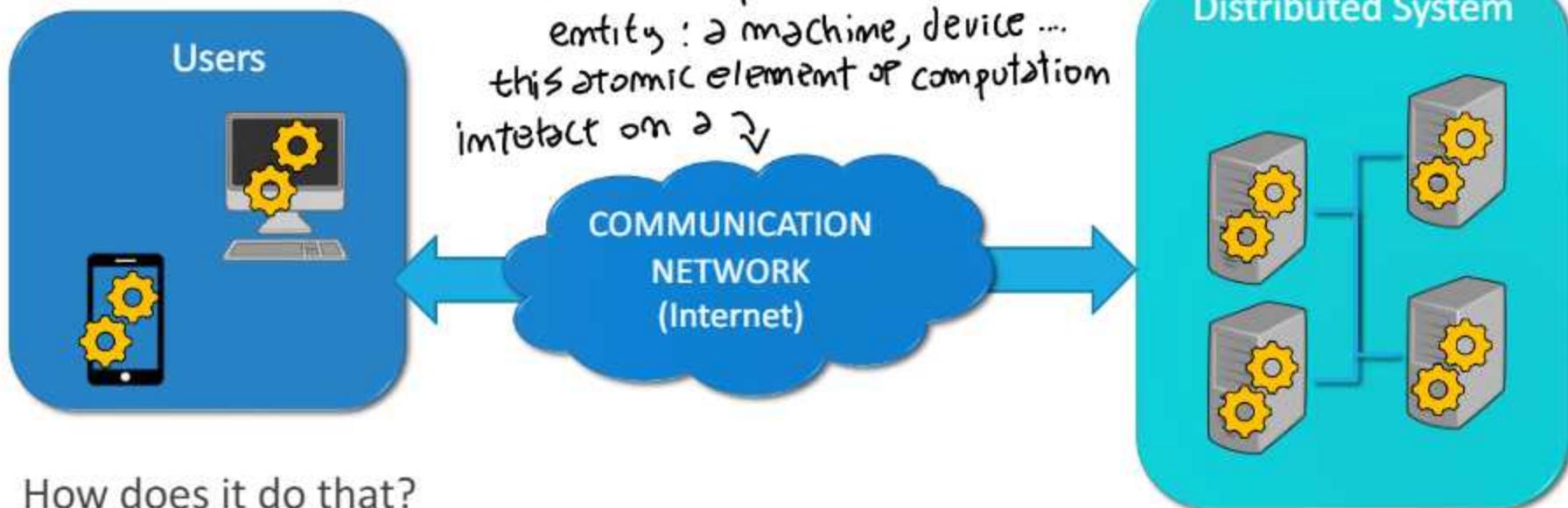
We have a set of users that are the client of the system, there is a distributed system that seem to user a blackbox, that provide service to the user. interaction happen thought a communication network.

## 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

we call processes, an atomic entity : a machine, device ...  
this atomic element of computation  
interact on a ↗



How does it do that?

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

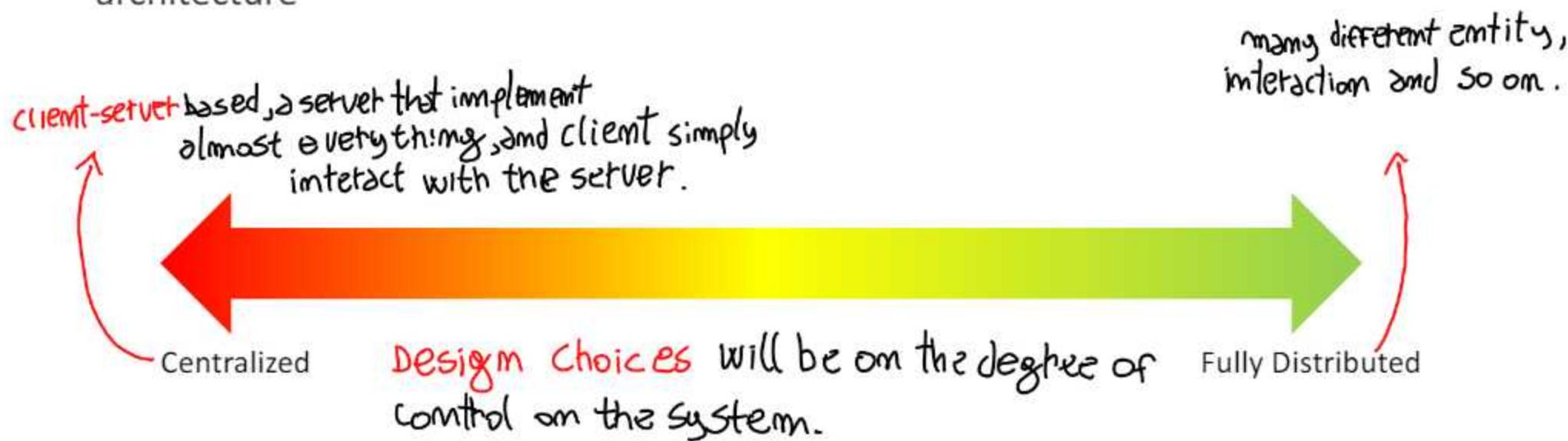
In practice, how can the user interact with the system, knowing that in back-end there are a lot of troubles? → we have to write software, put a layer of software on top the o.s. that mask all the problems (Ethetogeneity, ...), and provide to user just interfaces.

# System deployment → NEED TO UNDERSTAND REQUIREMENT FOR DESIGN THE SYSTEM!!

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

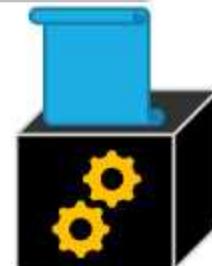


**component** is a unit of software, a black-box that you take, develop and deploy somewhere in the system. Have a specification, have an unspecified behavior and an interface.

# Architectural Style

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, need synchronization between components
- how these elements are jointly configured into a system



More about this  
in Software  
Engineering

Components can be abstracted by their specification and their interfaces

→ i know what the boxes supposed to do, the input and output by specification.  
All this architectural style can be define without knowing the detail of implementation

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 → capture essence of the problem, don't care about the issue of a specific platform, language ...
2. help distinguish the fundamental from the accessory → algorithm design focus only on fundamental things. correctness not performances!
3. prevent system designers and engineers from reinventing, over and over, the same solutions for slight variants of the very same problems.

→ for not start from scratch, design system as a composition of boxes and we just focus on a single box each time. We work incrementally without reinvent the alg= hithm each time.

# The road to build a distributed abstraction

## Step 1: definition of the system model, is the representation of the environment

- A system model must:

- describe the relevant elements in an abstract way
- identify their intrinsic properties
- characterize their interactions

↳ if latency is  
controllably, time  
of communication and so on,...

↳ communicated on internet,  
there is a shared memory...

↳ for starting is necessary to  
understand the characteristics of  
the system, identify the most  
common property.

## Step 2: build a distributed abstraction

- understand how to design a protocol that capture recurring interaction patterns in distributed applications  
↳ focusing on the way with the processing interface each other.  
how to talk with other machine for progress with computation

## 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

the methodology for design algorithm is **composition model** that describe the protocols, the behaviour of every box in terms of **PSEUDO-CODE**. because we have to abstract from specific language.

## Composition Model

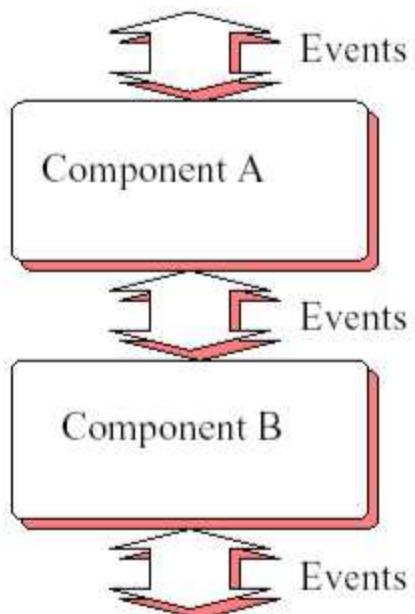
The protocols we will consider in this course are presented in pseudo-code

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.

→ the algorithm will don't follow a monadic flow, but will have many small piece of code that will be triggered as soon an event happen in the system, because for collaborate need to take an action when something happen in the system. (Ex. request from a process, need to do something ...)

# Composition Model and its code



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

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

component that capture the event  
generate a event for another component

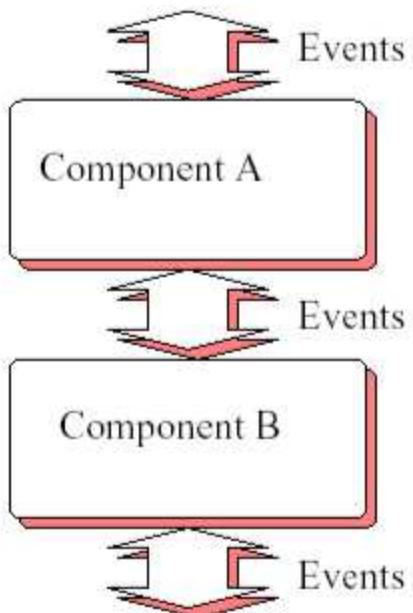
Keyword for identifying the beginning of the code when the event is captured from component

manage event code

input for event 4

Figure 1.1. Composition model

# Composition Model and its code



Variable updated, timer elaps ... a condition should be satisfied.

---

**upon condition do** // an internal event  
do something;  $\sim$  handler code

not related with interaction with other component

---

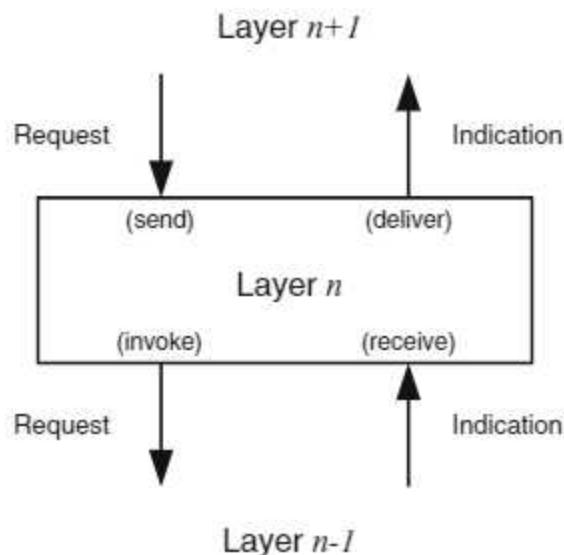
**upon event  $\langle co, Event \mid att_1^1, att_1^2, \dots \rangle$  such that condition do**  
do something;

mix events and condition  
for handle a situation

Figure 1.1. Composition model

# Programming Interface

when a layer asking something, arrow down



## 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

receive something from bottom layer, that you asked, allow incoming

## Indication events

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

functionalities implemented from other, you can use like interface, for provide of receive services

# Example - Job handler

---

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

Module:

Name: JobHandler, instance *jh*.

Events: *interface of the module*

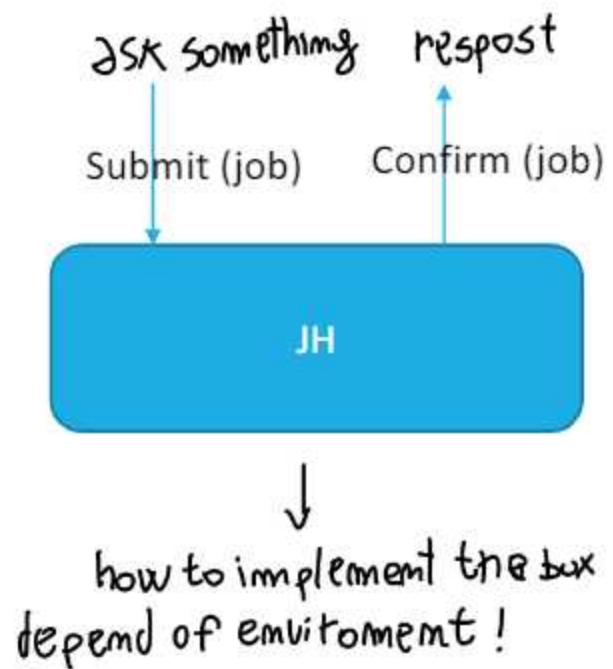
Request:  $\langle jh, \text{Submit} \mid job \rangle$ : Requests a job to be processed. *(request event)*

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

Properties: *behavior that module should satisfy*

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

*name of property*: *description*.



# Example – Job handler (synchronous implementation)

---

## Algorithm 1.1: Synchronous Job Handler

---

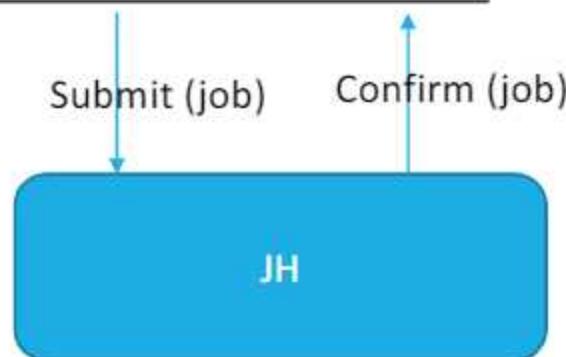
Implements:

JobHandler, **instance** *jh*.

**upon event**  $\langle jh, \text{Submit} \mid job \rangle$  **do**  
    process(*job*);  
    **trigger**  $\langle jh, \text{Confirm} \mid job \rangle$ ;

design a fragment of code  
for every arrow that is  
incoming inside  
the box. Here one!

remain  
blocked and at the  
and respond.



# Example – Job handler (asynchronous implementation)

---

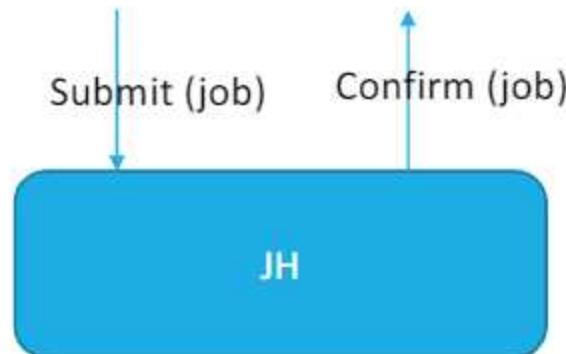
## Algorithm 1.2: Asynchronous Job Handler

---

Implements:

JobHandler, instance  $jh$ .

```
upon event <  $jh$ , Init > do
    buffer :=  $\emptyset$ ; Putting in buffer pending request
upon event <  $jh$ , Submit | job > do and executed according
    buffer := buffer  $\cup$  {job}; to a policy here FIFO.
    trigger <  $jh$ , Confirm | job >;
empty
upon  $buffer \neq \emptyset$  do internal event based
    job := selectjob(buffer); on a condition.
    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:**  $\langle th, Submit \mid job \rangle$ : Submits a job for transformation and for processing.

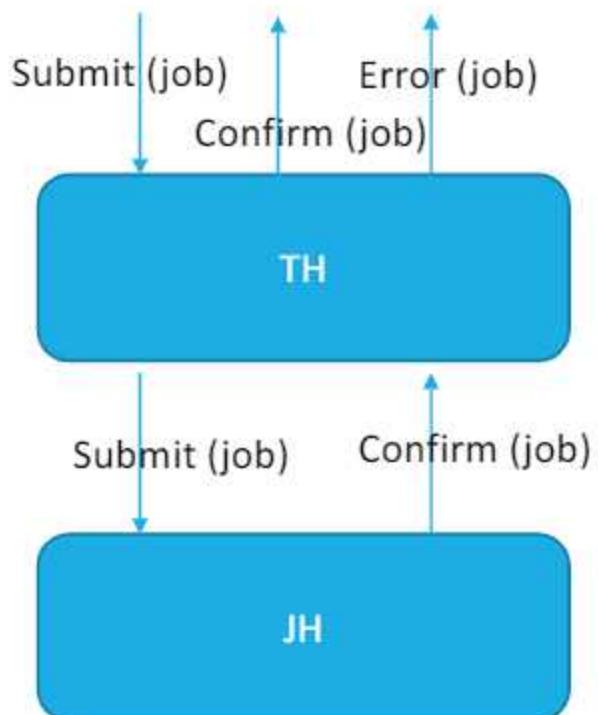
**Indication:**  $\langle th, Confirm \mid job \rangle$ : Confirms that the given job has been (or will be) transformed and processed.

**Indication:**  $\langle th, Error \mid job \rangle$ : 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

$top := 1;$   
     $bottom := 1;$  *don't full*  
     $handling := \text{FALSE};$   
     $buffer := [\perp]^M;$  *buffer with limited size*

upon event  $\langle th, Submit | job \rangle$  do

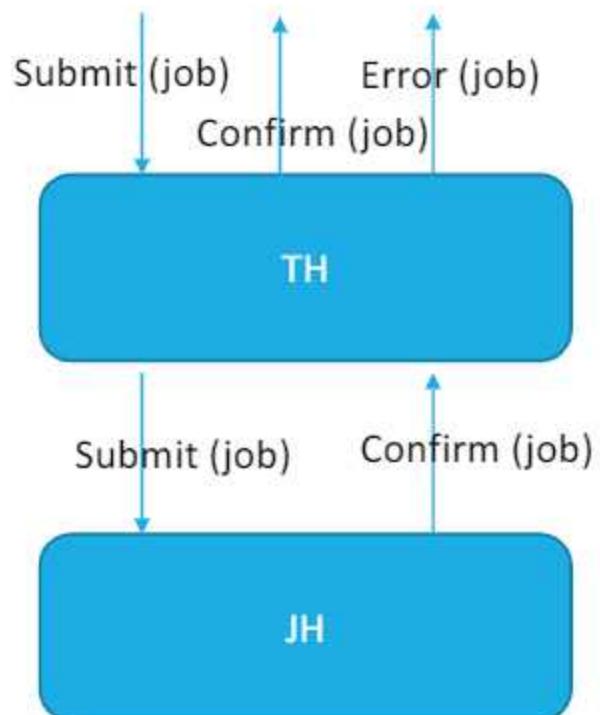
    if  $bottom + M = top$  then  
        trigger  $\langle th, Error | job \rangle;$   
    else  
         $buffer[top \bmod M + 1] := job;$   
         $top := top + 1;$   
        trigger  $\langle th, Confirm | job \rangle;$

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

$handling := \text{FALSE};$



# MODELLING DISTRIBUTED COMPUTATIONS

## Processes

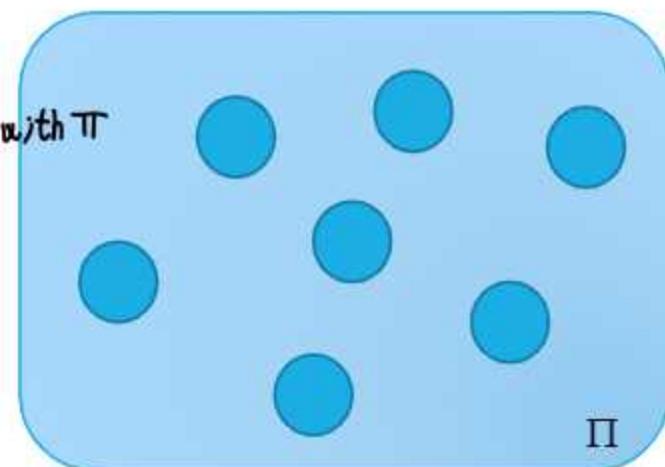
- $\Pi$  denotes the set of processes

- Unless stated otherwise, this set is static and does not change, and every process knows the identities of all processes.

- Sometimes, a function  $\text{rank} : \Pi \rightarrow \{1, \dots, N\}$  is used to associate every process with a unique index between 1 and N

fixed and mapped with  $\Pi$   
↑

identity by integer



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

↳ or also `my-id`

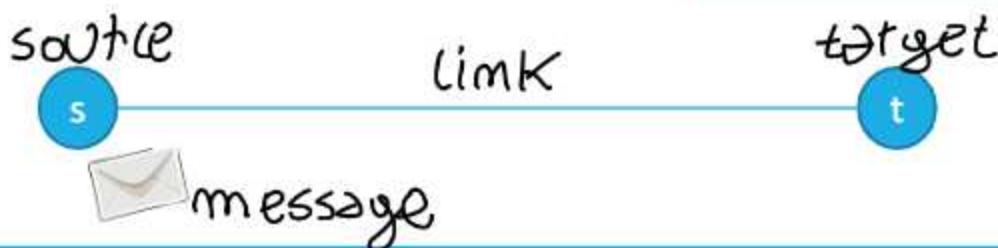
~~> process meat always communicates using messages

# Processes' interactions and Messages

- Processes in  $\Pi$  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.



- the algorithm can be represented as a **state machine**, in which the transition between a state to another is represented by the events that are happening in the system.
- the same state machine is executed by every process, is needed synchronization.

# Distributed Algorithms

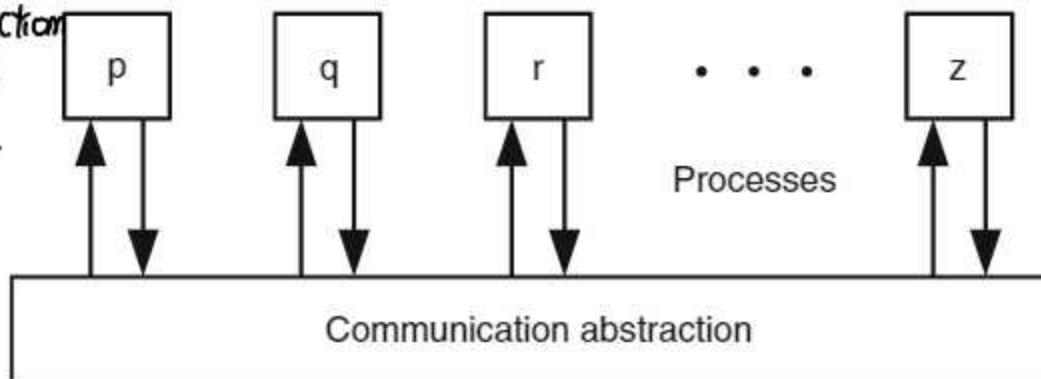
A distributed algorithm consists of a distributed collection of automata, one per process.

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.

the sequence of transition  
executed by processes  
over individual automata.



**SPECIFICATION OF DISTRIBUTED ALGORITHMS** is given in term of property : safety properties and liveness properties

## Safety and Liveness

→ describe the condition that the algorithm must satisfied without making mistake.

*Safety properties state that the algorithm should not do anything wrong*

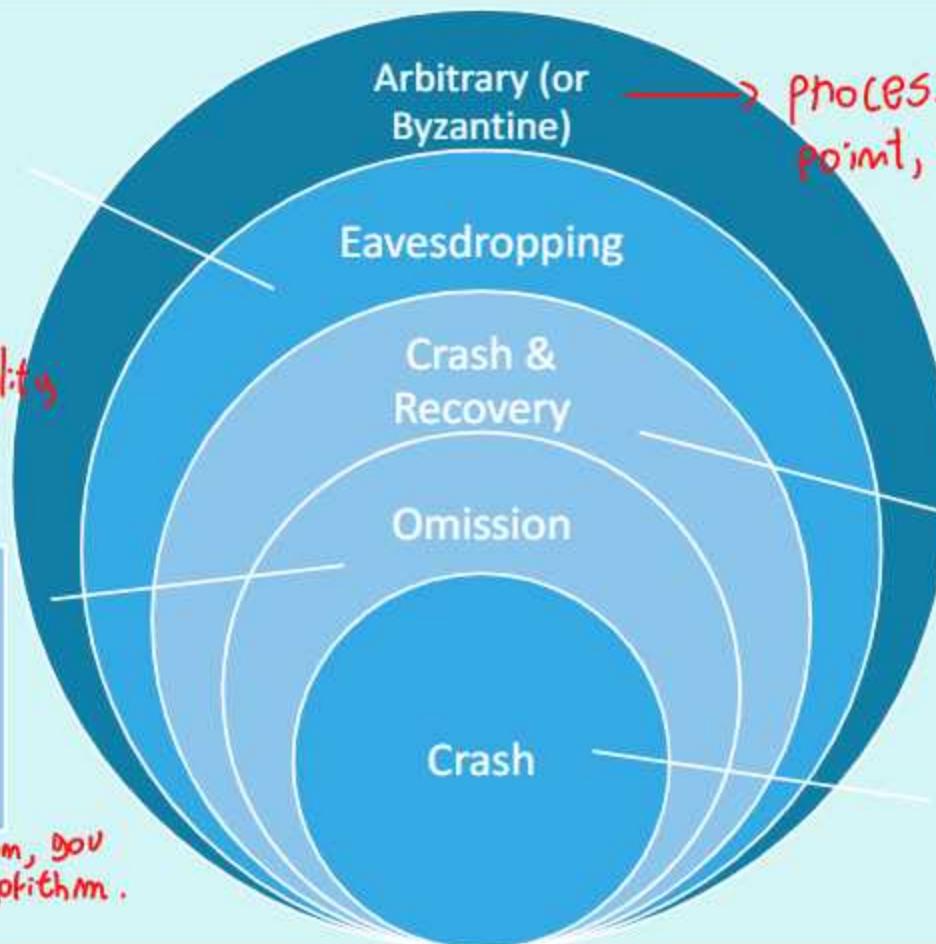
- a safety property is a property of a distributed algorithm that can be violated at some time t and never be satisfied again after that time
- a safety property is a property such that, whenever it is violated in some execution E of an algorithm, there is a partial execution E' of E such that the property will be violated in any extension of E'

→ identify all the properties that follow the process of the computation

*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'  $\geq t$

# Failure Models: failure we want to tolerate.



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

Problem related to confidentiality

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

send omission or delivery omission, you omit to run a small part of algorithm.

Stop and restart

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

component stop working at instant t the process stops its execution

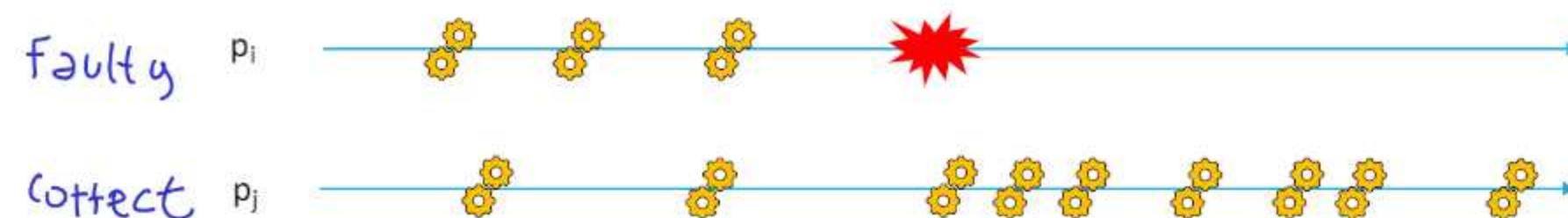
# CRASH-STOP MODEL

## 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



is capability of a system to provide a service that are justify trusted



key for ensure dependability.

in our model we will consider the process may fail, and write software

with the idea.

otherwise one fault will destroy the system!

# 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.

↳ robustness also



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

→ change the definition of correct and faulty

# 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 !

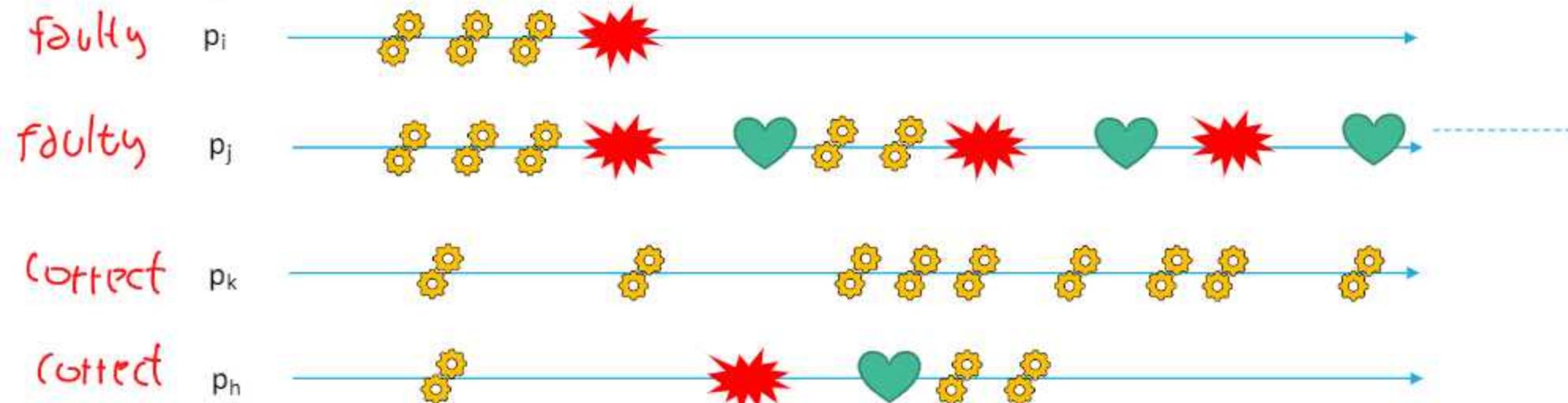
# Crash-recovery fault

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

*where log the progress of computation*

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 ASSUMPTION

over the time need for process information

### Synchronous System

Characterized by three properties

#### 1. Synchronous processing

- Known Upper Bound on the time taken by a process to execute a basic step

With this knowledge you can estimate the time need to run the algorithm and communicate with other, and use this to infer if other are working correctly or not.

↳ you can achieve mutual exclusion on access database without communication

#### 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

you can easily estimate performance of algorithms

- Timed failure detection, *Know Upper bound*
- 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 with one failure  $\rightarrow$  no agreement!

# 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

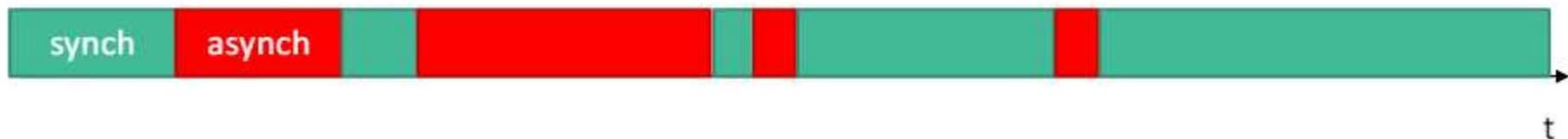
**WARNING:** Assuming Partial synchrony does not mean that

- IMP!!*
- 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

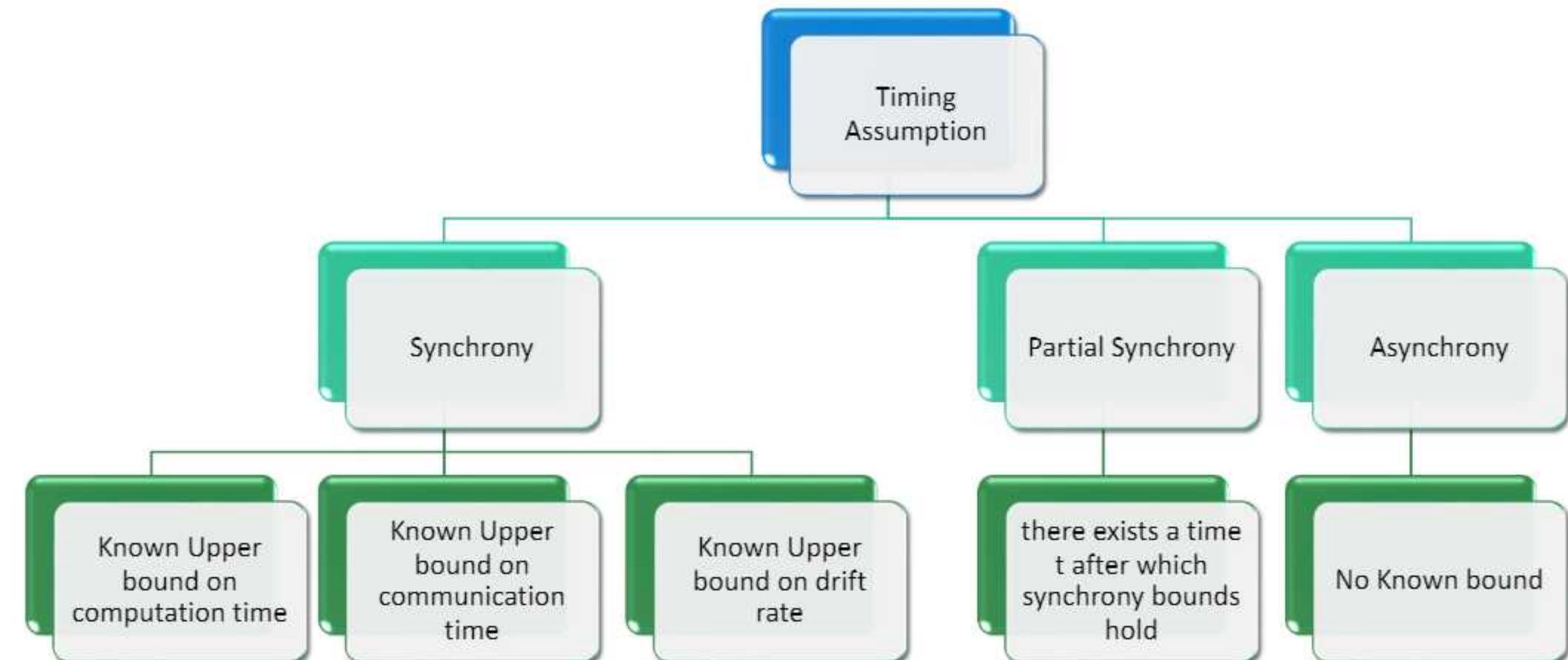
---

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



RECAP.

# 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

could be TCP, UDP, ... Abstract: Link is a connection between two processes, is an edge in the graph that represents the communication network.

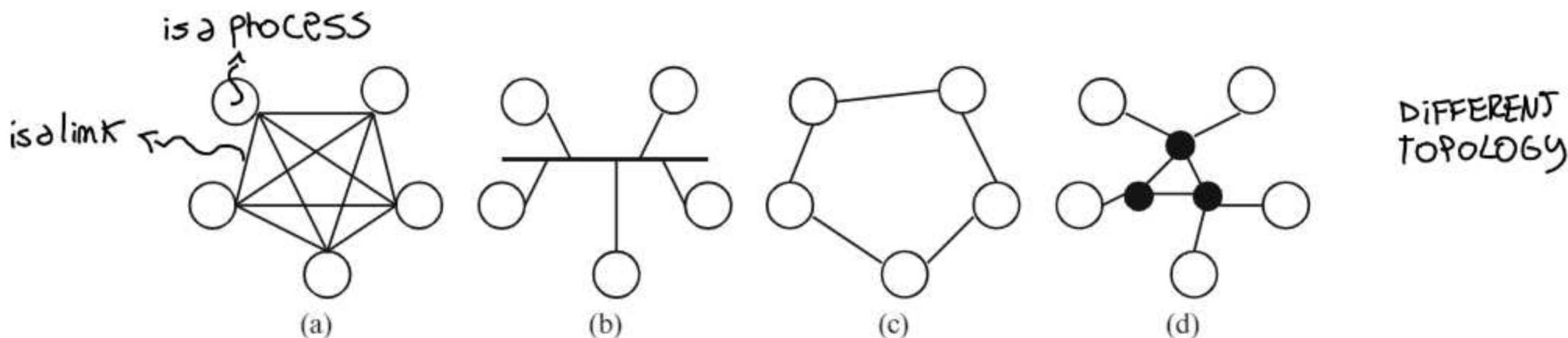
## Link Abstraction

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

→ not enter in detail of the specific technology used for communication between two nodes, reasoning in term of the property that communication link provide me.

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



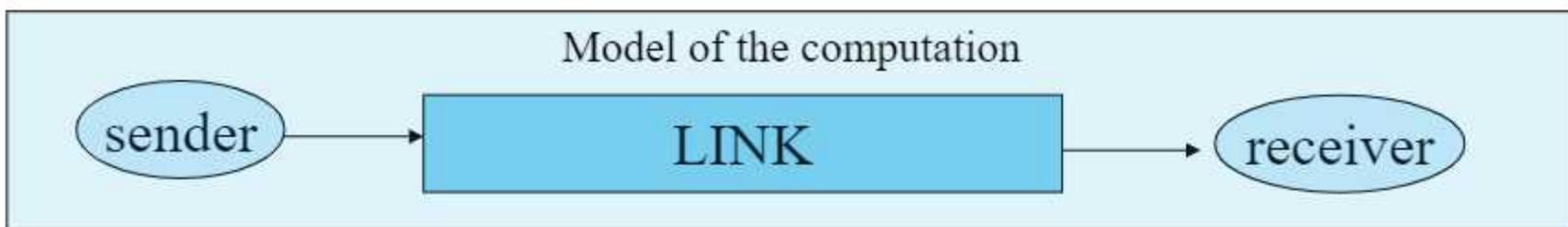
# Link Abstractions under crash failure assumption

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). even if the two endpoints are correct.  
Like UDP
  2. Stubborn links
  3. Perfect links
- use these two primitives to build up a communication link that is more reliable called perfect link that guarantees that if the endpoints are correct the message will be received.

# System Model

for implementing perfect link

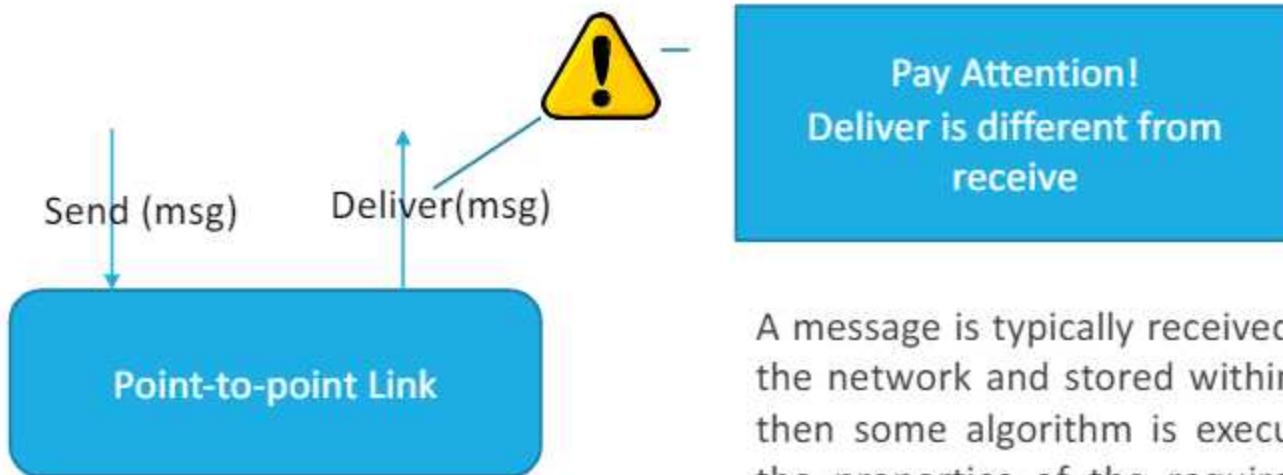
- Two processes (sender and receiver) endpoint of the link
- Messages interactions
  - can be lost
  - experience an unpredictable time to reach the destination (latency)
- Processes
  - can crash
  - The time taken by each process to execute an operation is bounded (such a bound can be unknown) Any assumption on computation time (asynchronous system)



# A generic link interface

*Deliver*: are giving message to application.  
*receive*: take message to communication medium.

*Deliver  $\neq$  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 → capture the behavior of UDP

---

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

---

Module:

Name: FairLossPointToPointLinks, instance *fl*.

Events:

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

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

Properties:

→ not lose systematically any message

**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*.

capture TCP  
because retransmitting  
a finite number of  
times

# 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.

*To avoid using:* **Stubborn Link**

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

*To avoid not necessary retransmitting:* **Quiescent Implementation**

Each message may be delivered more than once.

*To avoid duplicate:*

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

**Indication:**  $\langle sl, Deliver \mid p, m \rangle$ : 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 *fl*.

upon event  $\langle sl, Init \rangle$  do  
  *sent* :=  $\emptyset$ ; *keep looking at invocation of fair-loss*  
  starttimer( $\Delta$ );

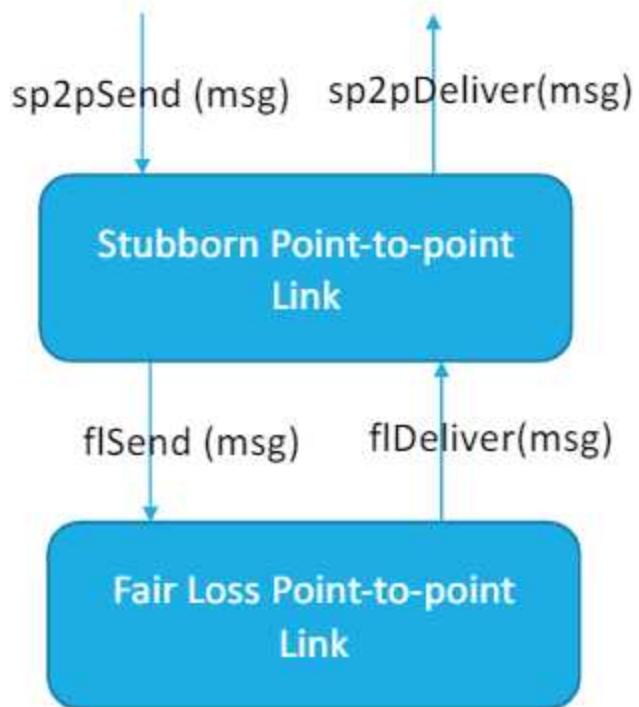
upon event  $\langle \text{Timeout} \rangle$  do  
  forall  $(q, m) \in \text{sent}$  do  
    trigger  $\langle fl, \text{Send} \mid q, m \rangle$ ;  
  starttimer( $\Delta$ );

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

upon event  $\langle fl, \text{Deliver} \mid p, m \rangle$  do  
  trigger  $\langle sl, \text{Deliver} \mid p, m \rangle$ ;

resent every  $\Delta$  messages  
sent to *fl*, infinite  
number of time eventually

will be triggered



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

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

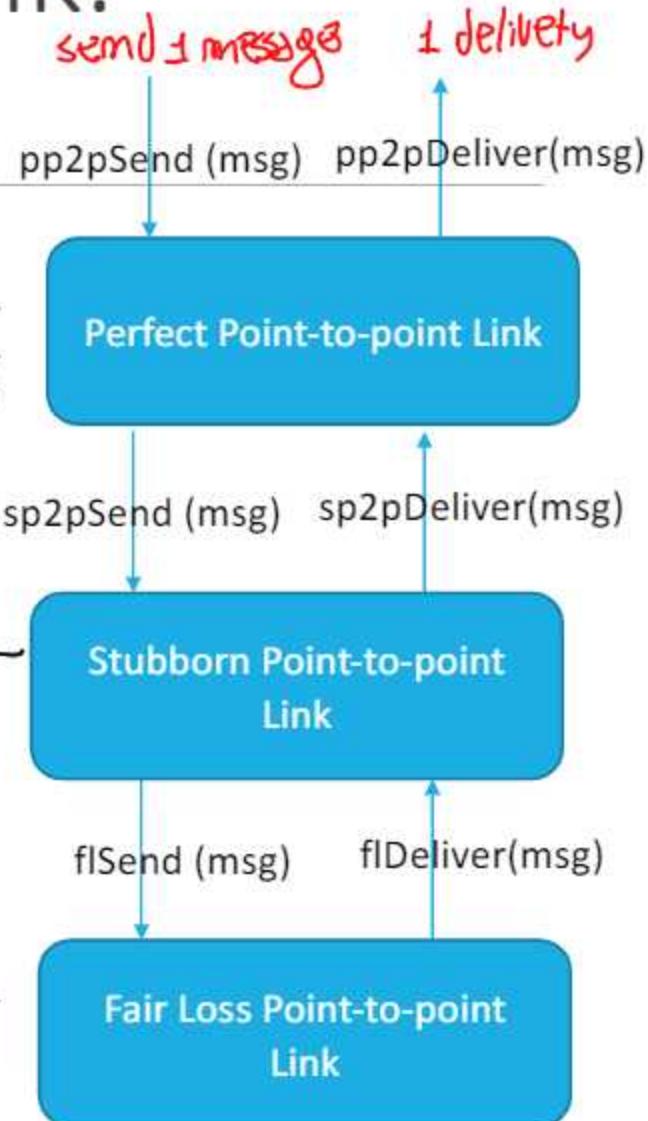
upon event  $\langle pl, \text{Init} \rangle$  do  
    *delivered* :=  $\emptyset$ ;

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

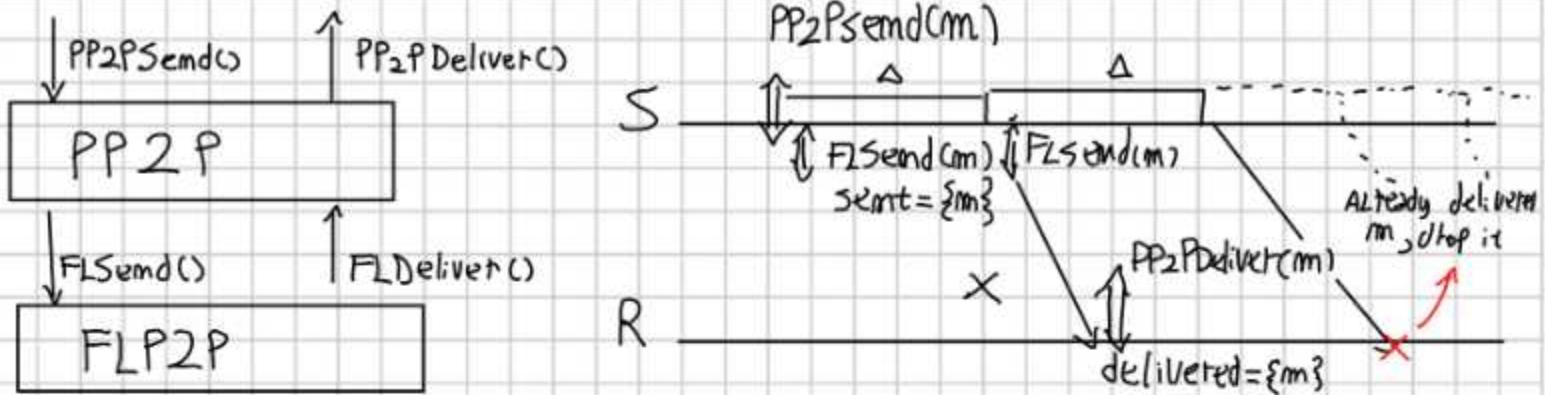
upon event  $\langle sl, \text{Deliver} \mid p, m \rangle$  do  
    if  $m \notin \text{delivered}$  then  
         $\text{delivered} := \text{delivered} \cup \{m\}$ ;  
        trigger  $\langle pl, \text{Deliver} \mid p, m \rangle$ ;

try to eliminate  
and do perfect  
directly on  
fLL!

check not duplication



perfect link directly on top on fast loss link



this implementation isn't quiescent use an infinite number of messages, sent over the failloss .

a solution not perfect.

INIT

sent<sub>i</sub> = empty  $\emptyset$

delivered; = empty  $\emptyset$

$$\text{time} = \Delta$$

Upon event  $PP_2PSend(m)$

$$\text{sent}_i^+ = \text{sent}_i^- \cup \{m\}$$

trigger: FLp2PSend(m) to p<sub>j</sub>

Open event FLP2PDeliver(m) from P<sub>j</sub>

if  $m \notin \text{deliveted}$ ;

delivered; = delivered; U {m}

trigger PP2P Deliver(m)

When  $t = 0$

for every  $m$  in  $Sent$ ;

trigger FLP2PSend(m) to P<sub>j</sub>

Problem of transmission  
every  $\Delta$  interval.

the perfect solution here:

INIT:

$sent_i = \emptyset$

$delivered_i = \emptyset$

$timet = \Delta$

$ACK_m = \text{false}$

Upon event  $PP2PSend(m)$

$sent_i = sent_i \cup \{m\}$

trigger  $FLP2PSend(m)$  to  $P_j$

Upon event  $FLP2PDeliver(ACK, m)$  from  $P_j$

$ACK_m = \text{true}$

Upon event  $FLP2PDeliver(m)$  from  $P_j$

trigger  $FLP2PDeliver(ack, m)$

if  $m \notin delivered_i$

$delivered_i = delivered_i \cup \{m\}$

trigger  $PP2PDeliver(m)$

when  $timet = 0$

for every  $m$  in  $sent_i$

if not  $ACK_m$

trigger  $FLP2PSend(m)$  to  $P_j$

$timet = \Delta$

# 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

# Dependable Distributed Systems

## Master of Science in Engineering in Computer Science

AA 2022/2023



---

### LECTURE 3: TIME IN DISTRIBUTED SYSTEMS

PHYSICAL CLOCKS AND CLOCK SYNCHRONIZATION

# Introduction

for collaborate and proceed  
in a common goal.

Many applications require **ordering between events** and **synchronization** to terminate correctly

- E.g. Air traffic control, Network monitoring, measurement and control, Stock market, buy and sell orders, etc...



# Why Time is so Important?

It is a quantity we are interested to measure

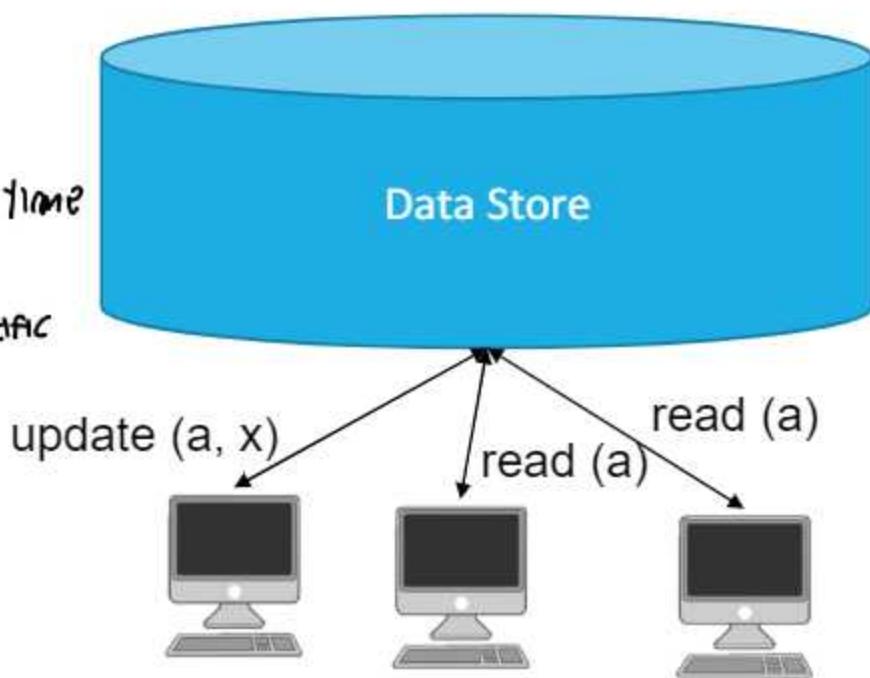
A lot of algorithms depends on time

- data consistency
- authentication
- double processing avoidance

↳ something already done, and do the same many time?

**data consistency:** need to understand which is the most updated version of a specific data, in order to resolve possible conflict (Ex. concurrent update).

**authentication:** we need to achieve authentication in a time window.

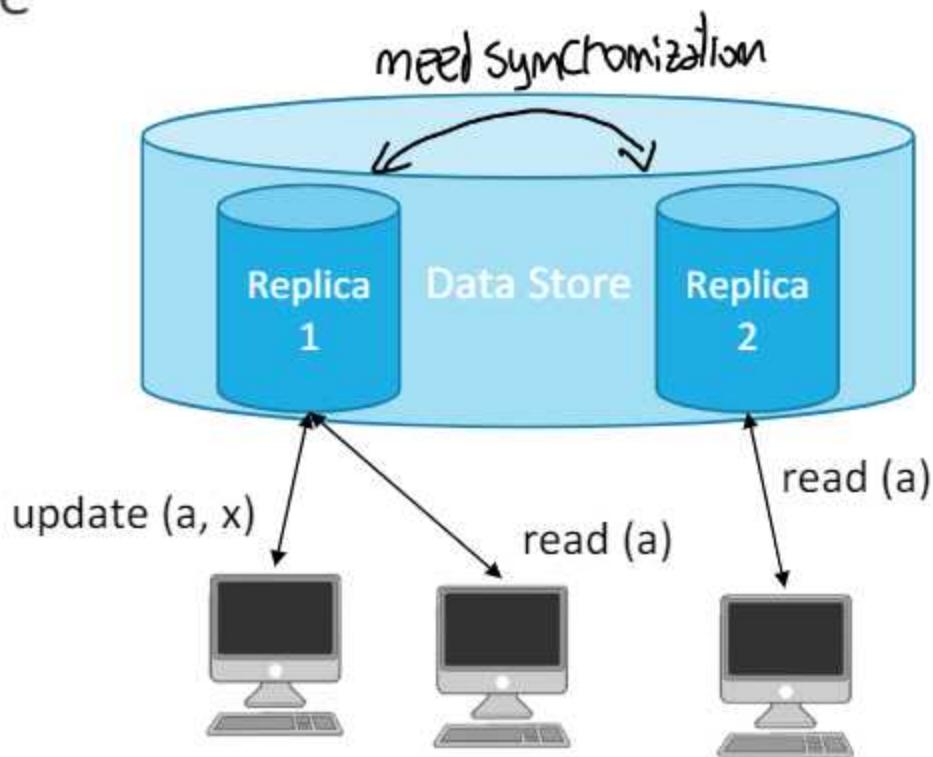


# Why Time is so Important?

It is a quantity we are interested to measure

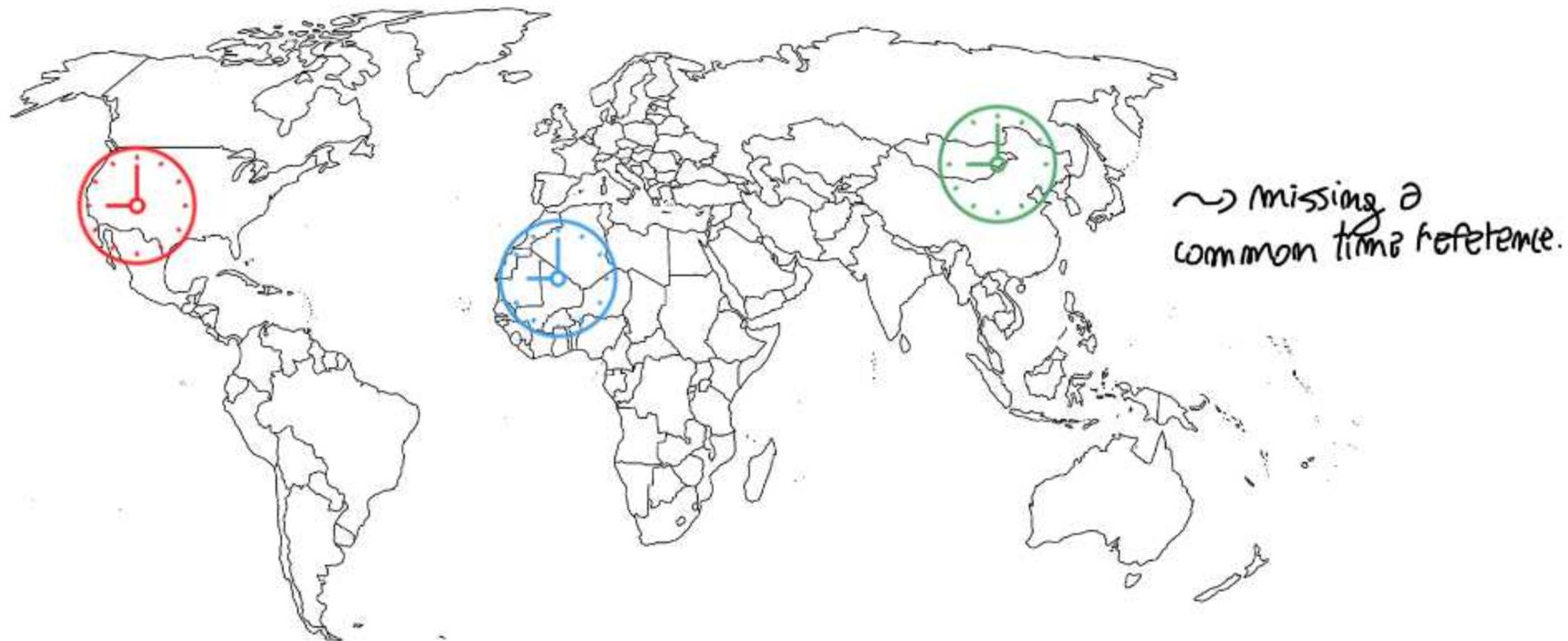
A lot of algorithms depends on time

- data consistency
- authentication
- double processing avoidance



# Why using time in a DS is difficult?

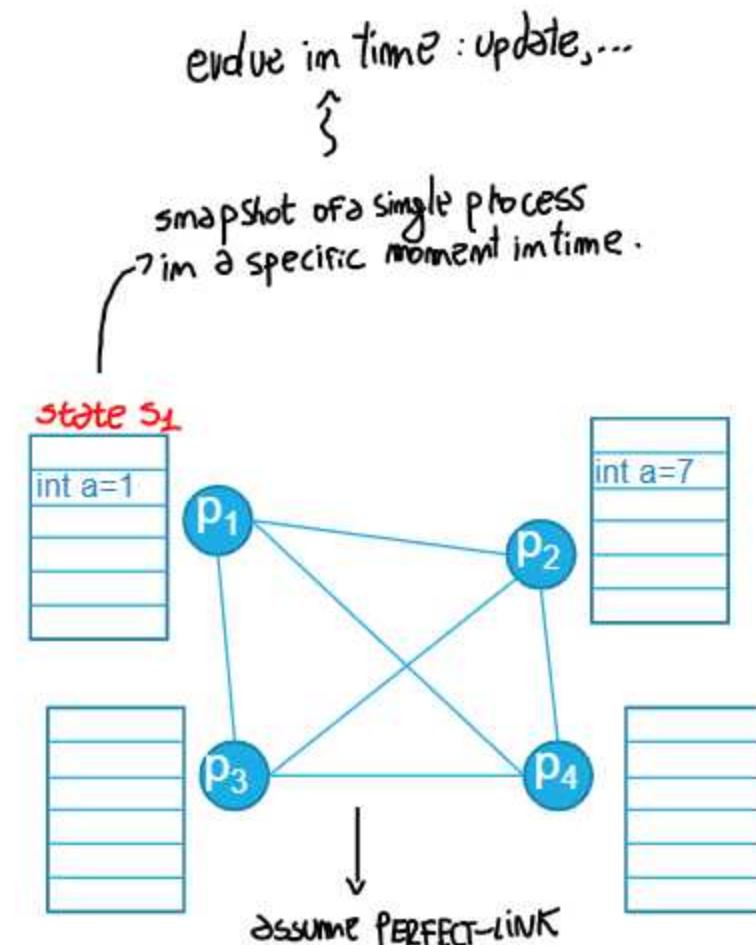
in distributed system, not all part of the system are in same place.



We need to agree on a **common time reference**

# System Model

- A distributed system is composed by a set  $\Pi = \{p_1, p_2, \dots, p_n\}$  of  $n$  processes
- Each process  $p_i$  has a state  $s_i$  that is changed by the actions it takes during the algorithm execution
  - The state  $s_i$  includes all the values of variables maintained by  $p_i$
- Each process can communicate with other processes only by exchanging messages



# Computation Model

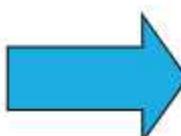
- on top of processes there are some events that occur (receive a message, complete an operation, ...).
- Everything is important in a specific process to understand the evolution of a particular computation, usually is model by an event.

## Each process generates a sequence of events

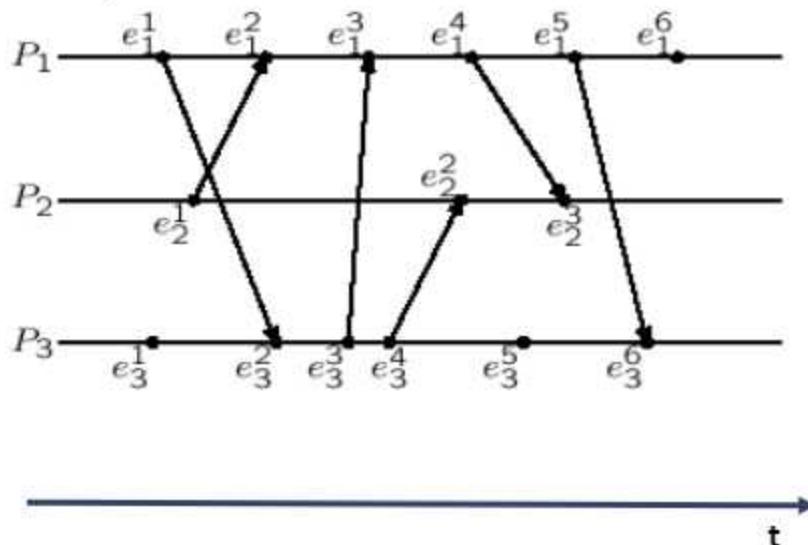
- Internal event (event that transforms the process state)
- external event (send/receive) ↗ related with messages exchange
- $e_i^k$ , k-th event generated by  $P_i$

↗ (locally inside the process)

The evolution of the computation  
can be represented with a  
space-time diagram.



usually you don't have the entire evolution of the system, only partial knowledge



# Execution History of Computations

## Local History

Sequence of events produced by a process

$$\text{history}(p_1) = h_1 = \langle e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_1^6 \rangle$$

at least usually this is a notion that you have.

most of time is missing the interrelation that exist between those processes.

## Partial Local History

Prefix of local history

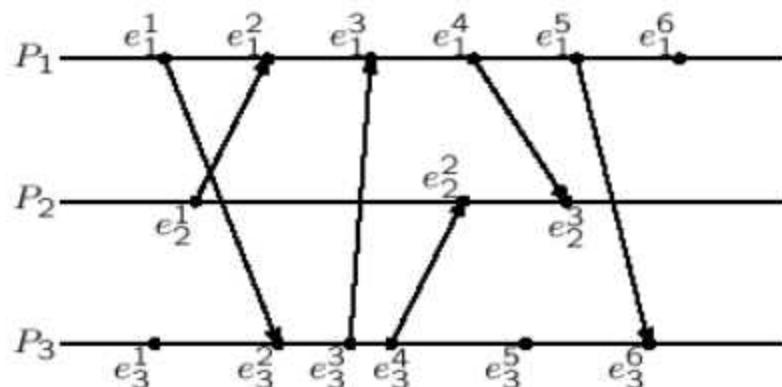
$$h_1^m = e_1^1 \dots e_1^m$$

## Global History

Set containing every local history

$$H = \bigcup_i h_i \text{ per } 1 \leq i \leq n$$

how all events are occurred and how are spaced over the time.



# Time Abstraction in Distributed Systems, How?

## Timestamping

- Each process attaches a label to each event (using a timestamp).  
\_> it should be possible to realize a global history of the system.

## “Naïf” solution

- Each process timestamps events by mean of its physical clock

 for other events, in a single process is easy, among all the process is very hard, because in D.S. most of the time have network delay, process delay,...

# Time Abstraction in Distributed Systems, How?

- Does **timestaming** allows to realize a obtain the **global execution history**?

- It is always possible to define an order among events produced by the same process
- But what's happen when we consider several distinct processes running on different PCs?

You don't have a common shared knowledge of a clock, only a distribute clock, there isn't a single clock accessible by all processes, each process have a clock.

1

In a distributed system in presence of *network delay, processing delay, etc...*

**It is impossible to realize a common clock shared among every process.**

# Time Abstraction in Distributed Systems, How?

Using timestamps it is possible to synchronize physical clocks (with a certain degree of approximation), through appropriate algorithms.

*there are some delay!*

\_> A process can label events using its physical local clock (synchronized with a certain *synchronization accuracy* or *synchronization error*).

# Physical and Software Clocks

Application processes access a **local clock** obtained by operating system reading a **local hardware clock**.

**Hardware clocks** consist of an **oscillator** (quartz crystal, electrical oscillations) **and a counting register** that is incremented at every tick of the oscillator.

assuming that exist a global knowledge of time.

At real time  $t$ , the operating system reads the **hardware clock  $H_i(t)$** , therefore it produces the **software clock  $C_i(t)$**  {os apply some correction:  $\alpha$  multiply oscillation to get the actual clock,  $\beta$  for shift forward or back in time.}

$$C_i(t) = \alpha H_i(t) + \beta$$

# Physical Computer Clocks

$C_i(t)$  approximates the physical time  $t$  at process  $p_i$

Example:  $C_i(t)$  may be implemented by a 64-bit word, representing nanoseconds that have elapsed at time  $t$ .

Generally this clock is not completely accurate

- it can be different from the real time  $t$
- It can be different at any process due to the precision of the approximation

$C_i$  can be used such as timestamp for event produced by  $p_i$ .

How much should be smaller the granularity (resolution) of software clocks (the time interval between two consecutive increments of software clock) to distinguish between two different events?

$T_{\text{resolution}} < \Delta T \text{ between two notable events}$

# Parameters affecting the Clock Synchronization accuracy

hardware clock deviate because depend from oscillation and those oscillation are influenced by temperature. At some interval of time  $t_i$  can be faster or slower.

For say that a specific physical clock is correct, we set a bounded for the deviation

Different local clocks can have different values:

- **Skew**: “the difference in time between two clocks”

$$\text{Skew}_{i,j}(t) = |C_i(t) - C_j(t)|$$

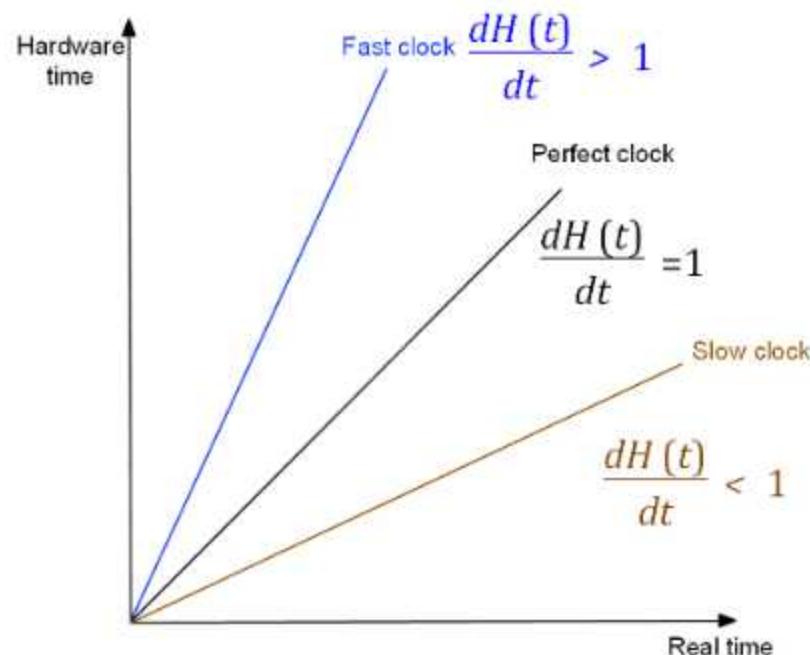
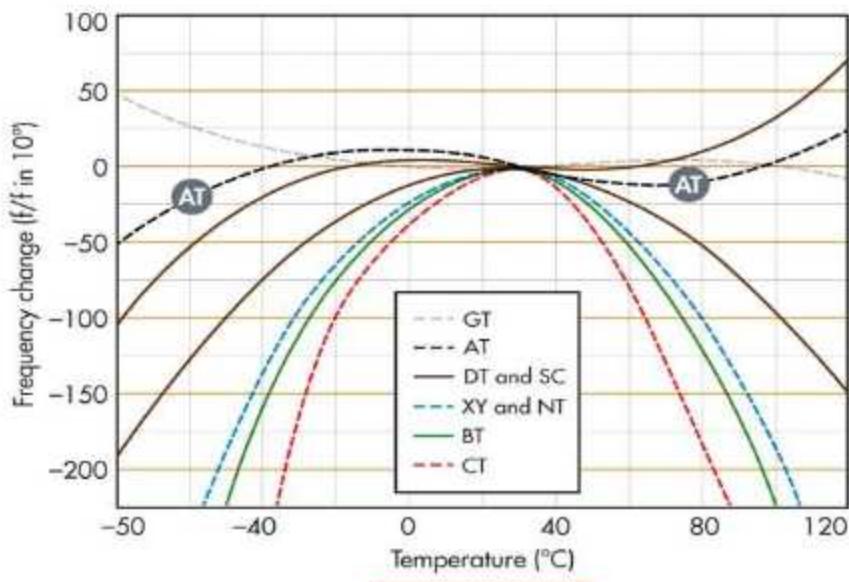
- **Drift Rate**: “the gradual misalignment of synchronized clocks caused by the slight inaccuracies of the time-keeping mechanisms

e.g. drift rate of 2 microsec/sec means clock increases its value of 1 sec+2 microsec for each second.

- Ordinary quartz clocks deviate nearly by 1 sec in 11-12 days. ( $10^{-6}$  secs/sec).
- High-precision quartz clock drift rate is  $10^{-7}$ - $10^{-8}$  secs/sec

# Parameters affecting the Clock Synchronization accuracy: Drift Rate

$$\frac{dH(t)}{dt}$$



## Correct Clock

An hardware clock  $H$  is **correct** if its drift rate is within a limited bound of  $\rho > 0$  (e.g.  $10^{-6}$  secs/ sec).

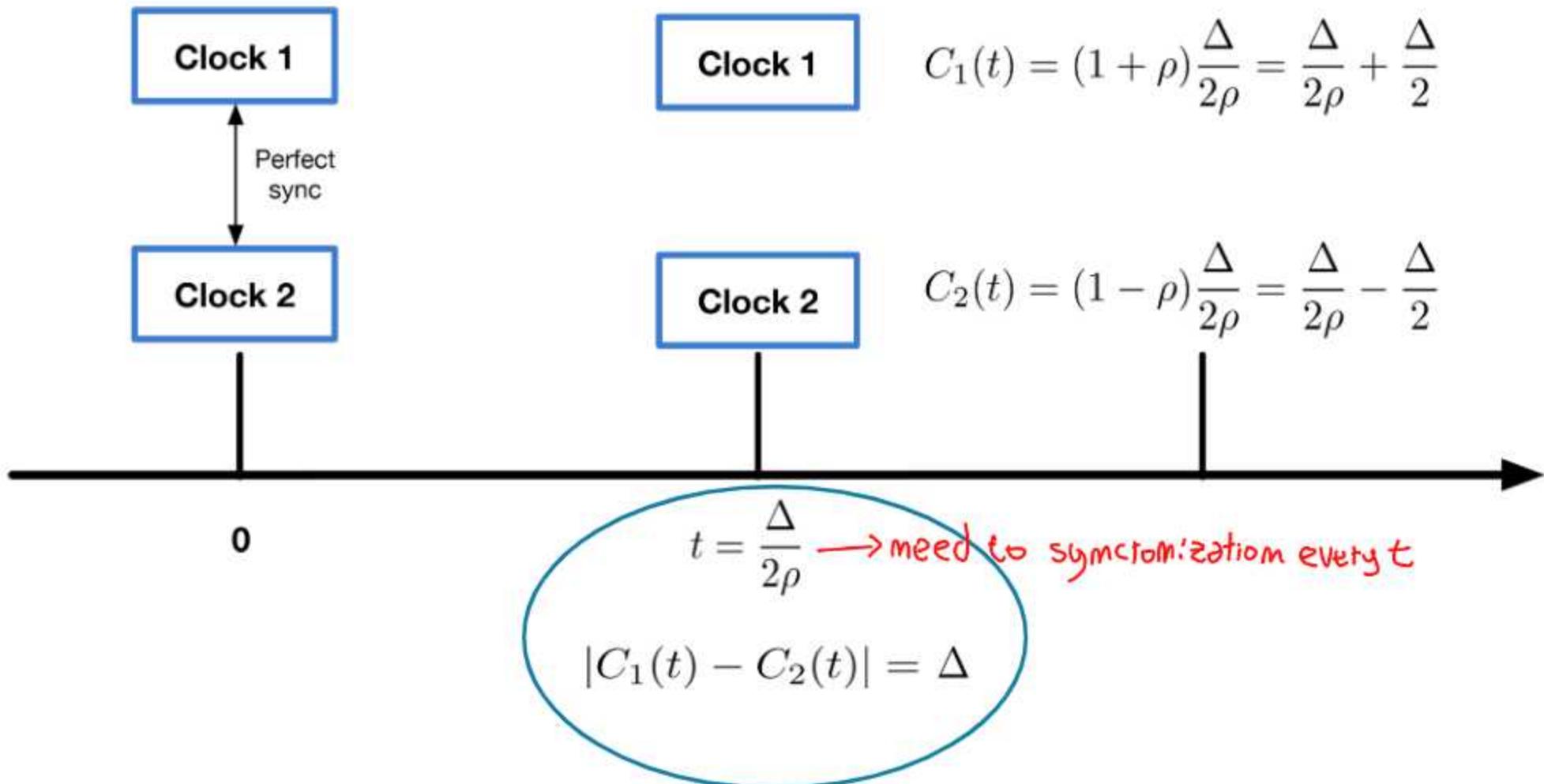
$$1 - \rho \leq \frac{dH(t)}{dt} \leq 1 + \rho$$

↳ deviation is bounded by  $\rho$

In presence of a correct hardware clock  $H$  we can measure a time interval  $[t, t']$  (for all  $t' > t$ ) introducing only limited errors.

$$(1 - \rho)(t_1 - t_0) \leq H(t_1) - H(t_0) \leq (1 + \rho)(t_1 - t_0)$$

# Bounding the Skew



# Bounding the Skew

$$C_1(t) = \frac{\Delta}{2\rho} + \frac{\Delta}{2} - \Delta$$

↑  
**Correction**

$$C_1(t) = \frac{\Delta}{2\rho} + \frac{\Delta}{2}$$

not return in a past time, because it will be errors  
between time events, most slower  
software process clock

$$C_2(t) = \frac{\Delta}{2\rho} - \frac{\Delta}{2}$$

Clock 1

Clock 2

# Monotonicity

Software clocks must be **monotone**

$$t' > t \rightarrow C(t') > C(t)$$

The monotonic property can be guaranteed choosing opportune values for  $\alpha$  and  $\beta$  (Note that  $\alpha$  and  $\beta$  can be a function of time).

How to apply negative correction? **Slowing down the software clock**

# Monotonicity

**It is not possible to impose a clock value in past.**

\_ > This action can violate the cause/effect ordering of the events produced by a process and the time monotonicity.

**We slow down clocks hiding interrupts.**

Hiding interrupts, the local clock is not updated so that we have to hide a number of interrupt equals to slowdown time divided by the interrupt period.

# Universal Time Coordinated (UTC)

for adjust time, periodically device connect to a specific server and set time in case differ from real time.

**UTC is an international standard: the base of any international measure.**

**Based on International Atomic Time:** 1 sec = time a caesium atom needs for 9192631770 state transitions.

- Physical clocks based on atomic oscillators are the most accurate clocks (drift rate 10-13)

**UTC-signals** come from shortwave radio broadcasting stations or from satellites (GPS) with an accuracy of

- 1 msec for broadcasting stations
- 1  $\mu$ sec for GPS

**UTC-signals receivers** can be connected to computers and can be used to synchronize local clocks with the real time

# Internal/External Synchronization

## External Synchronization

there is a common reference (UTC), we have to attempt to get closer to that time. There is an UTC server, a bound D

- Processes synchronize their clock  $C_i$  with an authoritative external source S (UTC)
- Let  $D > 0$  (accuracy) be the synchronization bound
- Clocks  $C_i$  (for  $i = 1, 2, \dots, N$ ) are externally synchronized with a time source S (UTC) if for each time interval I:

$$|S(t) - C_i(t)| < \overbrace{D}^{\text{bound}} \text{ for } i = 1, 2, \dots, N \text{ and for real time } t \text{ in } I$$

correct real time      software clock

We say that clocks  $C_i$  are accurate within the bound of D

# Internal/External Synchronization

## Internal Synchronization *not an external to time?*

- All the processes synchronize their clocks  $C_i$  between them
- Let  $D > 0$  (precision) be the synchronization bound and let  $C_i$  and  $C_j$  the clocks at processes  $p_i$  and  $p_j$  respectively
- Clocks are internally synchronized in a time interval  $I$ :

*for each possible pair*  $|C_i(t) - C_j(t)| < D$  for  $i, j = 1, 2, \dots, N$  and for all time  $t$  in  $I$

We say that clocks  $C_i, C_j$  agree within the bound of  $D$

# Physical Clock Synchronization

Notes:

Exam question

not achieve perfect synchronization because there are delay. You can bound drift, periodically synchronize.

\_> **Clocks that are internally synchronized are not necessarily externally synchronized.** i.e. even though they agree with each other, they drift collectively from the external time source.

\_> **A set of processes  $P$ , externally synchronized within the bound of  $D$ , is also internally synchronized within the bound of  $2D$ .**

- This property directly follows from the definition of internal and external clock synchronization.

# Synchronization Algorithms

# Synchronization by mean of a Time Server

how to periodically synchronize?

## Centralized Time Service

- Request-driven : make a request to a server and get a reply
  - Christian's Algorithm → for external synchronization, with real correct time.  
(synchronization with a probability  $p$ , there are unpredictable delay)
- Broadcast-based
  - Berkeley Unix algorithm - Gusella & Zatti (1989)

## Distributed Time Service (Network Time Protocol)

Use Christian's algorithm

# Christian's Algorithm

External synchronization algorithm

---

Use a time server S that receives a signal from an UTC source

---

Works (probabilistically) also in an asynchronous system

- Is based on message round trip time (RTT)
- Synchronization is reached only if RTTs are small with respect to the required accuracy

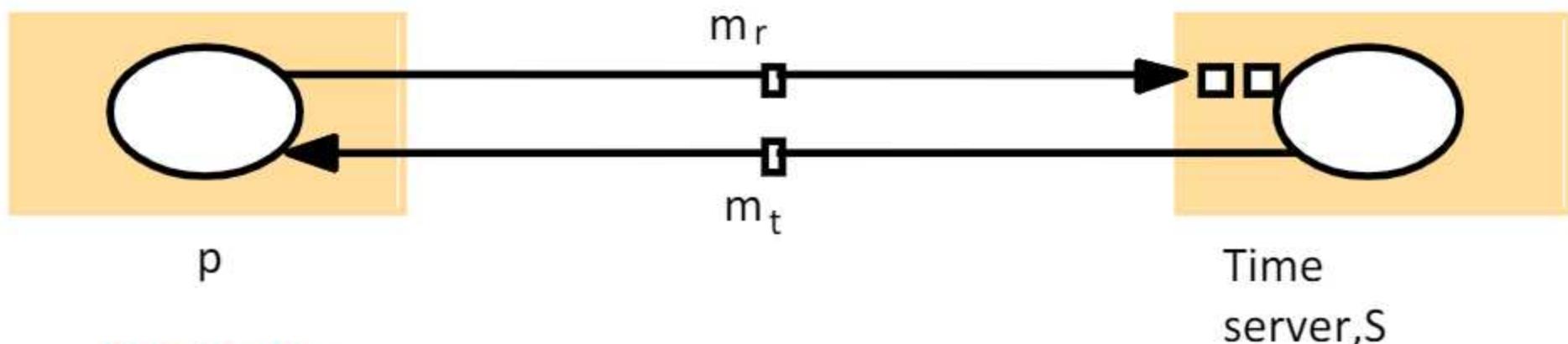
RTT: time that take a message to arrive from a point to another point and return.

# Christian's Algorithm

in general  $T_{\text{round}}/2$  because the channel is symmetric with delays. Same amount of time from p to S and from S to p.

\_> A process  $p$  asks the current time through a message  $m_r$  and receives  $t$  in  $m_t$  from S

\_>  $p$  sets its clock to  $t + T_{\text{round}}/2$ ,  $T_{\text{round}}$  is round trip time experienced by  $p$



Notes: **Weakness**

- A time server can crash
- Cristian suggests to use a cluster of synchronized time servers for overcome failure.
- A time server can be attacked... , believe in false time

# Christian's Algorithm Accuracy

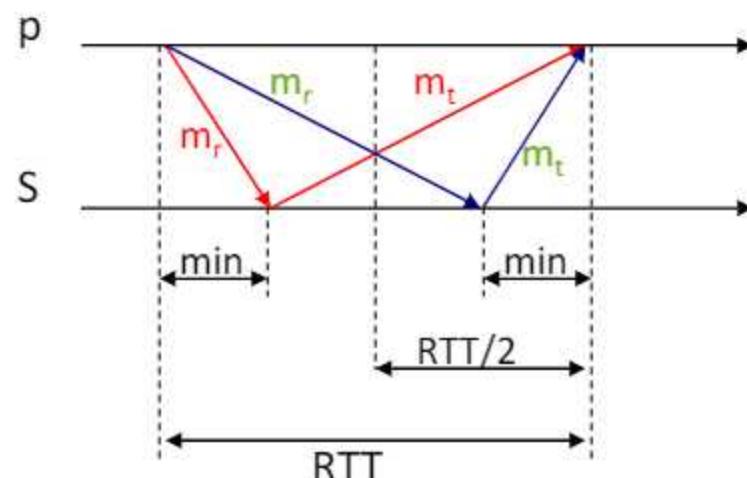
for improve accuracy, measure RTT but also minimum amount of time for send a message.

## Case 1

Reply time is greater than estimate one (obtained by RTT/2), in particular is equal to (RTT-min)

$$\Delta = \text{estimate of response} - \text{real time} = (\text{RTT}/2) - (\text{RTT} - \text{min}) =$$

$$(-\text{RTT} + 2\text{min})/2 = -\text{RTT}/2 + \text{min} = \underline{-\text{RTT}/2 + \text{min}}$$



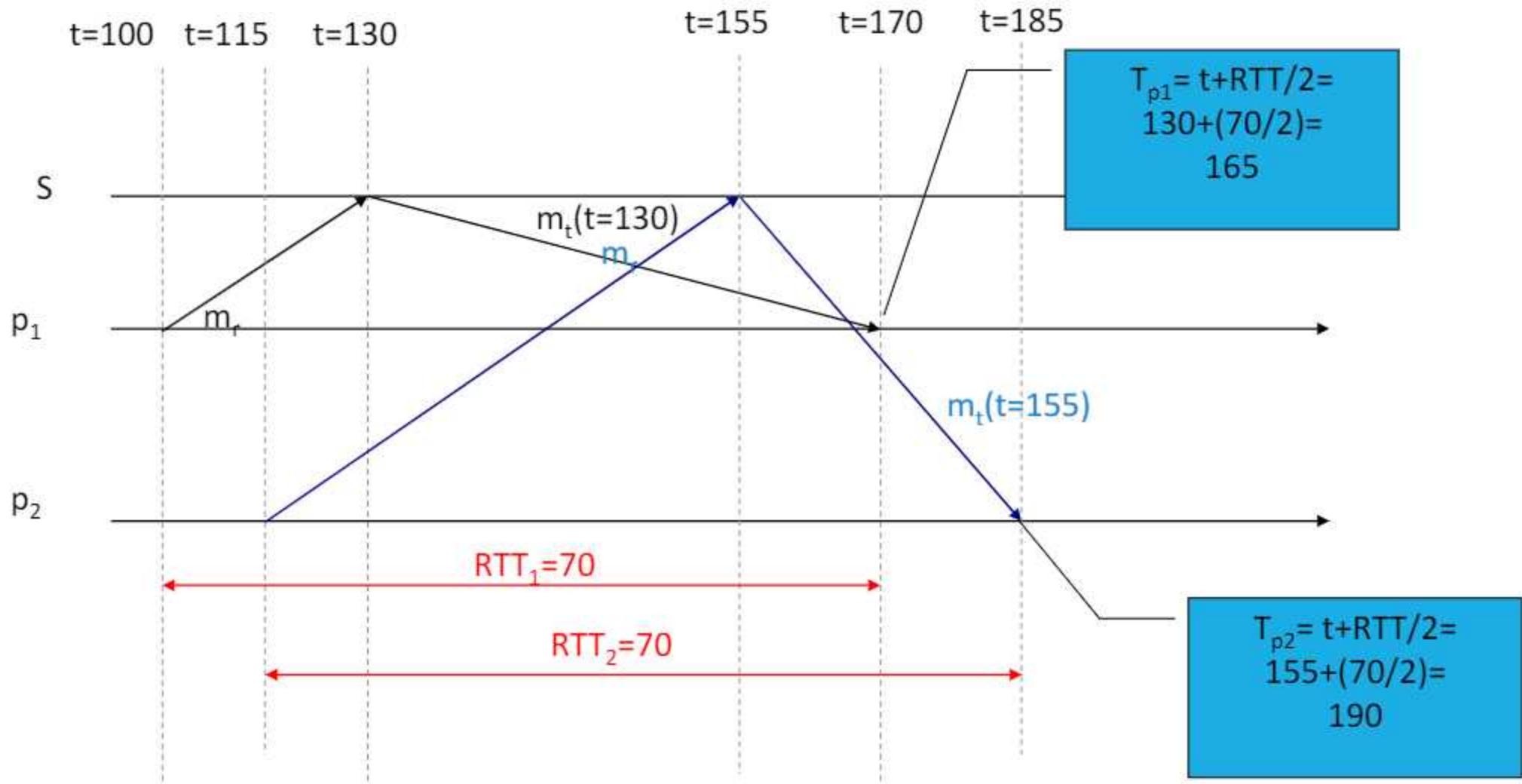
## Case 2

Reply time is smaller than estimate one (obtained by RTT/2), in particular is equal to (RTT-min)

$$\Delta = \text{estimate of response} - \text{real time} = (\text{RTT}/2) - \text{min} = \underline{+ (\text{RTT}/2 - \text{min})}$$

Consequently the accuracy of Cristian's algorithm is  $\pm (\text{RTT}/2 - \text{min})$  where  $\text{min}$  is the minimum transmission delay

# Christian's algorithm example



# Christian's algorithm example

In the previous scenario

- If the minimum message transmission time is  $t_{min} = 30$  then the accuracy is  $\pm 5$  (i.e.  $\pm RTT/2 - t_{min} = 70/2 - 30 = \pm 5$ )
- If the minimum message transmission time is  $t_{min} = 20$  then the accuracy is  $\pm 15$  (i.e.  $\pm RTT/2 - t_{min} = 70/2 - 20 = \pm 15$ )

# Discussion

The synchronization server is a single point of failure

- There could exist periods in which the synchronization is not possible  
    → Ask to multiple servers at the same time (synchronization group)

Servers in the group may be arbitrarily faulty or malicious

- Add redundancy
- Use authentication

# Berkeley's Algorithm

## Internal synchronization algorithm

(no external server, but only processes)

- master-slave structure  $\rightsquigarrow$  a single process that lead the protocol
- Based on steps
  - gathering of all the clocks from other processes and computation of the difference
  - computation of the correction

## Berkeley: Measuring the difference between clocks

The master process  $p_m$  sends a message with a timestamp  $t_1$  (local clock value) to each process of the system ( $p_m$  included)

When a process  $p_i$  receives a message from the master, it sends back a reply with its timestamp  $t_2$  (local clock value)

When the master receives the reply message it reads the local clock ( $t_3$ ) and compute the difference between the clocks  $\Delta = (t_1 + t_3)/2 - t_2$

# Berkeley: Synchronization Algorithm

## Master behaviour

- Computes of the differences  $\Delta p_i$  between the master clock and the clock of every other process  $p_i$  (including also the master)
- Computes the average avg of all  $\Delta p_i$  without considering faulty<sup>1</sup> processes
- Computes the correction of each process (including faulty processes)

$$Adg_{p_i} = avg - \Delta p_i$$

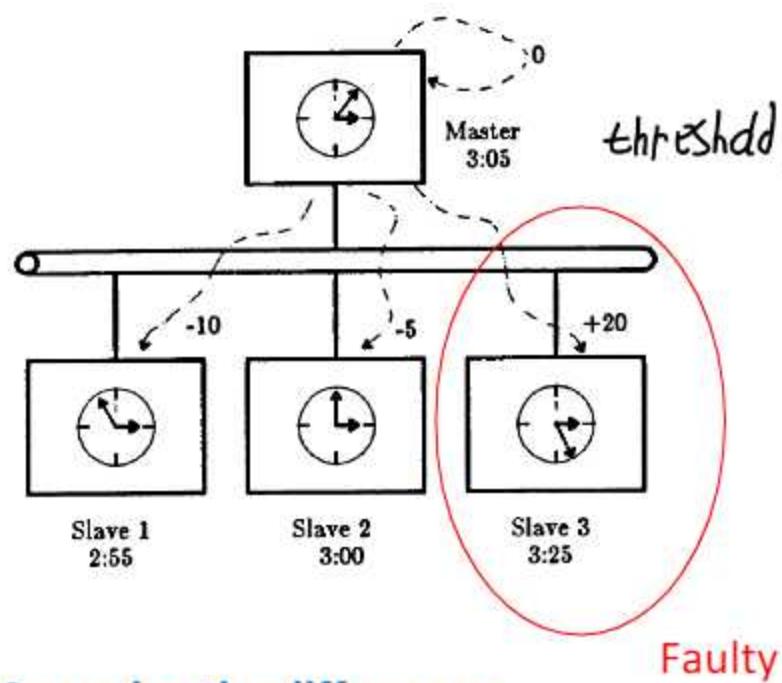
tolerant to some failures,  
only below the y

## Slaves behaviour

- When a process receives the correction, it is applied to the local clock
- If the correction is a negative one, the process do not adjust the value but it slow down its clock

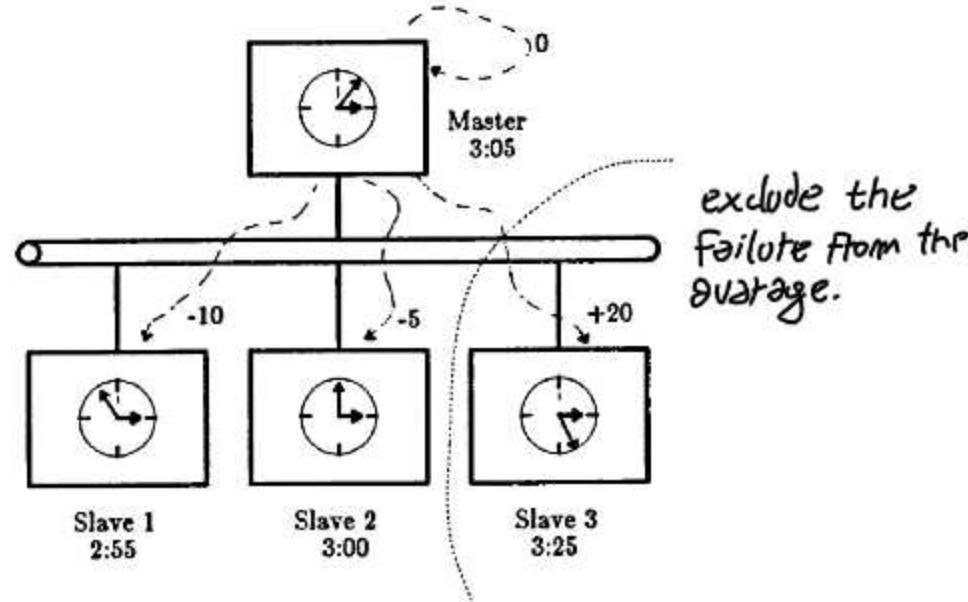
1. A faulty process is a process that has a clock which differ from the one of the master more than a given threshold y

# Berkeley: Example



## 1. Measuring the differences

- $\Delta p_m = 3:05 - 3:05 = 0$
- $\Delta p_1 = 3:05 - 2:55 = -10$
- $\Delta p_2 = 3:05 - 3:00 = -5$
- $\Delta p_3 = 3:05 - 3:25 = 20$  not include faulty!

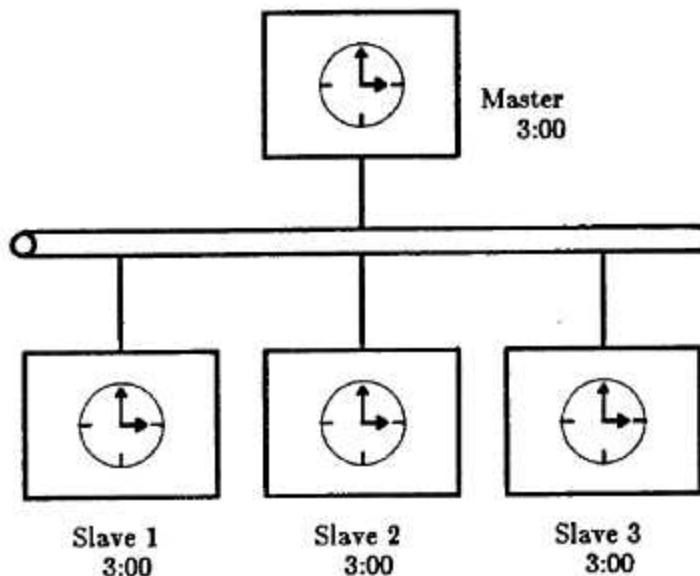
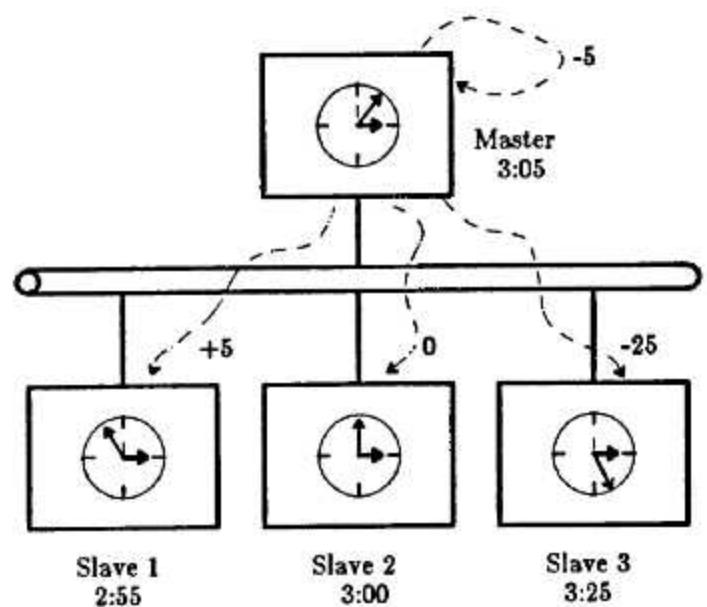


## 2. Computing the average

- $\text{Avg} = (0 - 10 - 5) / 3 = -5$

# Berkeley: Example

*Synchronized!*



## 3. Compute and send the correction

- $Adj_m = \text{Avg} - \Delta p_m = -5 - 0 = -5$
- $Adj_1 = \text{Avg} - \Delta p_1 = -5 - (-10) = 5$
- $Adj_2 = \text{Avg} - \Delta p_2 = -5 - (-5) = 0$
- $Adj_3 = \text{Avg} - \Delta p_3 = -5 - 20 = -25$

## 4. Apply the correction

# Berkeley's algorithm: accuracy

The protocol accuracy depends on the maximum round-trip time

- The master does not consider clock values associated to RTT greater than the maximum one

Fault tolerance:

- If the master crashes, another master is elected (in an unknown time)
- It is tolerant to arbitrary behaviour (e.g. slaves that send wrong values)
  - Master process consider a certain number of clock values and these values do not differ between them over a certain threshold

# Berkeley Algorithm: Slowing Down

- **Observation:** what does slowing down a clock mean?
- It is not possible to impose a clock value in past to slaves that have a clock value greater than the new computed mean.
  - This action can violate the cause/effect ordering of the events produced by the slave and the time monotonicity.
- Consequently we **slow down clocks hiding interrupts.**
  - Hiding interrupts, the local clock is not updated so that we have to hide a number of interrupt equals to slowdown time divides the interrupt period.

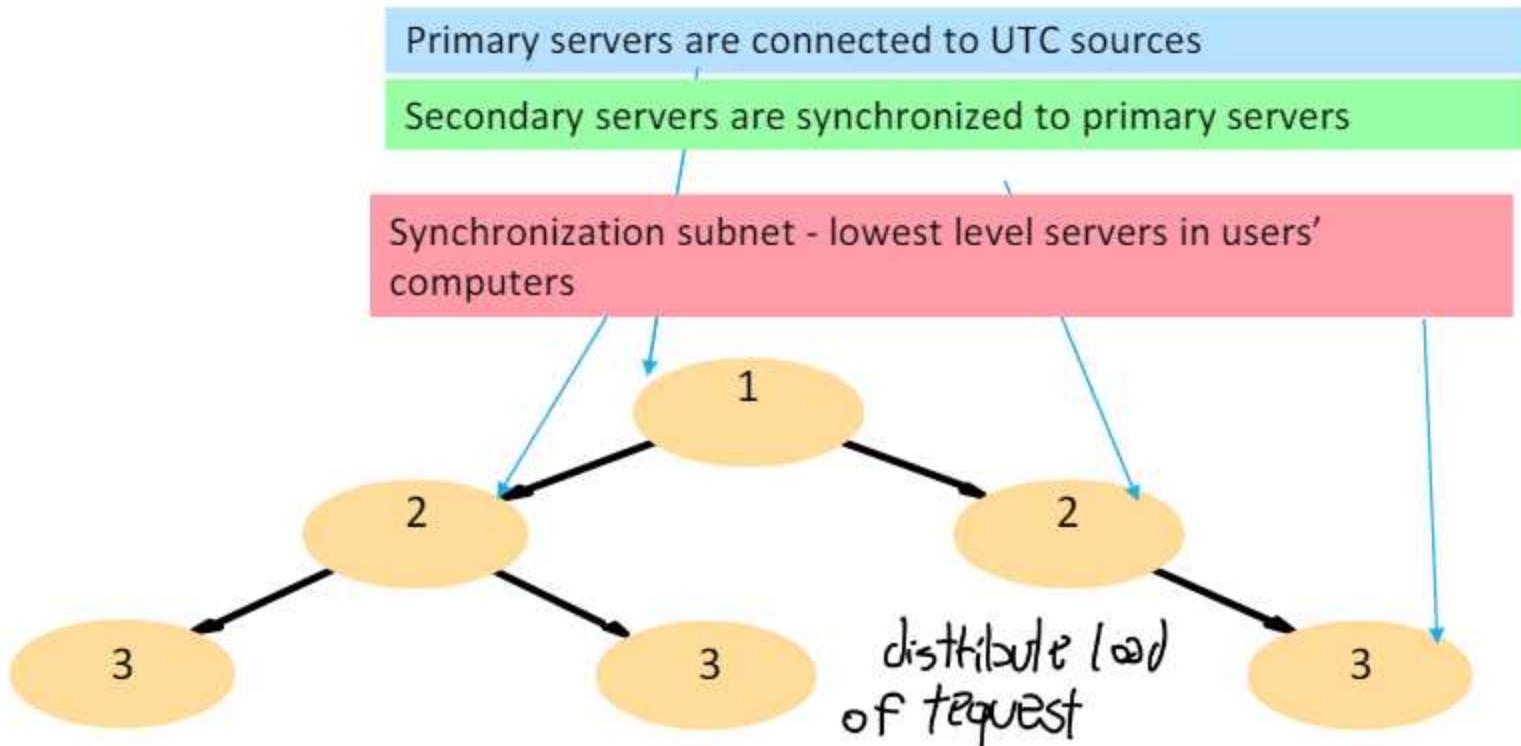
# Network Time Protocol (NTP)

: one generally used in machine, based on internet, there are some UTC servers organized in 3 level hierarchy

Time service over **Internet** - synchronizes clients with UTC:

Reliability by mean of redundant server and path

Scalable



# Network Time Protocol (NTP)

Synchronization of clients relative to UTC on an Internet-wide scale

- NTP is a standard de facto for external clock synchronization of distributed system on Internet
- NTP employs several security mechanisms (e.g. mechanisms for authentication of time references) usually they are not required in a local area network

Based on a remote reading procedure like **Cristian's algorithm**

- NTP specification adds to the basic algorithm mechanisms for clustering, filtering and evaluating data quality in order to minimize the synchronization

# Network Time Protocol (NTP)

**The NTP hierarchy is reconfigurable in presence of faults**

- Primary server that loses its connection with UTC-signal can become a secondary server
- Secondary server that loses its connection with a primary server (e.g. a crash of the primary server) can contact and connect itself to another primary.

# NTP Synchronization Modes

**Multicast**: server periodically sends its actual time to its leaves in the LAN. Leaves set their time using the received time assuming a certain delay. It is used in quick LANs but it shows a low accuracy

**Procedure call**: server replies to requests with its actual timestamp (like Cristian's algorithm). High Accuracy and it is useful when it is not available hw multicast.

**Symmetrical**: used to synchronize between pairs of time servers using messages containing timing information. Only used in high level of hierarchy.

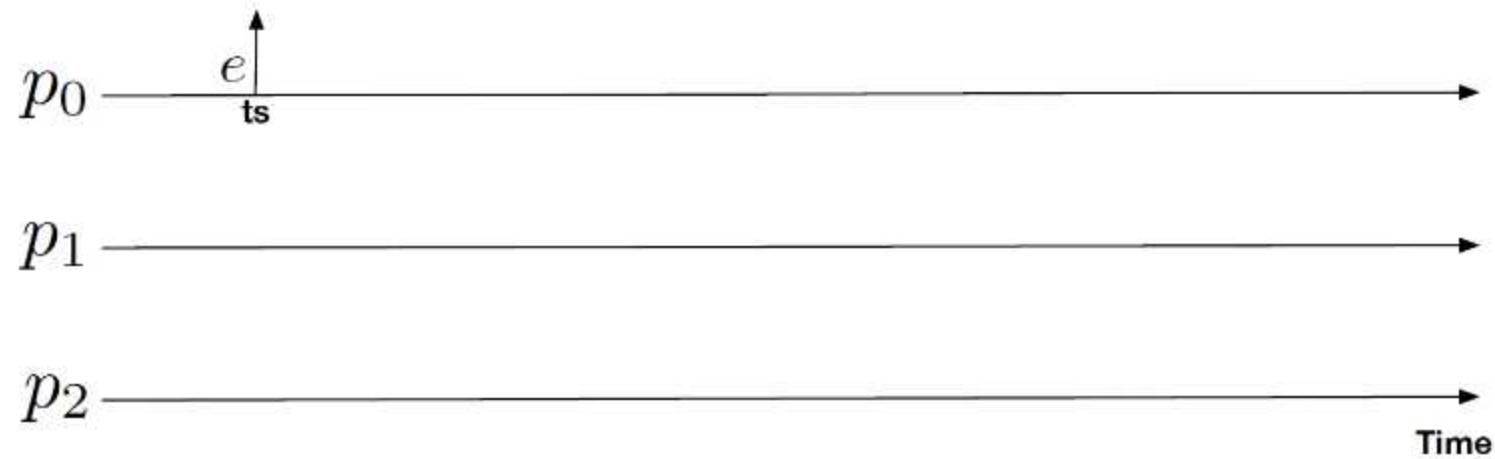
# Time in Asynchronous Systems

Physical Time: A global property...  
Observable? ?

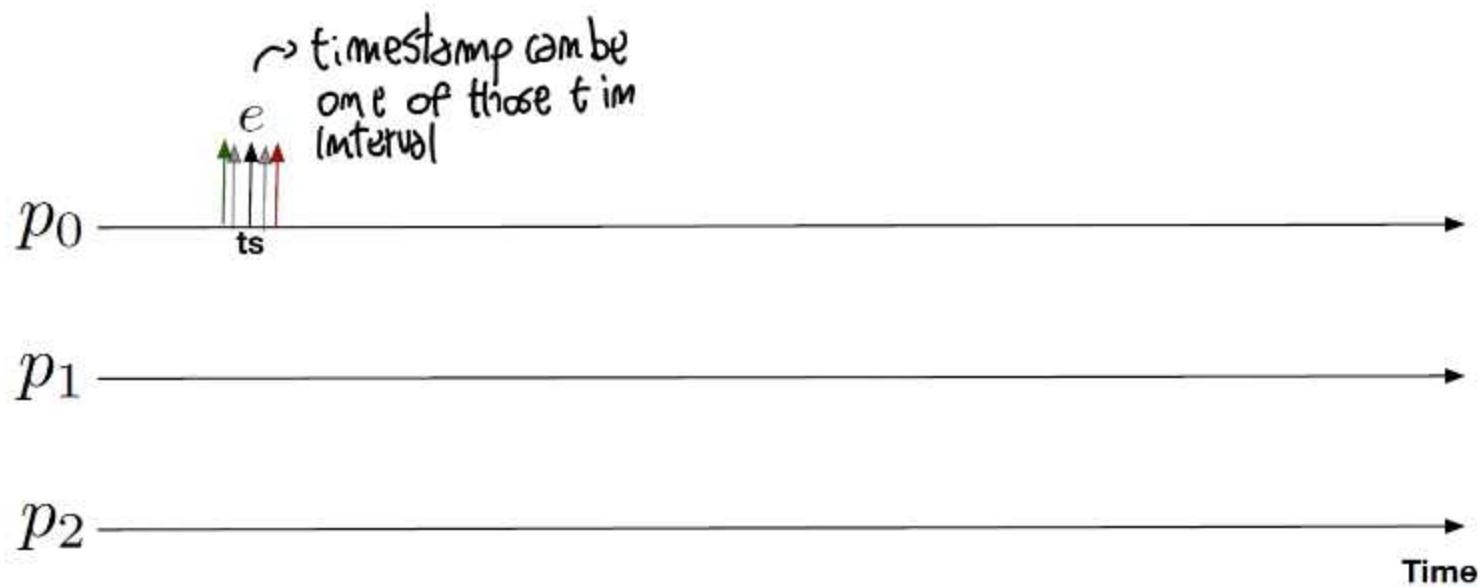
NO in a distributed asynchronous system: different clocks are synchronized only with a certain probability

- The impossibility of perfect accuracy is due to unpredictability of communication delay.
  - We can introduce a bound for the accuracy only when we known the upper and lower bounds for communication delays.

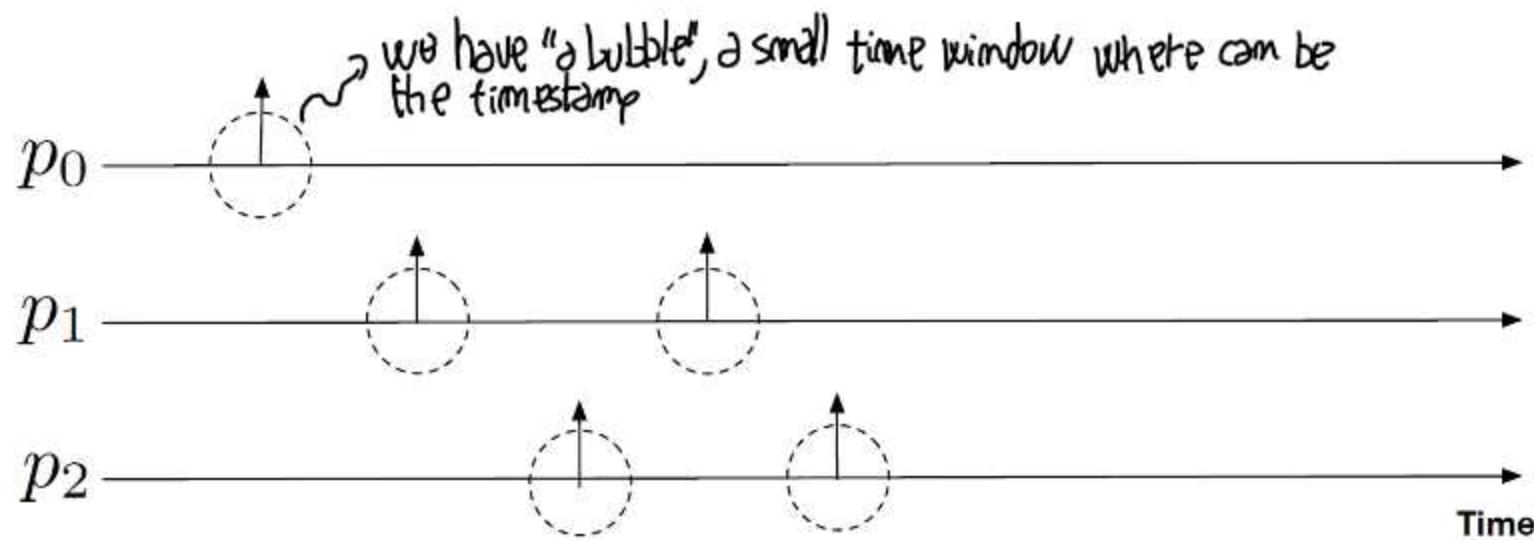
# Ordering with Timestamps



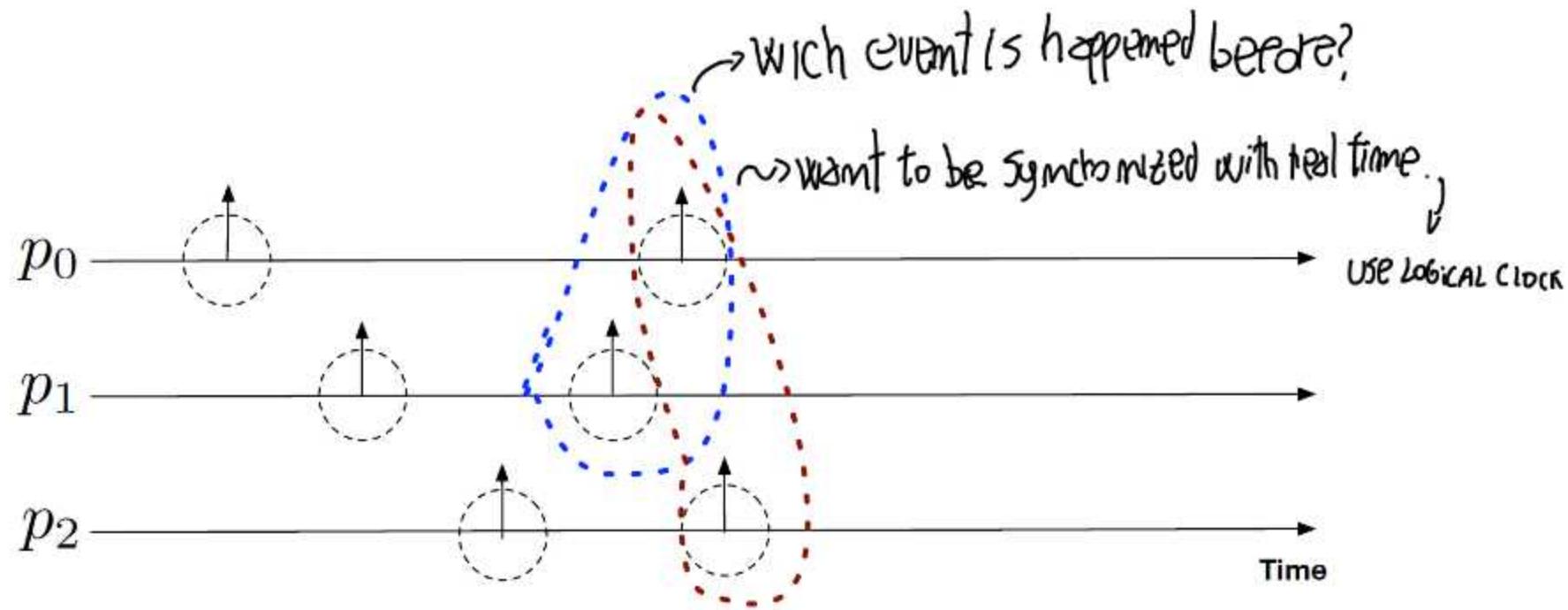
# Ordering with Timestamps



# Ordering with Timestamps



# Ordering with Timestamps



# References

- Andrew S. Tanenbaum, Maarten van Steen:  
Distributed systems - principles and paradigms, Chapter 6.1  
Free available at <https://www.distributed-systems.net/index.php/books/ds3/>

you can make some assumptions about synchronous of the system and run some algorithm getting perfect clock synchronization or at least synchronize among a certain bound, but if the system is asynchronous the best thing that can do is getting synchronization with very high probability.

could be some period in which two processes are not synchronized within the bound defined, that could be a problem for application that require a strong knowledge of ordering. some application need ordering events in a consistent way

## Dependable Distributed Systems

Master of Science in Engineering in Computer Science

AA 2022/2023

---

LECTURE 4: **LOGICAL CLOCK** usually used when system not synchronous

## Recap

---

Physical clock synchronization algorithms have the aim to coordinate processes to reach an agreement on a common notion of time

The accuracy of the synchronization is strongly dependent on the estimation of transmission delay

- **ISSUE:** it can be hard to find a good estimation

### OBSERVATION

- In several applications it is not important when things happened but in which order they happened

We need to find a reliable way to order events without using clock synchronization!

logical clock: clock not taken by machine, not a physical parameter of a process, but is a logical parameter, an abstraction of the clock.

## Happened-Before relation

### OBSERVATION

- Two events occurred at some process  $p_i$  happened in the same order as  $p_i$  observes them
- When  $p_i$  sends a message to  $p_j$  the send event happens before the deliver event

↳ logical relation between some pair of events existing in computation that somehow capture notion of causality.

Lamport introduces the happened-before relation to capture causal dependencies between events (causal order relation)

- We note with  $\rightarrow_i$  the ordering relation between events in a process  $p_i$  (local notion of the events)
- We note with  $\rightarrow$  the happened-before between any pair of events

When order event want capture that something happened before something else, no need to know the time in which they happened, but want capture the causal relation between a cause and a consequence.

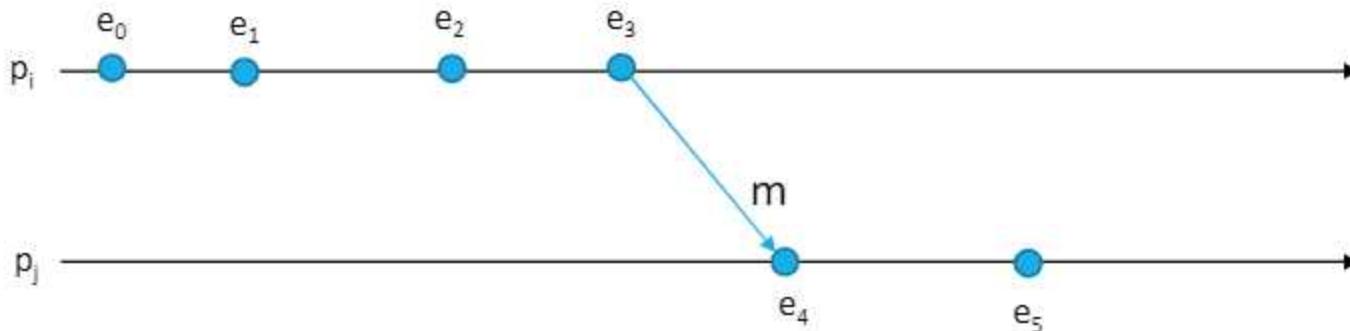
Abstract the notion of the time in which an event happened, simply look at causality between events → formalize CAUSALITY by defining happened-before

two events communicate exchange messages, there is always an event (send) an event (deliver). Send happened before deliver always

Happened-Before Relation: **Definition** *based on the fact that local everything is easy, globally use the communication motion.*

Two events  $e$  and  $e'$  are related by happened-before relation ( $e \rightarrow e'$ ) if: *satisfy one of this condition*

- $\exists p_i \mid e \rightarrow_i e'$  locally on process  $i$ ,  $e_1$  is happened before  $e_2$ .
- $\exists m \mid e=\text{send}(m)$  and  $e'=\text{deliver}(m)$   $e_3$  (send) is happened before  $e_4$  (deliver) of  $m$ .
- $\exists e, e', e'' \mid (e \rightarrow e'') \wedge (e'' \rightarrow e')$  (happened-before relation is transitive)  
Ex.  $e_2$  is happened before  $e_5$ , because  $e_2$  h-b  $e_3$  to rule 1.,  $e_3$  h-b  $e_4$  to rule 2. and  $e_4$  h-b  $e_5$  for 1.



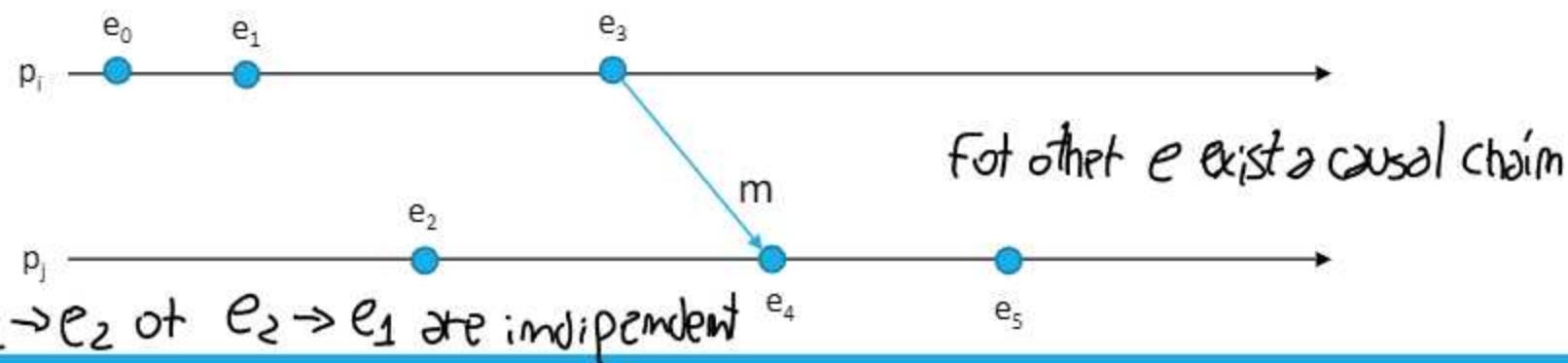
we can simply generate a partial order of the events

# Happened-Before Relation

## OBSERVATIONS

- Happened-before relation imposes a partial order over events of the execution history
  - It may exists a pair of events  $\langle e_i, e_j \rangle$  such that  $e_i$  and  $e_j$  are not in happened-before relation
  - If  $e_i$  and  $e_j$  are not in happened-before relation then they are concurrent ( $e_i \parallel e_j$ )
- For any pair of events  $e_i$  and  $e_j$  in a distributed system only one of the following holds
  - $e_i \rightarrow e_j$
  - $e_j \rightarrow e_i$
  - $e_i \parallel e_j$

for  $e_1$  and  $e_2$  is not possible to find any path link, the concurrent



for other events we want emulate the behavior of physical clock, something that always increasing each time by one, here instead of incrementing at any physical time of the computation, increment the logical clock each time that happens a relevant event in computation.

## Logical Clock

---

The Logical Clock, introduced by Lamport, is a software counter that monotonically increases its value, always increase never go back

A logical clock  $L_i$  can be used to timestamp events

---

$ts_e = L_i(e)$  is the “logical” timestamp assigned by a process  $p_i$  to events  $e$  using its current logical clock

---

### PROPERTY

<sup>h-b</sup>

- If  $e \rightarrow e'$  then  $ts_e < ts_{e'}$

keep track causality with a logical clock

### Observation

- The ordering relation obtained through logical timestamps is only a partial order

## Scalar Logical Clock: an implementation

Each process  $p_i$  initializes its logical clock  $L_i = 0$  ( $\forall i = 1 \dots N$ )

$p_i$  increases  $L_i$  of 1 when it generates an event (either send or receive)

- $L_i = L_i + 1$

When  $p_i$  sends a message  $m$

- creates an event  $send(m)$
- increases  $L_i$
- timestamps  $m$  with  $ts = L_i$

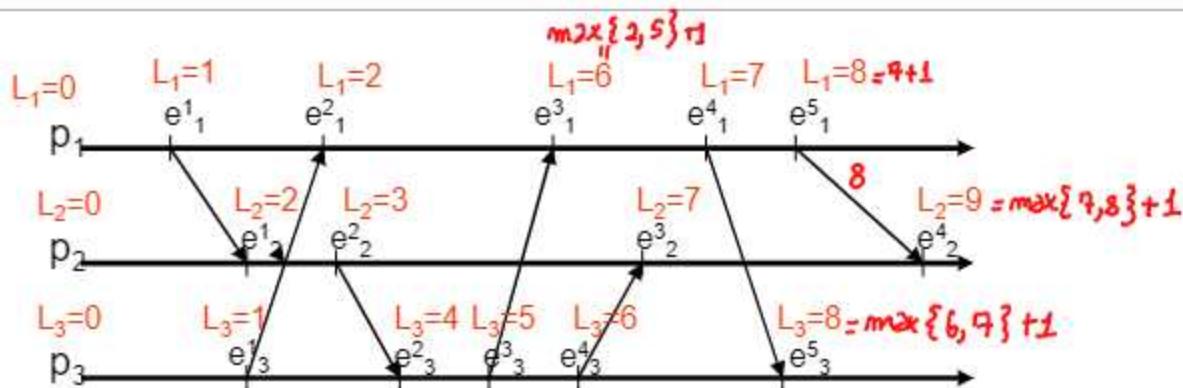
increment clock  $\Delta m$  and attach the value to message

When  $p_i$  receives a message  $m$  with timestamp  $ts$

- Updates its logical clock  $L_i = \max(ts, L_i)$
- Produces an event  $receive(m)$
- Increases  $L_i$

for preserve causality when update not simply increase by one, but compare with received timestamp

## Scalar Logical Clock: example



$e^j_i$  is  $j$ -th event of process  $p_i$

$L_i$  is the logical clock of  $p_i$

### NOTE

- $e^1_1 \rightarrow e^2_1$  and timestamps reflect this property
- $e^1_1 \parallel e^1_3$  and respective timestamps have the same value
- $e^1_2 \parallel e^3_1$  but respective timestamps have different values

OK are concurrent

## Limits of Scalar Logical Clock

Scalar logical clock can guarantee the following property

- If  $e \rightarrow e'$  then  $ts_e < ts_{e'}$

But it is not possible to guarantee

- If  $ts_e < ts_{e'}$  then  $e \rightarrow e'$

Consequently:

*problem in some application*

- Using scalar logical clocks, it is not possible to determine if two events are concurrent or related by the happened-before relation

Mattern [1989] and Fridge [1991] proposed an improved version of logical clock where events are timestamped with local logical clock and node identifier

- ***Vector Clock***

is a data structure, again a logical clock that you associate to every event with a timestamp, but instead use a simple number, you use a vector based on some rule

## Vector Clock : definition

Vector Clock for a set of N processes is composed by an array of N integer counters

Each process  $p_i$  maintains a Vector Clock  $V_i$ , and timestamps events by mean of its Vector Clock

Similarly to scalar clock, Vector Clock is attached to message  $m$

- in this case the timestamp will be an integer vector (i.e., an array of integer)

Vector Clock allows nodes to order events in happens-before just looking at their timestamps

- Scalar clocks:  $e \rightarrow e'$  implies  $L(e) < L(e')$
- Vector clocks:  $e \rightarrow e'$  iff  $L(e) < L(e')$



# Vector Clock : an implementation

---

Each process  $p_i$  initializes its Vector Clock  $V_i$

- $V_i[j] = 0 \quad \forall j = 1 \dots N$

$p_i$  increases  $V_i[i]$  of 1 when it generates an event

- $V_i[i] = V_i[i] + 1$

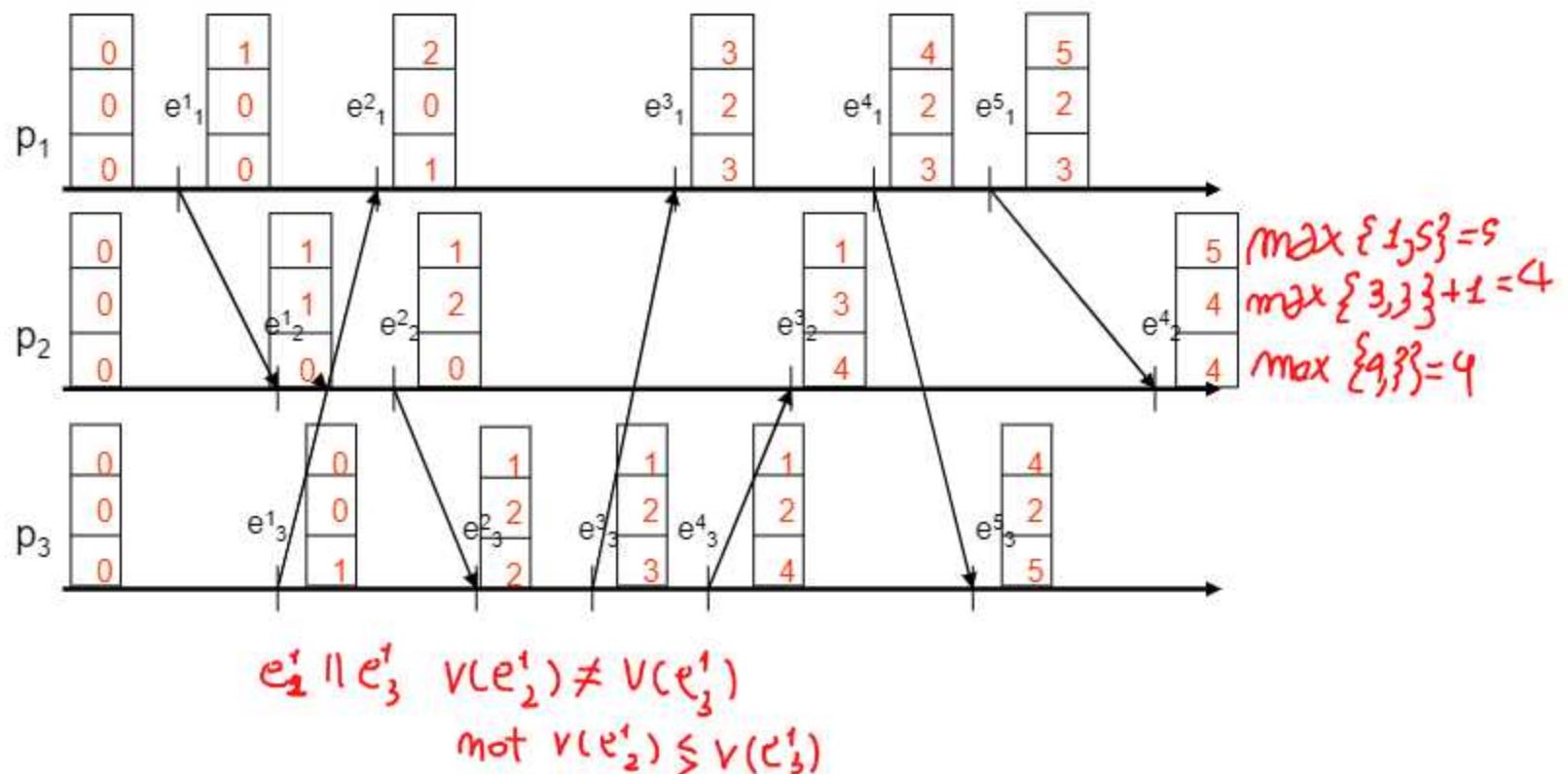
When  $p_i$  sends a message  $m$

- Creates an event  $send(m)$
- Increases  $V_i$
- timestamps  $m$  with  $ts = V_i$

When  $p_i$  receives a message  $m$  containing timestamp  $ts$

- Updates its logical clock  $V_i[j] = \max(ts[j], V_i[j]) \quad \forall j = 1 \dots N$
- Generates an event  $receive(m)$
- Increases  $V_i$

## Vector Clock: an example



## Vector Clock: properties

---

A Vector Clock  $V_i$

- $V_i[i]$  represents the number of events produced by  $p_i$
- $V_i[j]$  with  $i \neq j$  represents the number of events generated by  $p_j$  that  $p_i$  can know

$V = V'$  if and only if

- $V[j] = V'[j] \forall j = 1 \dots N$

$V \leq V'$  if and only if

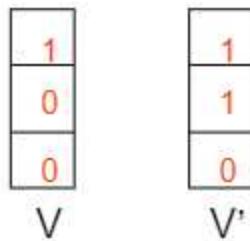
- $V[j] \leq V'[j] \forall j = 1 \dots N$

$V < V'$  therefore the event associated to  $V$  happened before the event associated to  $V'$  if and only if

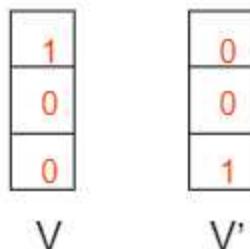
- $V \leq V' \wedge V \neq V'$ 
  - $\forall i = 1 \dots N V'[i] \geq V[i]$
  - $\exists i \in \{1 \dots N\} | V'[i] > V[i]$

## A comparison of Vector Clocks

---



$V(e) \leq V'(e')$  then  $e \rightarrow e'$   
and  $V(e) \neq V'(e')$



$V(e) \neq V'(e')$  then  $e \parallel e'$   
not  $V(e) \leq V'(e')$

Differently from Scalar Clock, Vector Clock allows to determine if two events are concurrent or related by a happened-before relation

---

## Logical clock in distributed algorithms

---

We have seen two mechanisms to represent logical time

- Scalar Clock
- Vector Clock

Each mechanism can be used to solve different problems, depending on the problem specification

---

- Scalar Timestamp → Lamport's Mutual Exclusion
- Vector Timestamp → Causal Broadcast

# Distributed Mutual Exclusion

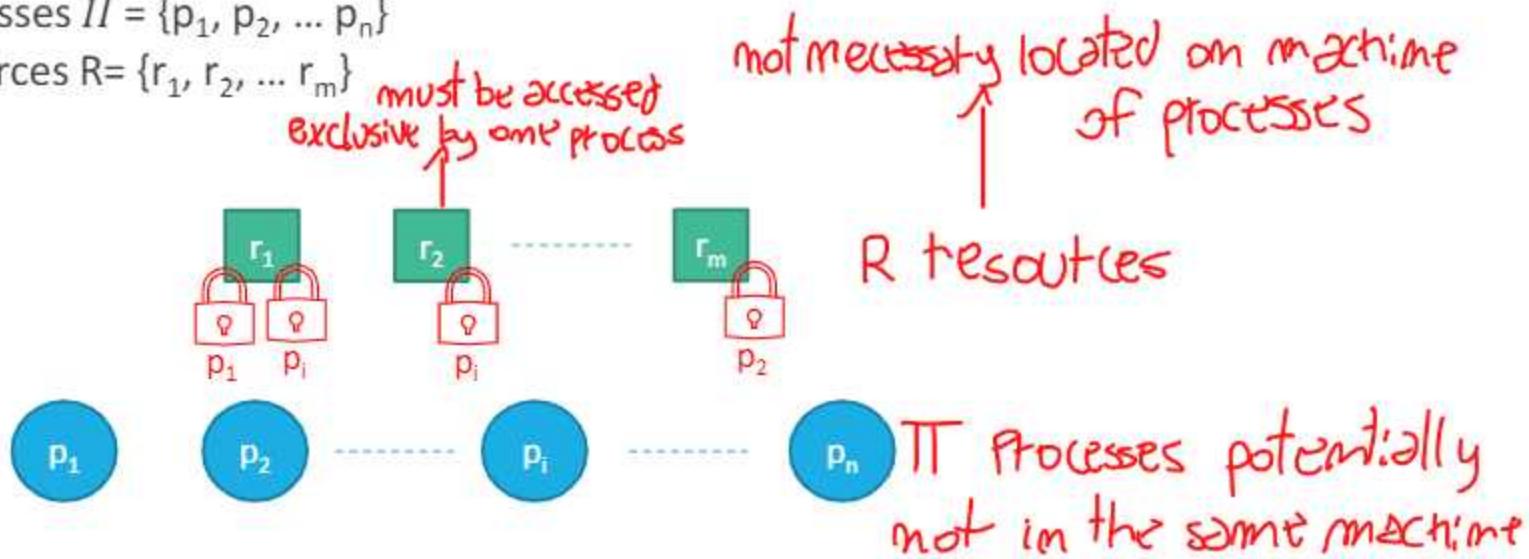
---

Avoid that one resource is taken more than one process in the same time

# The Mutual Exclusion Problem

Let us consider

- a set of processes  $\Pi = \{p_1, p_2, \dots, p_n\}$
- a set of resources  $R = \{r_1, r_2, \dots, r_m\}$



## PROBLEM

- Processes need to access resources exclusively and we need to design a distributed abstraction that allows them to coordinate to get access to resources

want to specify the problem in terms of safety and liveness  
also in distributed system want to avoid deadlock

# System Model

Let us consider

- a set of processes  $\Pi = \{p_1, p_2, \dots, p_n\}$
- a set of resources  $R = \{r_1, r_2, \dots, r_m\}$
- For the sake of simplicity let us assume  $|R| = 1$

fairness: no one be even blocked, if you ask eventually you get

no-deadlock: simply say that some one go on, not necessarily all the processes

The system is asynchronous : not putting any bound on latency of communication or the time needed for turn global computation

Processes are not going to fail (they will be always correct) fundamental

Processes communicate by exchanging messages on top of perfect point-to-point links

↳ no failure on sending messages

# The Mutual Exclusion abstraction

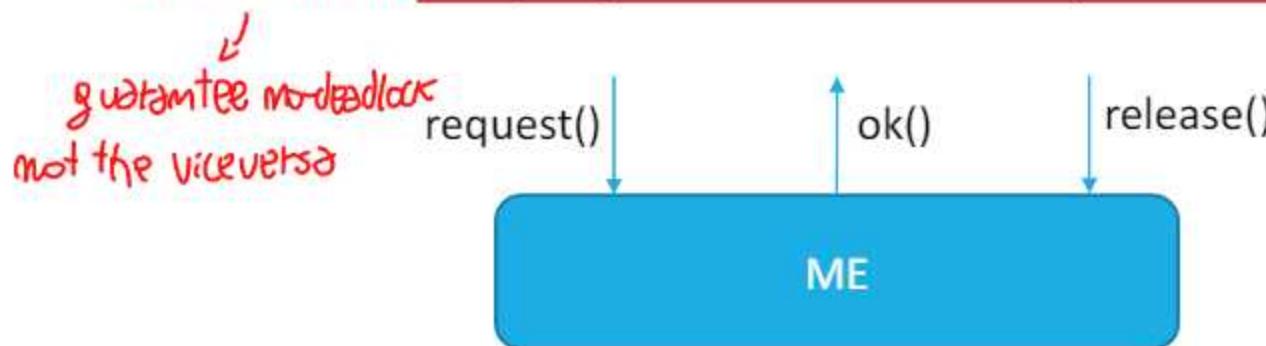
## EVENTS

- request (): it issues a request to enter into the critical section
- ok (): it notifies the process that it can now access the critical section *confirmation of free cs*
- release (): it is invoked to leave the critical section and to allow someone else to enter

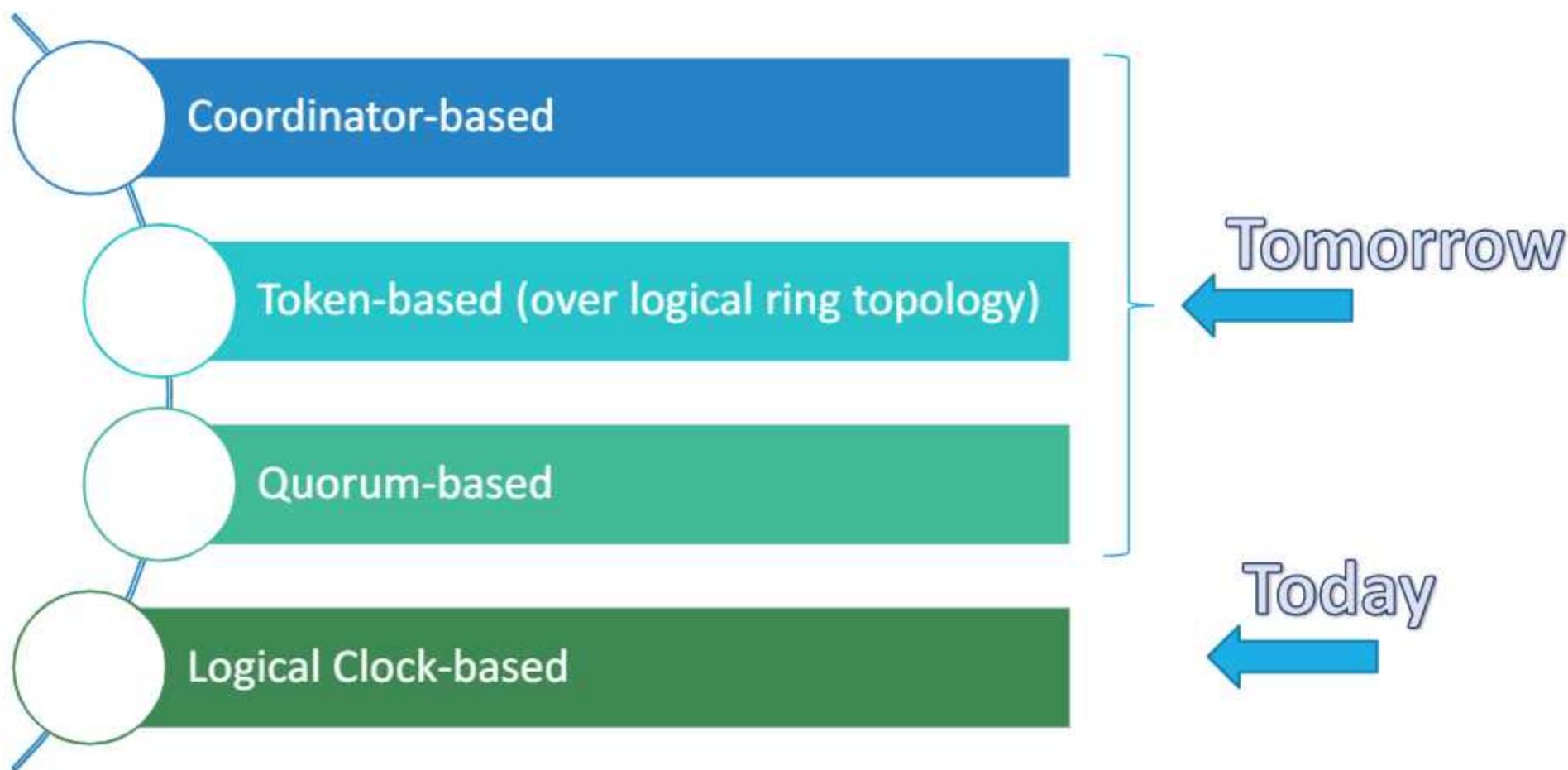
## PROPERTIES *that want to guarantee*

→ safety property

- Mutual Exclusion: at any time t, at most one process  $p$  is running the critical section
- No-Deadlock: there always exists a process  $p$  able to enter the critical section
- No-Starvation: every request () and release () operation eventually terminate



# Different Approaches to Distributed Mutual Exclusion



Timestamp-based algorithm: Lamport's Distributed Mutual Exclusion

*processes need to collaborate to get CS*

*similarly to bakery algorithm*

Difference from concurrent system

- When a process wants to enter the CS sends a request message to all the other

An history of the operations is maintained by using a counter (timestamp)

Each transmission and reception event is relevant to the computation

- The counter is incremented for each send and receive event
- The counter is incremented also when a message, not directly related to the mutual exclusion computation, is sent or received.

## Lamport's algorithm: implementation

---

Local data structures to each process  $p_i$

- $ck$  integer
    - Is the counter for process  $p_i$
  - $Q$ 
    - Is a queue maintained by  $p_i$  where CS access requests are stored

---
- $\leadsto$  processes behind me for CS

Algorithm rules for a process  $p_i$

- Request to access the CS
  - $p_i$  sends a request message, attaching  $ck$ , to all the other processes
  - $p_i$  adds its request to  $Q$
- Request reception from a process  $p_j$ 
  - $p_i$  puts  $p_j$  request (including the timestamp) in its queue
  - $p_i$  sends back an ack to  $p_j$

## Lamport's algorithm: implementation

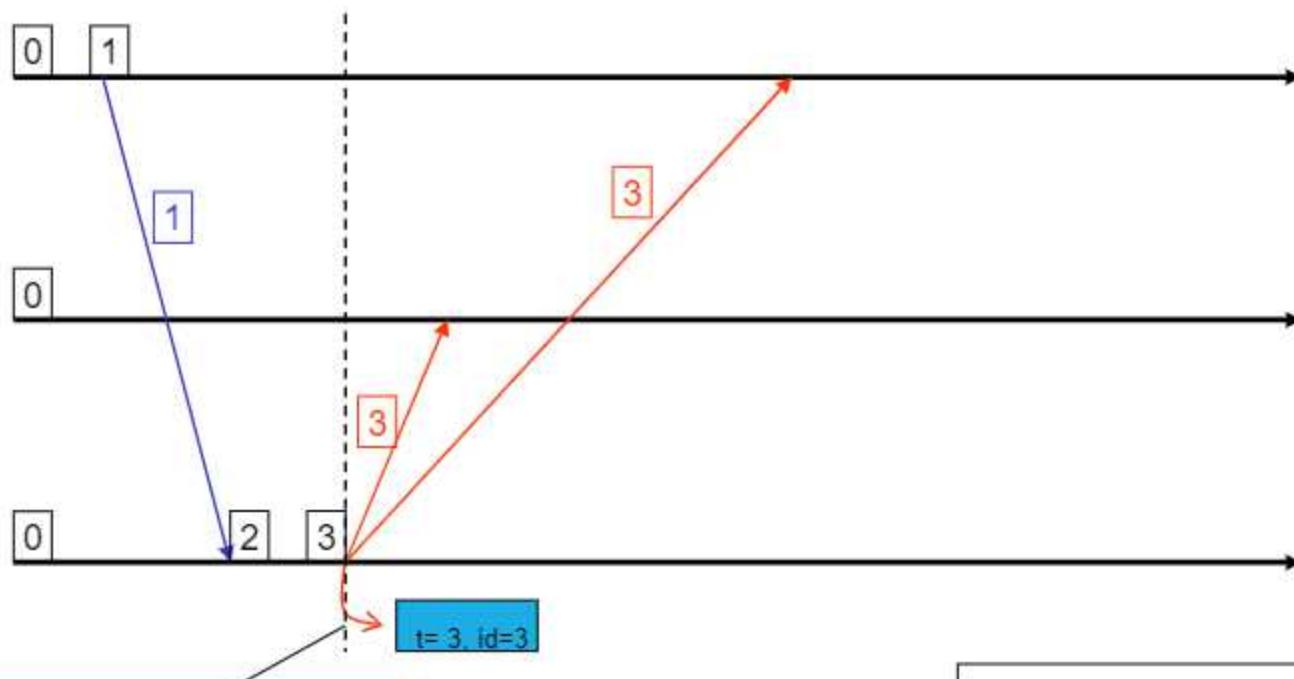
Algorithm rules for a process  $p_i$

- $p_i$  enters the CS iff
  - $p_i$  has, in its queue, a request with timestamp  $t$
  - $t$  is the small timestamp in the queue
  - $p_i$  has already received an ack with timestamp  $t'$  from any other process and  $t' > t$
- Release of the CS
  - $p_i$  sends a RELEASE message to all the other processes
  - $p_i$  deletes its request from the queue
- Reception of a release message from a process  $p_j$ 
  - $p_i$  deletes  $p_j$ 's request from the queue

if there are two timestamp equal use  
the identifiers of processes

Q USE FIFO way

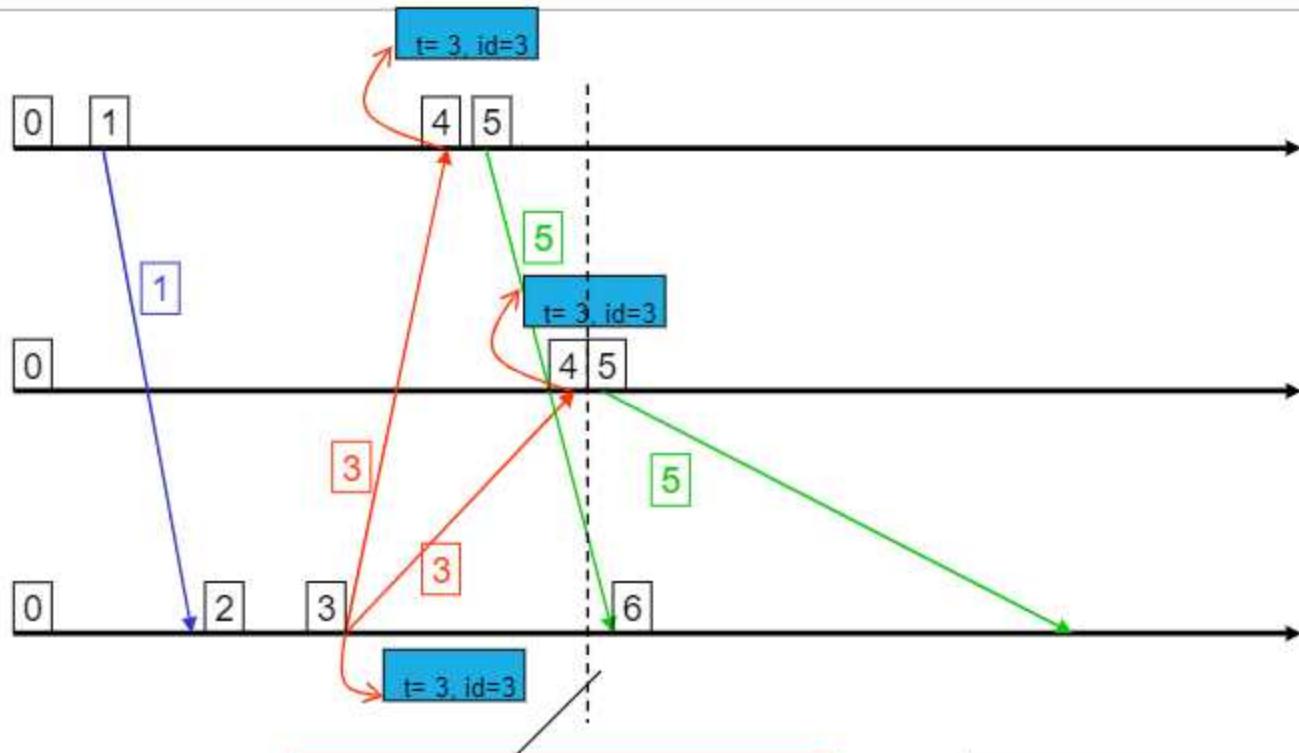
## Lamport's algorithm: example



p<sub>3</sub> sends its CS access request

→ Program Message *not related with mutual exclusion*  
→ CS access Message

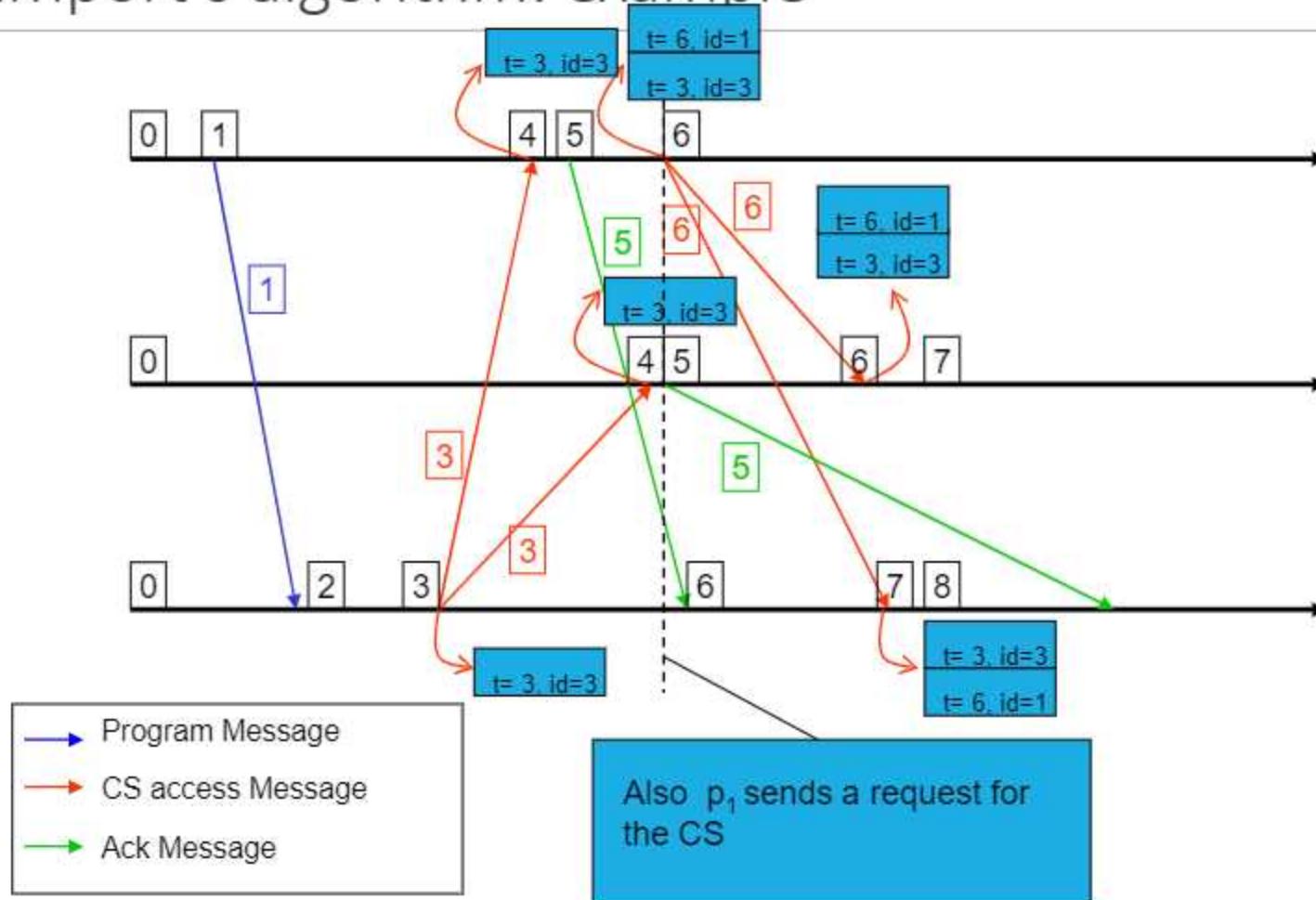
## Lamport's algorithm: example



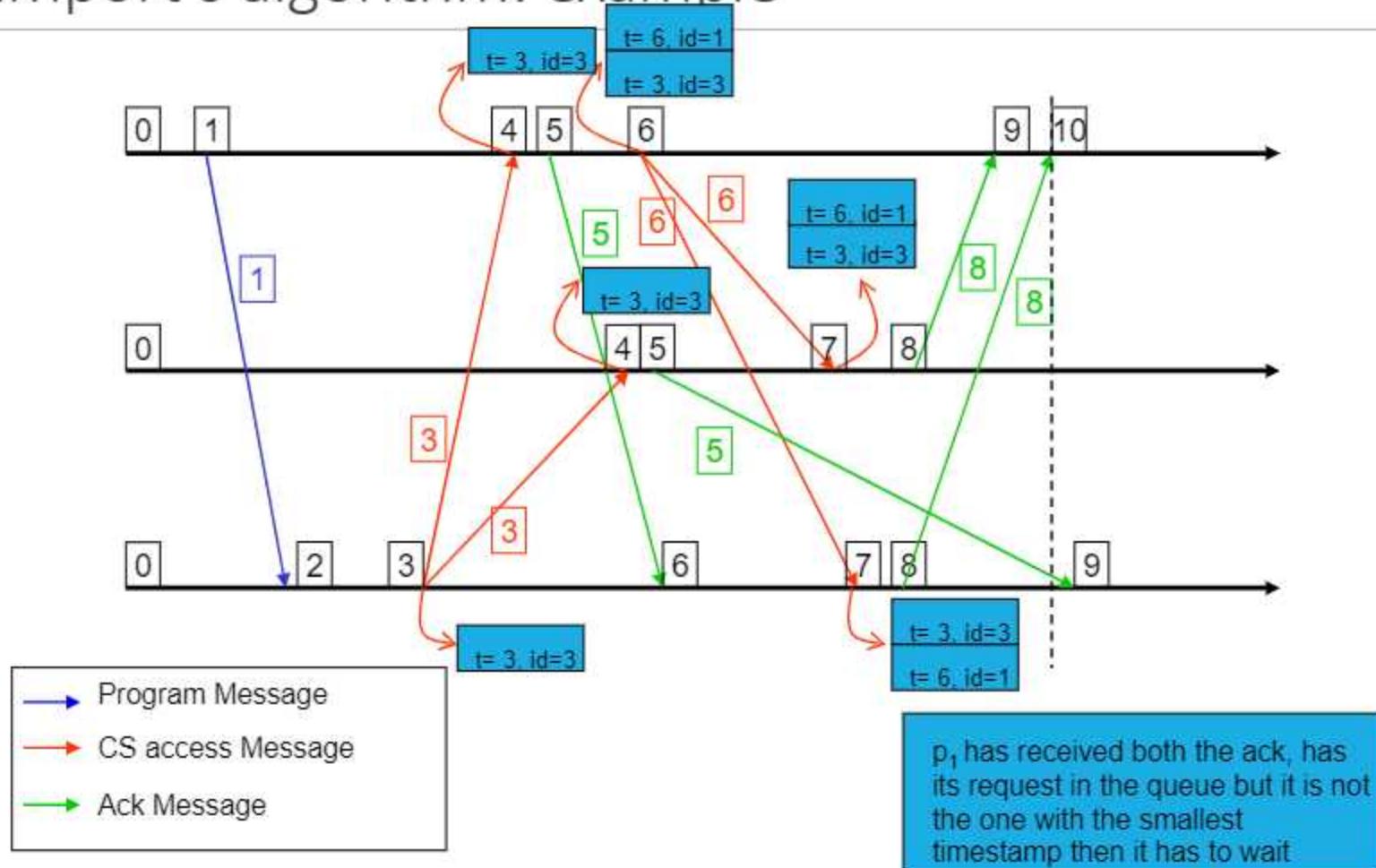
p<sub>1</sub> e p<sub>2</sub> receive p<sub>3</sub> request, they put it in the queue and send back an ack

- Program Message (blue arrow)
- CS access Message (red arrow)
- Ack Message (green arrow)

## Lamport's algorithm: example

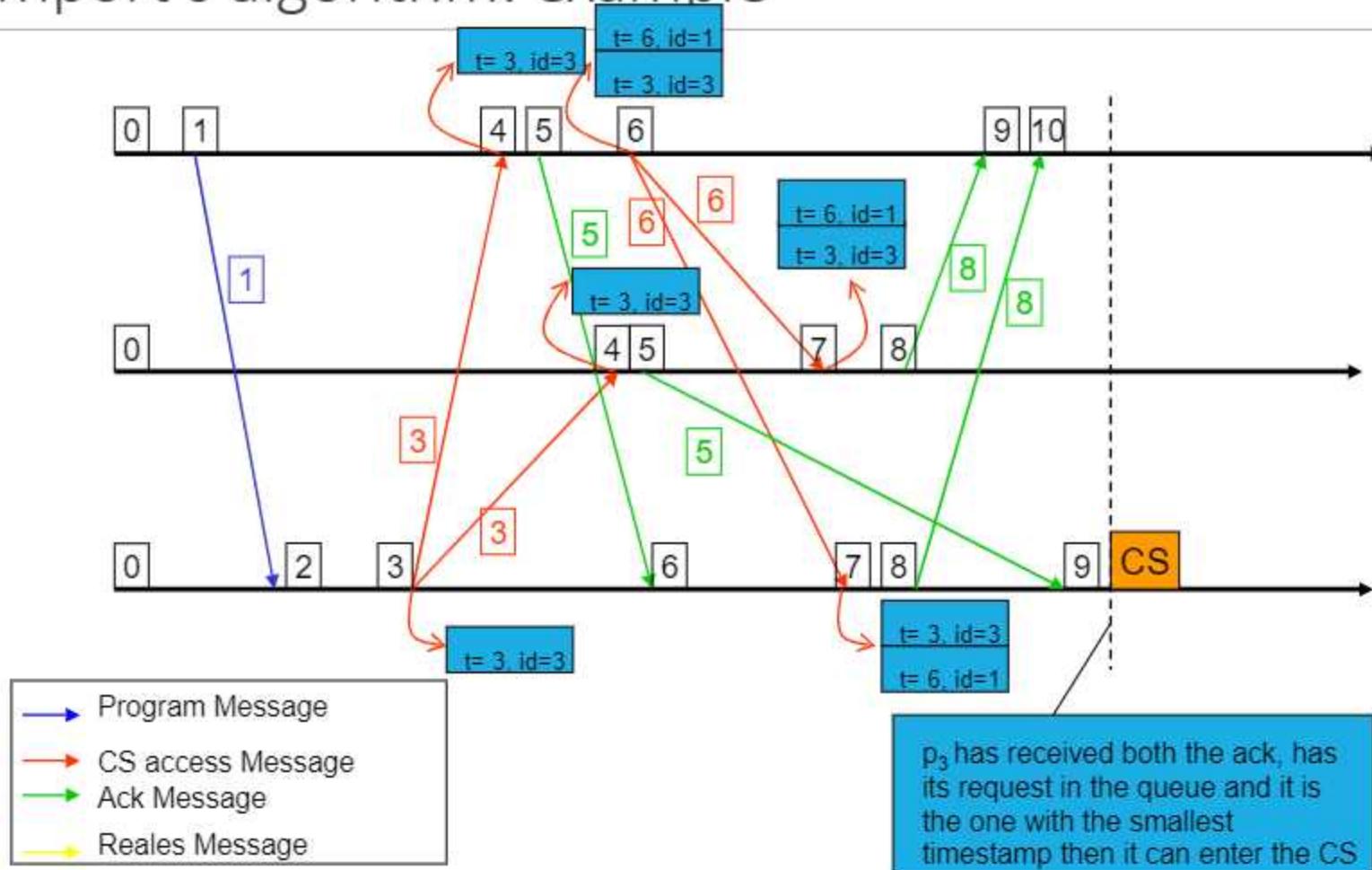


## Lamport's algorithm: example

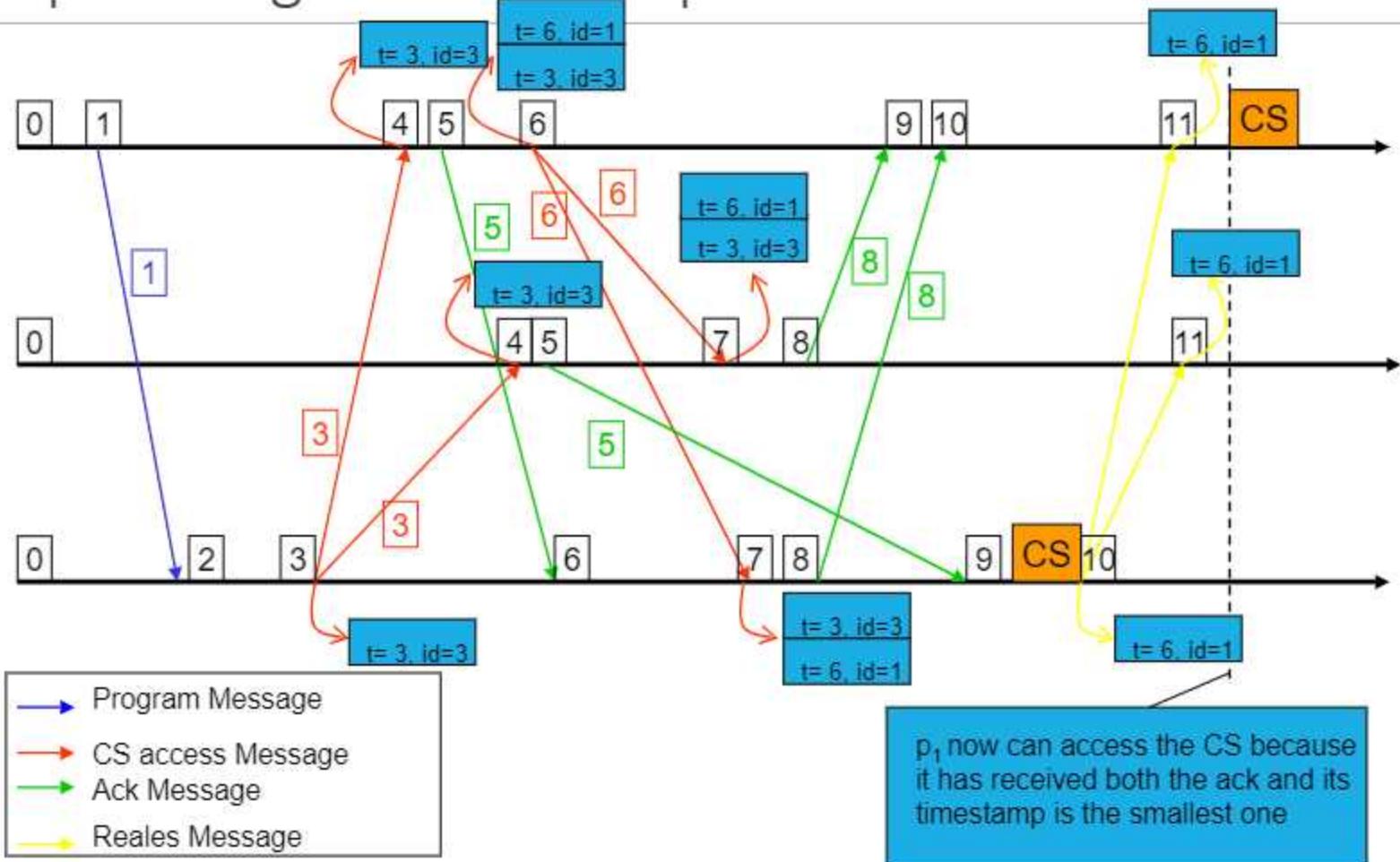


at bit at a time to deliver messages  
but without failure all work

## Lamport's algorithm: example



## Lamport's algorithm: example



for safety show that there always exist at most one process in CS. By contradiction algorithm works and exist two processes that are in CS ( $p_i$  and  $p_j$ ). Both of them verify the 3 conditions of the algorithm.  $p_i$  received the 3 acknowledgement,  $p_i$  also requesting the  $q$  and  $p_i$  has the smallest timestamp in  $q$ . Same for  $p_j$ . This is not possible because there is a causality that link the requests

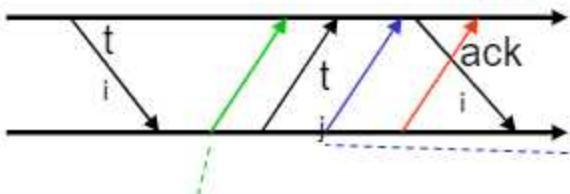
Lamport's algorithm: **safety proof** with the release with the timestamp of the acknowledgement. Show that

Let us suppose by contradiction that both  $p_i$  and  $p_j$  enter the CS

- both the processes have received an ack from any other process and, to enter the CS, the timestamp has to be the smallest in the queue
  - $t_i < t_j < \text{ack}_i.\text{ts}$
  - $t_j < t_i < \text{ack}_j.\text{ts}$

independent from whether message is sent there is

contradiction



Both processes receive the ack when the two requests are in the queue but ME is guaranteed by the total order on the timestamps

$p_j$  ack arrives before  $p_i$  request then  $p_i$  enters the CS without any problem

$p_i$ 's ack arrives after  $p_j$ 's request but before  $p_i$ 's ack then  $p_i$  enters the CS without any problem and sends its ack after executing the CS

## Lamport's algorithm: properties

---

Fairness is satisfied: different requests are satisfied in the same order as they are generated

- Such order comes from the happened-before relation:
  - If two requests are in happened-before relation then they are satisfied in the same order.
  - If two request are concurrent with respect to the happened before relation then the access can happen in any order

for measure performance of the algorithm given that the system is asynchronous is not related to the time, but on the load imposing on the network  $\rightarrow$  messages needed for the algorithm

Lamport's algorithm: performances use messages complexity,

Lamport's algorithm needs  $3(N-1)$  messages for the CS execution

- N-1 requests
- N-1 acks
- N-1 releases

↓  
number of processes

↓  
for each access

overall messages  
that are exchanged

In the best case (none is in the CS and only one process ask for the CS) there is a delay (from the request to the access) of 2 messages

use same philosophy of Lamport but use  $2(N-1)$  messages

## Ricart-Agrawala's algorithm: implementation

### Local variables

- #replies (initially 0)
- State  $\in \{\text{Requesting, CS, NCS}\}$  (initially NCS)
- Q pending requests queue (initially empty)
- Last\_Req  $\hookrightarrow$  timestamp of last request
- Num

not answer if don't want access to CS or not the first to enter

### Algorithm

**begin**

1. State=Requesting
2. Num=num+1; Last\_Req=num
3.  $\forall i=1 \dots N$  send REQUEST(num) to  $p_i$
4. Wait until #replies=n-1
5. State=CS
6. CS  $\hookrightarrow$  only to  $t \in Q$
7.  $\forall r \in Q$  send REPLY to  $r$
8.  $Q = \emptyset$ ; State=NCS; #replies=0

#### **Upon receipt REQUEST( $t$ ) from $p_j$**

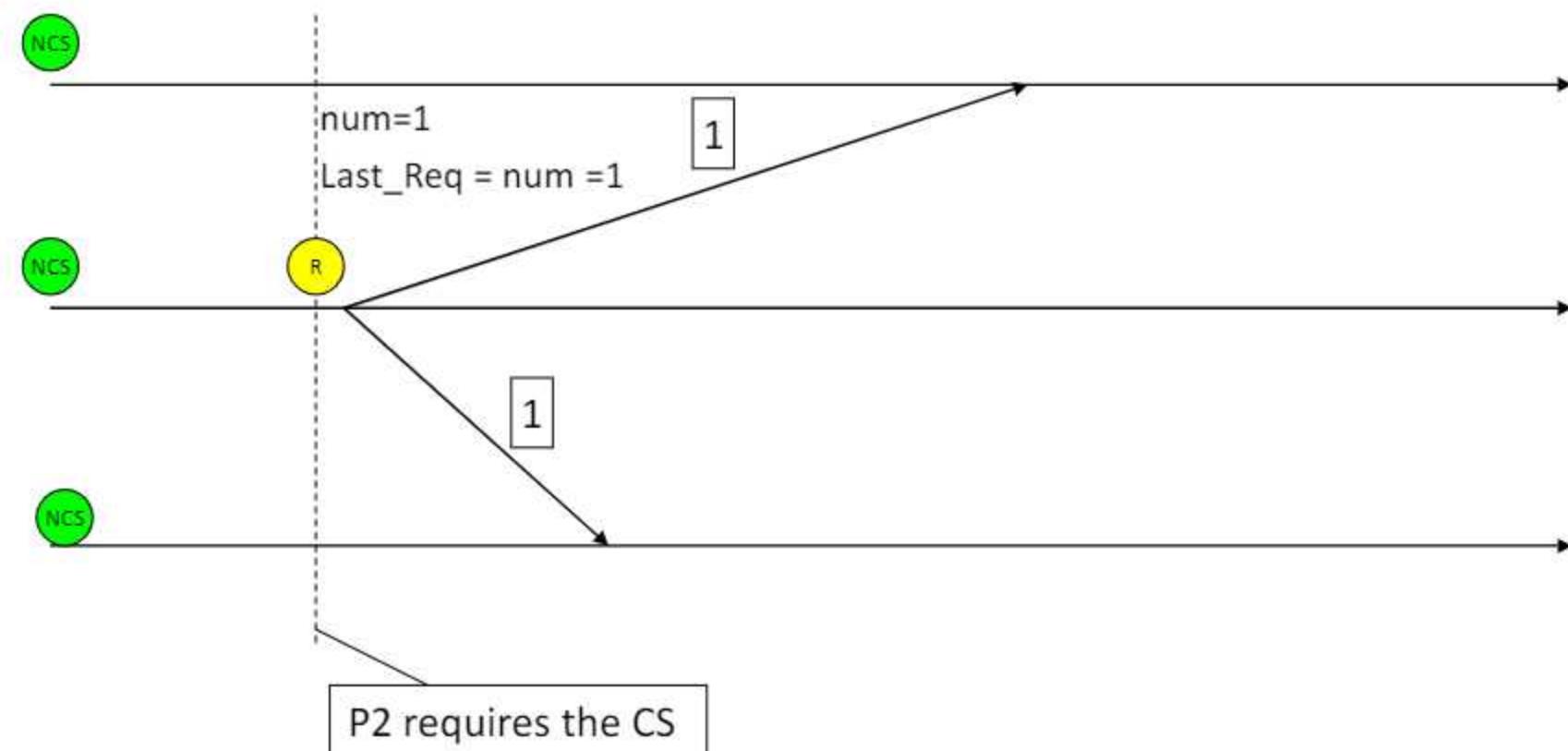
1. If State=CS or (State=Requesting and {Last\_Req, i}  $\subset \{t, j\}$ )
2. Then insert in  $Q[t, j]$
3. Else send REPLY to  $p_j$
4. Num=max( $t, \text{num}$ )

#### **Upon receipt of REPLY from $p_j$**

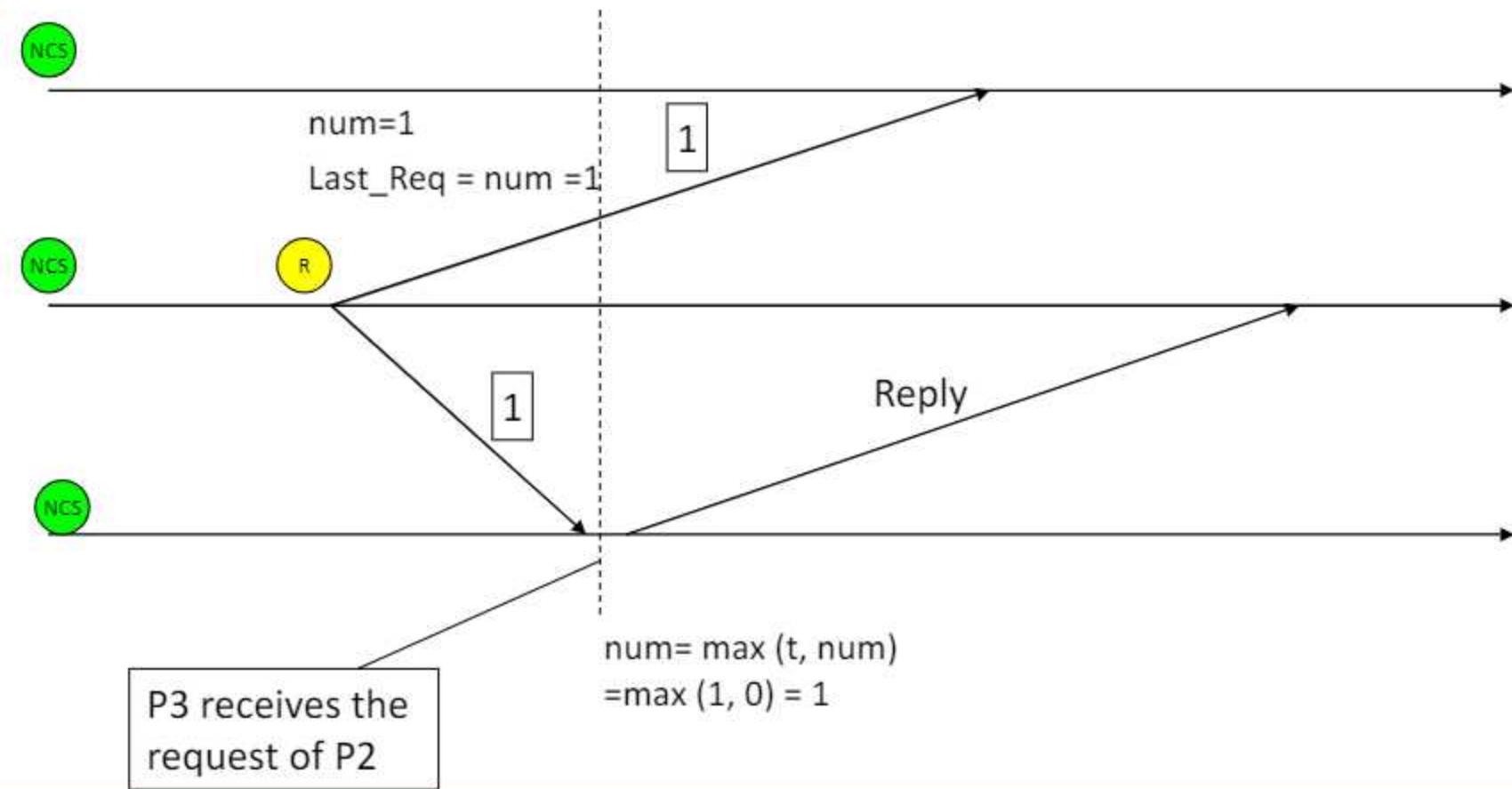
1. #replies=#replies+1

See example for understand !!!

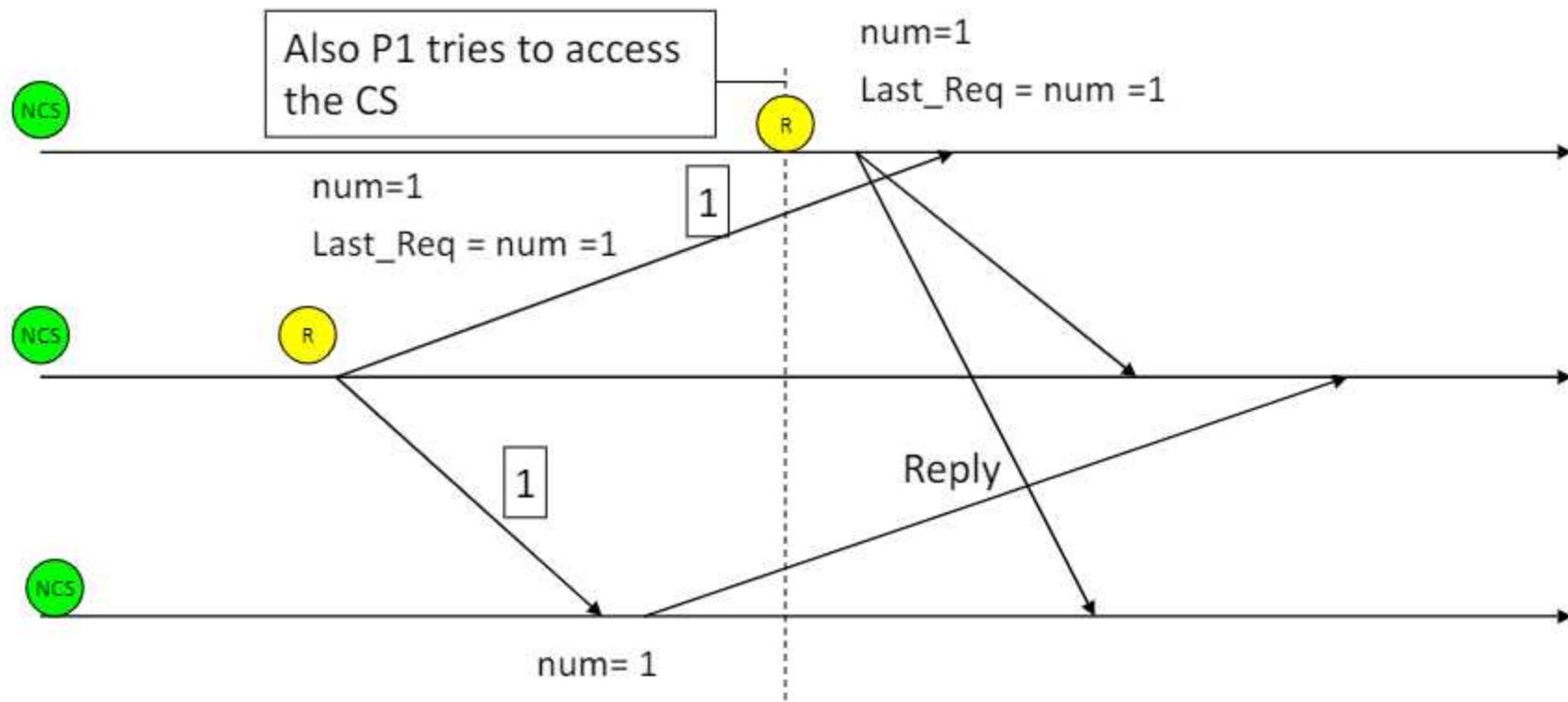
## Ricart-Agrawala's algorithm: example



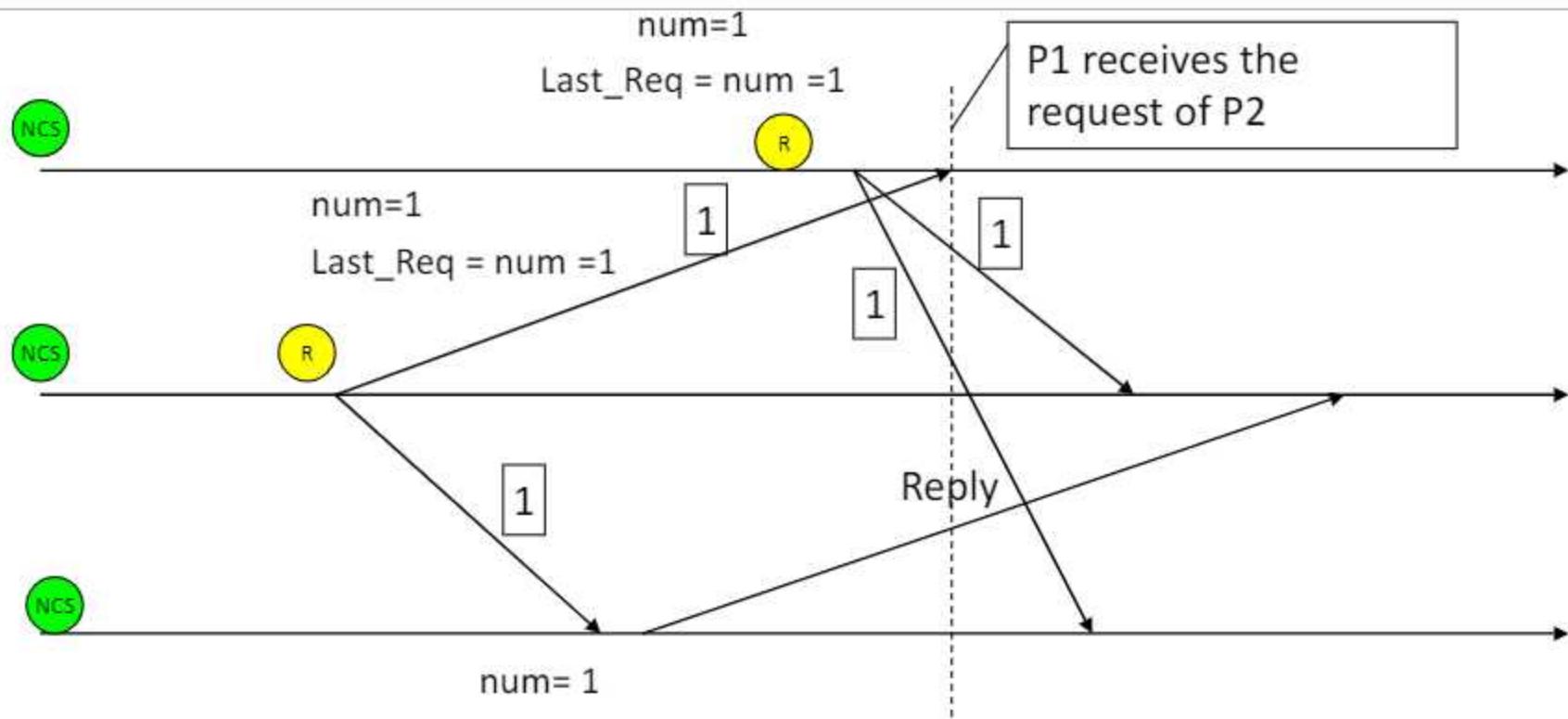
# Ricart-Agrawala's algorithm: example



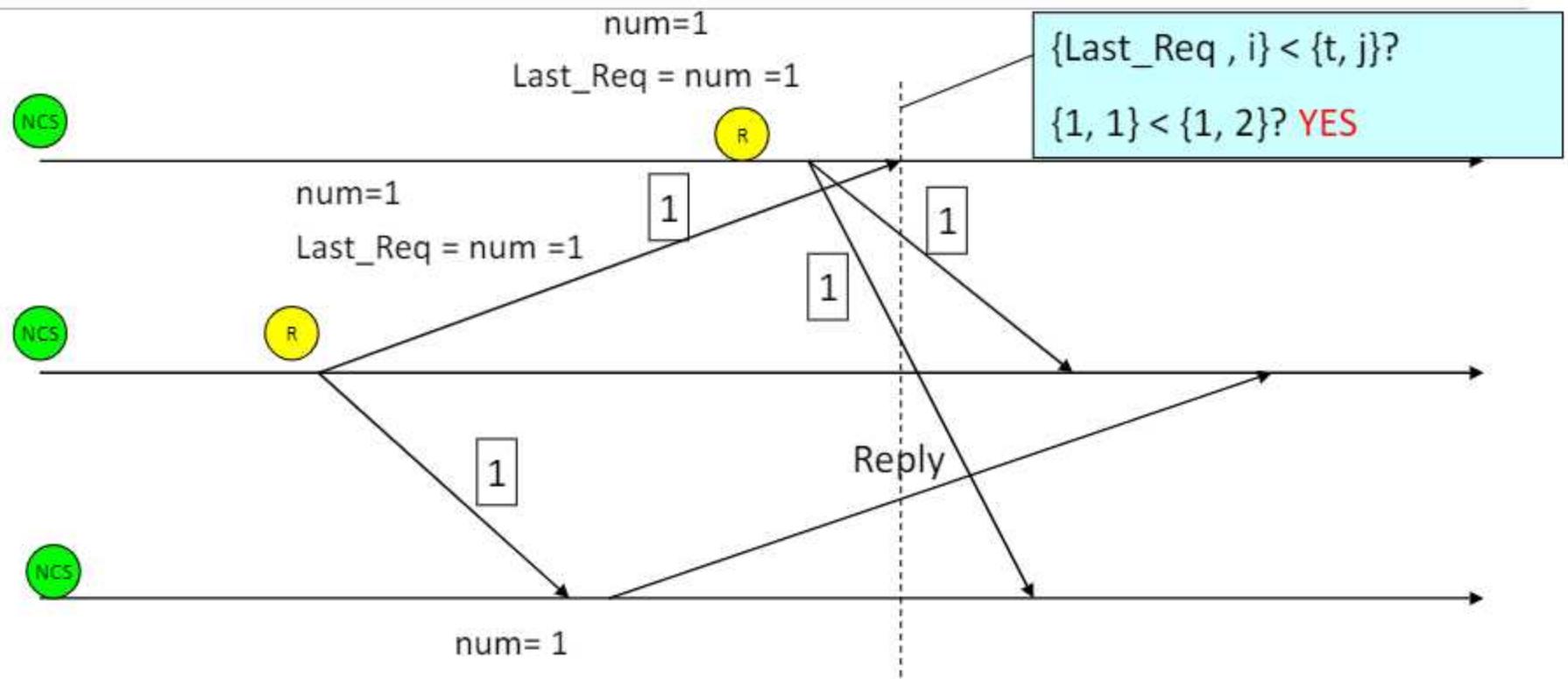
# Ricart-Agrawala's algorithm: example



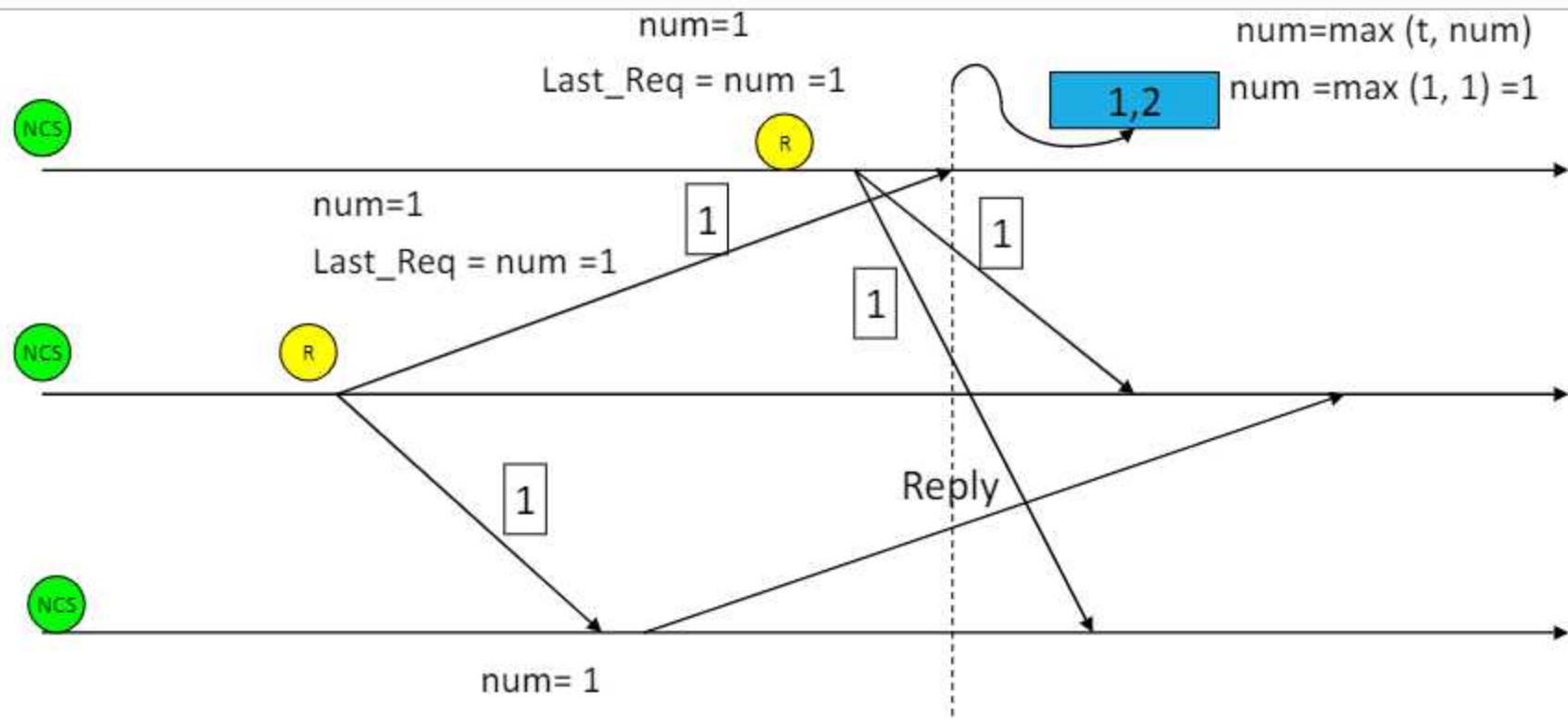
## Ricart-Agrawala's algorithm: example



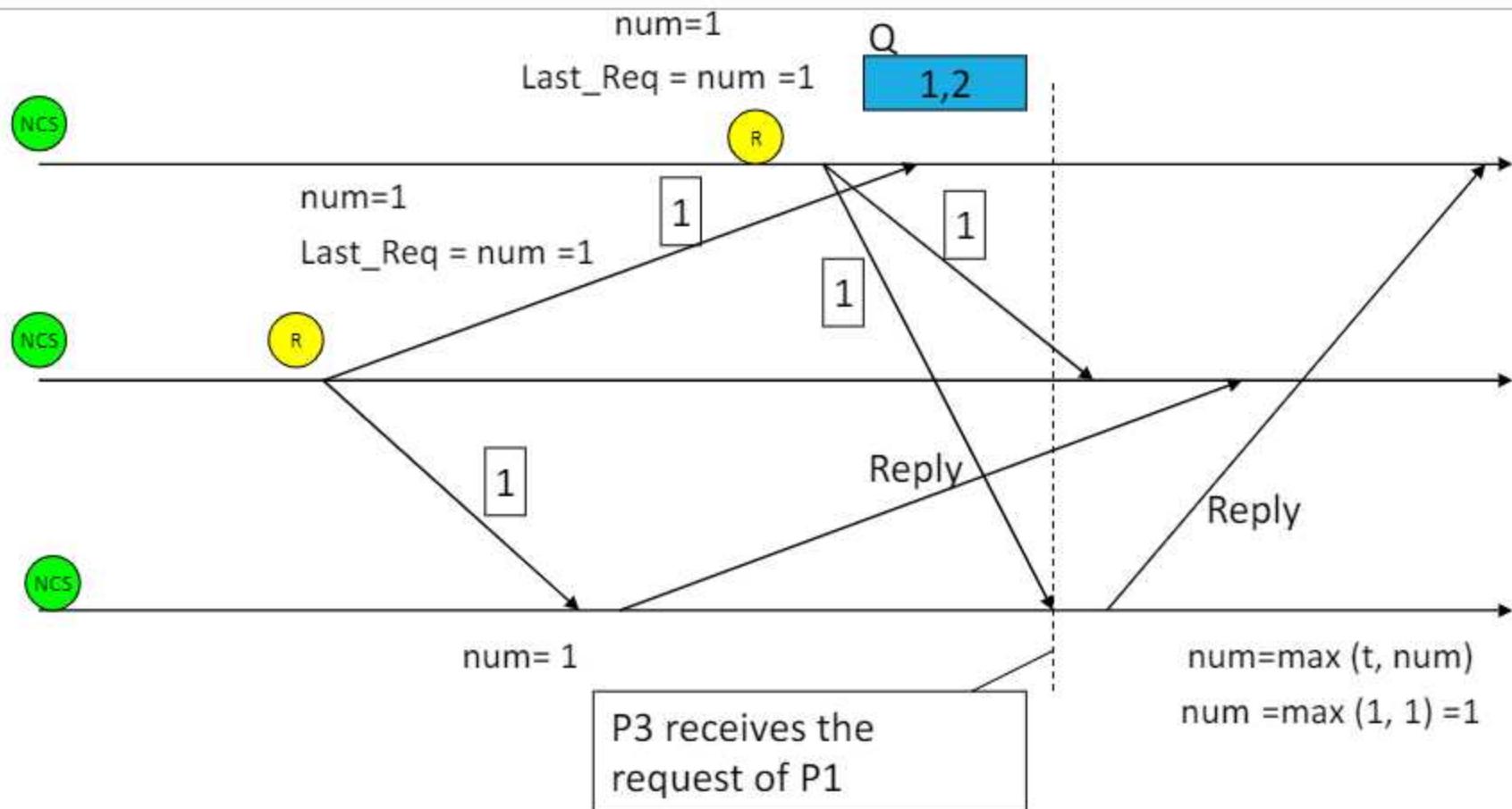
## Ricart-Agrawala's algorithm: example



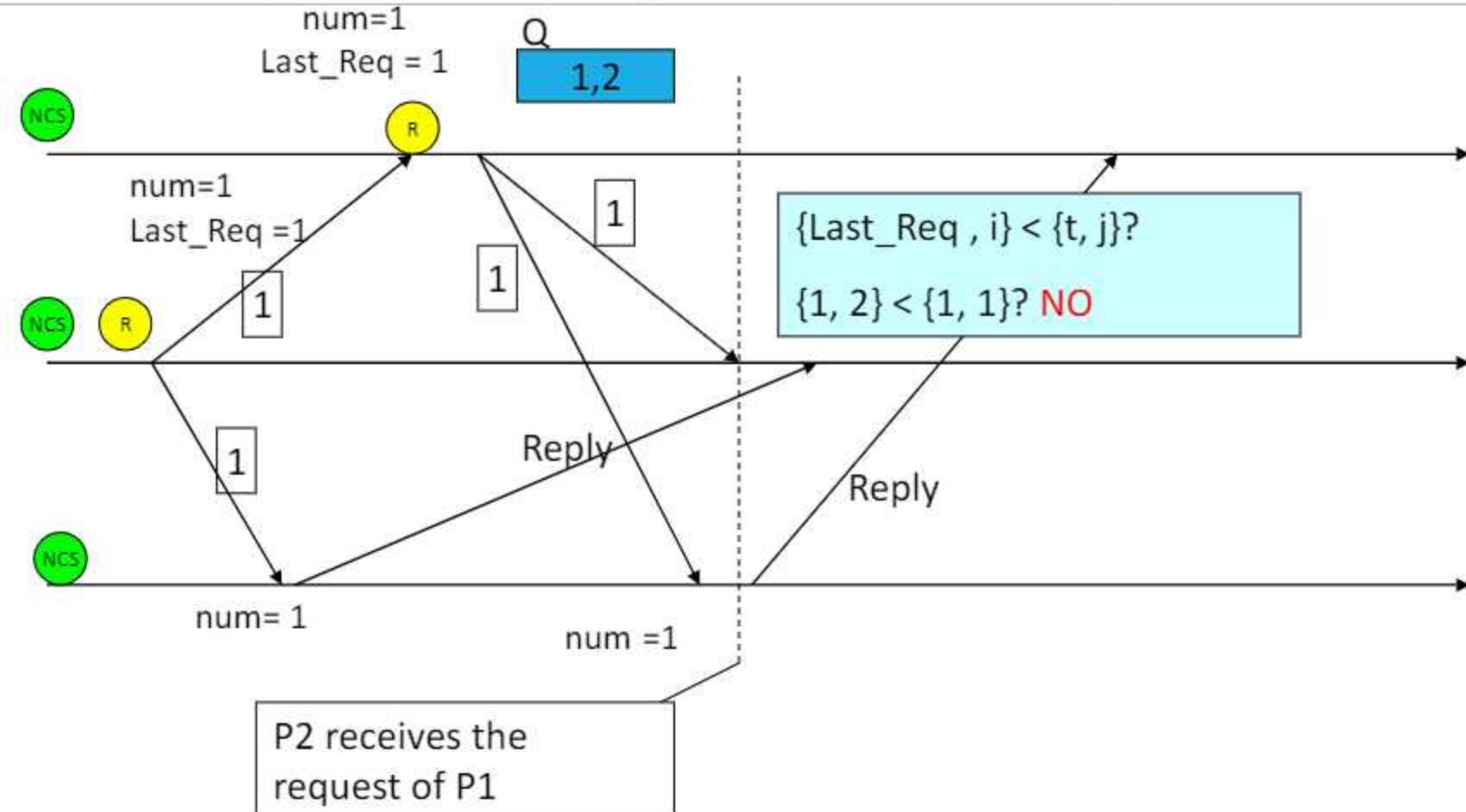
## Ricart-Agrawala's algorithm: example



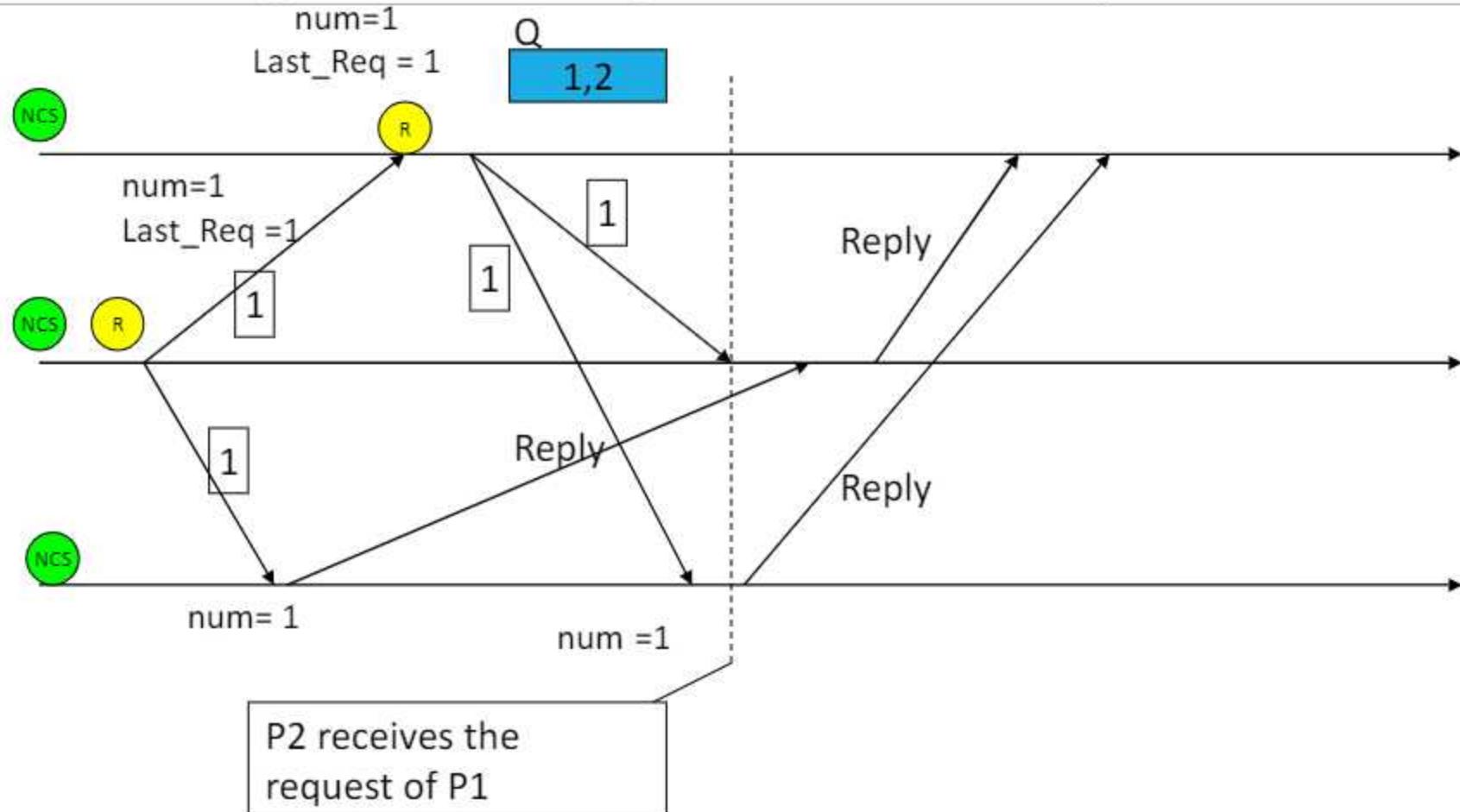
## Ricart-Agrawala's algorithm: example



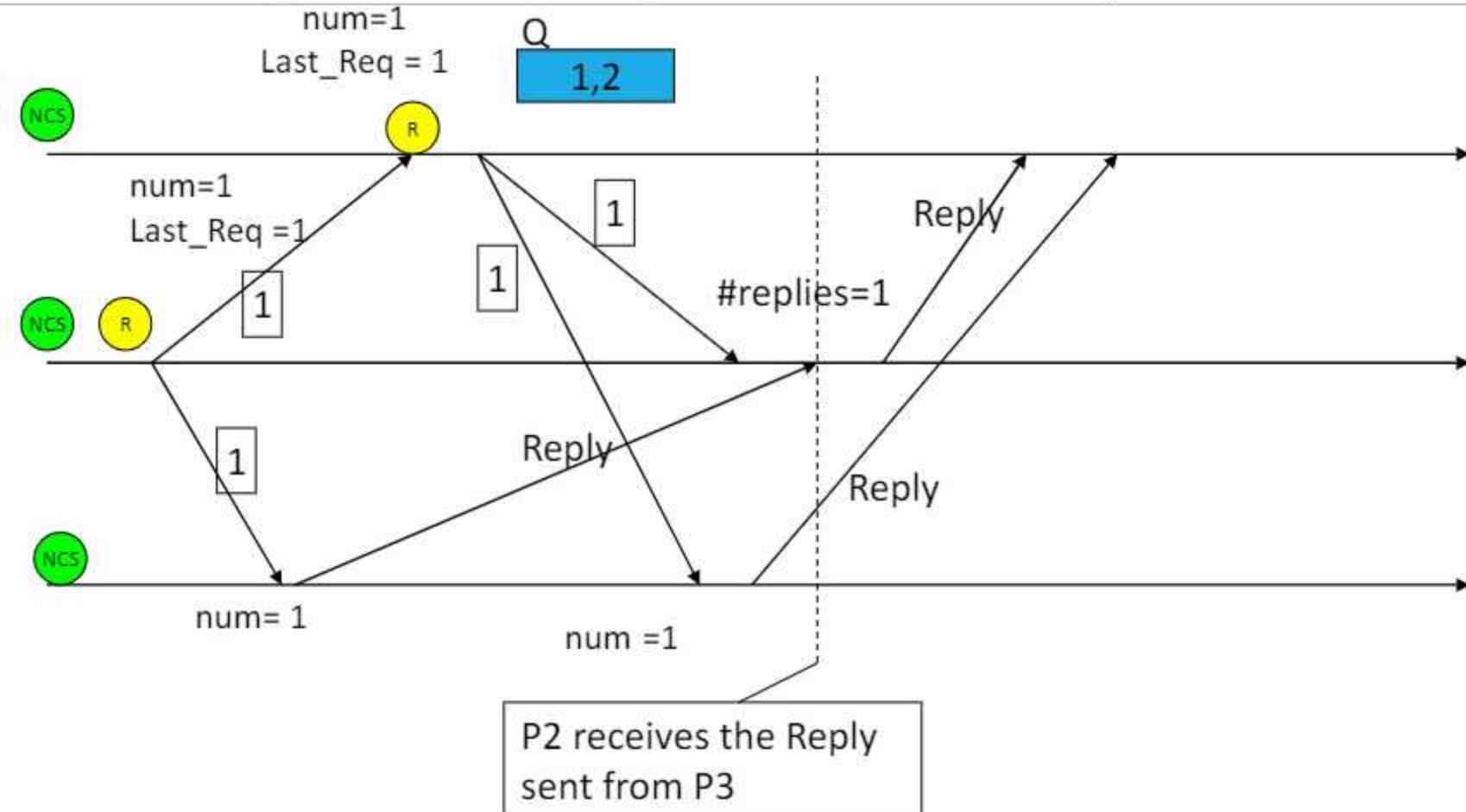
## Ricart-Agrawala's algorithm: example



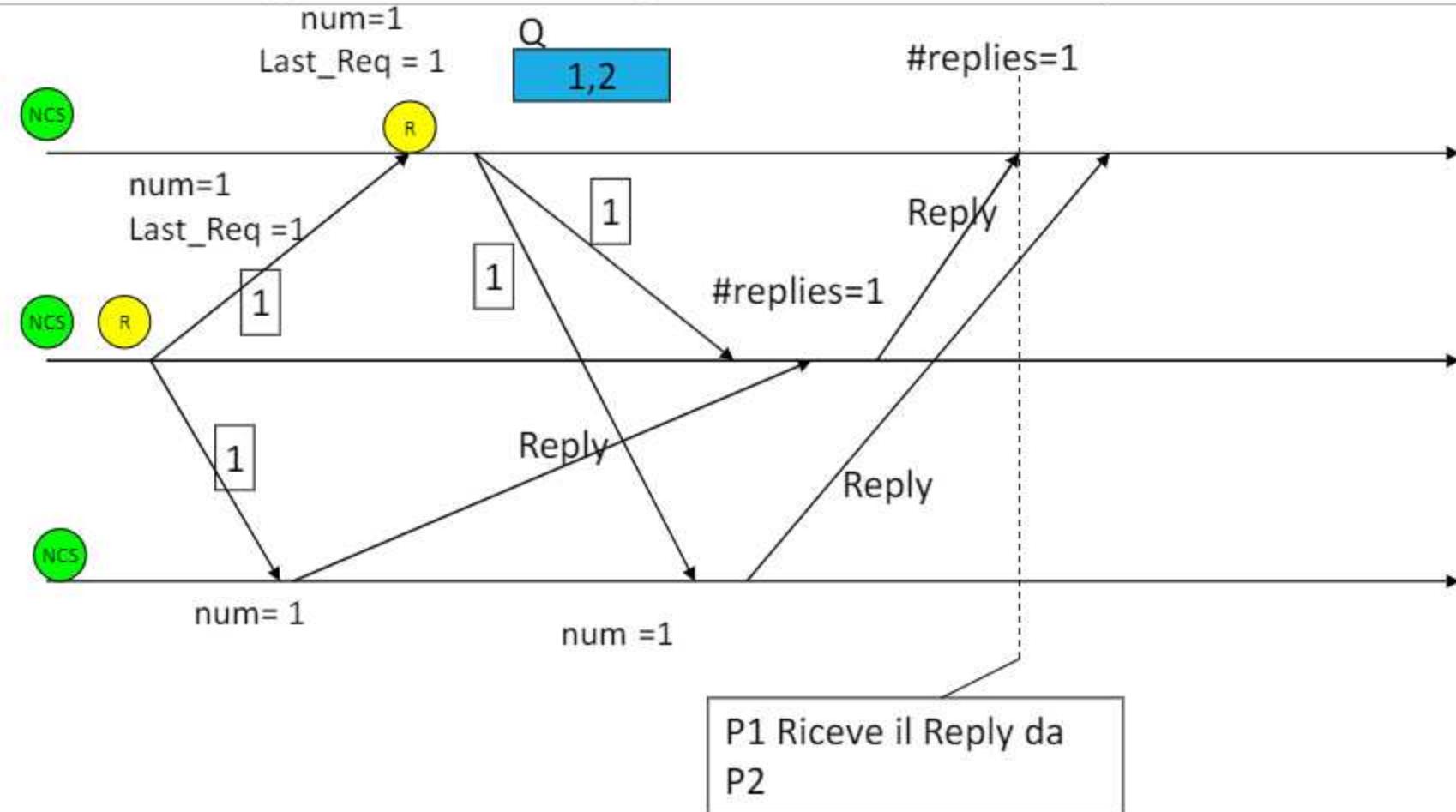
# Ricart-Agrawala's algorithm: example



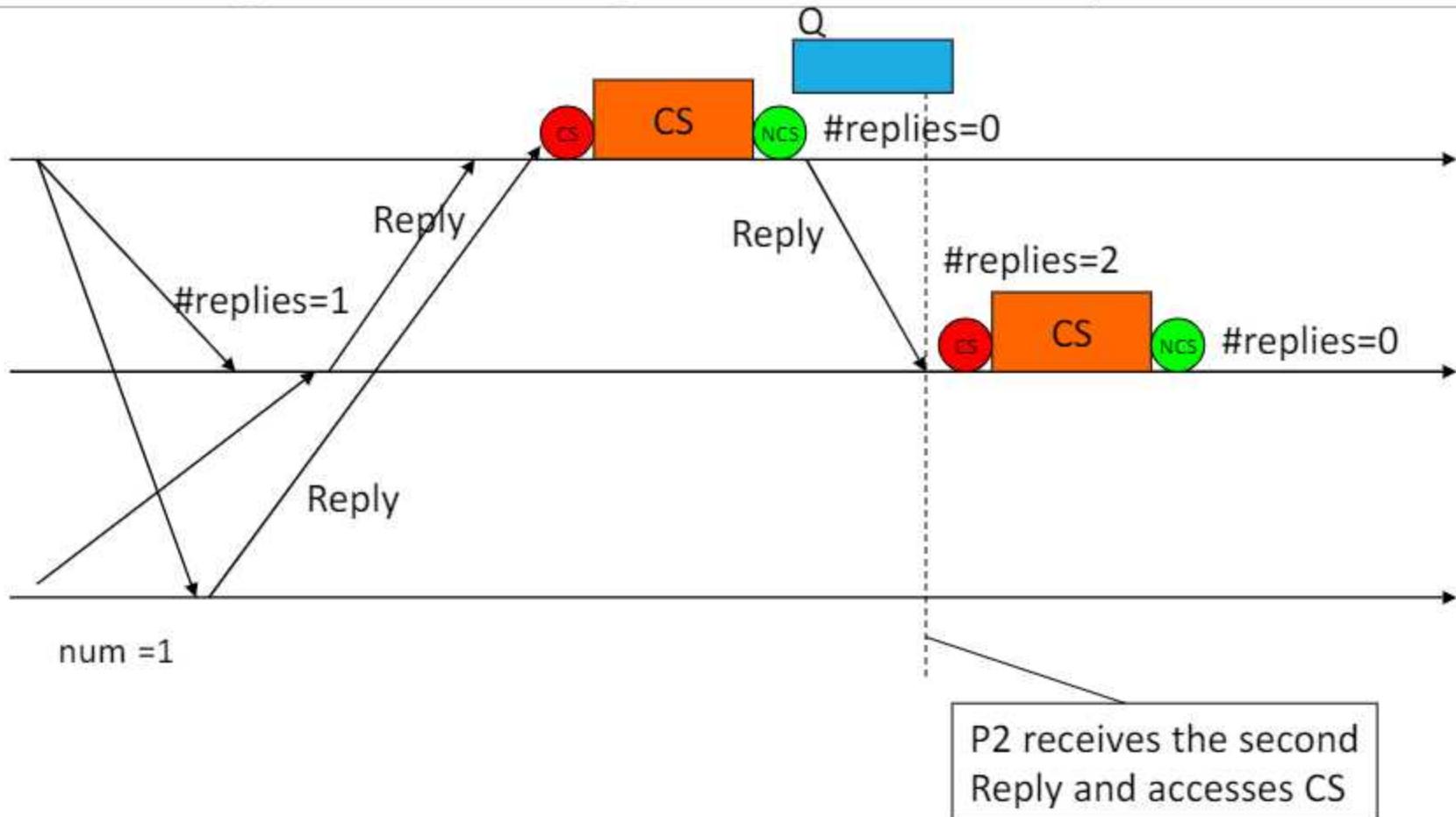
## Ricart-Agrawala's algorithm: example



# Ricart-Agrawala's algorithm: example

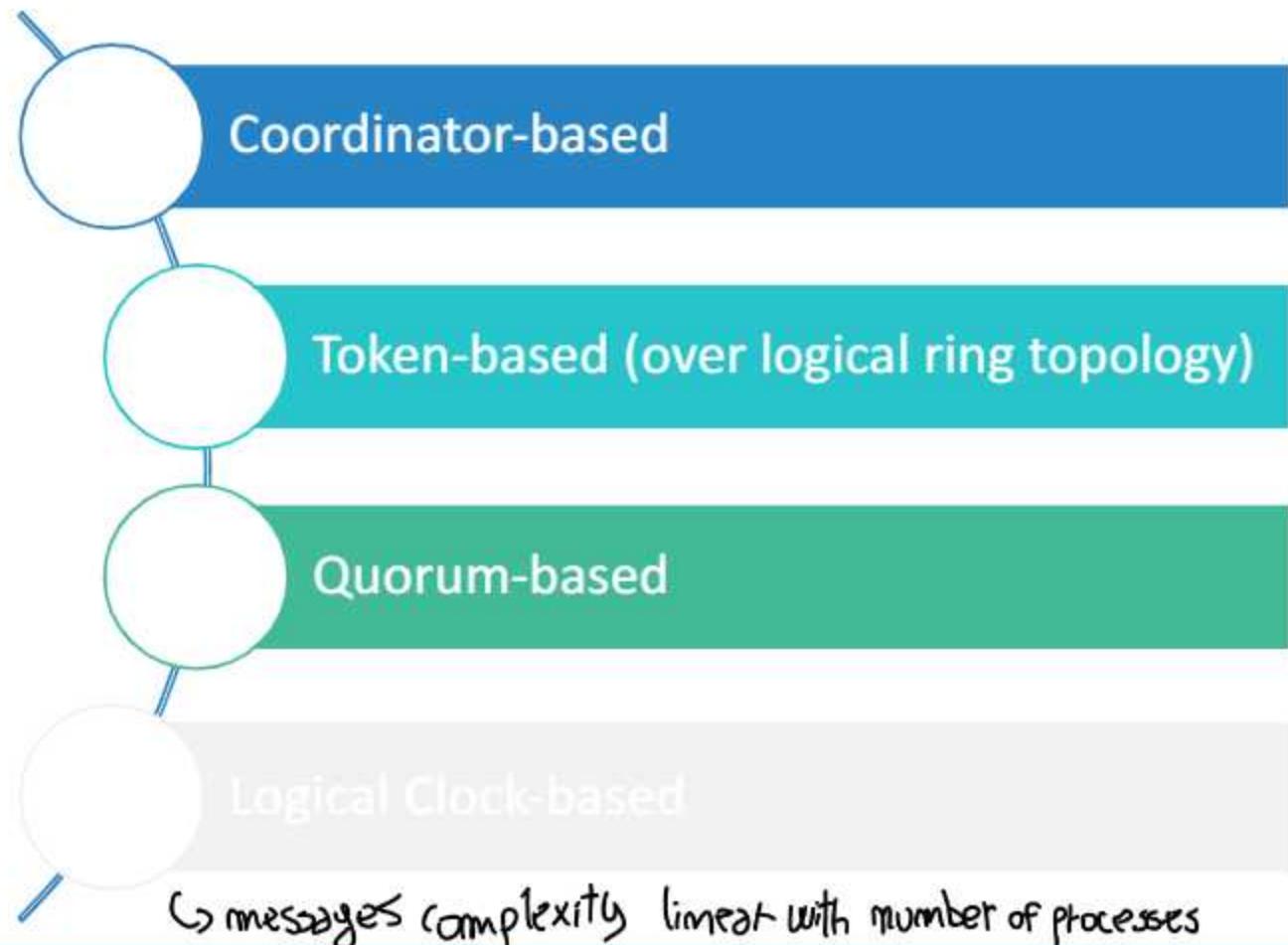


## Ricart-Agrawala's algorithm: example



# Recap - Different Approaches to Distributed Mutual Exclusion

---



every process have exactly  
some role , FIFO policy  
based on causality between request

# Coordinator-based Distributed Mutual Exclusion

sat: sfy no deadlock, always someone is selected, and no starvation if FIFO policy is used in pending

help synchronization of the processes

## BASIC IDEA

- There exist a special process (i.e., a coordinator) that collects requests and grant permission to enter into the critical section

**local variable**

```
Init
state = idle
coordinator = getCoordinatorId() → id of coordinator

upon event request()
state = waiting
trigger pp2pSend(REQ, i) to coordinator

upon event pp2pDeliver(GRANT_CS)
state = CS
trigger ok()

upon event release()
state = idle
trigger pp2pSend(REL, i) to coordinator
```

state in which the process is  
→ state in which the process is  
→ id of coordinator

~ left from CS, communicate it to coordinator

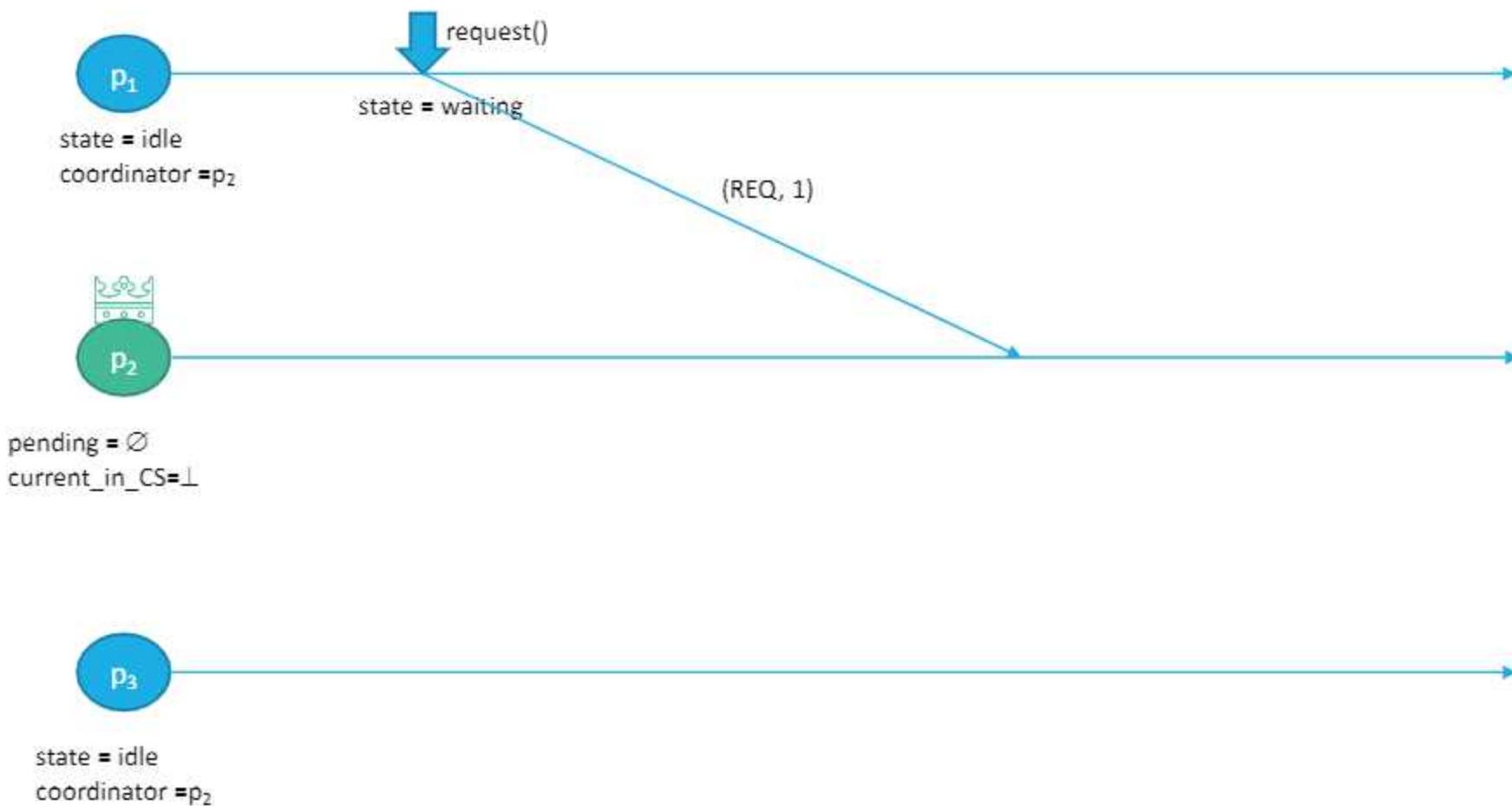
```
Init → request for CS
pending = ∅
current_in_CS = ⊥ → current process in CS

upon event pp2pDeliver(REQ, j) from pj
pending = pending ∪ {pj}

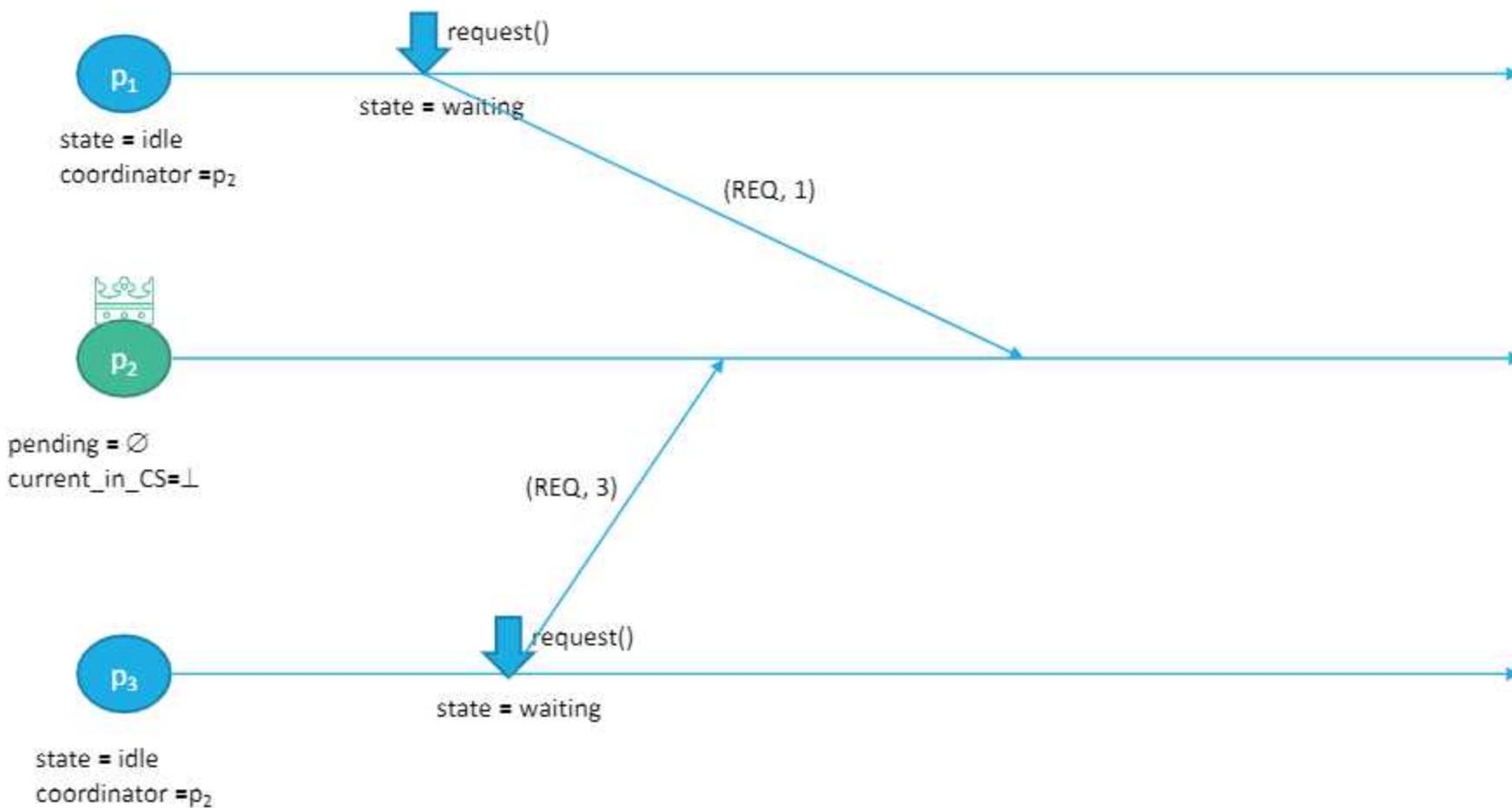
when pending ≠ ∅ and current_in_CS = ⊥
candidate = select_process(pending)
pending = pending \ candidate
current_in_CS = candidate
trigger pp2pSend(GRANT_CS) to candidate

upon event pp2pDeliver(REL, j) from pj
if current_in_CS = j
current_in_CS = ⊥
```

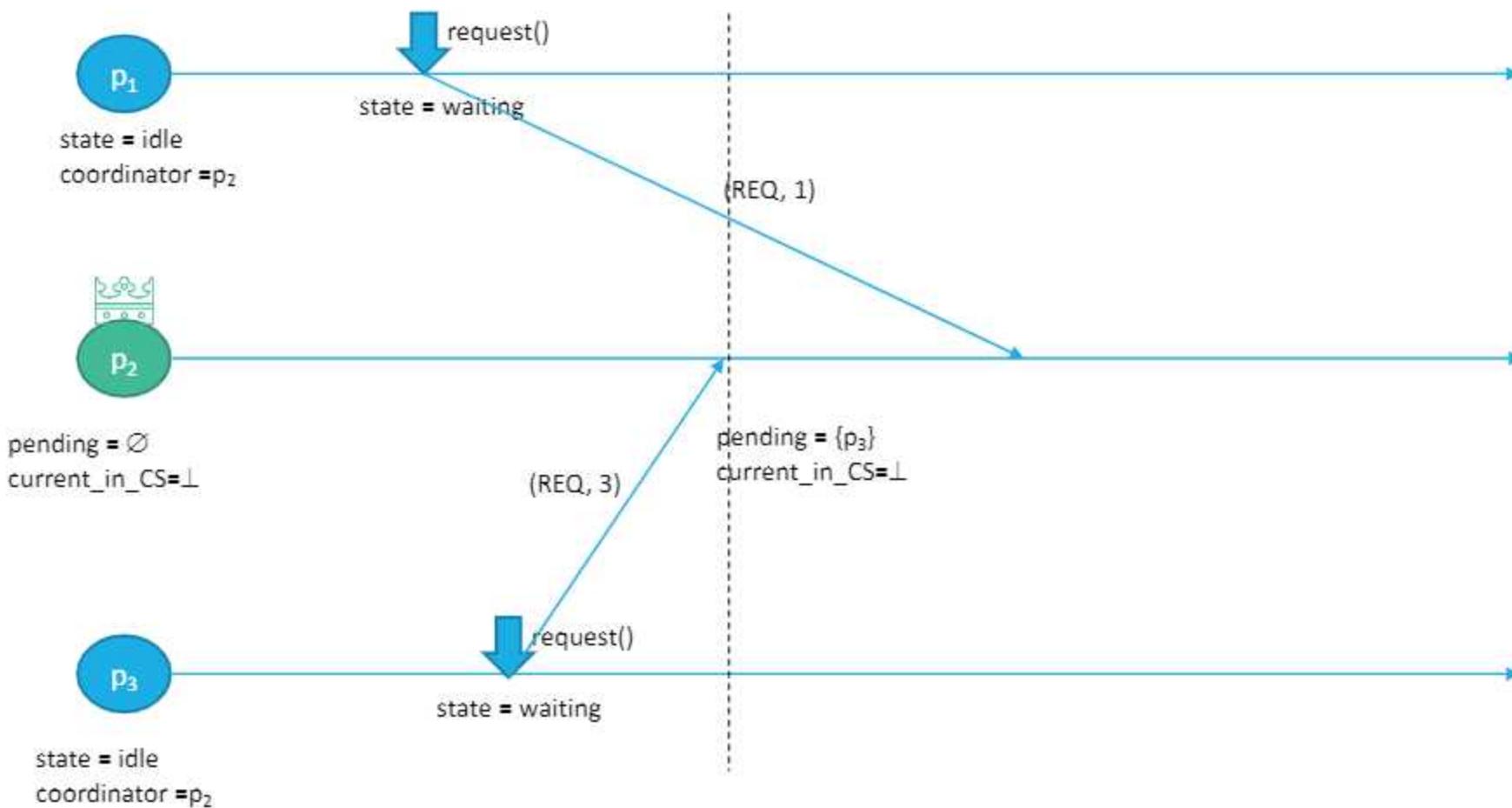
# Example



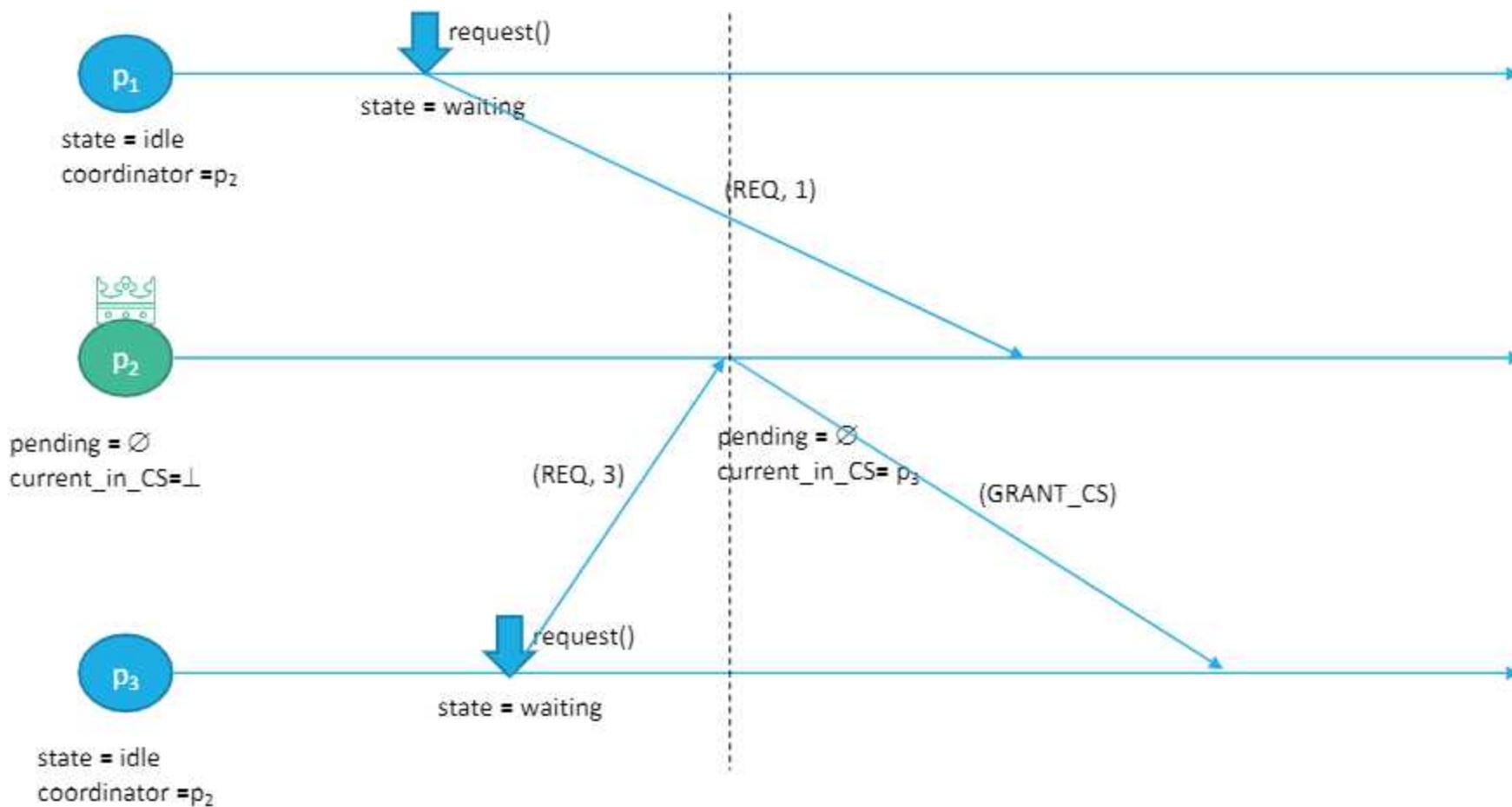
# Example



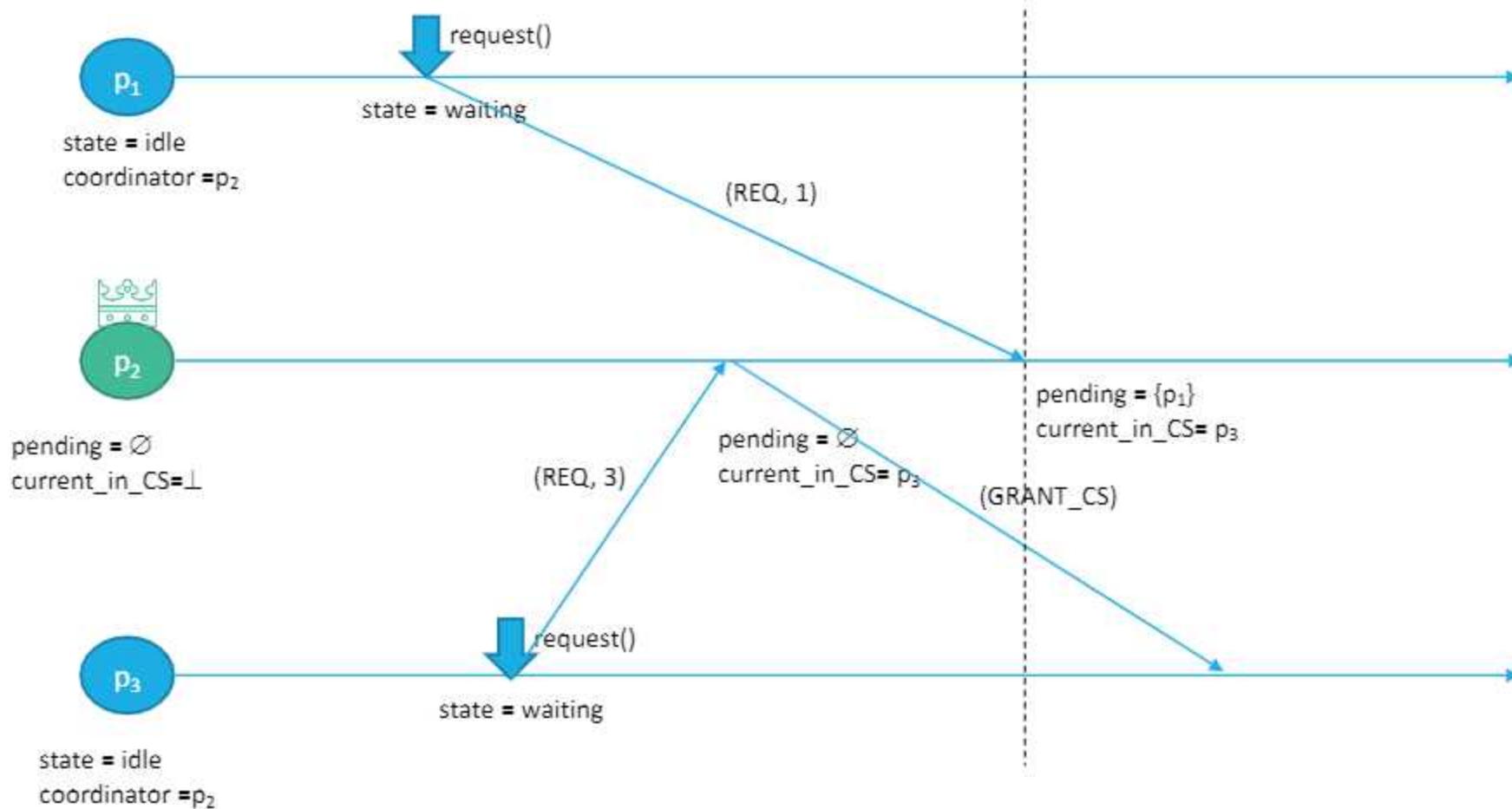
# Example



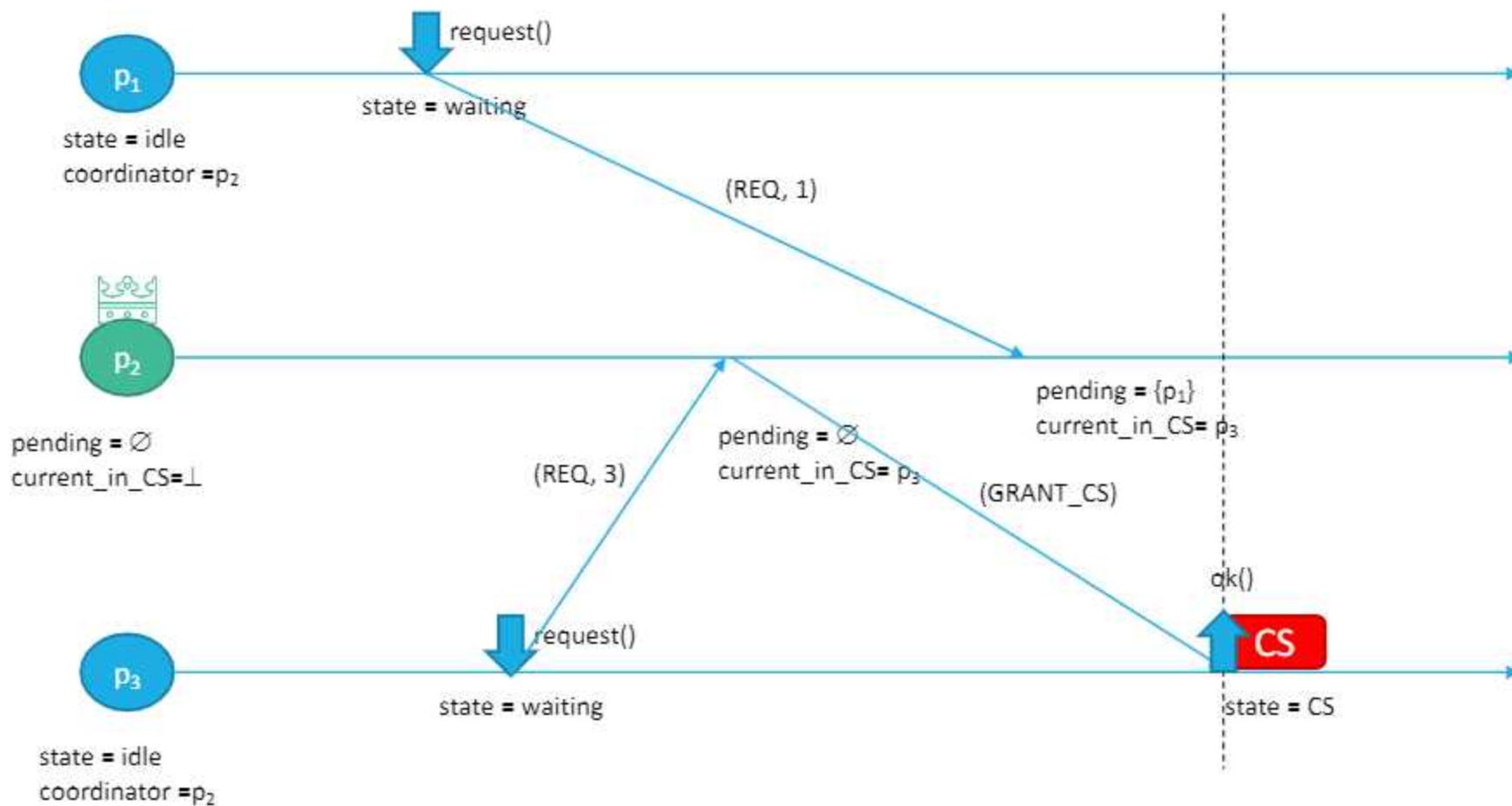
# Example



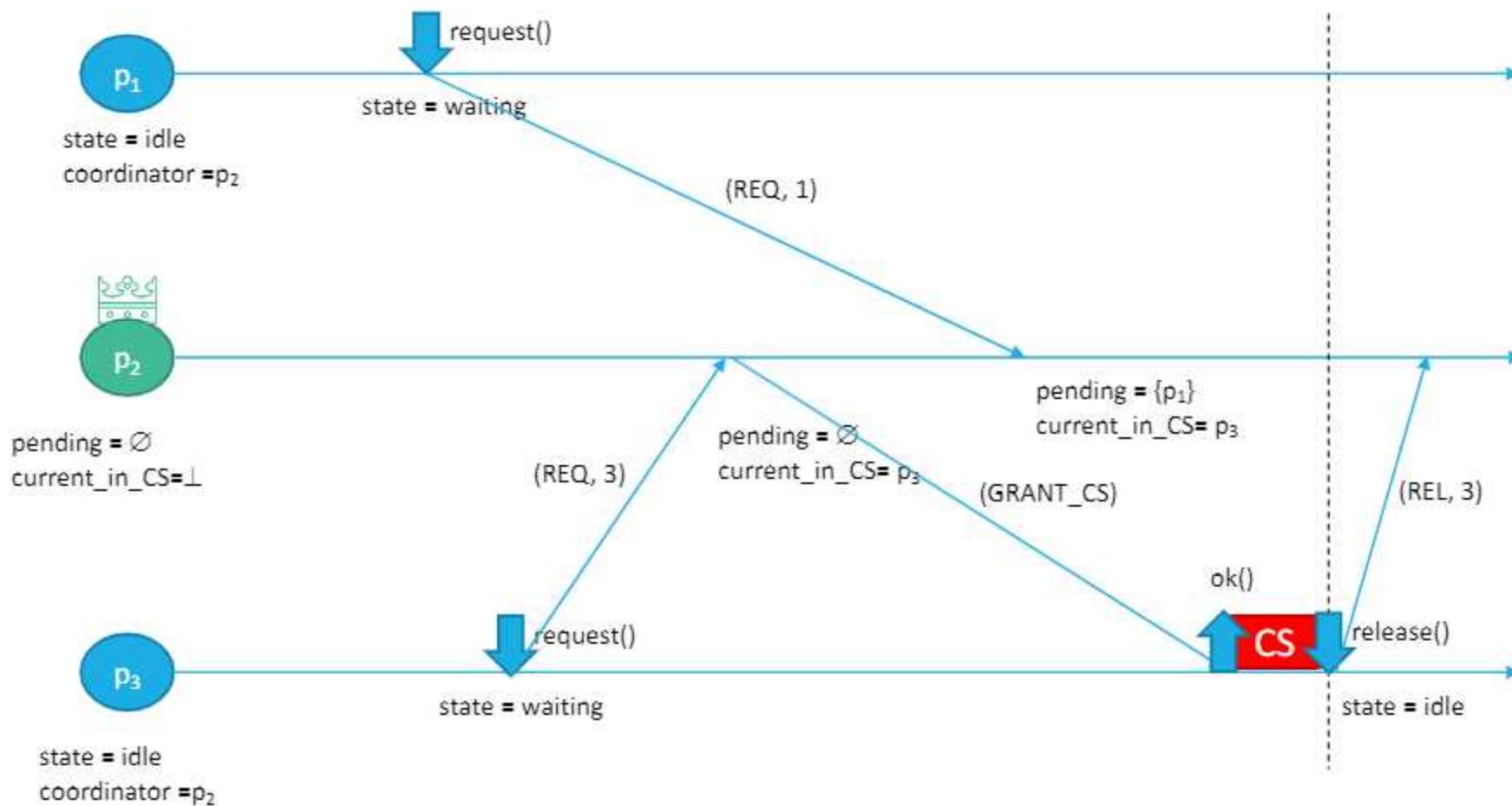
# Example



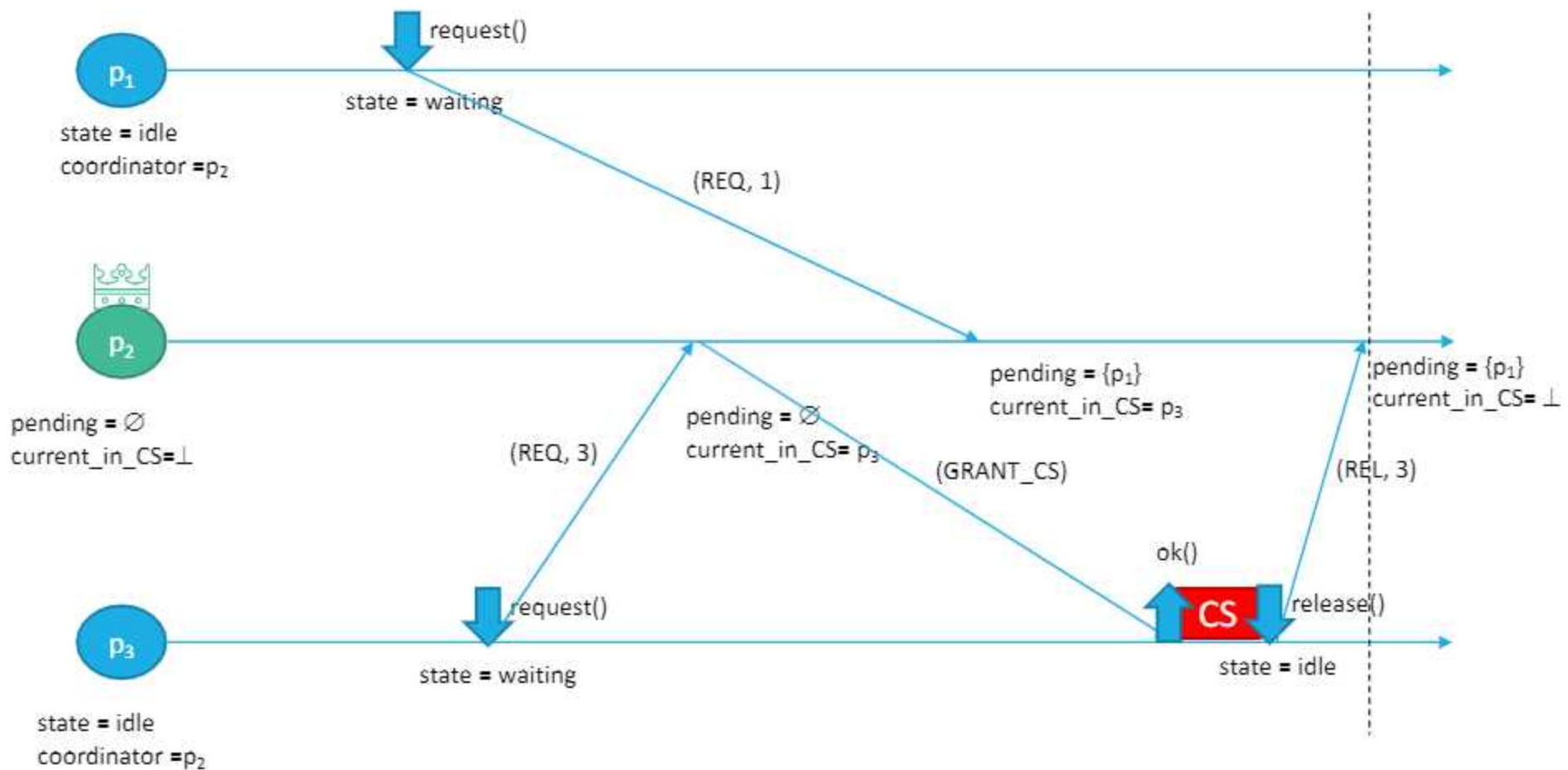
# Example



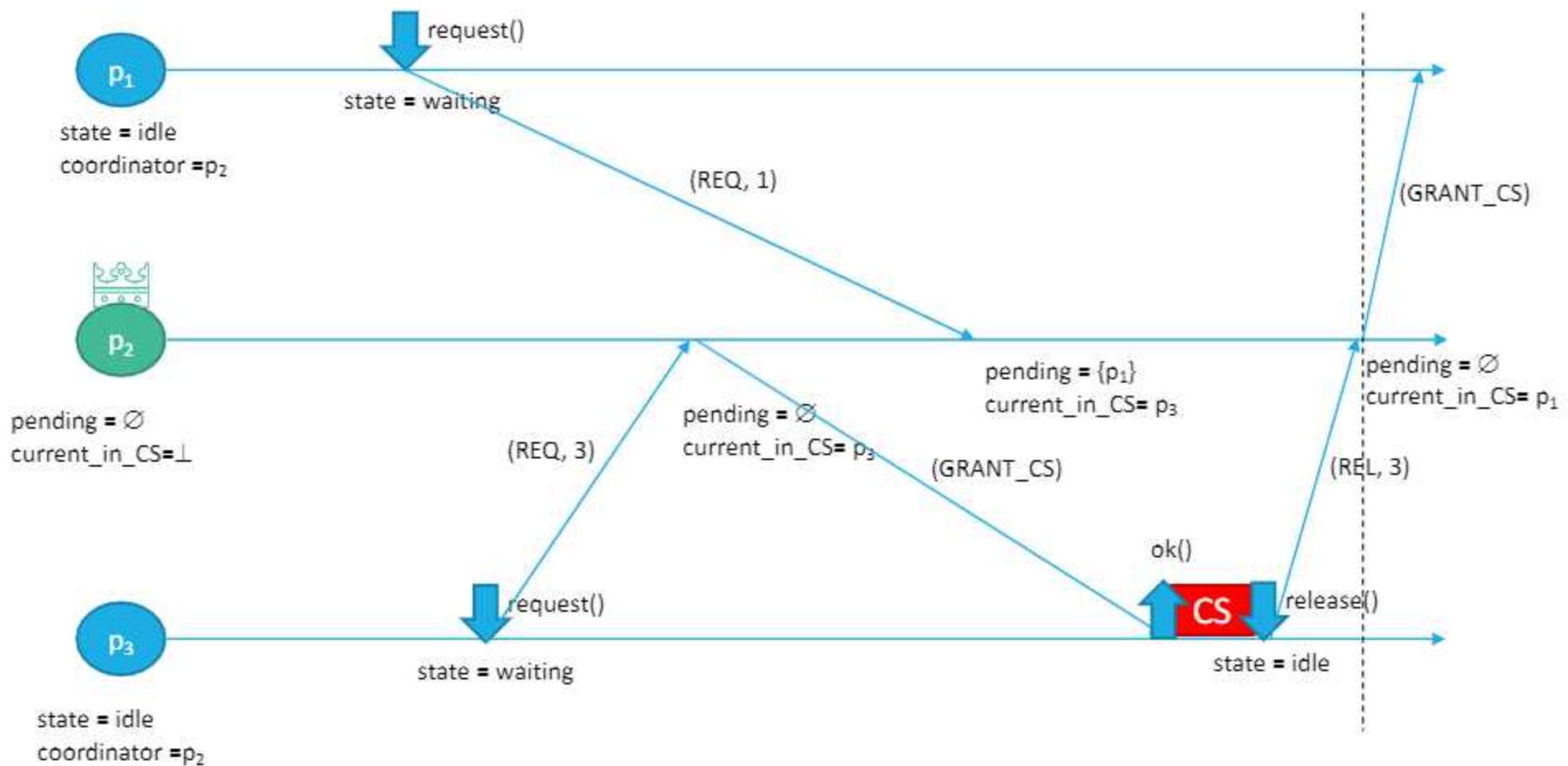
# Example



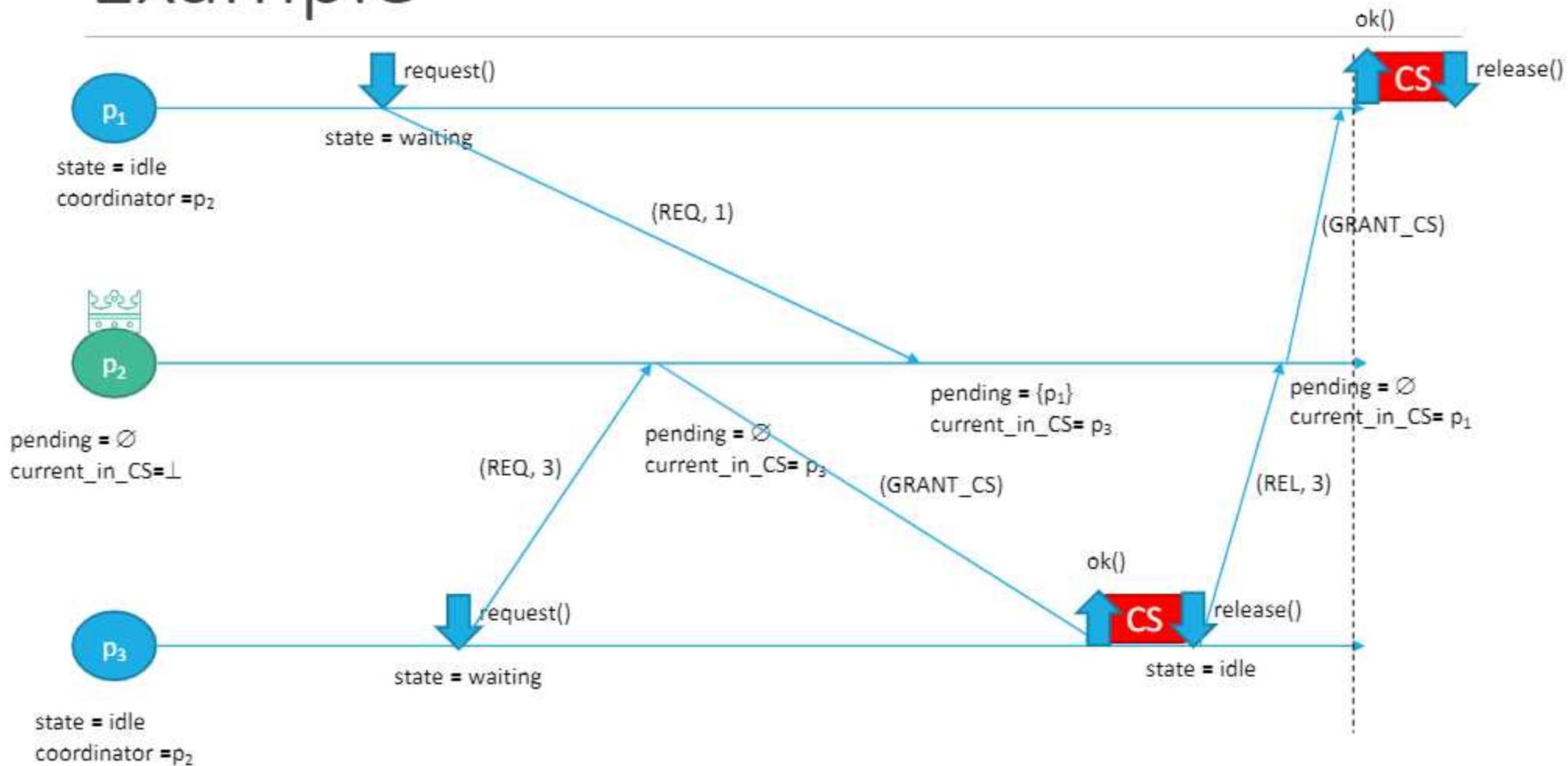
# Example



# Example



# Example



messages complexity is constant  $O(1)$  only two messages for entry in CS

# Discussion *analysis of the algorithm*

## PERFORMANCE

- entering the CS always requires 2 messages (i.e., REQ and GRANT) taking one RTT
- releasing the CS only requires 1 message
  - such message represent the delay between two different accesses to the CS

by the coordinator  
↑



- if the system is very large, the coordinator could be overloaded became a bottleneck
- if failed coordinator the system stuck no one else that could coordinate access to CS

don't exist a special process but an extra data structure that manage access to CS

## Token-based Algorithm on Logical Ring

### BASIC IDEA

- a process interested to the CS can access it only when it receives a token
- The token is unique and it is exchanged between processes
- to guarantee fairness, we can exploit a structured logical topology (i.e., a ring) for exchanging messages related to the mutual exclusion protocol

# Token-based Algorithm on Logical Ring

---

## INTUITION OF THE ALGORITHM

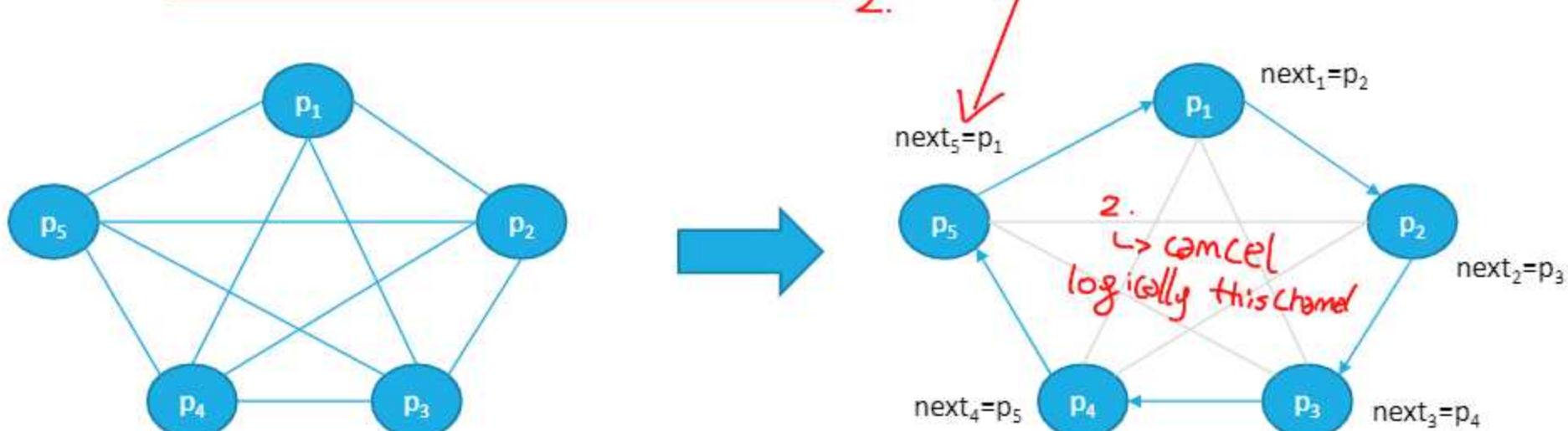
1. we construct an overlay (i.e., a logical network) as a ring exploiting existing point-to-point communication channels
2. A token is created and inserted in the ring during the initialization phase (i.e., it is assigned to a process of the system)
3. When a process requests the CS
  - a. it waits until it gets the token
  - b. enter the CS and upon release it sends the token to its next on the ring
4. If a process receives the token and it is not interested in the CS, it simply passes it to the next in the ring

# Token-based Algorithm on Logical Ring

## INTUITION

1. we construct an overlay (i.e., a logical network) as a ring exploiting existing point-to-point communication channels
  - The ring is obtained by:
    - storing in a local variable the name of the next process in the ring and
    - allowing the communication only with the next

*r>talk with only the neighbour*

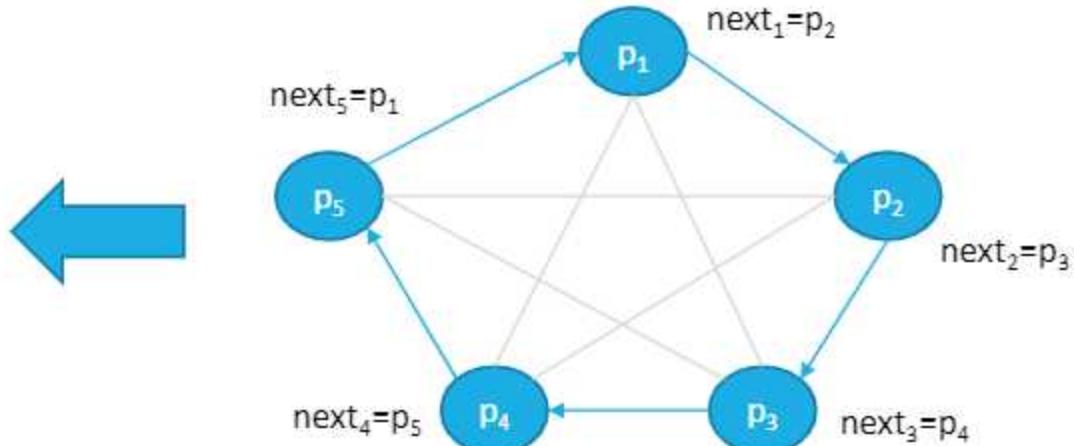


# Token-based Algorithm on Logical Ring

## INTUITION

1. we construct an overlay (i.e., a logical network) as a ring exploiting existing point-to-point communication channels
  - The ring is obtained by:
    - storing in a local variable the name of the next process in the ring and
    - allowing the communication only with the next

```
Init
state = idle
next = p(i+1)mod N
...  
define next
```



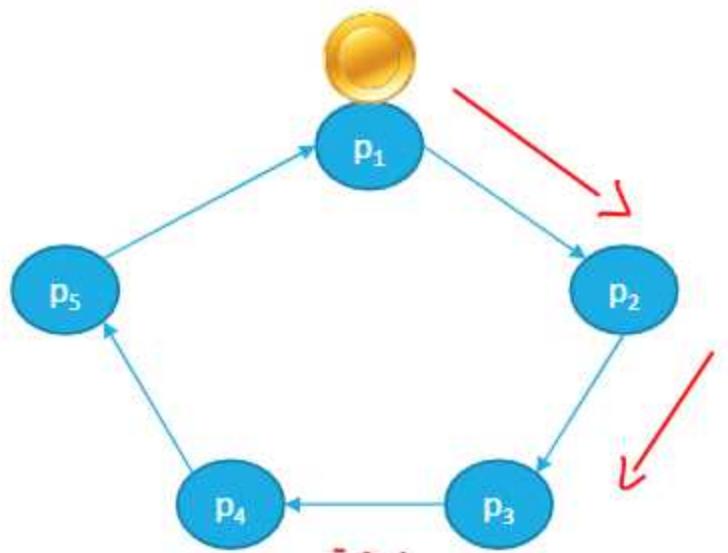
# Token-based Algorithm on Logical Ring

## INTUITION OF THE ALGORITHM

2. A token is created and propagated in the ring during the initialization phase (i.e., it is assigned to a process of the system)

**WARNING: The token must be unique to guarantee mutual exclusion**

Only one process (selected through a deterministic function) can create the token during the init



### Init

state = idle

next =  $p_{(i+1) \bmod N}$

if self =  $p_0$

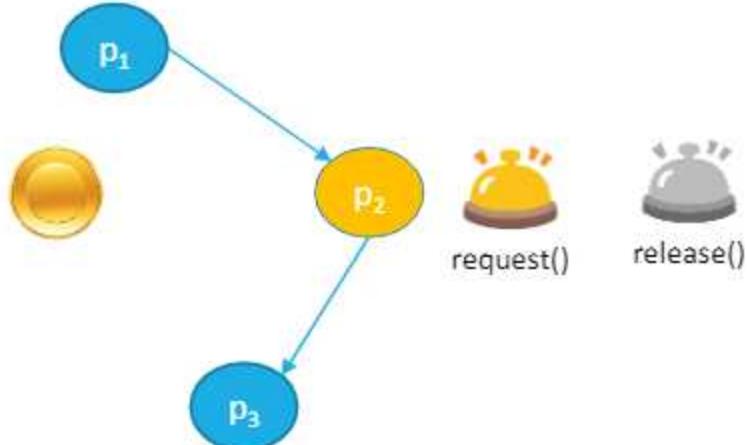
trigger pp2pSend(TOKEN) to next

...

# Token-based Algorithm on Logical Ring

## INTUITION OF THE ALGORITHM

3. When a process requests the CS
  - a. it waits until it gets the token
  - b. enter the CS and upon release it sends the token to its next on the ring
4. If a process receives the token and it is not interested in the CS, it simply passes it to the next in the ring



```
upon event request()
    state = waiting for the token

upon event pp2pDeliver(TOKEN)
    if state == waiting
        state = CS
        trigger ok()
    else
        trigger pp2pSend(TOKEN) to next → pass on next on the ring

upon event release()
    state = idle
    trigger pp2pSend(TOKEN) to next
```

# Token-based Algorithm on Logical Ring

*complete algorithm ↓*

```
Init
state = idle
next = p(i+1)mod N
if self = p0
    trigger pp2pSend(TOKEN) to next

upon event request()
    state = waiting

upon event pp2pDeliver(TOKEN)
    if state == waiting
        state = CS
        trigger ok()
    else
        trigger pp2pSend(TOKEN) to next

upon event release()
    state = idle
    trigger pp2pSend(TOKEN) to next
```

*to the application that turn mutual exclusion*

# Discussion *on performances*

*number  
of  
messages*

The algorithm continuously consume communication resources (even if no one is interested to the CS)

↳ continuously turn over the network, not the best solution

*latency*

The delay experienced by every process between the request and the grant varies between 0 (it just receives the token) and N messages (it just forwarded the token)

↳ good if system is small, remove the problem of one point failure and there isn't a special process. Usefull for collect information and manage it.

very high overhead but simple algorithm

# Quorum-based Algorithm – Maekawa's voting algorithm

→ util when need  
to manage failure

**BASIC IDEA** not asking permission to everybody, but only to a part, a **quorum**

- to enter the CS every process waits to get the acknowledgement only by a subset of processes large enough to guarantee conflicts

→ need to vote for permission

Each process  $p_i$  has associated a **voting set**  $V_i$

- Voting sets must satisfy the following properties
  - $p_i \in V_i$
  - $\forall i, j, V_i \cap V_j \neq \emptyset$  (i.e., there is at least one common member for each pair of voting sets)
  - $|V_i| = K$  (voting sets have all the same size for fairness – same load principle)
  - each  $p_i$  is contained in  $M$  voting sets (same responsibility principle)

that could block a  
double access to CS

for defining these groups must  
be always an intersection,  
everybody participate in the same  
number of groups

↳ equal distribution of load

# Quorum-based Algorithm – Maekawa's voting algorithm

**Init**  
state = released *or idle*  
voted = false *already voted for a request*  
 $V_i = \text{get\_voting\_set}(i)$  *guy to contact for enter CS*  
replies =  $\emptyset$   
pending =  $\emptyset$

**upon event** request()  
    state = wanted  
    **for each**  $p_j \in (V_i \setminus p_i)$  do  
        trigger pp2pSend(REQ, i) to  $p_j$

**upon event** pp2pDeliver(REQ, j) *give permission to someone*  
    *if* state == held OR voted == true  
        pending = pending  $\cup \{j\}$   
    *else*  
        trigger pp2pSend(ACK, i) to  $p_j$   
        voted = true

**upon event** pp2pDeliver(ACK, j)  
    replies = replies  $\cup \{j\}$

**when** |replies| ==  $|V_i| - 1$   
    state = held  
    *trigger OK()*

**upon event** release()  
    state = released  
    replies =  $\emptyset$   
    **for each**  $p_j \in (V_i \setminus p_i)$  do  
        trigger pp2pSend(REL, i) to  $p_j$

**upon event** pp2pDeliver(REL, j)  
    *if* |pending| > 0  
        candidate = select\_next(pending)  
        pending = pending \ candidate  
        *trigger pp2pSend(ACK, i) to candidate*  
        voted = true  
    *else*  
        voted = false

# Quorum-based Algorithm – Maekawa's voting algorithm

---

CHALLENGE: How to compute the values K and M to balance load and responsibilities?

- Maekawa showed that the optimal solution which minimize k and allows to get ME is having
  - $K \sim \sqrt[2]{N}$  *processes for groups*
  - $M = K$  *every process participate in K groups*
- An approximation to define  $V_i$  is having sets such that  $|V_i| \sim 2\sqrt[2]{N}$  defined as follows
  - Place processes in a matrix of size  $\sqrt[2]{N} \times \sqrt[2]{N}$
  - for each  $p_i$ , let  $V_i$  be the union of the rows and columns containing  $p_i$

# Example

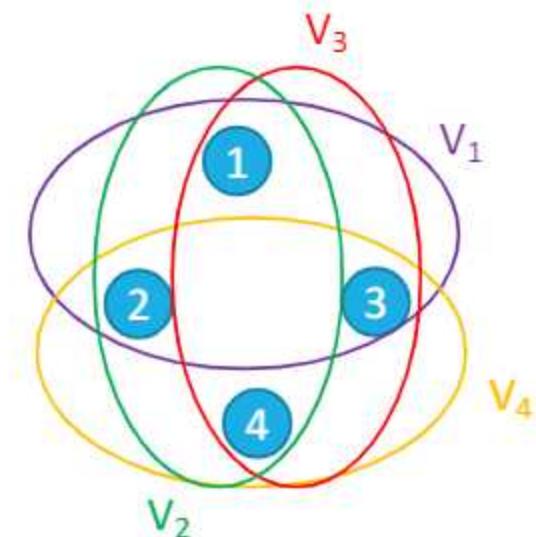
Let us consider a system composed by  $N = 4$  processes

- ( $M = K \sim \sqrt[2]{4}$  and  $|V_i| \sim 2\sqrt[2]{4}$ )

Let's place processes in the matrix and compute  $V_i$

p1	p2
p3	p4

Process id	V
1	p <sub>1</sub> , p <sub>2</sub> , p <sub>3</sub>
2	p <sub>1</sub> , p <sub>2</sub> , p <sub>4</sub>
3	p <sub>1</sub> , p <sub>3</sub> , p <sub>4</sub>
4	p <sub>2</sub> , p <sub>3</sub> , p <sub>4</sub>



# Example

You asking to  $\sqrt{m}$  process in average  
and not  $m$  like in first algorithm

Let us consider a system composed by  $N = 9$  processes

- ( $M = K \sim \sqrt[2]{9}$  and  $|V_i| \sim \sqrt[2]{9} = 6$ )

↪ about here is 5

Let's place processes in the matrix

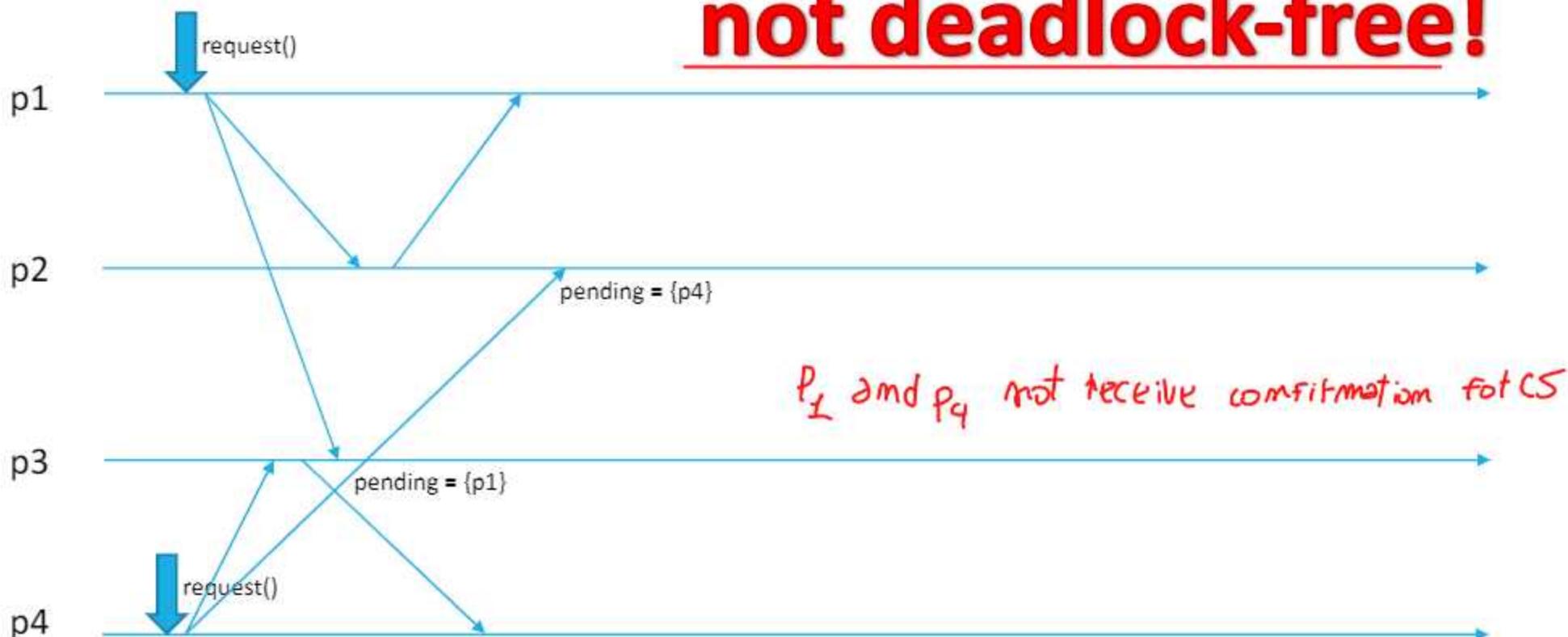
p1	p2	p3
p4	p5	p6
p7	p8	p9

Process id	V
1	p <sub>1</sub> , p <sub>2</sub> , p <sub>3</sub> , p <sub>4</sub> , p <sub>7</sub>
2	p <sub>1</sub> , p <sub>2</sub> , p <sub>3</sub> , p <sub>5</sub> , p <sub>8</sub>
3	p <sub>1</sub> , p <sub>2</sub> , p <sub>3</sub> , p <sub>6</sub> , p <sub>9</sub>
4	p <sub>4</sub> , p <sub>5</sub> , p <sub>6</sub> , p <sub>1</sub> , p <sub>7</sub>
5	p <sub>4</sub> , p <sub>5</sub> , p <sub>6</sub> , p <sub>2</sub> , p <sub>8</sub>
6	p <sub>4</sub> , p <sub>5</sub> , p <sub>6</sub> , p <sub>3</sub> , p <sub>9</sub>
7	p <sub>7</sub> , p <sub>8</sub> , p <sub>9</sub> , p <sub>1</sub> , p <sub>4</sub>
8	p <sub>7</sub> , p <sub>8</sub> , p <sub>9</sub> , p <sub>2</sub> , p <sub>5</sub>
9	p <sub>7</sub> , p <sub>8</sub> , p <sub>9</sub> , p <sub>3</sub> , p <sub>6</sub>

# WARNING!

## Maekawa's Algorithm is not deadlock-free!

Example



# Reference

---

George Coulouris, Jean Dollimore and Tim Kindberg, Gordon Blair "Distributed Systems: Concepts and Design (5th Edition)". Addison - Wesley, 2012.

- Chapter 11 – section 11.2

Dependable Distributed Systems  
Master of Science in Engineering in  
Computer Science

AA 2022/2023

---

LECTURE 6: FAILURE DETECTION

# Recap on Timing Assumptions

## Synchronous

- timing assumptions are explicit on
  - Upper bounds on process execution time
  - Upper bounds on communication time ↗ every message that you exchange inside the system has an upper bound on maximum latency on delivery
  - Existence of a common global clock ↗ an upper bound on clock drift exist between all clocks in distributed system

## Asynchronous

- there are no timing assumptions

## Partial synchrony ↗ more closeness

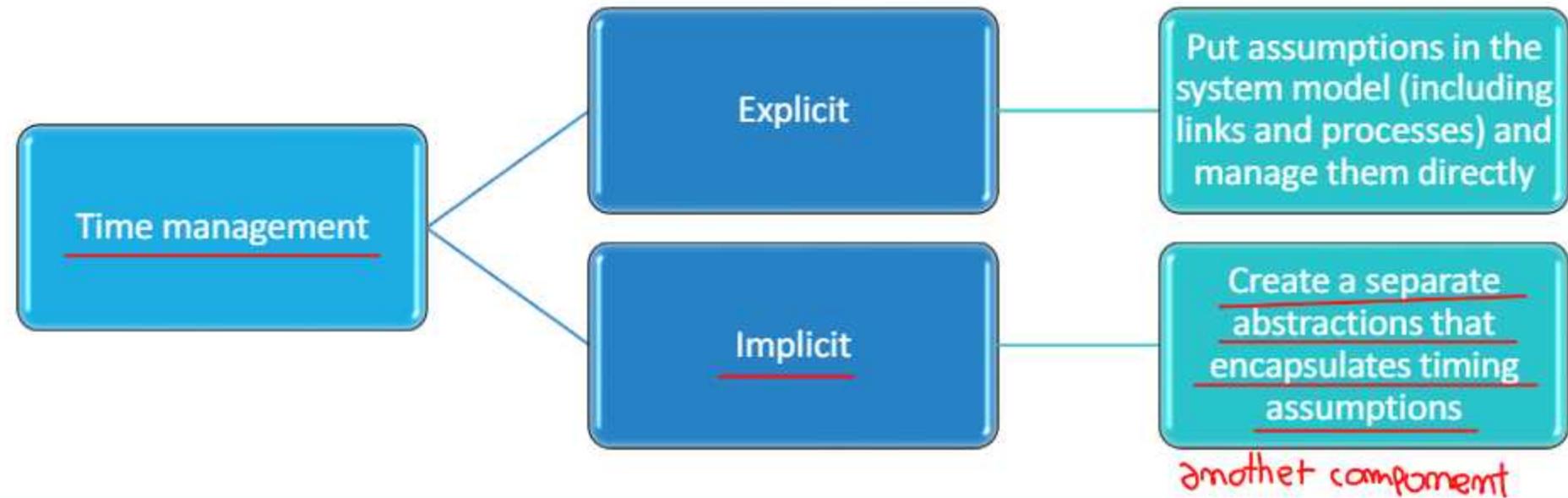
- requires abstract timing assumptions (after an unknown time t the system becomes synchronous)

Given that you have or not those time assumption on top of the system, you may directly use these bound as a variable in your protocol/algorithm. Then, explicit manage these upper bound, but another way is to manage them implicitly —> not always take care about how much time is passed from the time you send a message..., but add an additional component that notify to you each time happen something bad, like a failure detector.

# Recap on Timing Assumptions

## NOTE

- manipulating time inside a protocol/algorithm is complex and the correctness proof may become very involved and sometimes prone to errors



oracle: something that define like a component (Petri link...)

# Failure Detector Abstraction

A Failure Detector is an oracle providing information about the failure state of a process (the other process in the system)

- it is a software module to be used together with process and link abstractions
- It encapsulates timing assumptions of either partially synchronous or fully synchronous system
- Stronger are the timing assumption, more accurate the information provided by a failure detector will be

A failure detector is generally described by two properties:

- Accuracy (informally is the ability to avoid mistakes in the detection)
- Completeness (informally is ability to detect all failures)

Detect a specific type of failure, **crash Failure**: you have a set of process and some of that just stop working, not repaired, some processes may fail at a time t

# Failure Detectors ~~Classification~~

↪ not make mistakes

Correct processes are never suspected

Some correct process is never suspected

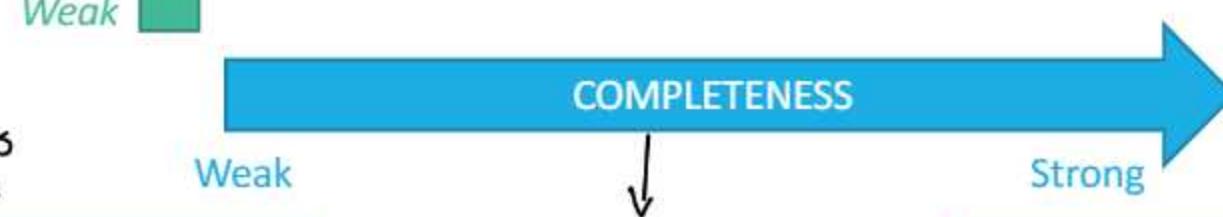
↪ few correct process is suspected to be crash

Eventually every process that crashes is permanently suspected by some correct process

↪ not all notice a crash

several kind of failure detectors

Q (Quasi-perfect Failure Detector)	P (Perfect Failure Detector)
W (Weak Failure Detector)	S (Strong Failure Detector)



how many correct process notice a particular process has crashed, something bad has happened.

Eventually every process that crashes is permanently suspected by every correct process

# Failure Detectors Classification

---

## OBSERVATION

- W guarantees that there is at least one correct process that is never suspected.
- Practically speaking, this could be difficult to achieve and thus it is worth to consider accuracy properties to hold eventually

## EVENTUAL STRONG ACCURACY

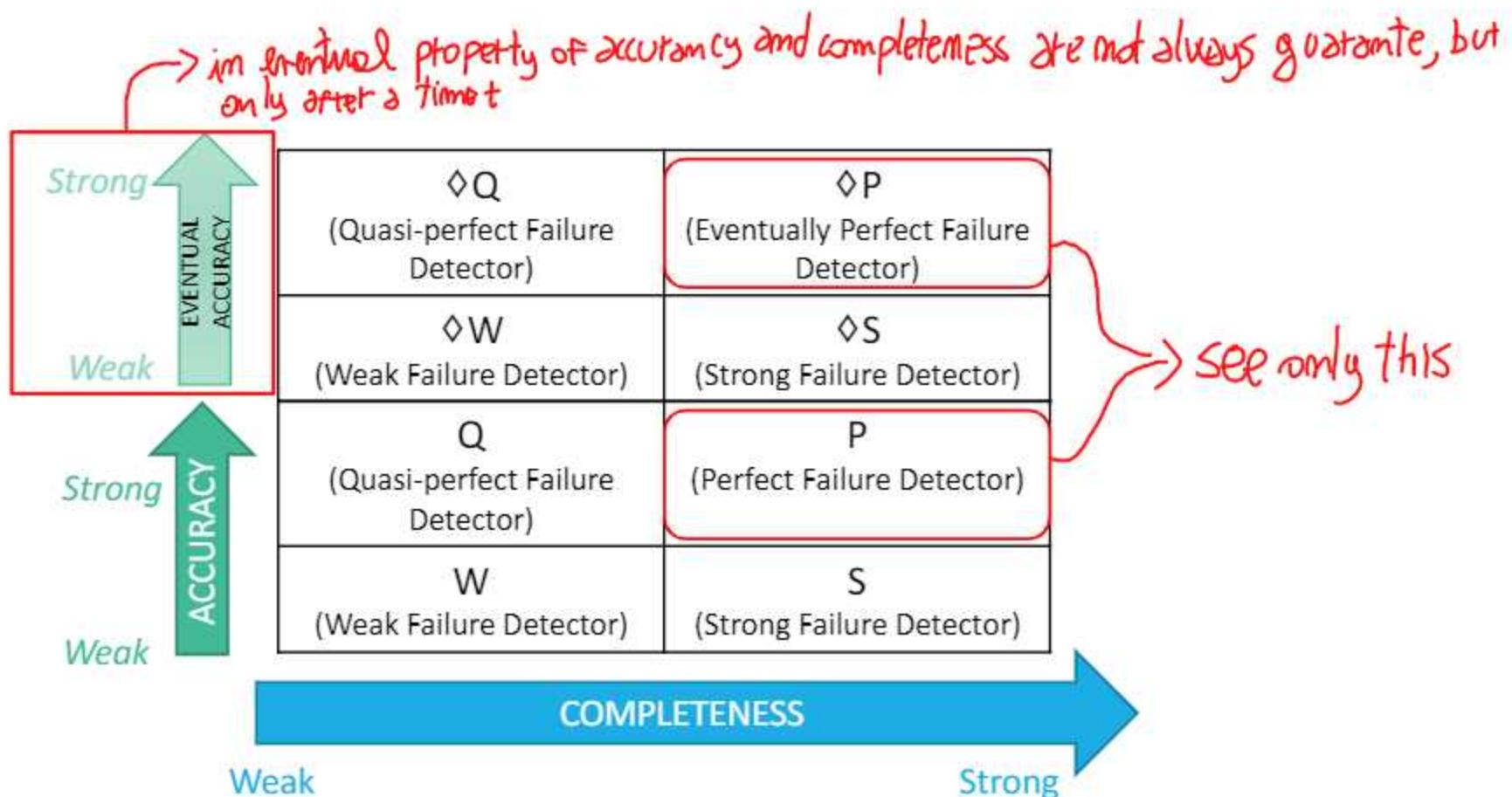
- There is a time after which correct processes are not suspected

## EVENTUAL WEAK ACCURACY

- There is a time after which some correct process is not suspected

EVENTUAL → after a time + the failure detector act like it should be

# Failure Detectors Classification



# Perfect Failure detectors (P)

---

## System model

- synchronous system *→ all the strong assumption on the system*
- crash failures

Using its own clock and the bounds of the synchrony model, a process can infer if another process has crashed

---

# Perfect failure detectors (P) Specification

---

## Module 2.6: Interface and properties of the perfect failure detector

---

- Module:

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

- Events:

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

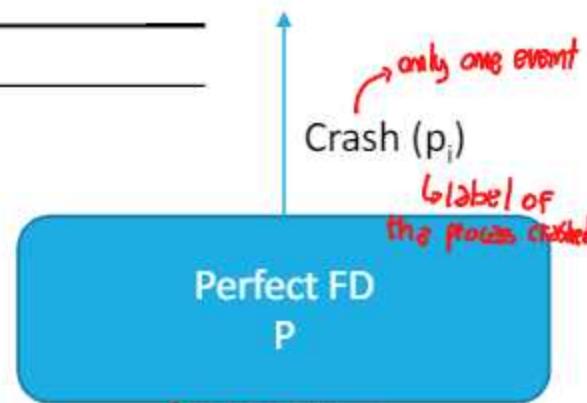
Properties:

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.

→ after a time  $t$

no mistake in detection



# Perfect failure detectors (P) Implementation

Algorithm 2.5: Exclude on Timeout

Implements:

PerfectFailureDetector, instance  $\mathcal{P}$ .

Uses:

PerfectPointToPointLinks, instance  $pl$ .

link that:

not loose messages and always delivered between correct processes

upon event  $\langle \mathcal{P}, \text{Init} \rangle$  do

$alive := \Pi$ ;  $\rightarrow$  all processes in the system

$detected := \emptyset$ ;  $\rightarrow$  initially empty, all processes detected crashed

$\text{startimer}(\Delta)$ ;

upon event  $\langle \text{Timeout} \rangle$  do

forall  $p \in \Pi$  do  $\rightarrow$  in the system

if  $(p \notin alive) \wedge (p \notin detected)$  then

$detected := detected \cup \{p\}$ ;

trigger  $\langle \mathcal{P}, \text{Crash} \mid p \rangle$ ;

trigger  $\langle pl, \text{Send} \mid p, [\text{HEARTBEATREQUEST}] \rangle$ ;

$alive := \emptyset$ ;

$\rightarrow$  ask to  $p$  if he is alive

$\text{startimer}(\Delta)$ ;

upon event  $\langle pl, \text{Deliver} \mid q, [\text{HEARTBEATREQUEST}] \rangle$  do

trigger  $\langle pl, \text{Send} \mid q, [\text{HEARTBEATREPLY}] \rangle$ ;

upon event  $\langle pl, \text{Deliver} \mid p, [\text{HEARTBEATREPLY}] \rangle$  do

$alive := alive \cup \{p\}$ ;

just send a message to every process in the system and wait for an answer for a certain timeout. After timeout  $\Delta$  concludes that a process is crashed

$\Delta$  timer is equal or greater to  $RTT + \text{Processing time}$

Crash ( $p_i$ )

Perfect FD  
 $\mathcal{P}$

pp2pSend (msg) pp2pDeliver(msg)

send a message to a specific process

Perfect Point-to-point Link

to deliver a message from a specific sender only once.

if not use detected variable, you modify more time that a process p is crashed

## Correctness

---

- To prove the correctness, we must prove that both Strong Completeness and Strong Accuracy are satisfied
- What if links are fair loss? *something bad may happen, maybe you send a message and everyone is correct, but you loose the message, will be detected crashed*
- What if we select a timeout too long? *↳ no problem, but bad performance*
- What if we select a timeout too short?

for guarantee Strong A and C. At necessary  $2\Delta$

# Eventually perfect failure detectors ( $\Diamond P$ )

---

## System model

- partial synchrony *relax synchrony assumption*
  - Crash failures
  - Perfect point-to-point links
- 

Crashes can be accurately detected only after a (unknown) time  $t$

- Before time  $t$  the systems behaves as an asynchronous one
- The failure detector may make mistake before time  $t$  considering correct processes as crashed.
- The notion of detection becomes suspicious

# Eventually perfect failure detectors ( $\diamond P$ ) Specification

---

**Module 2.8:** Interface and properties of the eventually perfect failure detector

---

**Module:**

**Name:** `EventuallyPerfectFailureDetector`, **instance**  $\diamond P$ .

**Events:**

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

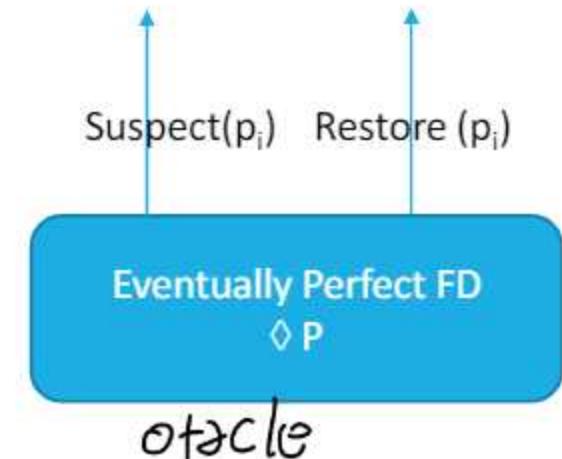
**Indication:**  $\langle \diamond P, \text{Restore} \mid p \rangle$ : Notifies that process  $p$  is not suspected anymore.

**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.

---



# Basic constructions rules of an eventually perfect FD

---

Use timeouts to suspect processes that did not sent expected messages

---

A suspect may be wrong

- A process  $p_i$  may suspect another one  $p_j$  as the current timeout is too short

*the message is not lost, use perfect link*

- ◊  $P$  is ready to change its judgment as soon as it receives a message from  $p_j$ 
  - In this case, the timeout value is updated

If  $p_j$  has actually crashed,  $p_i$  does not change its judgment anymore

---

# Eventually perfect failure detectors ( $\diamond P$ ) Implementation

Algorithm 2.7: Increasing Timeout

Implements:

EventuallyPerfectFailureDetector, instance  $\diamond P$ .

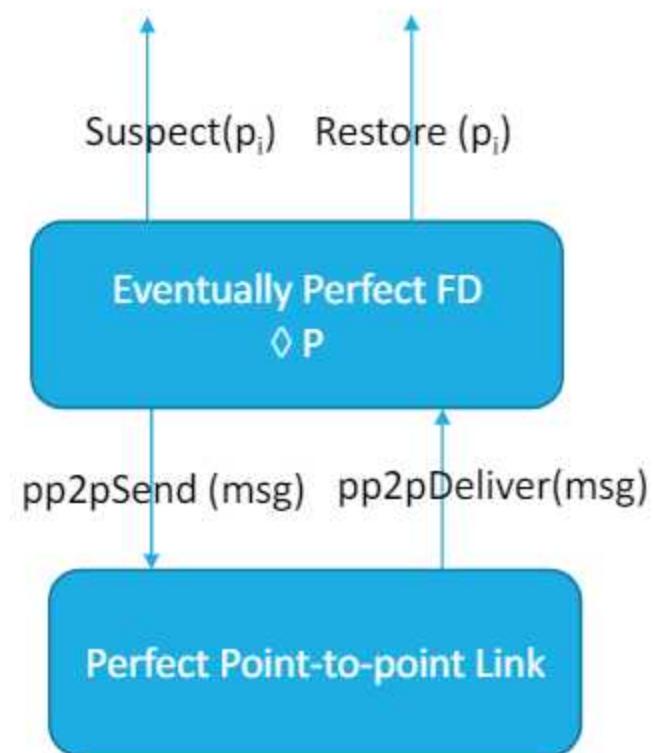
we have increasing delay

Uses:

PerfectPointToPointLinks, instance  $pl$ .

```
upon event <  $\diamond P$ , Init > do
    alive :=  $\Pi$ ;
    suspected :=  $\emptyset$ ; ↳ suspected to be crashed
    delay :=  $\Delta$ ; ↳ timeout is not static
    starttimer(delay);

upon event < Timeout > do
    if alive  $\cap$  suspected  $\neq \emptyset$  then
        delay := delay +  $\Delta$ ; ↳ change timeout double it
    forall  $p \in \Pi$  do
        if ( $p \notin alive$ )  $\wedge$  ( $p \notin suspected$ ) then
            suspected := suspected  $\cup \{p\}$ ;
            trigger <  $\diamond P$ , Suspect |  $p$  >;
        else if ( $p \in alive$ )  $\wedge$  ( $p \in suspected$ ) then
            suspected := suspected  $\setminus \{p\}$ ;
            trigger <  $\diamond P$ , Restore |  $p$  >;
        trigger <  $pl$ , Send |  $p$ , [HEARTBEATREQUEST] >;
    alive :=  $\emptyset$ ;
    starttimer(delay);
```



# Correctness

---

## Strong completeness

- If a process crashes, it will stop to send messages. Therefore the process will be suspected by any correct process and no process will revise the judgement.
- 

## Eventual strong accuracy

- After time  $T$  the system becomes synchronous. i.e., after that time a message sent by a correct process  $p$  to another one  $q$  will be delivered within a bounded time. If  $p$  was wrongly suspected by  $q$ , then  $q$  will revise its suspicion.
-

# References

---

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

- Chapter 2 – from Section 2.6.1 to Section 2.6.5

T. Chandra, S. Toueg *Unreliable Failure Detectors for Reliable Distributed Systems*

- <https://dl.acm.org/doi/pdf/10.1145/226643.226647>

Dependable Distributed Systems  
Master of Science in Engineering in  
Computer Science

AA 2021/2022

---

LECTURE 7: LEADER ELECTION

# Recap on Timing Assumptions

---

## Synchronous

- timing assumptions are explicit either on
  - Bounds on process executions and communication channels, or
  - Existence of a common global clock, or
  - Both

## Asynchronous

- there are no timing assumptions

encapsulated all synchronous assumption inside a detector that provide all the information related to a failure state of a process

in synchronous system we are able to identify correctly all the processes that failed upon a certain amount of time without any mistake. When the system became eventually synchronous (partial), you loose some degree of accuracy, move from strong accuracy to eventual strong accuracy.

# Recap on Timing Assumptions

Partial synchrony requires abstract timing assumptions (after an unknown time  $t$  the system becomes synchronous)

Two choices:

1. Put assumption on the system model (including links and processes)
2. Create a separate abstractions that encapsulates those timing assumptions

also called oracle  
↑  
detector

Note: manipulating time inside a protocol/algorithm is complex and the correctness proof may become very involved and sometimes prone to errors

the perfect failure detector always give you the system information, not make any mistake when report there is a fault process, instead eventually perfect make some mistake during the evolution. with suspect and testore there is liveness guarantee that eventually detector stop make mistake.

# An alternative

---

Sometimes, we may be interested in knowing one process that is alive instead of monitoring failures

- E.g., Need of a coordinator

→ Leader detector oracle, encapsulated the timing assumption over the system.

We can use a different oracle (called leader election module) that reports a process that is alive

not need to know the correct state of every process, if there is some correct process and give him a particular role

# Leader Election Specification

**Module 2.7:** Interface and properties of leader election

**Module:**

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

**Events:**

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

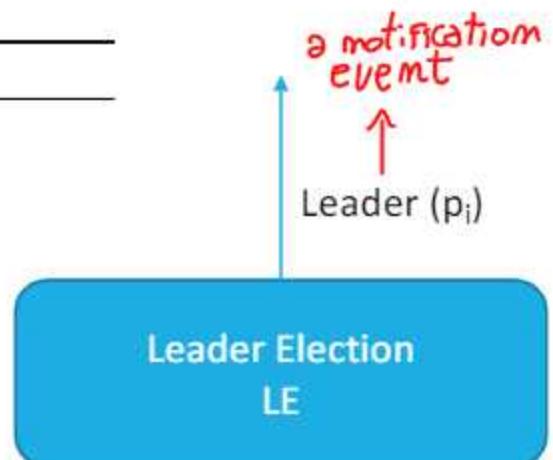
**Properties:**

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

↳ liveliness property because indicate a progress in computation, eventually take a step.

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

*↳ if you are the leader, you will trust the same leader until you are alive in the system. Always correct.*



every process trusts one leader, if you change idea, then the previous one failed.

# Leader Election Implementation

## Algorithm 2.6: Monarchical Leader Election

Implements:

LeaderElection, instance *le*.

maxrank: give process with highest Identifier

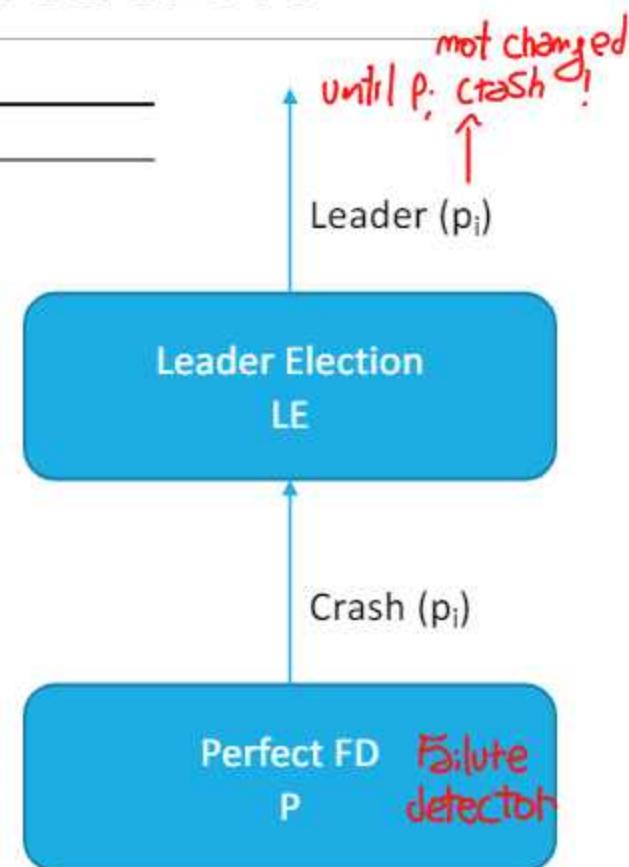
Uses:

↳ PerfectFailureDetector, instance *P*.

upon event  $\langle le, \text{Init} \rangle$  do  
     $\text{suspected} := \emptyset;$       *list of processes that became faulty during execution.*  
     $\text{leader} := \perp;$       *name of current leader.*  
immediately trigger Leader event!

upon event  $\langle P, \text{Crash} | p \rangle$  do  
     $\text{suspected} := \text{suspected} \cup \{p\};$

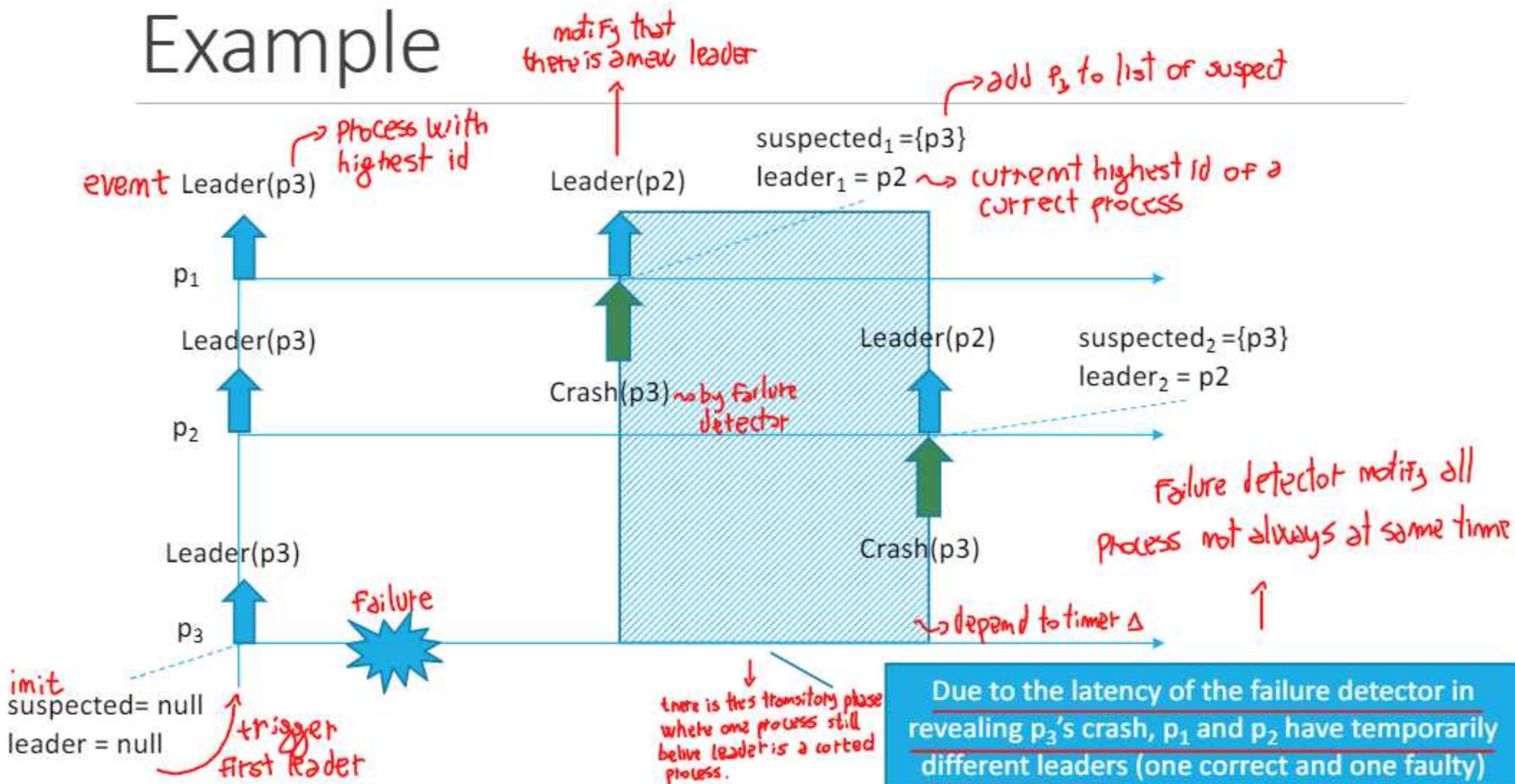
upon  $\text{leader} \neq \text{maxrank}(II \setminus \text{suspected})$  do  
     $\text{leader} := \text{maxrank}(II \setminus \text{suspected});$   
    trigger  $\langle le, \text{Leader} | \text{leader} \rangle;$



↳ this function is deterministic, is the same in every process in the system

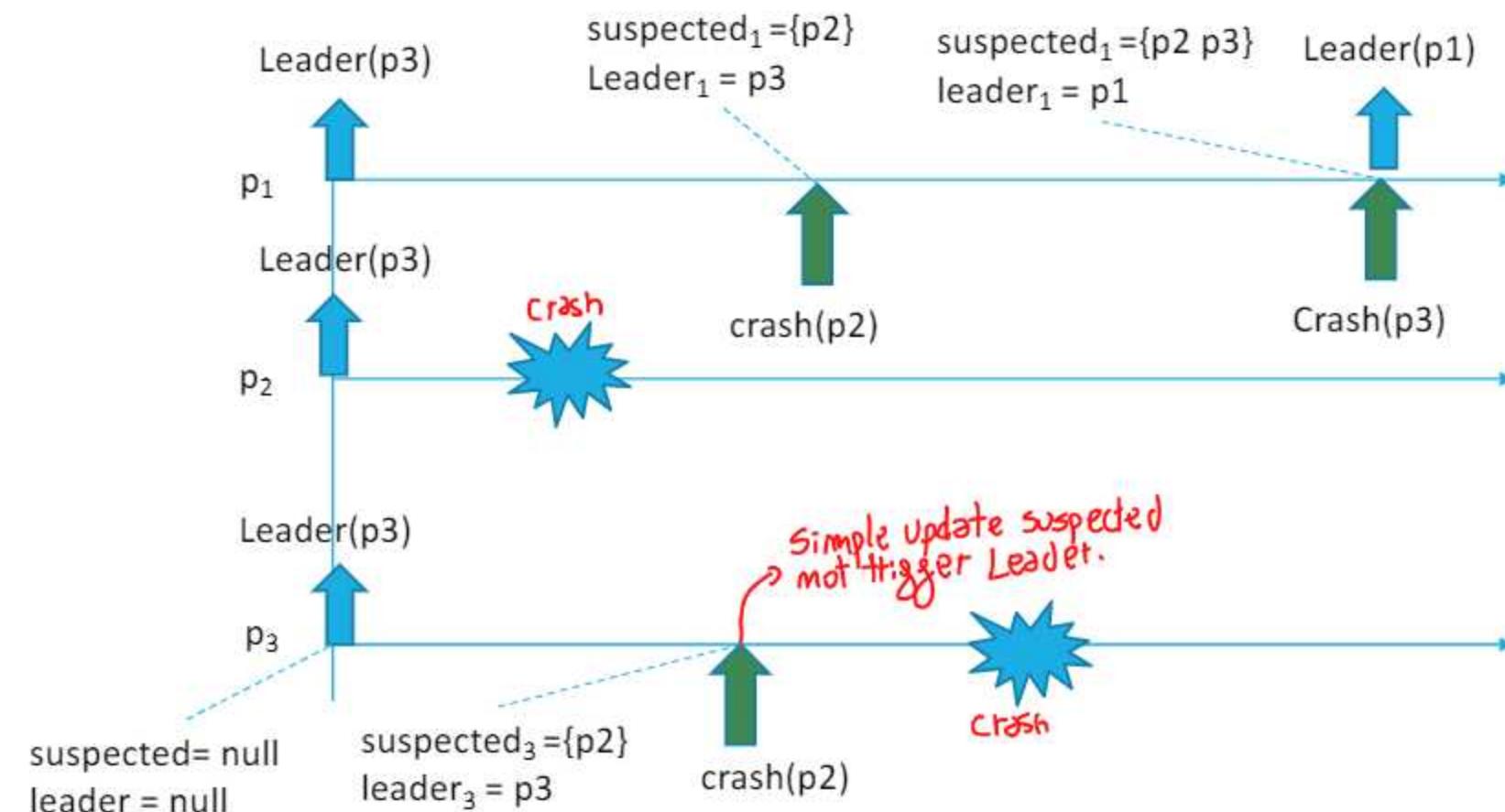
Faulty process is the leader

## Example

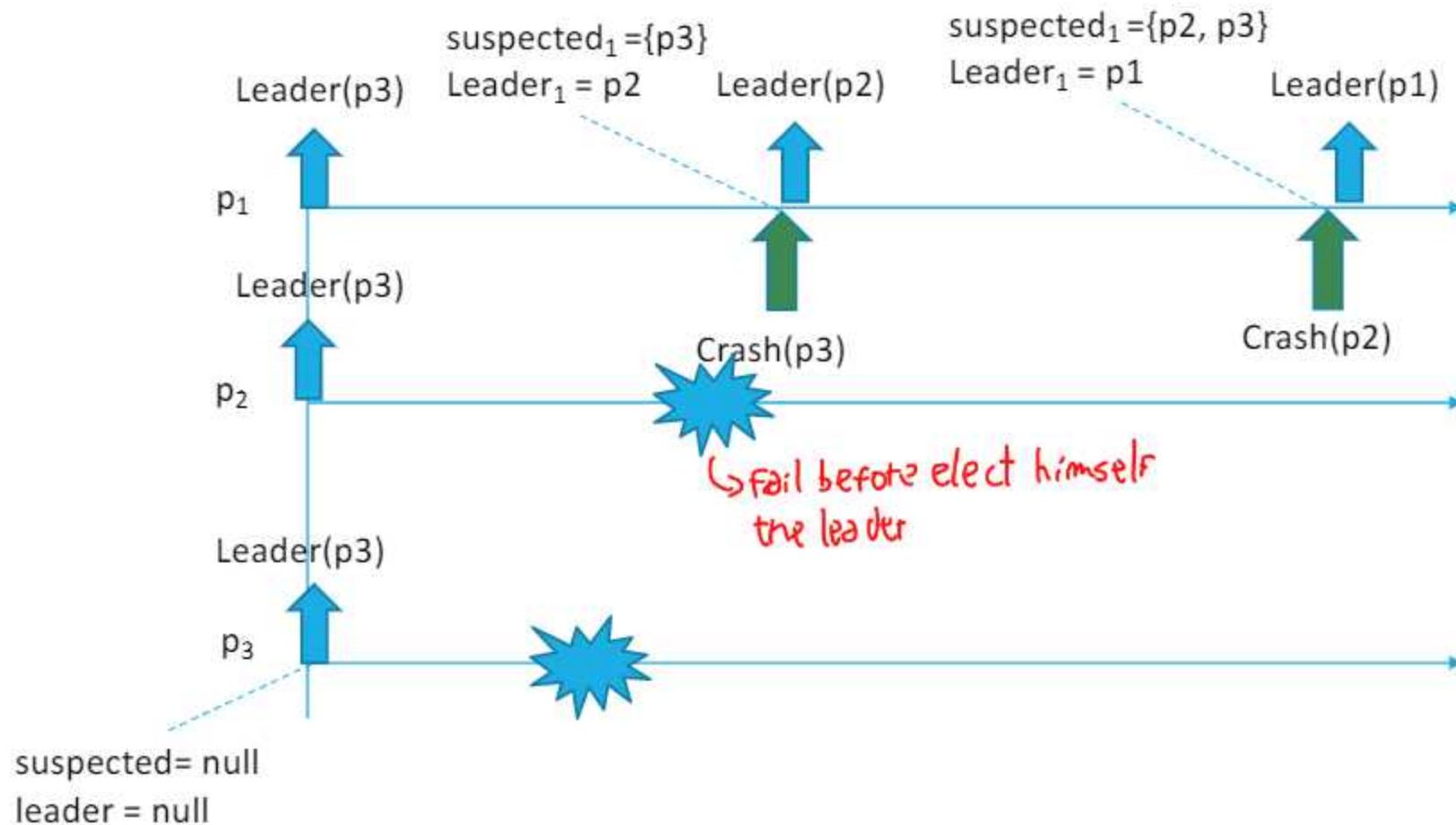


Process that Fail is not the leader

## Example 2

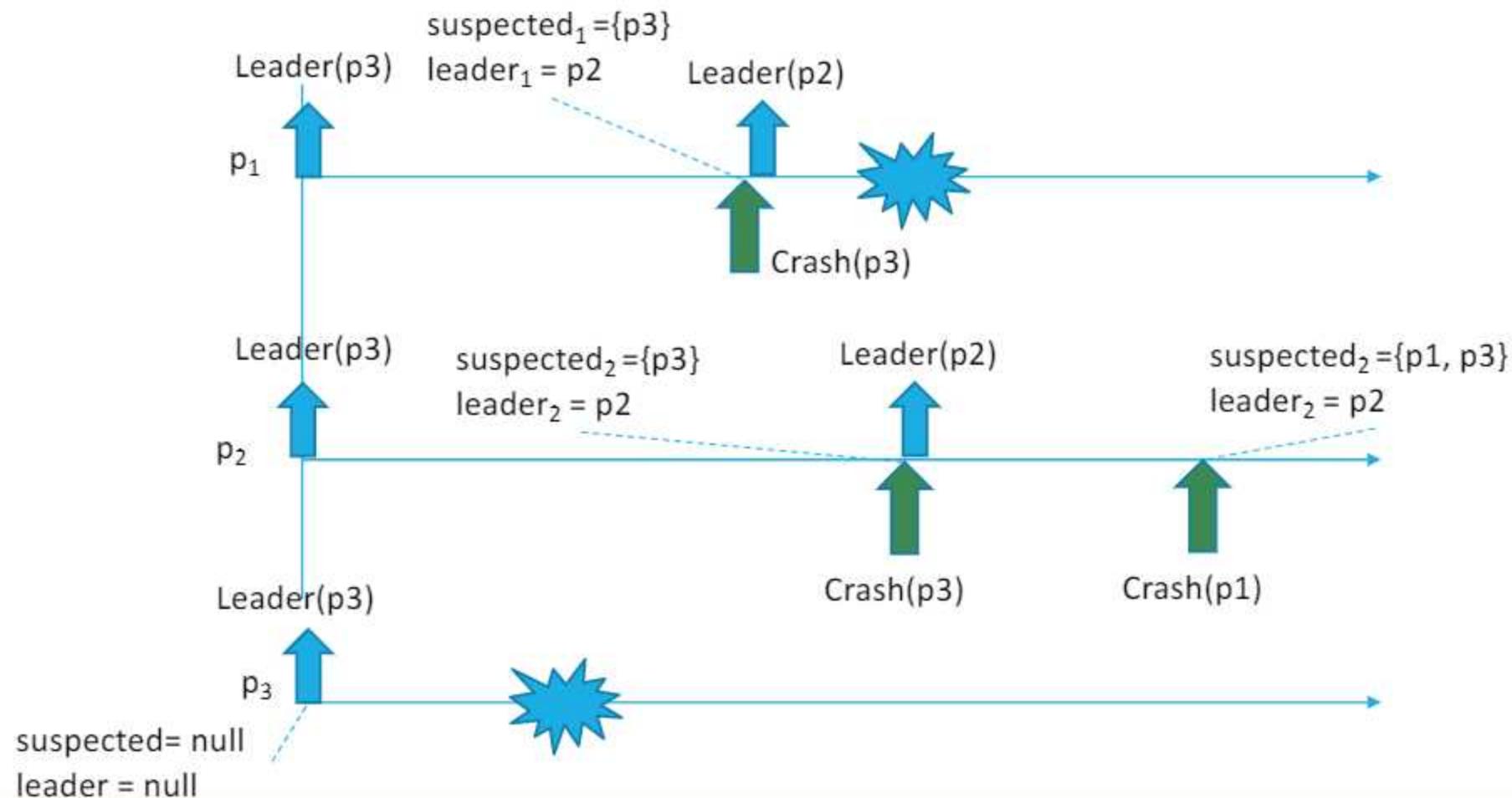


# Example 3



# Example 4

*symmetric to Example 1*



# Correctness

---

What if the Failure detector is not perfect?

if we have an eventually failure detector we can not satisfy the same specification ,  
someone else can change errortily the leader . Period in which i have two leader  
both correct , loose accuracy property

# Eventual leader election ( $\Omega$ )

For mutual exclusion  
need a double check, and  
guarantee mutual

**Module 2.9:** Interface and properties of the eventual leader detector

**Module:**

**Name:** EventualLeaderDetector, instance  $\Omega$

**Events:**

**Indication:**  $\langle \Omega, \text{Trust} | p \rangle$ : Indicates that process  $p$  is trusted to be leader.

**Properties:**

**ELD1:** Eventual accuracy: There is a time after which every correct process trusts some correct process.

$\hookrightarrow$  after a time  $t$  between leader election  $\rightsquigarrow$  elect a good leader

**ELD2:** Eventual agreement: There is a time after which no two correct processes trust different correct processes.

$\rightsquigarrow$  all processes after  $t$  trust some leader

Eventual Leader Election  
 $\Omega$

not sure if  $p_i$   
is the leader  
↑  
Trust( $p_i$ )

# Observation on $\Omega$

---

$\Omega$  ensures that *eventually* correct processes will elect the same correct process as their leader

---

$\Omega$  does not guarantee that

- Leaders change in an arbitrary manner and for an arbitrary period of time
  - many leaders might be elected during the same period of time without having crashed
- 

Once a unique leader is determined, and does not change again, we say that the leader has *stabilized*

---

# Eventual leader election ( $\Omega$ )

---

## Using Crash-stop process abstraction

- Obtained directly by  $\text{<}P$  by using a deterministic rule on processes that are not suspected by  $\text{<}P$
- trust the process with the highest identifier among all processes that are not suspected by  $\text{<}P$

## IMPORTANT ASSUMPTION

- There always exists at least one correct process (otherwise  $\Omega$  cannot be built)

↳ resilience of algorithm is  $m-1$  or otherwise there is no convergence

# $\Omega$ Implementation

---

## Algorithm 2.8: Monarchical Eventual Leader Detection

---

Implements:

EventualLeaderDetector, instance  $\Omega$ .

Uses:

EventuallyPerfectFailureDetector, instance  $\diamond P$ .

upon event  $\langle \Omega, \text{Init} \rangle$  do

$\text{suspected} := \emptyset$ ;

$\text{leader} := \perp$ ;

upon event  $\langle \diamond P, \text{Suspect } p \rangle$  do

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

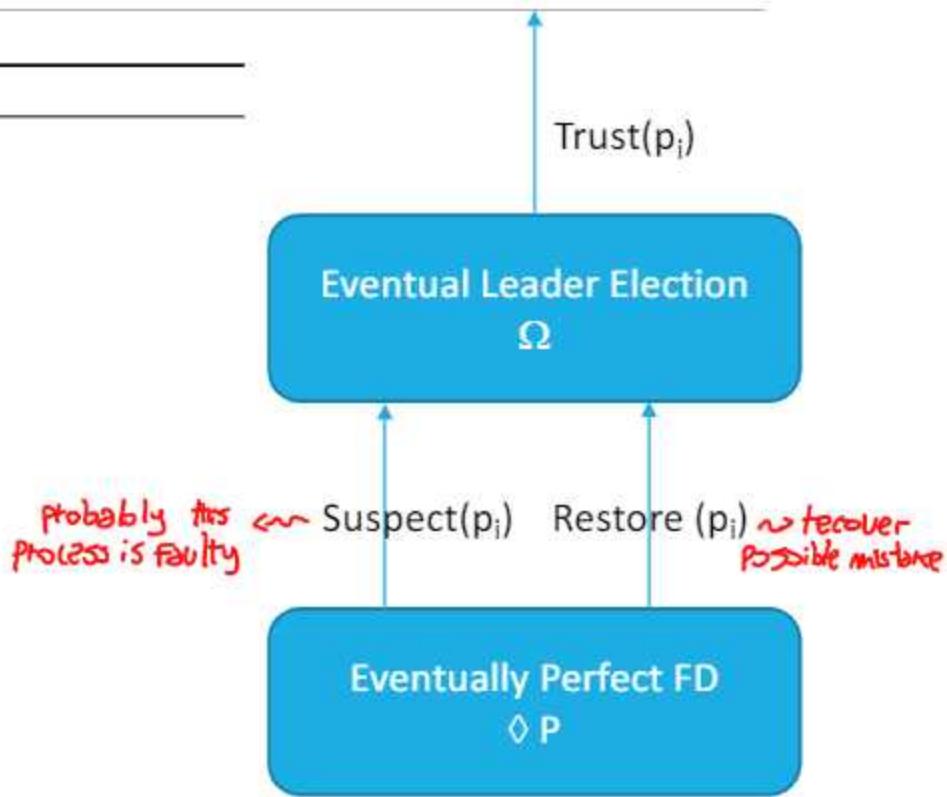
upon event  $\langle \diamond P, \text{Restore } | p \rangle$  do

$\text{suspected} := \text{suspected} \setminus \{p\}$ ;

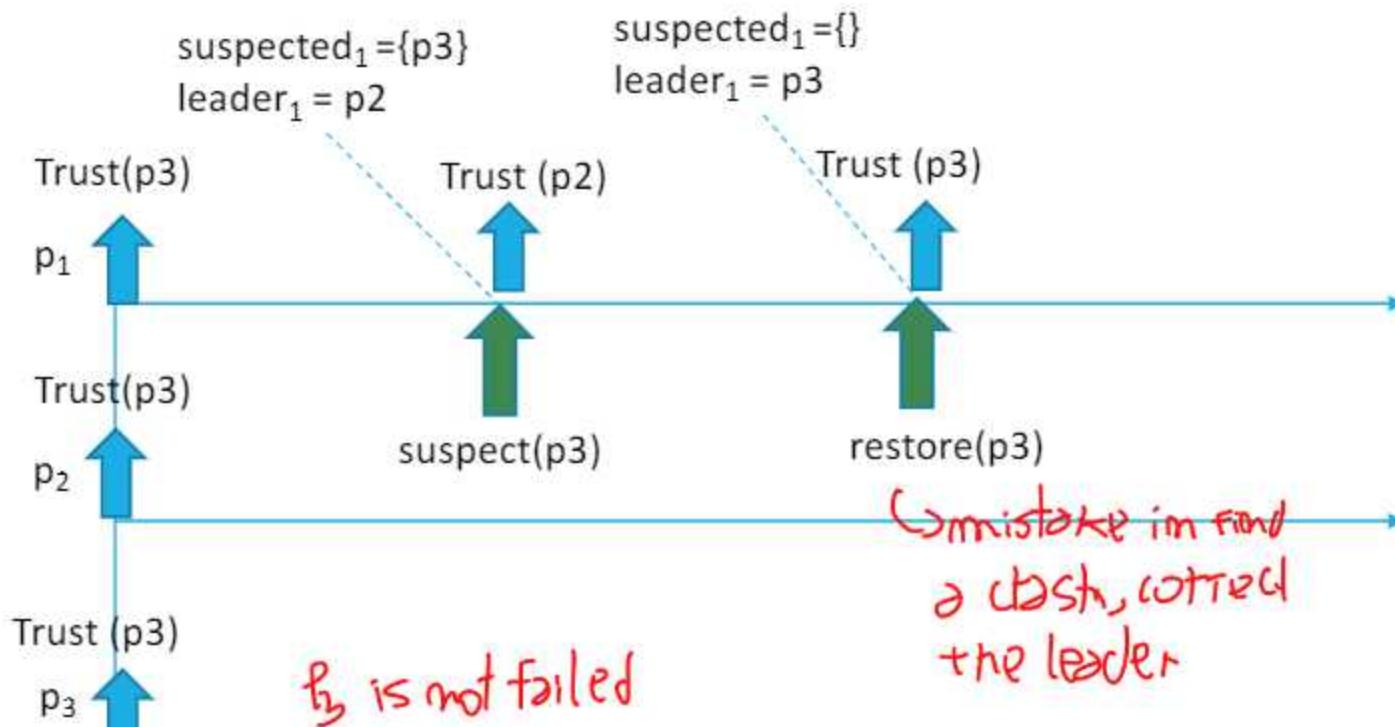
upon  $\text{leader} \neq \text{maxrank}(\Pi \setminus \text{suspected})$  do

$\text{leader} := \text{maxrank}(\Pi \setminus \text{suspected})$ ;

    trigger  $\langle \Omega, \text{Trust } | \text{leader} \rangle$ ; *→ change the event!*



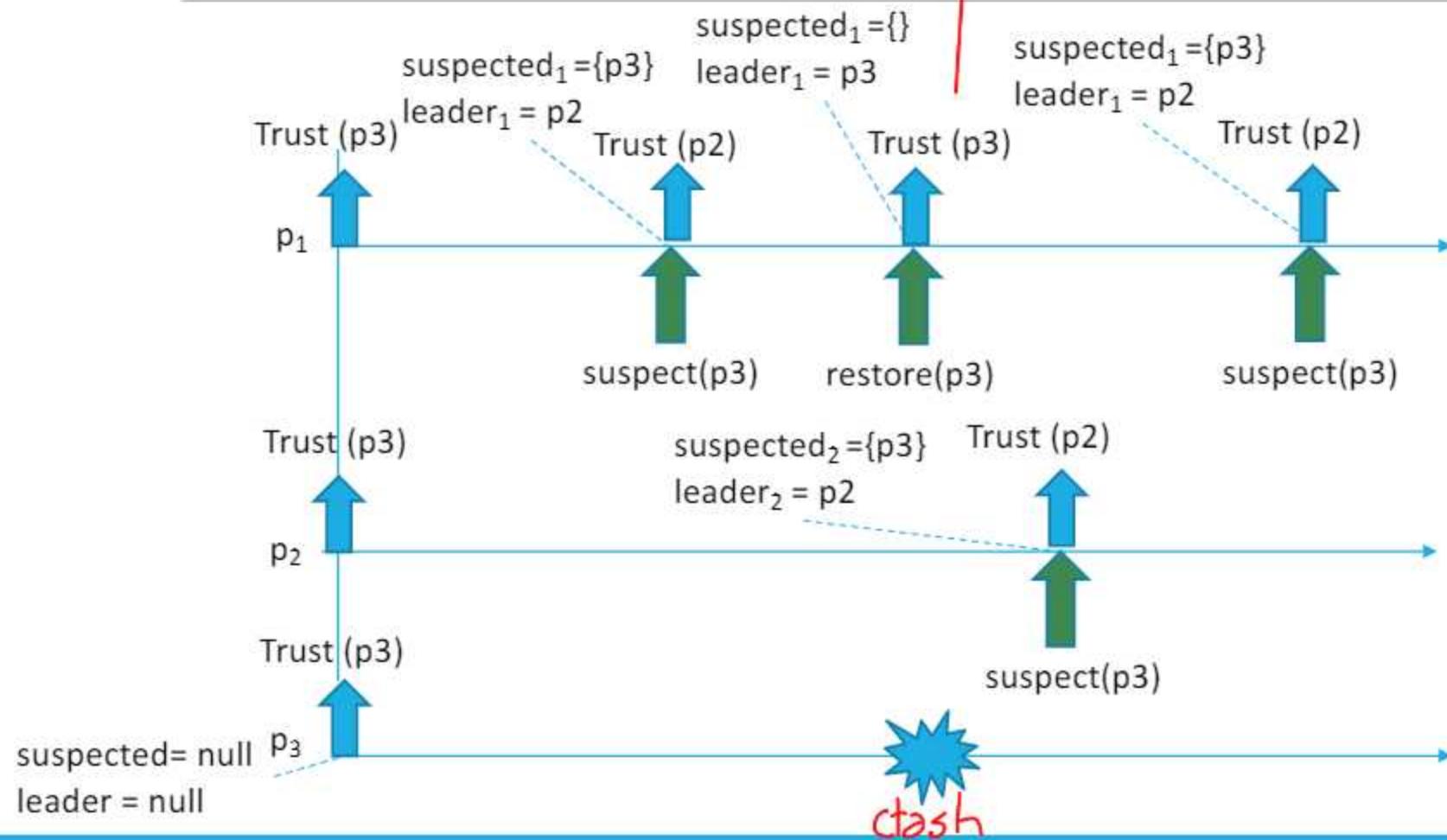
# Example



$\text{suspected} = \text{null}$   
 $\text{leader} = \text{null}$

## Example 2

correct the assumption of  $p_3$  failed



# Eventual leader election ( $\Omega$ )

## System model

- Crash-Recovery
- Partial synchrony  $\rightsquigarrow$  not guarantee always synchrony, like reality, not ask for a perfect P2P link

weakest failure model

Under this assumption, a correct process means:

1. A process that does not crash or
2. A process that crashes, eventually recovers and never crashes again

$\hookrightarrow$  crash a finite number of times

We can use a failure detector on a crash-recovery, use only use message passing, manage manually time assumption

# $\Omega$ With crash-recovery, fair lossy links and timeouts

## Algorithm 2.9: Elect Lower Epoch

Implements:

EventualLeaderDetector, instance  $\Omega$ .

Uses:

FairLossPointToPointLinks, instance  $fl$ .

```
upon event  $\langle \Omega, \text{Init} \rangle$  do
  epoch := 0;
  store(epoch);
  candidates :=  $\emptyset$ ;
  trigger  $\langle \Omega, \text{Recovery} \rangle$ ;
```

*external memory to store state of computation for recovery*

*to become leader*

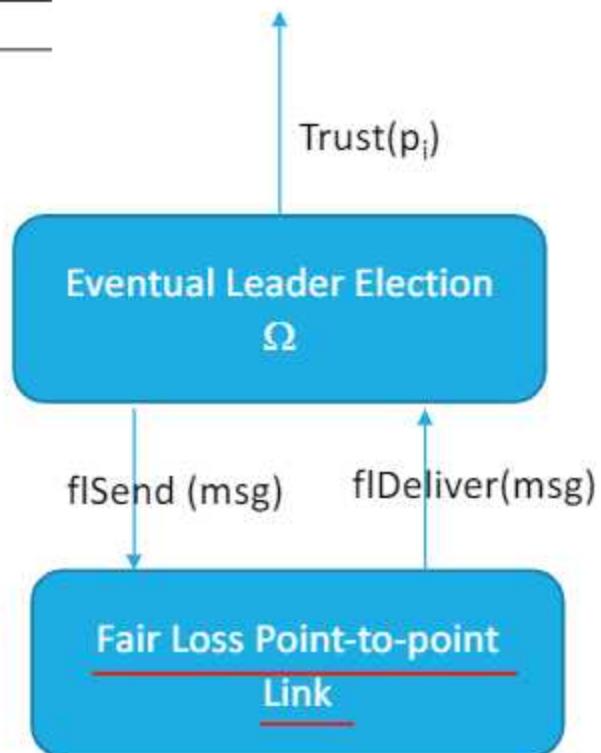
*help to identify correct processes*

*internal function that simply struct of algorithm*

*the first time at startup use init, when reboot only execute RECOVERY*

```
upon event  $\langle \Omega, \text{Recovery} \rangle$  do
  leader := maxrank( $\Pi$ );
  trigger  $\langle \Omega, \text{Trust} \mid \text{leader} \rangle$ ;
  delay :=  $\Delta$ ; to compute timing assumption
  retrieve(epoch);
  epoch := epoch + 1;
  store(epoch);
  forall  $p \in \Pi$  do
    trigger  $\langle fl, \text{Send} \mid p, [\text{HEARTBEAT}, epoch] \rangle$ ;
    candidates :=  $\emptyset$ ;
    starttimer(delay);
```

*keeps track of how many times the process crashed and recovered*



# $\Omega$ With crash-recovery, fair lossy links and timeouts

---

## Algorithm 2.9: Elect Lower Epoch

---

Implements:

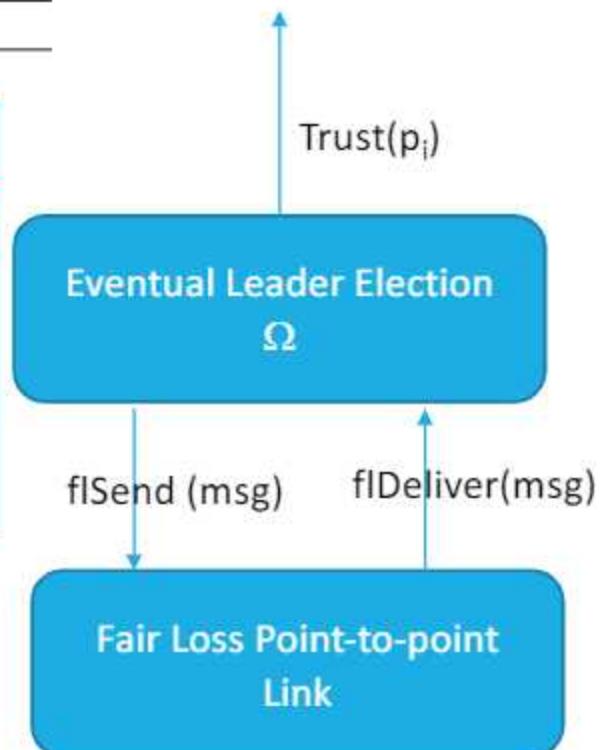
EventualLeaderDetector, instance  $\Omega$ .

Uses:

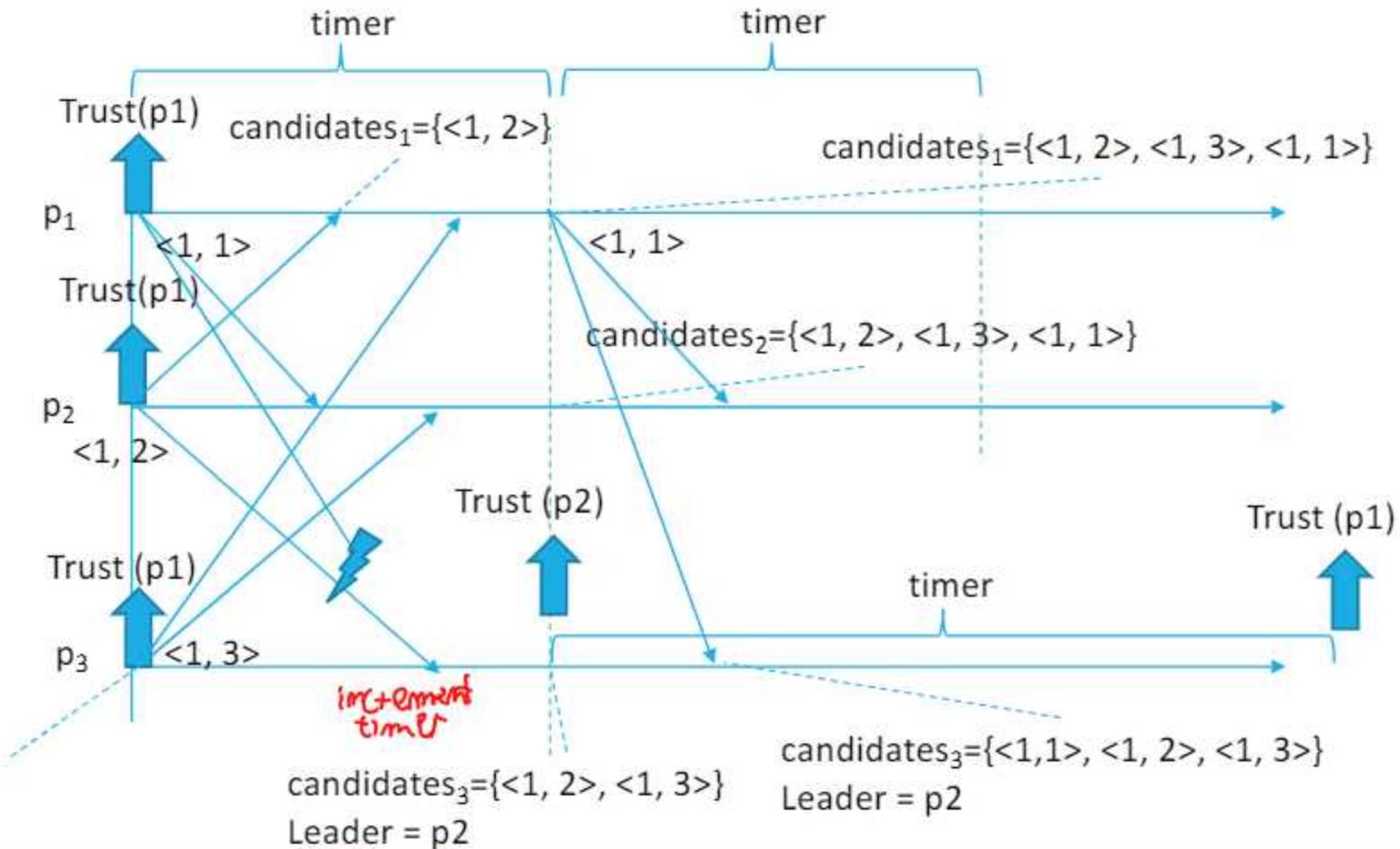
FairLossPointToPointLinks, instance  $fll$ .

```
upon event < Timeout > do
    newleader := select(candidates);
    if newleader  $\neq$  leader then
        delay := delay +  $\Delta$ ; increase time
        leader := newleader;
        trigger <  $\Omega$ , Trust | leader >;
    forall  $p \in \Pi$  do
        trigger <  $fll$ , Send |  $p$ , [HEARTBEAT, epoch] >;
        candidates :=  $\emptyset$ ;
        starttimer(delay);
upon event <  $fll$ , Deliver |  $q$ , [HEARTBEAT, ep] > do
    if exists  $(s, e) \in$  candidates such that  $s = q \wedge e < ep$  then
        candidates := candidates  $\setminus \{(q, e)\}$ ;
        candidates := candidates  $\cup (q, ep)$ ;
```

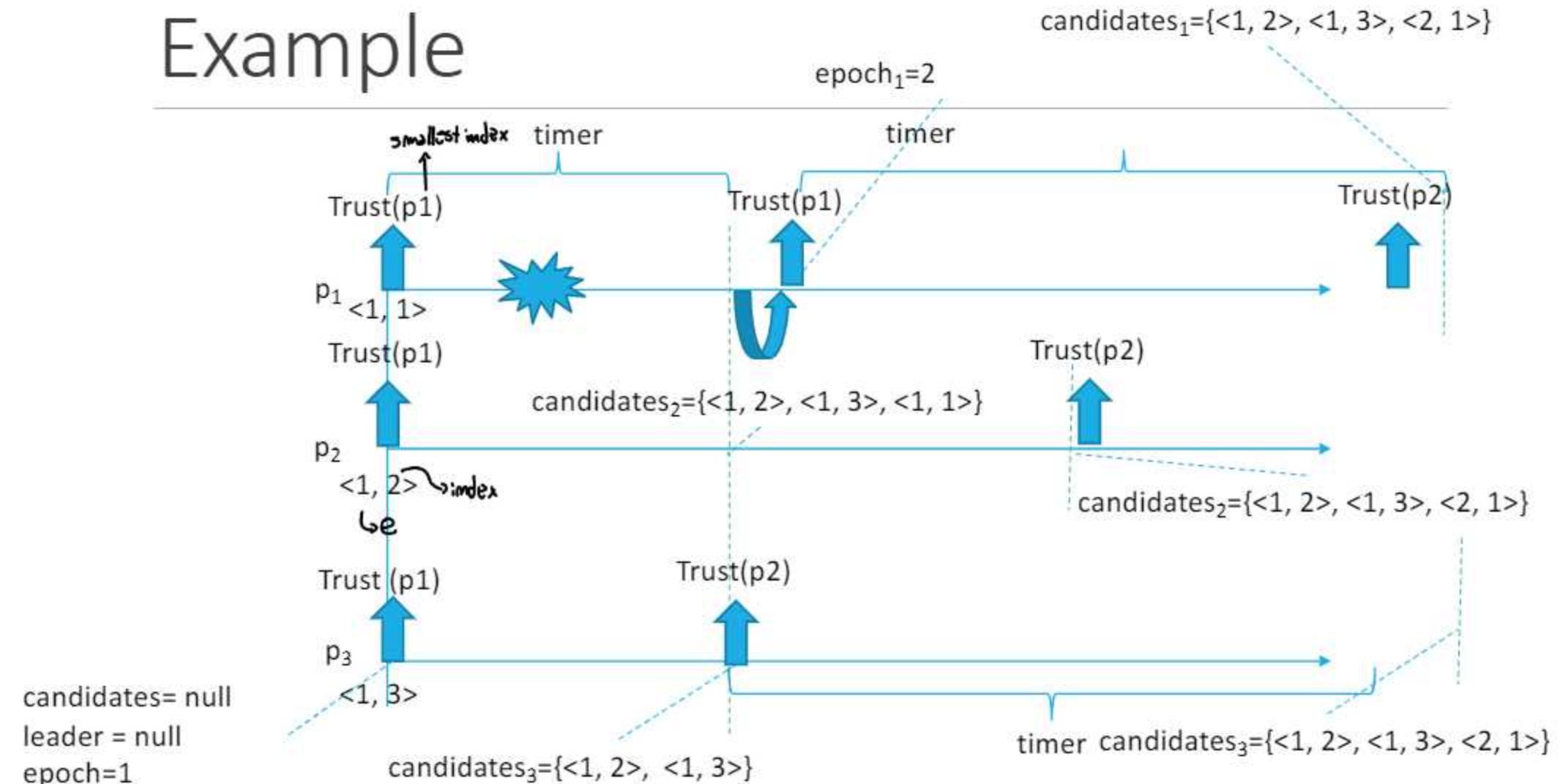
deterministic function returning  
one process among all candidates  
(i.e., process with the lowest  
epoch number and among the  
ones with the same epoch  
number the one with the lowest  
identifier)



# Example



# Example



# References

---

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

- Chapter 2 – from Section 2.6.1 to Section 2.6.5

Dependable Distributed Systems  
Master of Science in Engineering in  
Computer Science

AA 2021/2022

---

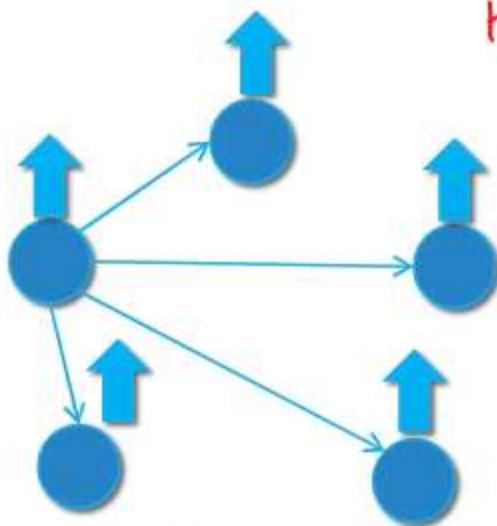
LECTURE 9: **BROADCAST COMMUNICATIONS**

# Recap: what we know up to now

- Define a system model and specify a problem or an abstraction in terms of safety and liveness
  - point-to-point communication abstractions
    - fair-loss, stubborn or perfect links
  - how to timestamp events
    - physical clocks
    - logical clocks
  - handling failures
    - Failure Detector
    - Leader Election
- mask to programme timing assumption*
- timing assumption and synchronization*
- Up to now, the focus has been on the interaction between two processes (like in a client/server environment)
- until now only P2P communication, only two processes communicate*

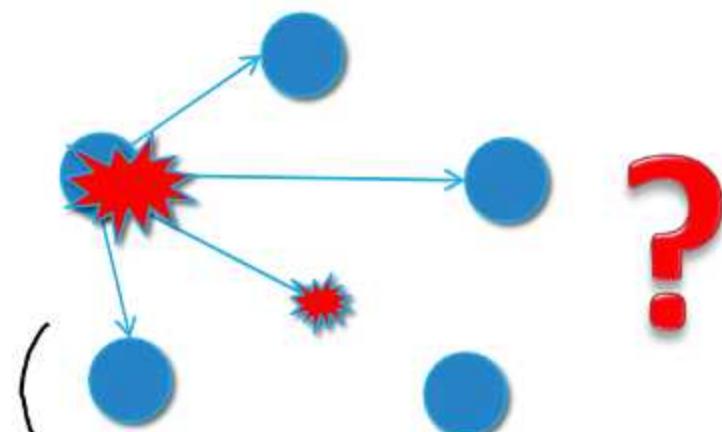
# Communication in a group: Broadcast

one to many communication paradigm,  
where one source generate message  
and spread all over the system, without  
retransmitting one by one.



→ if no failure, the source simply  
send the message over all the links, just  
a for loop with a send to every channel.

No Failures



→ In real-case, the messages can fail to arrive

Crash Failures

# Best Effort Broadcast (BEB) Specification

weakest specification

Module 3.1: Interface and properties of best-effort broadcast

Module:

Name: BestEffortBroadcast, instance *beb*. *is best effort*

Events:

Request:  $\langle \text{beb}, \text{Broadcast} \mid m \rangle$ : Broadcasts a message *m* to all processes.

Indication:  $\langle \text{beb}, \text{Deliver} \mid p, m \rangle$ : Delivers a message *m* broadcast by process *p*.

Properties:

**BEB1: Validity:** If a correct process broadcasts a message *m*, then every correct process eventually delivers *m*.

**BEB2: No duplication:** No message is delivered more than once.

**BEB3: No creation:** If a process delivers a message *m* with sender *s*, then *m* was previously broadcast by process *s*.

→ Preserve P2P communication

compact interface that spread the message to every body without giving much assurance about the outcome of spreading with failure



VALIDITY: if sender is correct, every correct process will deliver the message

# Best Effort Broadcast (BEB) Implementation

---

## Algorithm 3.1: Basic Broadcast

Implements:

BestEffortBroadcast, instance *beb*.

Uses:

PerfectPointToPointLinks, instance *pl*.

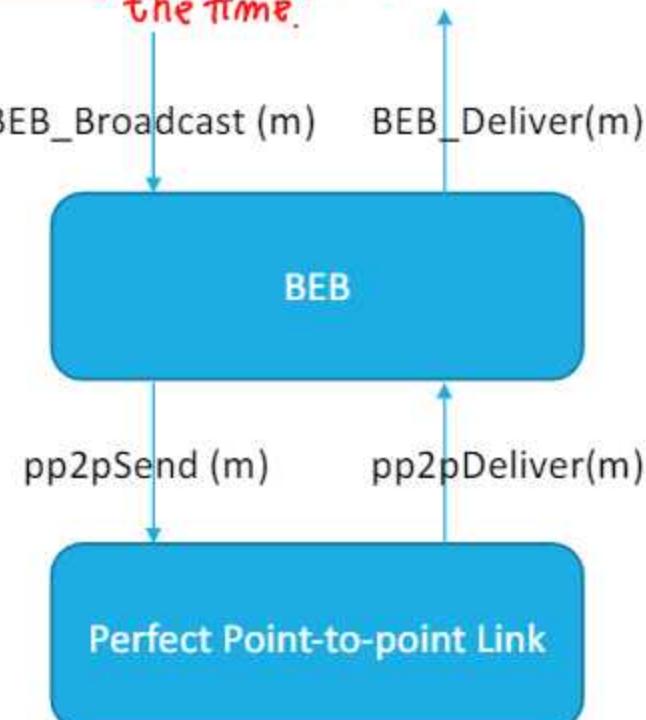
```
upon event < beb, Broadcast | m > do
    forall q ∈  $\Pi$  do
        trigger < pl, Send | q, m >;
```

```
upon event < pl, Deliver | p, m > do
    trigger < beb, Deliver | p, m >;
```

---

## System model

- Asynchronous system, *work independent from the time*.
- perfect links
- crash failures  
*↳ process may fail*



# Correctness

---

## Validity

- It comes from the *reliable delivery* property of perfect links + the fact that the sender sends the message to every other process in the system.
- 

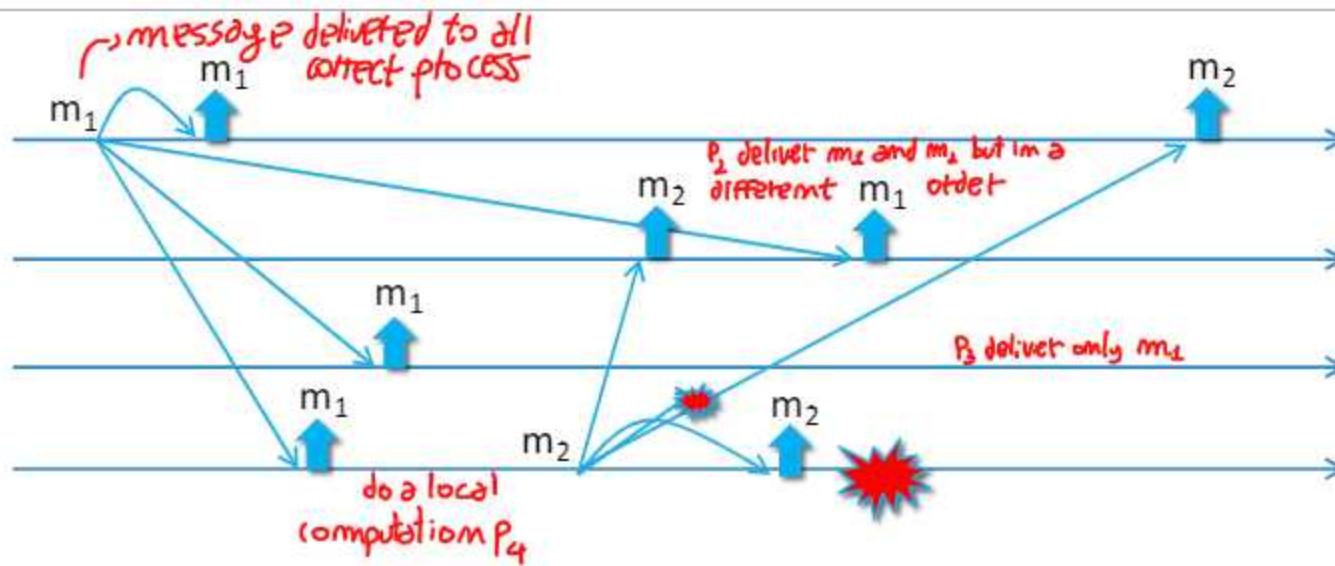
## No Duplication

- it directly follows from the No Duplication of perfect links + assumption on the uniqueness of messages (i.e., different messages have different identifiers).
- 

## No Creation

- it directly follows from the corresponding property of perfect links.

# Observations on Best Effort Broadcast (BEB) *~> there isn't consistent*



- BEB ensures the delivery of messages as long as the sender does not fail
- If the sender fails processes may disagree on whether or not deliver the message

→ you can trust that you deliver the message to everybody or none

# (Regular) Reliable Broadcast (RB)

## Module 3.2: Interface and properties of (regular) reliable broadcast

### Module:

Name: ReliableBroadcast, instance  $rb$ .

### Events:

Request:  $\langle rb, \text{Broadcast} \mid m \rangle$ : Broadcasts a message  $m$  to all processes.

Indication:  $\langle rb, \text{Deliver} \mid p, m \rangle$ : Delivers a message  $m$  broadcast by process  $p$ .

### Properties:

**RB1: Validity:** If a correct process  $p$  broadcasts a message  $m$ , then  $p$  eventually delivers  $m$ .

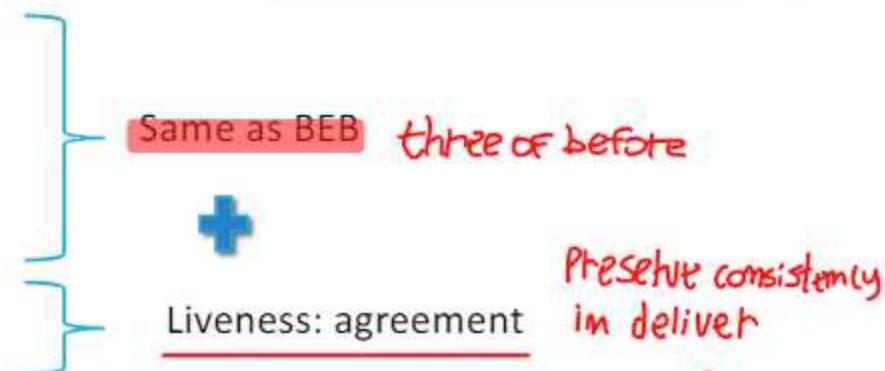
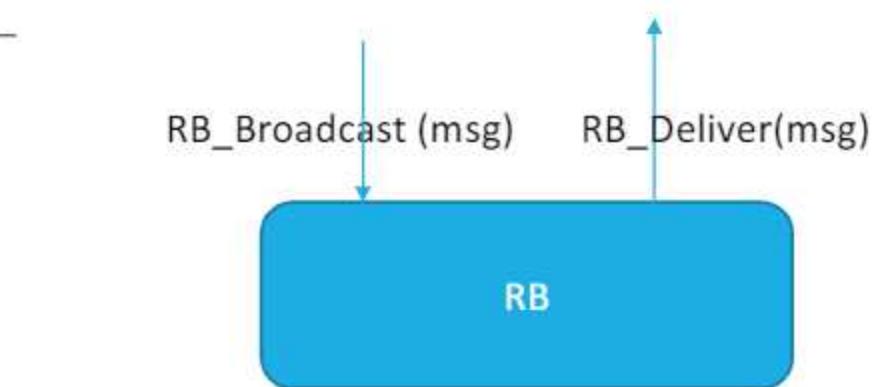
**RB2: No duplication:** No message is delivered more than once.

**RB3: No creation:** If a process delivers a message  $m$  with sender  $s$ , then  $m$  was previously broadcast by process  $s$ .

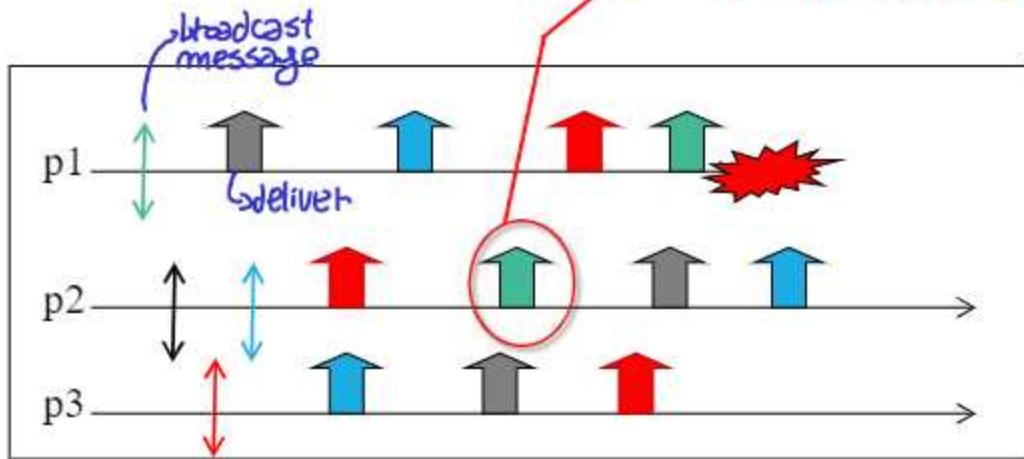
**RB4: Agreement:** If a message  $m$  is delivered by some correct process, then  $m$  is eventually delivered by every correct process.

either all correct message deliver the message

or none of them deliver it, correct processes deliver same set of messages



## BEB vs RB



Validate agreement because m or  $P_2$  is delivered only to  $P_2$  that is a correct process

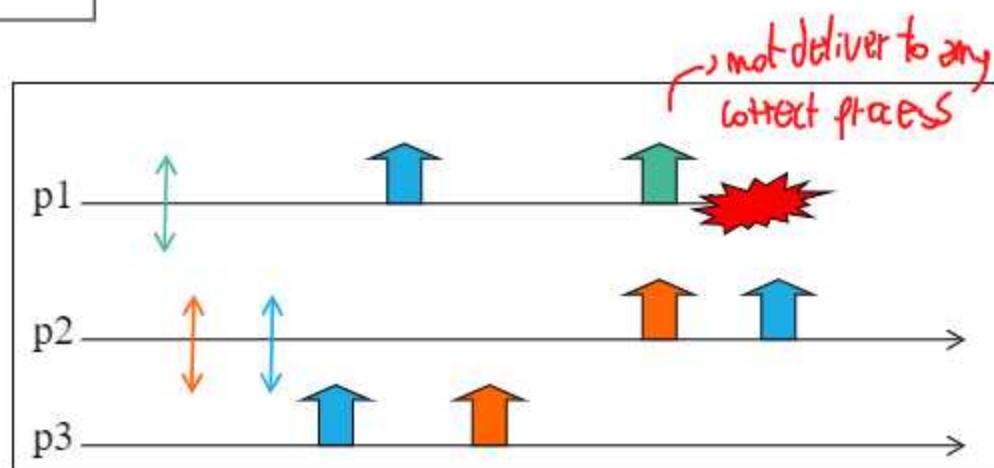
is best effort

Satisfies BEB but not RB  
(violation of the Agreement  
Property)

→ all correct processes that broadcast message will be delivered to other

Satisfies RB

↳ also best effort



# (Regular) Reliable Broadcast (RB) Implementation in Synchronous Systems

works using perfect failure detector

## Algorithm 3.2: Lazy Reliable Broadcast

Implements:

ReliableBroadcast, instance  $rb$ .

resend messages is the idea  
only when is needed

Uses:

BestEffortBroadcast, instance  $beb$ ;  
PerfectFailureDetector, instance  $\mathcal{P}$ .

```

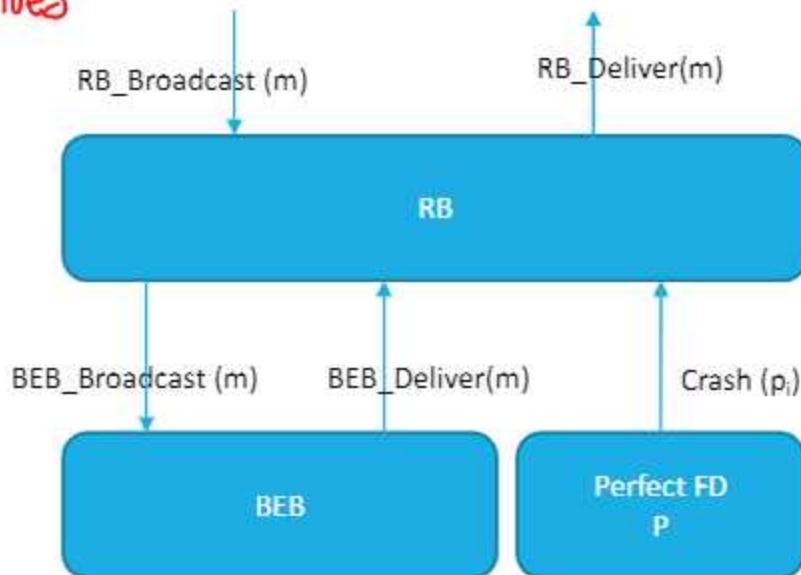
upon event <  $rb$ , Init > do
  correct :=  $\Pi$ ;
  from[ $p$ ] :=  $[\emptyset]^N$ ;
  matrix with m rows, one for every process and in each row state a set of messages got from that process

upon event <  $rb$ , Broadcast |  $m$  > do
  trigger <  $beb$ , Broadcast | [DATA, self,  $m$ ] >;
  s: sender
  m: information

upon event <  $beb$ , Deliver |  $p$ , [DATA,  $s$ ,  $m$ ] > do
  if  $m \notin from[s]$  then
    trigger <  $rb$ , Deliver |  $s$ ,  $m$  >;
    from[ $s$ ] := from[ $s$ ]  $\cup$  { $m$ };
    if  $s \notin correct$  then
      trigger <  $beb$ , Broadcast | [DATA,  $s$ ,  $m$ ] >;
  if m is not in from[s] then deliver to s, update from[s]

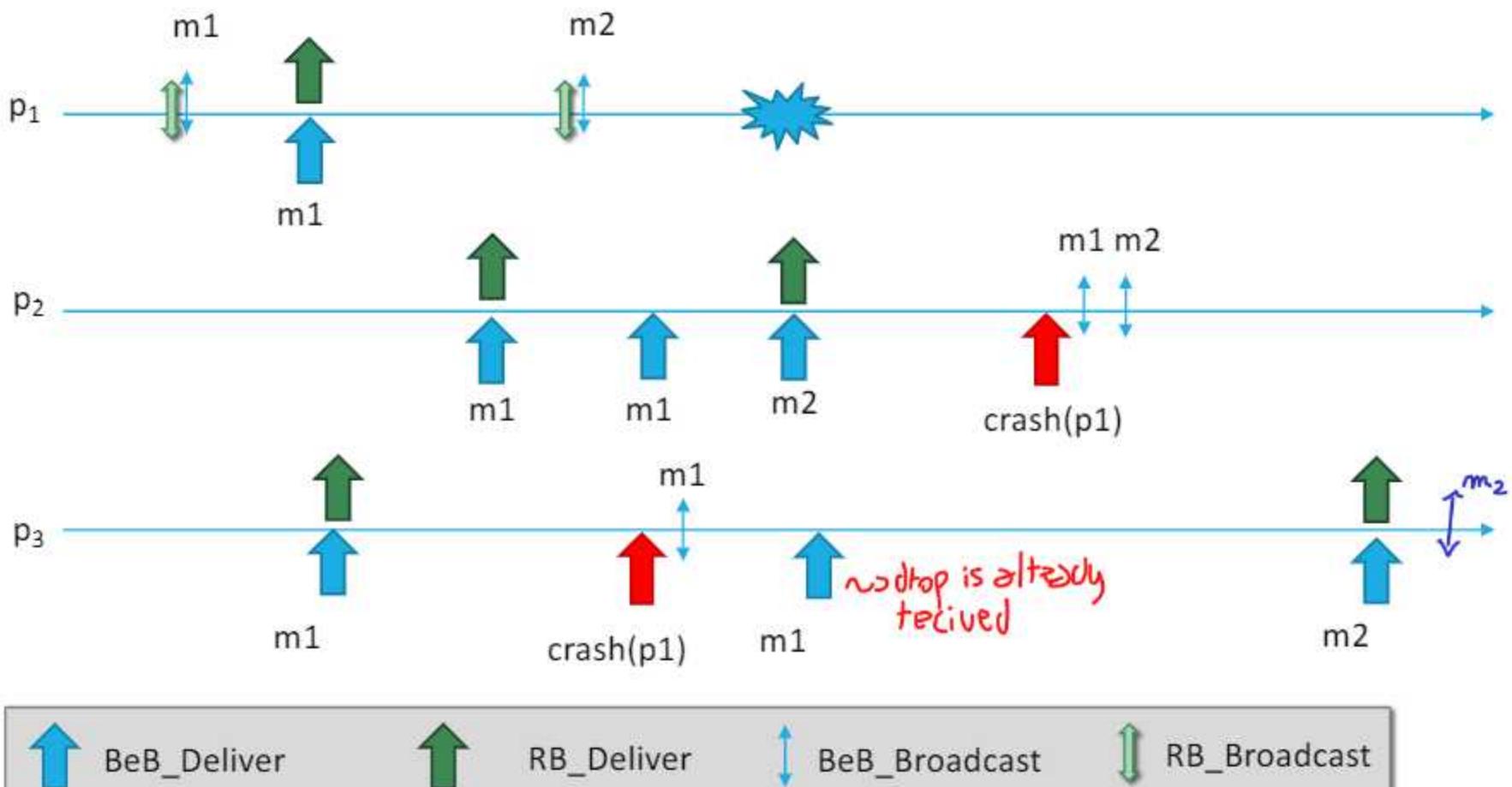
upon event <  $\mathcal{P}$ , Crash |  $p$  > do
  correct := correct  $\setminus$  { $p$ };
  forall  $m \in from[p]$  do
    trigger <  $beb$ , Broadcast | [DATA,  $p$ ,  $m$ ] >;
  if p crashes, remove p from correct, trigger broadcast for all m in from[p]

```



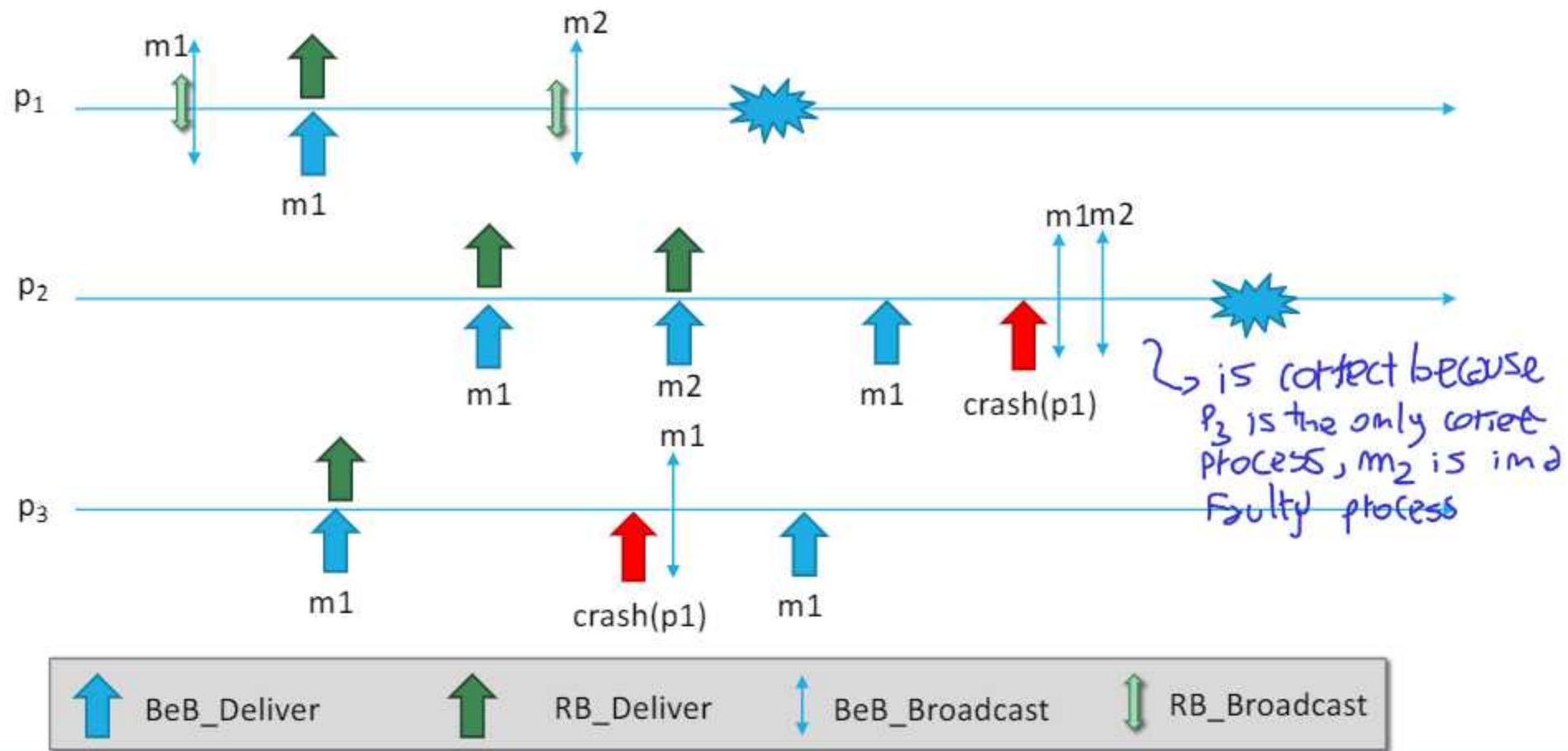
The algorithm is Lazy in the sense that it retransmits only when necessary

# Example

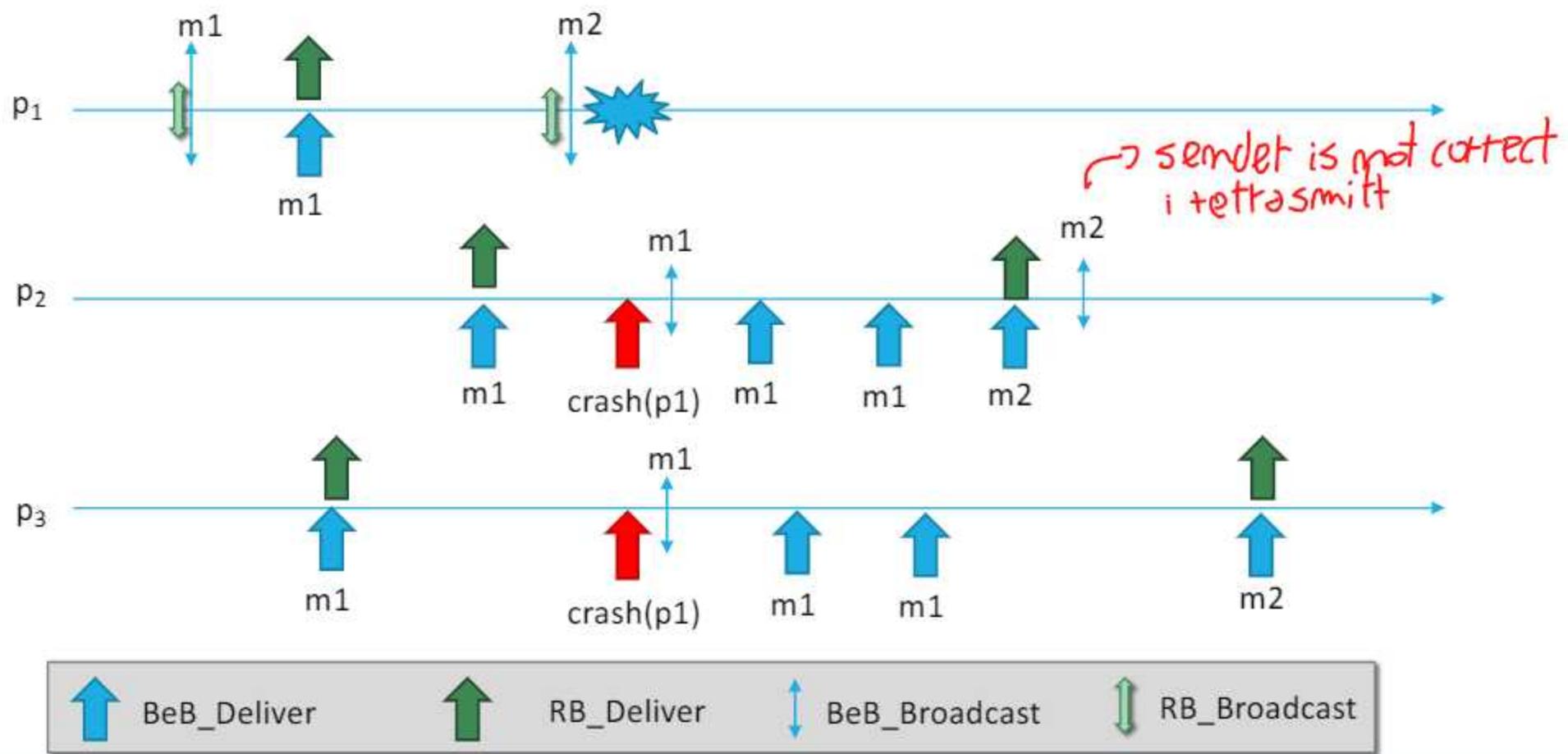


## Example

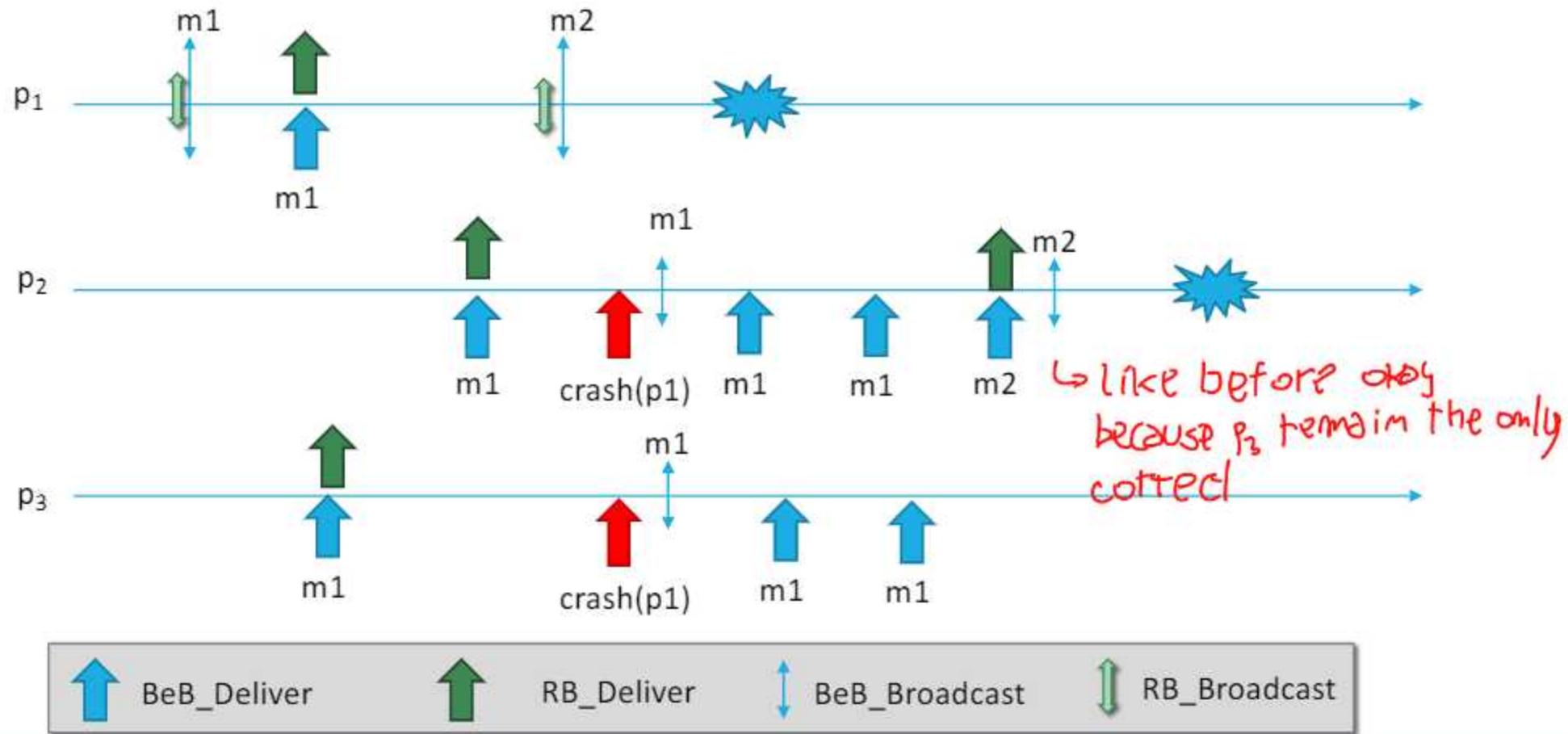
i receive something and other lost, help



# Example

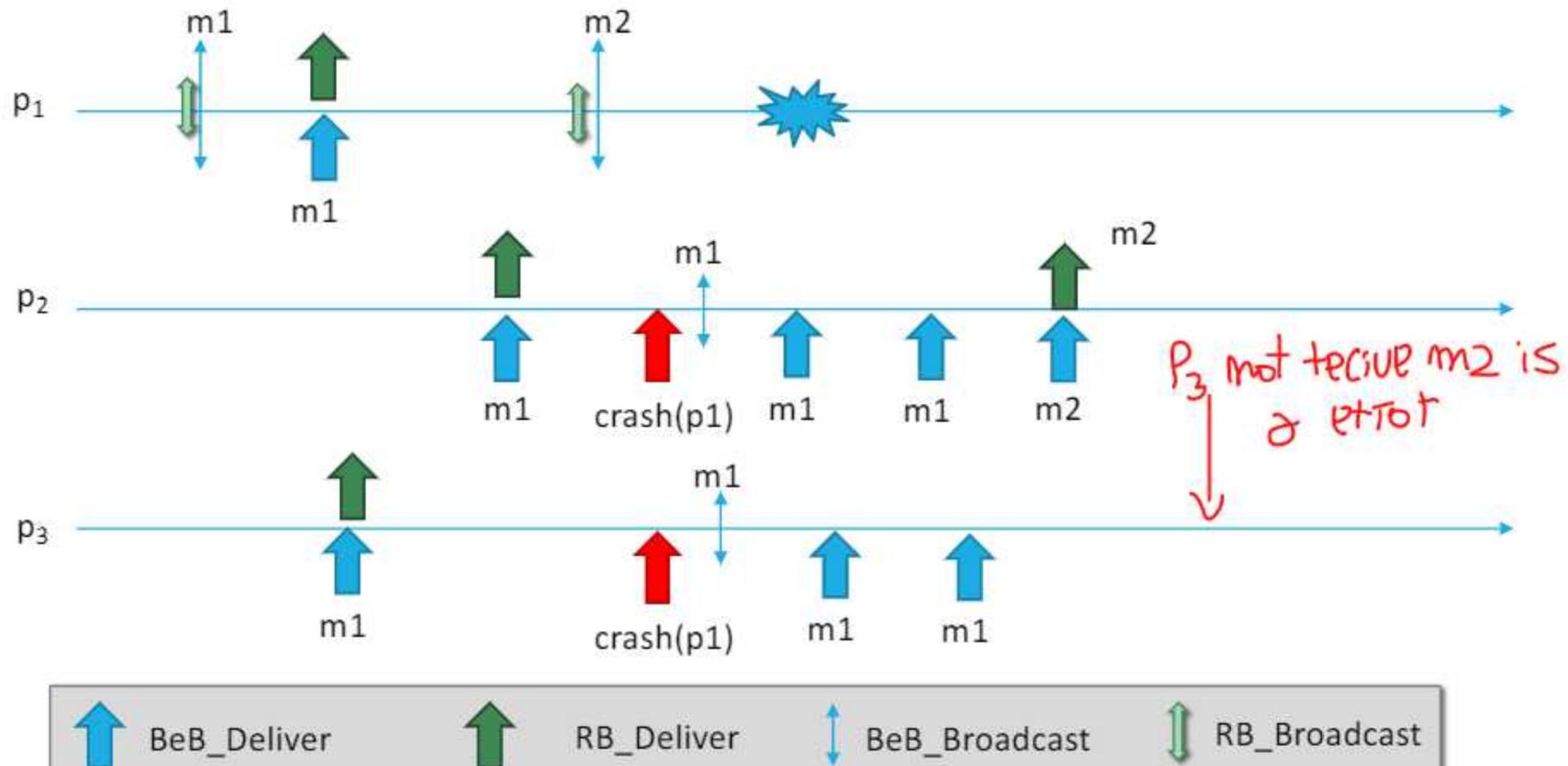


# Example



# Example with removal of retransmission

this is a mistake



# Performance of Lazy RB Algorithm

## ➤ Best case

1 BEB message per one RB message , there are no failure

## ➤ Worst case

$n-1$  BEB messages per one RB (this is the case with  $n-1$  failures) , have a sequence of failure

↳ complexity is the chain of  $n-1$  BEB , that use  $n$  messages , we have a quadratic complexity

## ➤ What if the FD is not perfect?

↳ If failure detector is not perfect , the accuracy is not a problem because we retransmit messages , but for strong completeness (if you are faulty , i discover it) violate property

# (Regular) Reliable Broadcast (RB)

## Implementation in Asynchronous Systems

### Algorithm 3.3: Eager Reliable Broadcast

Implements:

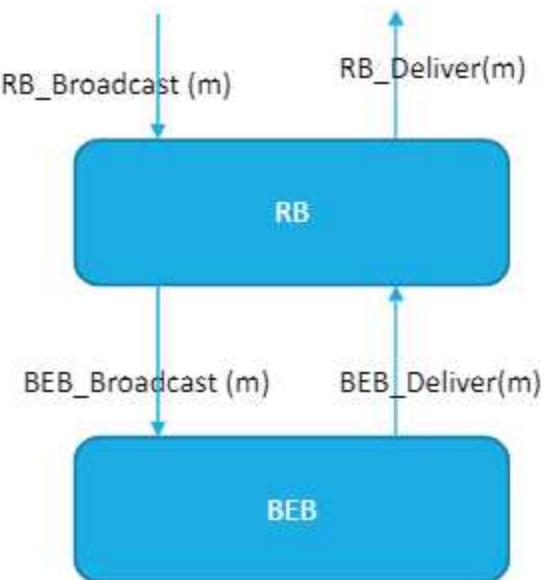
ReliableBroadcast, instance *rb*.

Uses:

BestEffortBroadcast, instance *beb*.

```
upon event < rb, Init > do
    delivered :=  $\emptyset$ ;
upon event < rb, Broadcast | m > do
    trigger < beb, Broadcast | [DATA, self, m] >;
upon event < beb, Deliver | p, [DATA, s, m] > do
    if m  $\notin$  delivered then
        delivered := delivered  $\cup$  {m};
        trigger < rb, Deliver | s, m >;
        trigger < beb, Broadcast | [DATA, s, m] >;
```

↓  
retransmit everytime  
i receive a message



The algorithm is Eager in the  
sense that it retransmits  
every message

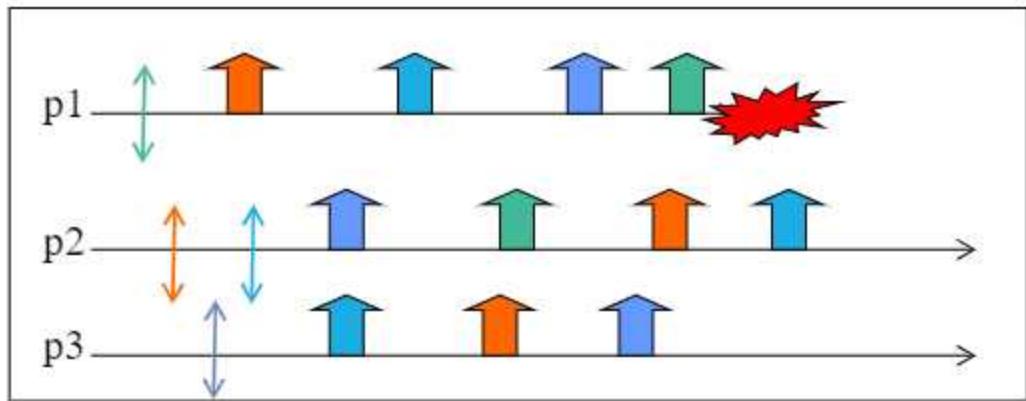
# Performance of Eager RB Algorithm

---

- Best case = Worst case  
n BEB messages per one RB

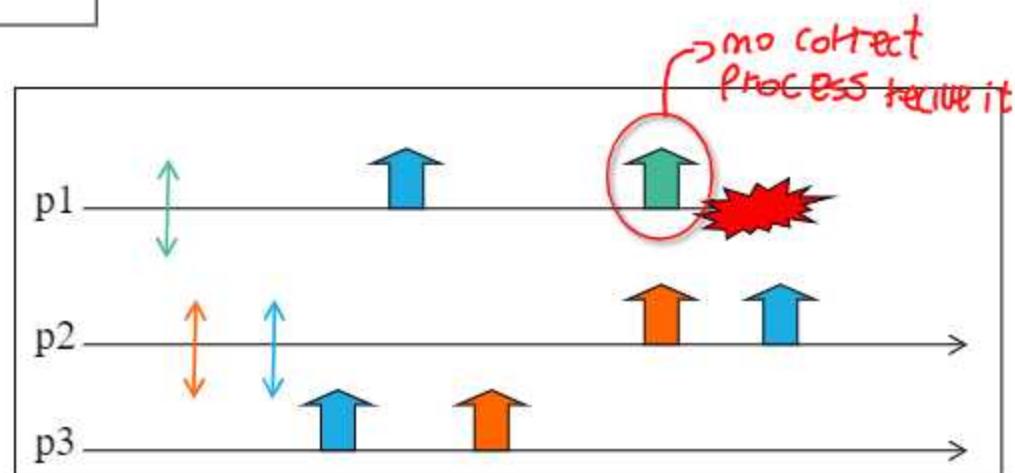
Cost less of failure detector because if there are no communication, there are no messages

# BEB vs RB



Satisfies BEB but not RB  
(violation of the Agreement Property)

Satisfies RB  
not uniform RB



→ no correct process to use it

# Uniform Reliable Broadcast (URB) Specification

**Module 3.3:** Interface and properties of uniform reliable broadcast

**Module:**

**Name:** UniformReliableBroadcast, instance *urb*.

**Events:**

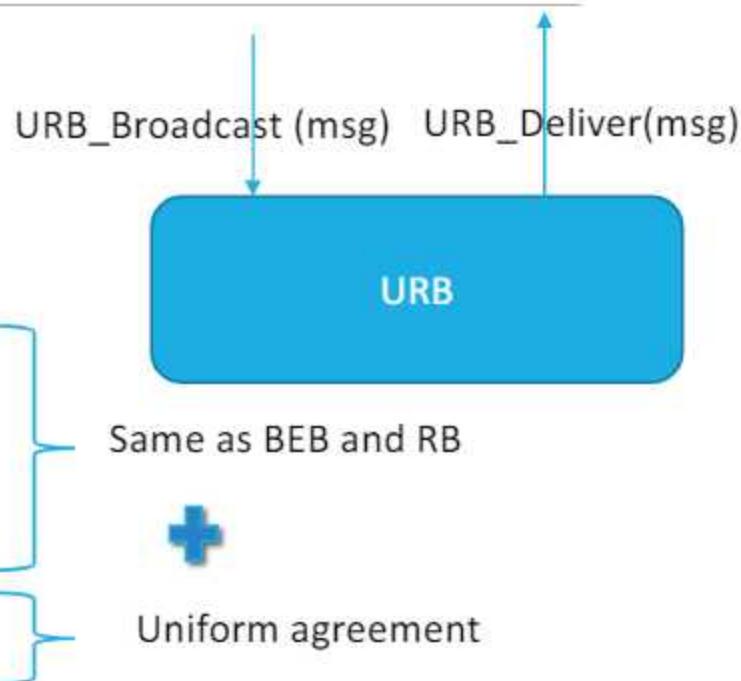
**Request:**  $\langle \text{urb}, \text{Broadcast} \mid m \rangle$ : Broadcasts a message *m* to all processes.

**Indication:**  $\langle \text{urb}, \text{Deliver} \mid p, m \rangle$ : Delivers a message *m* broadcast by process *p*.

**Properties:**

**URB1–URB3:** Same as properties RB1–RB3 in (regular) reliable broadcast (Module 3.2).

**URB4: Uniform agreement:** If a message *m* is delivered by some process (whether correct or faulty), then *m* is eventually delivered by every correct process.

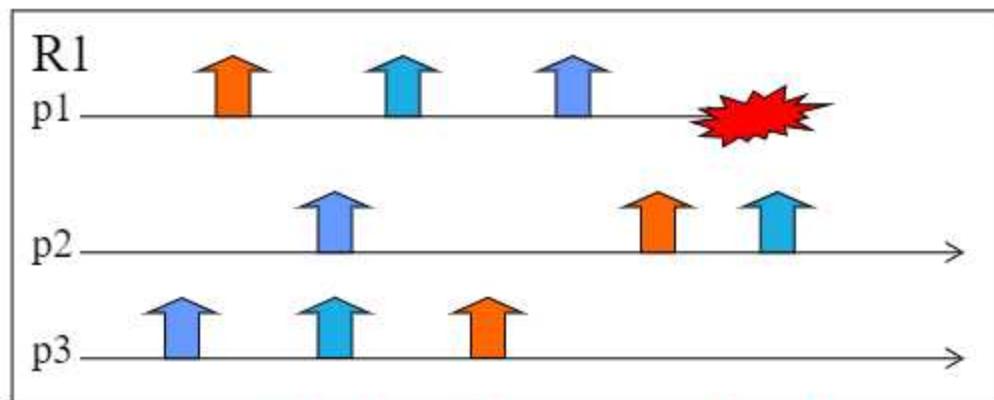


**Agreement on a message delivered by any process (crashed or not)!**



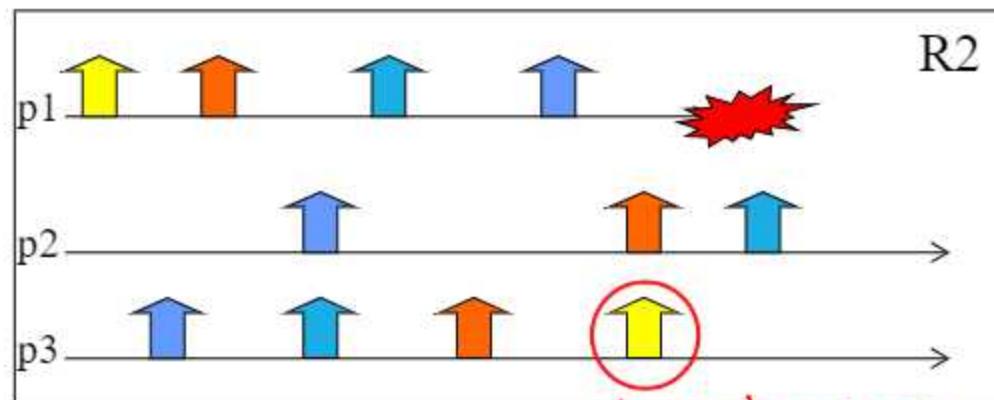
the set of messages delivered by a correct process is a superset of the ones delivered by a faulty one

# BEB vs RB vs URB



everybody deliver exactly the same set of messages

URB



is not reliable  
because p2 is correct  
and not have it

BEB if yellow message is sent by p1

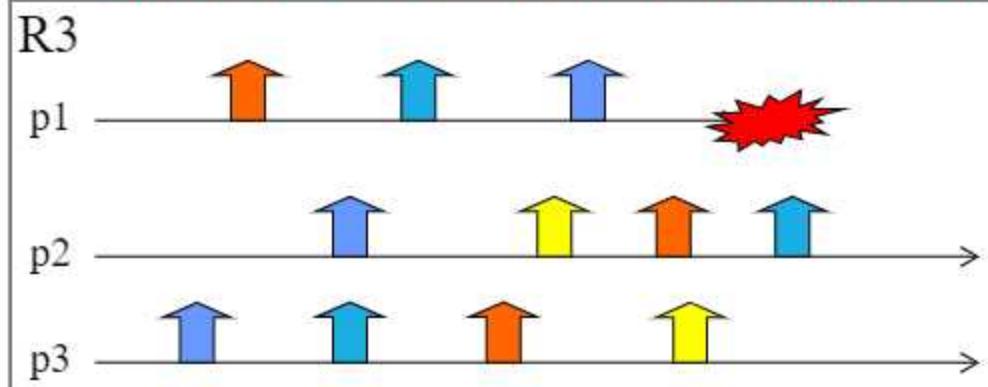
Non-correct otherwise

↳ if sent by p2 is obviously, if is p3 violate validity

# BEB vs RB vs URB

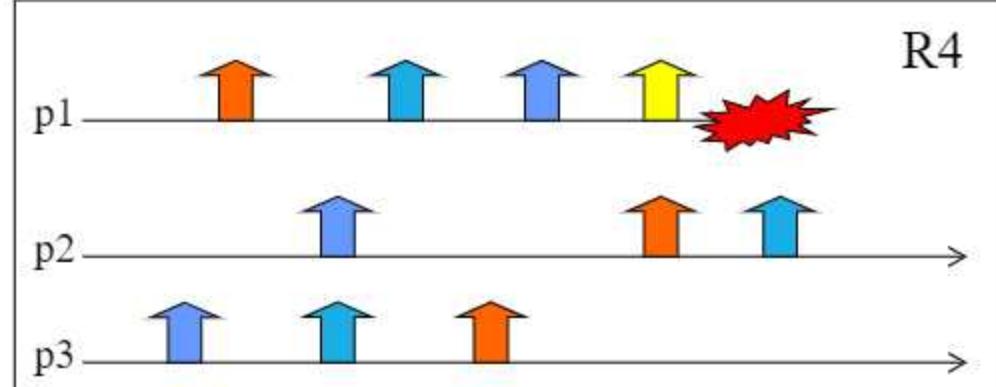
URB

*faulty deliver is a subset of correct processes*



RB if yellow message is sent by p1

Non-correct otherwise



idea is not deliver as soon we can, but wait that everybody could potential deliver the message in synchronous case we use the power of P.F.D. , we use a pending list and ask to other

# Uniform Reliable Broadcast (URB) Implementation in Synchronous System

## Algorithm 3.4: All-Ack Uniform Reliable Broadcast

Implements:

UniformReliableBroadcast, instance *urb*.

Uses:

BestEffortBroadcast, instance *beb*.

PerfectFailureDetector, instance *P*.

```
upon event { urb, Init } do
  delivered := {};
  pending := {};
  correct := {};
  forall m do ack[m] := {};
  same as RB, for filter possible retransmission

  upon event { urb, Broadcast | m } do
    pending := pending ∪ {(self, m)};
    set of messages that i receive, but didn't deliver yet
    trigger { beb, Broadcast | [DATA, self, m] };

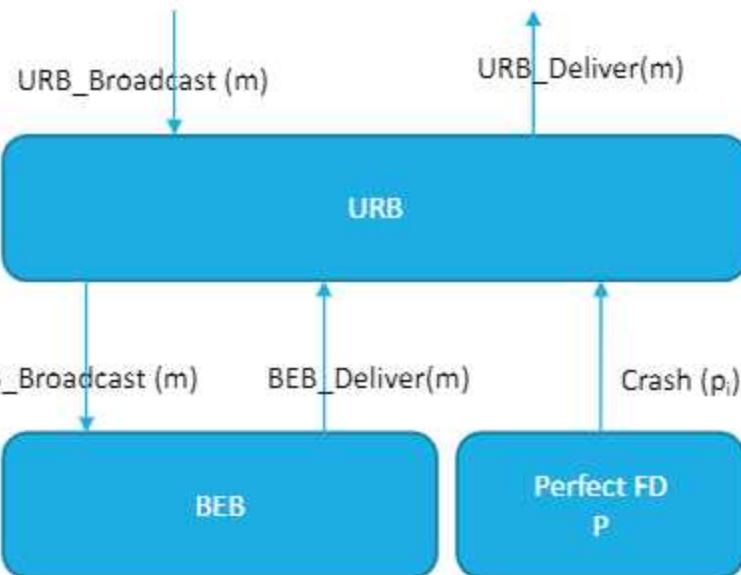
  upon event { beb, Deliver | p, [DATA, s, m] } do
    pending := pending ∖ {(s, m)};
    trigger { beb, Broadcast | [DATA, s, m] };

  upon event { P, Crash | p } do
    correct := correct ∖ {p};
    check whatever the set of correct processes is
    ↗ contained in the set of processes that provided the acknowledgement for m.

  function candeliver(m) returns Boolean is
    return (correct ⊆ ack[m]);

  upon exists (s, m) ∈ pending such that candeliver(m) ∧ m ∉ delivered do
    delivered := delivered ∪ {m};
    trigger { urb, Deliver | s, m };

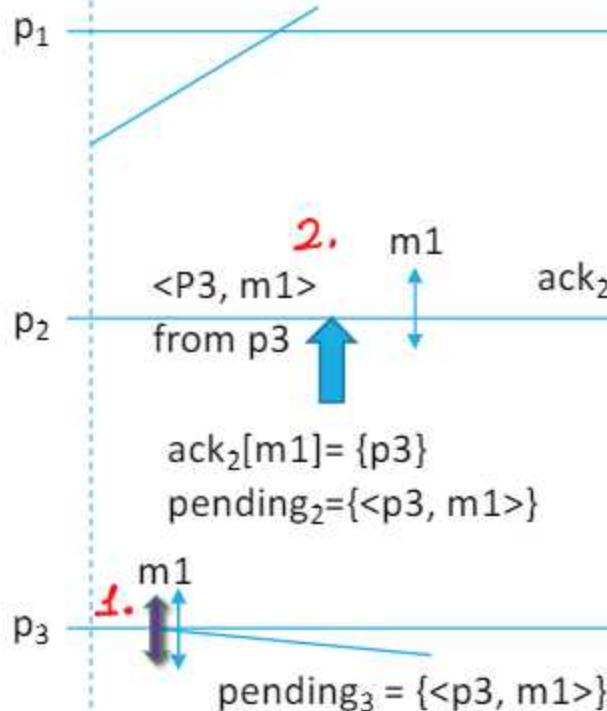
  if have in pending, not deliver and all correct processes provide a copy of the message
```



for deliver one UDRB we use a quadratic number of messages, each process need to retransmit

# Example

Delivered = empty  
Pending = empty  
Correct = {p1, p2, p3}



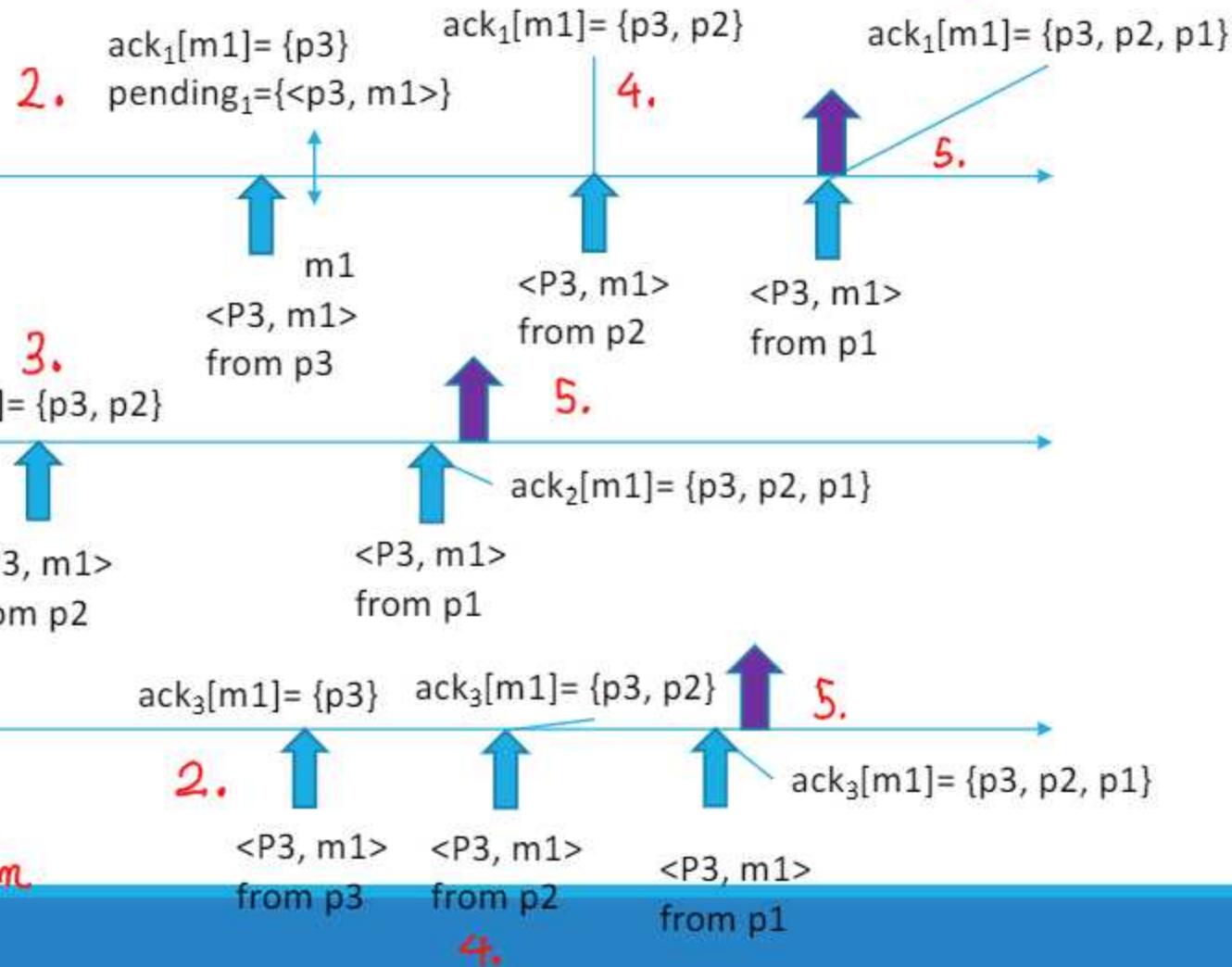
when  $m_1$  is delivered is triggered

upon event  $\langle beb, Deliver \mid p, [DATA, s, m] \rangle$  do  
 $ack[m] := ack[m] \cup \{p\}$ ;  
if  $(s, m) \notin pending$  then  
 $pending := pending \cup \{(s, m)\}$ ;  
trigger  $\langle beb, Broadcast \mid [DATA, s, m] \rangle$ ;

function  $cadeliver(m)$  returns Boolean is  
return ( $correct \subseteq ack[m]$ );

upon exists  $(s, m) \in pending$  such that  $cadeliver(m) \wedge m \notin delivered$  do  
 $delivered := delivered \cup \{m\}$ ;  
trigger  $\langle urb, Deliver \mid s, m \rangle$ ;

1. start to monitoring Pending list

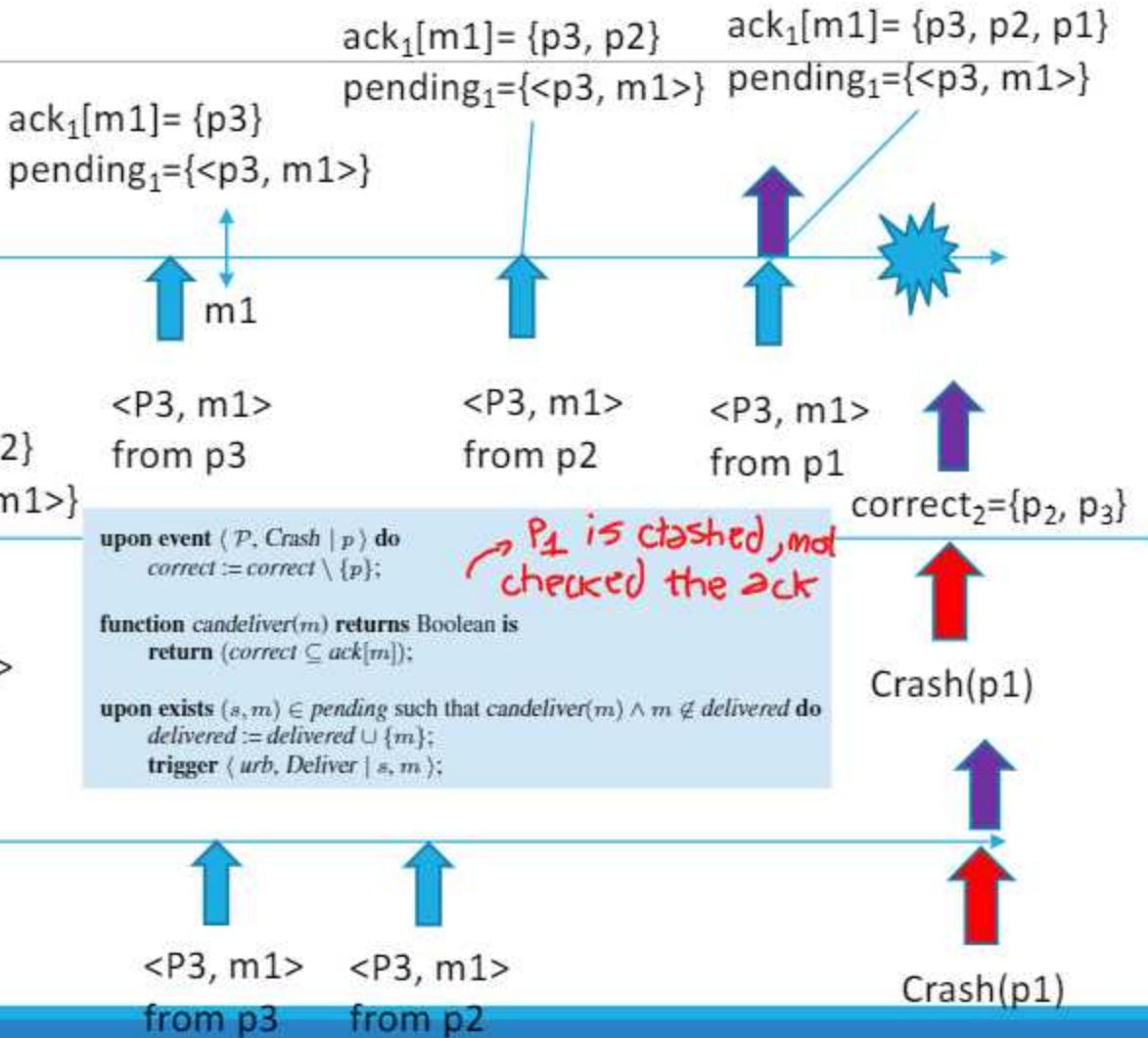
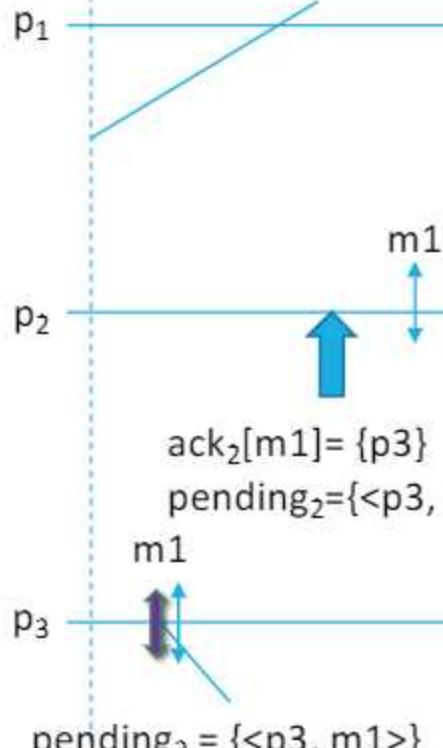


upon event  $\langle urb, Broadcast \mid m \rangle$  do  
 $pending := pending \cup \{(self, m)\}$ ;  
trigger  $\langle beb, Broadcast \mid [DATA, self, m] \rangle$ ;

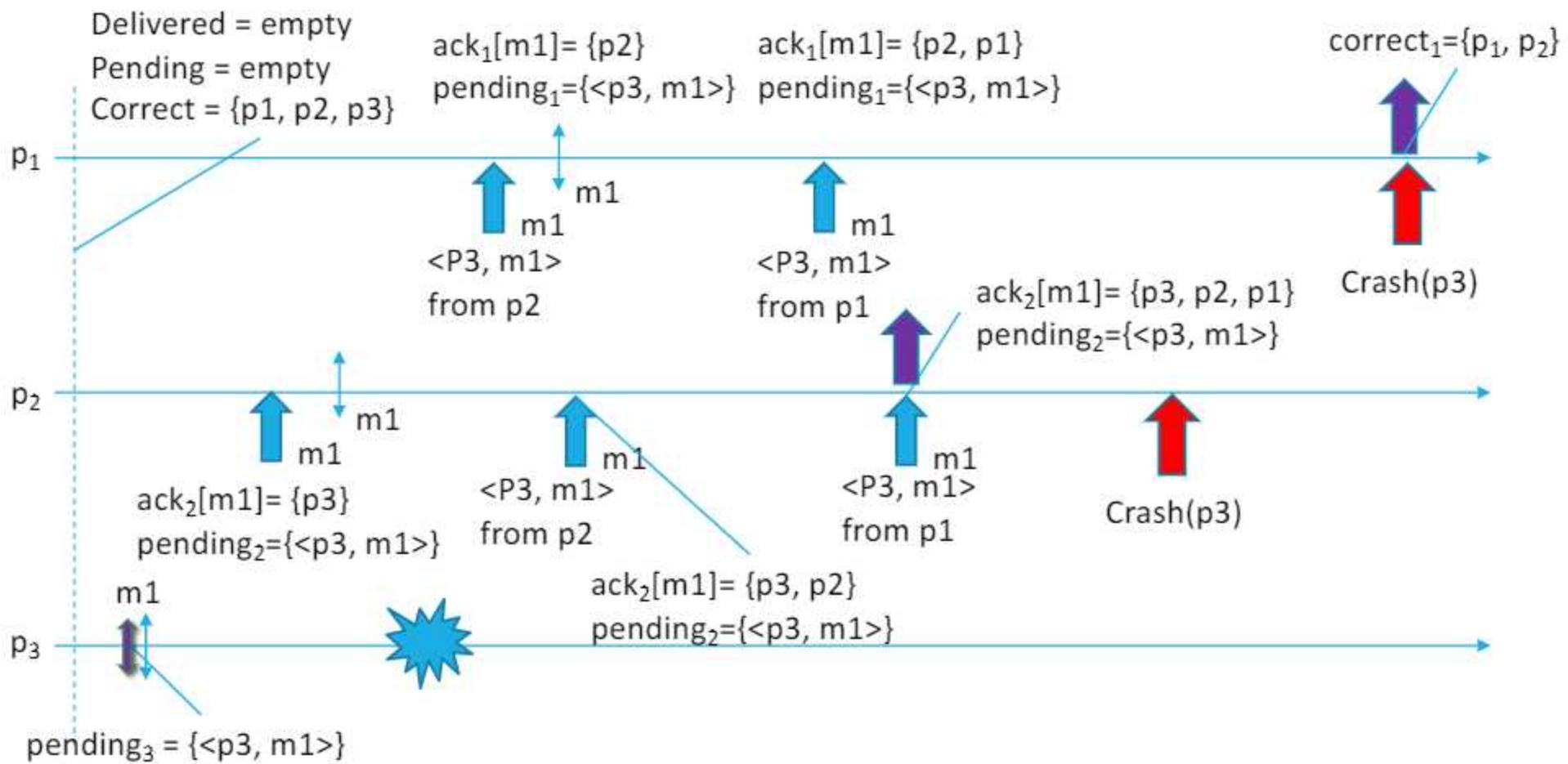
first transmission

# Example

Delivered = empty  
 Pending = empty  
 Correct = {p1, p2, p3}



# Example



we can't use perfect failure detector, without strong completeness can't guarantee liveness, without strong accuracy may loose uniform agreement

# Uniform Reliable Broadcast (URB)

## Implementation in Asynchronous System

can't simply remove failure detector and transmitting, need to bound the number of failure processes.  $F > N/2$   $\Rightarrow$  majority of correct processes remain in the system ( $l=6, F < 3$ )  $\Rightarrow$  decrease robustness of the system

### Algorithm 3.5 Majority-Ack Uniform Reliable Broadcast

Implements:

UniformReliableBroadcast (urb).

Extends:

All-Ack Uniform Reliable Broadcast (Algorithm 3.4).

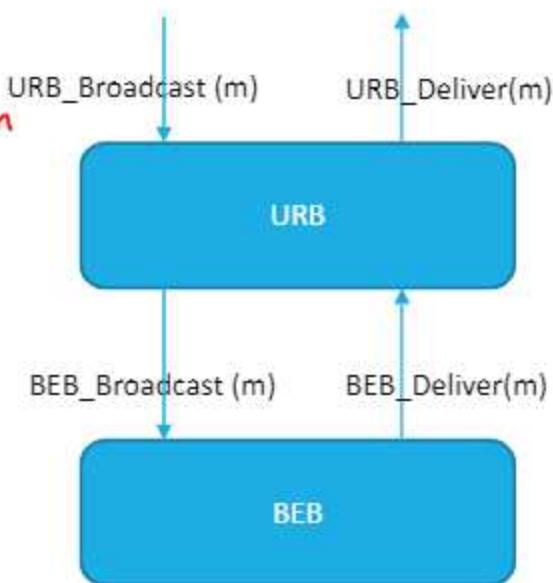
Uses:

BestEffortBroadcast (beb).

function canDeliver(m) returns boolean is  
return  $(|ack_m| > N/2)$ ;

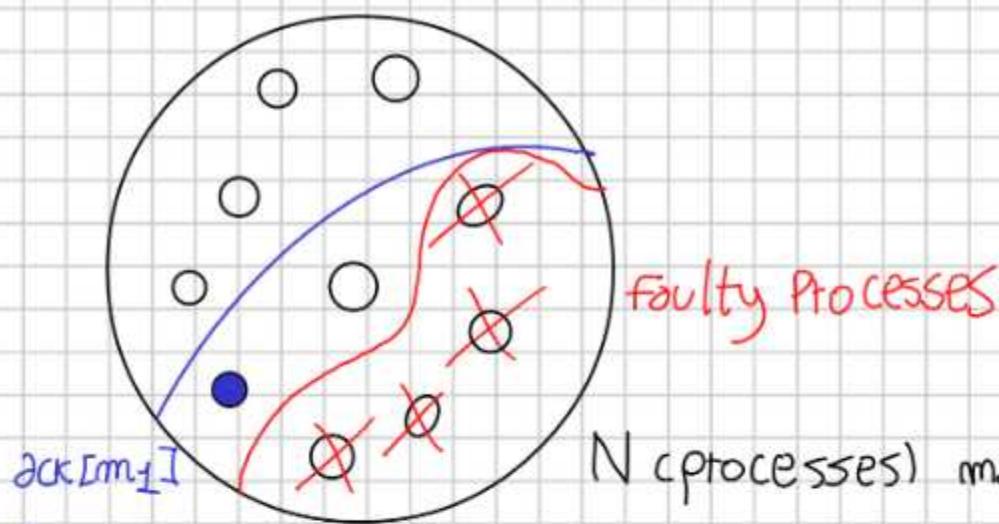
~only changed

// Except for the function above, and the non-use of the  
// perfect failure detector, same as Algorithm 3.4.

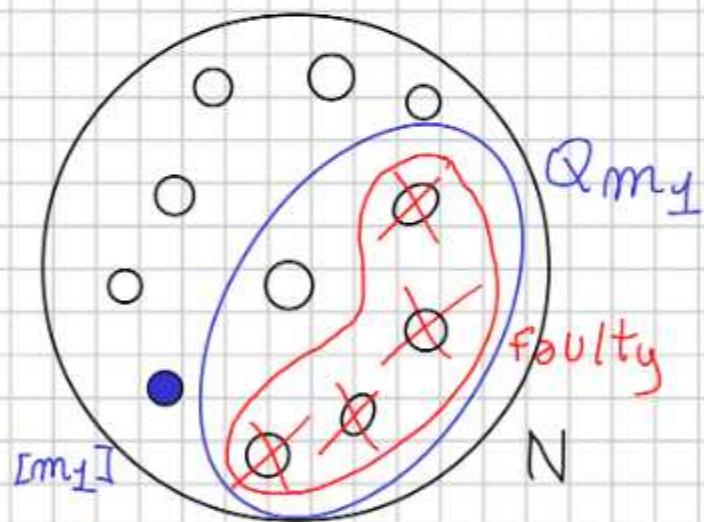


We need to assume a majority of correct processes

URB asynchronous algorithm idea:



↳ if i am unlucky in my majority, i have these faulty processes, but at least in intersection will be some correct processes (processes that provide an acknowledgement), guarantee that the message will not be lost.



original sender send the message and everytime is retransmitted, until a quorum  $Q_{m_1}$ , these processes acknowledgement the message, in any case of failure the quorum contain at least a correct process.

Complexity of three primitives (cost of each broadcast in number of messages)

	Synchronous	Asynchronous
BEB	$O(m)$	$O(m)$
RB	assume PerfectFailureDetector, P good case: failures = $\emptyset$ worst case: failures = $m-1$ $O(m)$	No PerfectFailureDetector, failure = $m-1$ $O(m^2)$
URB	no difference good or worst case, assume P failures = $m-1$ $O(m^2)$	need to be failure $< N/2$ $O(m^2)$

- $m$  = number of processes

When move from synchronous to asynchronous most of the time i must pay. In terms of resilience, the primitive is less to bust, tolerate a less number of failure, or pay in complexity of the algorithm that be in any case quadratic.

When move from weakest to strongest specification also i have a price to pay.

these algorithm are deterministic

# Uniform Reliable Broadcast

---

- There exists an algorithm for synchronous system using Perfect failure detector
- There exists an algorithm for asynchronous system when assuming a “majority of correct processes”
- Can we devise a uniform reliable broadcast algorithm for a partially synchronous system (using an eventually perfect failure detector) but without the assumption of a majority of correct processes?

even if in the system are not any failure, the primitive does not guarantee that everybody get the message

## Probabilistic broadcast

- Message delivered 99% of the times
- Not fully reliable
- Large & dynamic groups ↗ use this in practice, like block-chain
- Acks make reliable broadcast not scalable

deterministic used in small system, because of quadratic cost

# Ack Implosion and ack tree

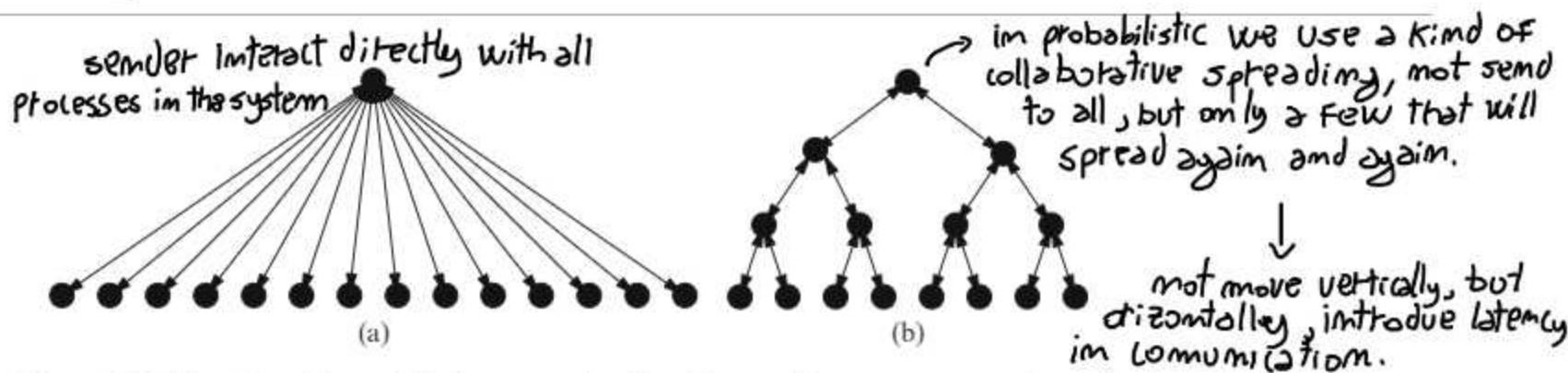


Figure 3.5: Direct vs. hierarchical communication for sending messages and receiving acknowledgments

↳ tree is a fragile structure, maintaining it is complex, for this probabilistic.

Problems:

Process spends all its time by doing the ack task

Maintaining the tree structure

# Probabilistic Broadcast

---

## Module 3.7: Interface and properties of probabilistic broadcast

---

Module:

Name: ProbabilisticBroadcast, instance  $pb$ .

Events:

Request:  $\langle pb, \text{Broadcast} \mid m \rangle$ : Broadcasts a message  $m$  to all processes.

Pb\_Broadcast(msg)

Indication:  $\langle pb, \text{Deliver} \mid p, m \rangle$ : Delivers a message  $m$  broadcast by process  $p$ .

Pb\_Deliver(msg)

Properties:

**PB1: Probabilistic validity:** There is a positive value  $\varepsilon$  such that when a correct process broadcasts a message  $m$ , the probability that every correct process eventually delivers  $m$  is at least  $1 - \varepsilon$ .

PbB

↳  $\varepsilon$  depend on specific implementation

**PB2: No duplication:** No message is delivered more than once.

**PB3: No creation:** If a process delivers a message  $m$  with sender  $s$ , then  $m$  was previously broadcast by process  $s$ .

---

# Gossip Dissemination

*based on epidemiology theory*

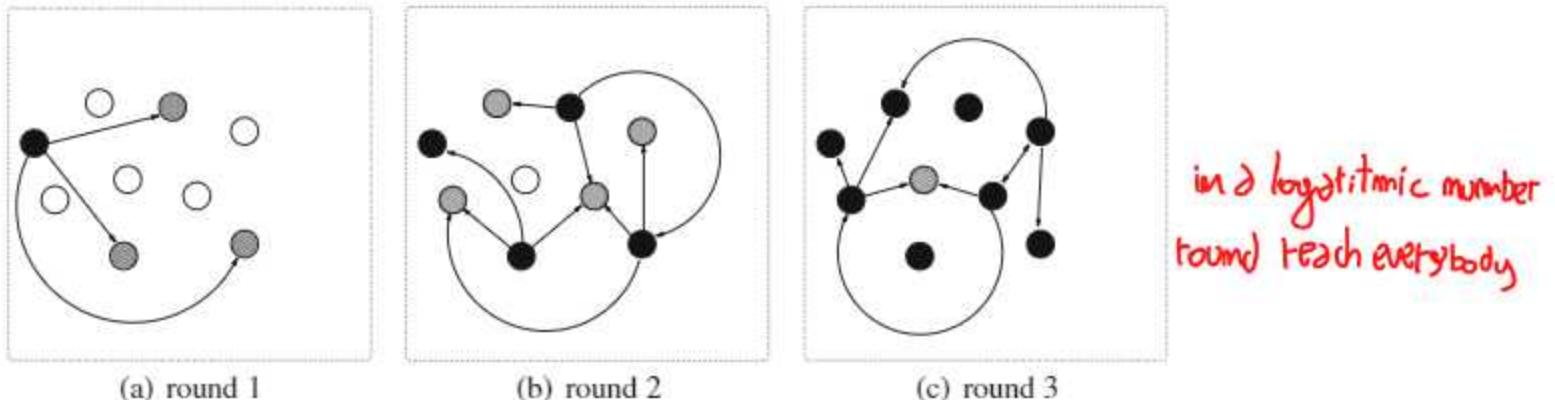


Figure 3.6: Epidemic dissemination or gossip (with fanout 3)

- A process sends a message to a set of randomly chosen  $k$  processes
- A process receiving a message for the first time forwards it to a set of  $k$  randomly chosen processes (this operation is also called a round)
- The algorithm performs a maximum number of  $r$  rounds

# Eager Probabilistic Broadcast

---

## Algorithm 3.9: Eager Probabilistic Broadcast

---

Implements:

ProbabilisticBroadcast, instance  $pb$ .

Uses:

FairLossPointToPointLinks, instance  $fl$ . *not Perfect!*

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

procedure  $gossip(msg)$  is  
    forall  $t \in \text{picktargets}(k)$  do trigger  $\langle fl, Send \mid t, msg \rangle$ ;

upon event  $\langle pb, Broadcast \mid m \rangle$  do  
     $delivered := delivered \cup \{m\}$ ;  
    trigger  $\langle pb, Deliver \mid self, m \rangle$ ;  
     $gossip([\text{GOSSIP}, \text{self}, m, R])$ ;

upon event  $\langle fl, Deliver \mid p, [\text{GOSSIP}, s, m, r] \rangle$  do  
    if  $m \notin delivered$  then  
         $delivered := delivered \cup \{m\}$ ;  
        trigger  $\langle pb, Deliver \mid s, m \rangle$ ;  
    if  $r > 1$  then  $gossip([\text{GOSSIP}, s, m, r - 1])$ ;

*generate the set to spread gossip*

```
function picktargets( $k$ ) returns set of processes is
    targets :=  $\emptyset$ ;
    while # (targets) <  $k$  do
        candidate := random( $fl \setminus \{\text{self}\}$ );
        if candidate  $\notin$  targets then
            targets := targets  $\cup$  {candidate};
    return targets;
```

*↳ round number ~ greater it is and greater is the probability  
of spread to all processes the message*

*↳ also depend  
on  $K$  the probability*

# References

---

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

- Chapter 3 - from Section 3.9 (except 3.9.6)
- Chapter 6 – Section 6.1

Dependable Distributed Systems  
Master of Science in Engineering in  
Computer Science

AA 2022/2023

---

LECTURE 10: CONSENSUS

# Consensus Problem

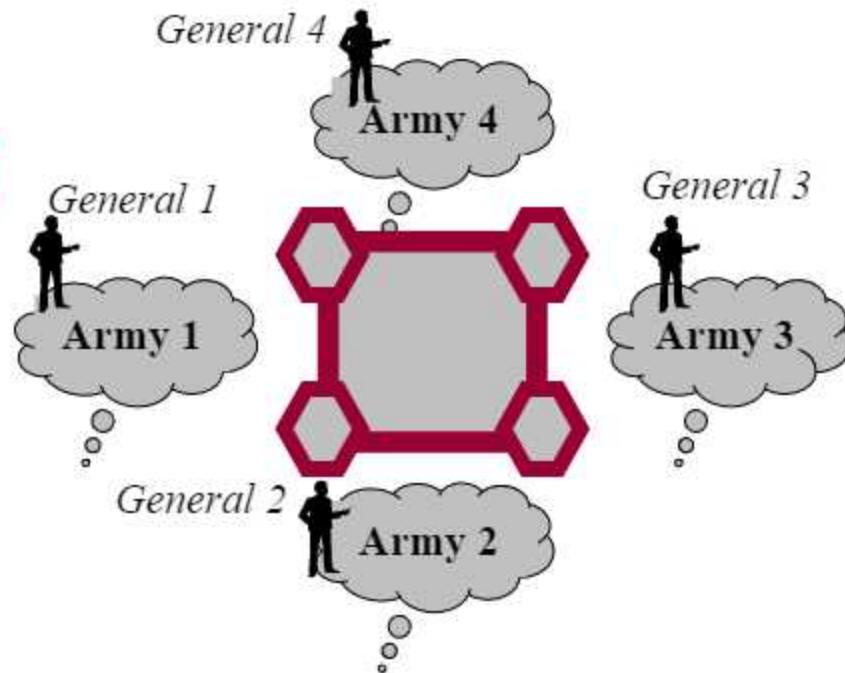
---

- A group of processes must agree on a value that has been proposed by one of them (e.g., commit/abort of a transaction).
- It is an abstraction of a class of problems where processes start with their opinion and then converge on one of them
- It is a fundamental problem
- We study algorithms working on weak models

# Consensus Example

Lamport example

all generals have  
a own perspective,  
need to swap their  
opinion

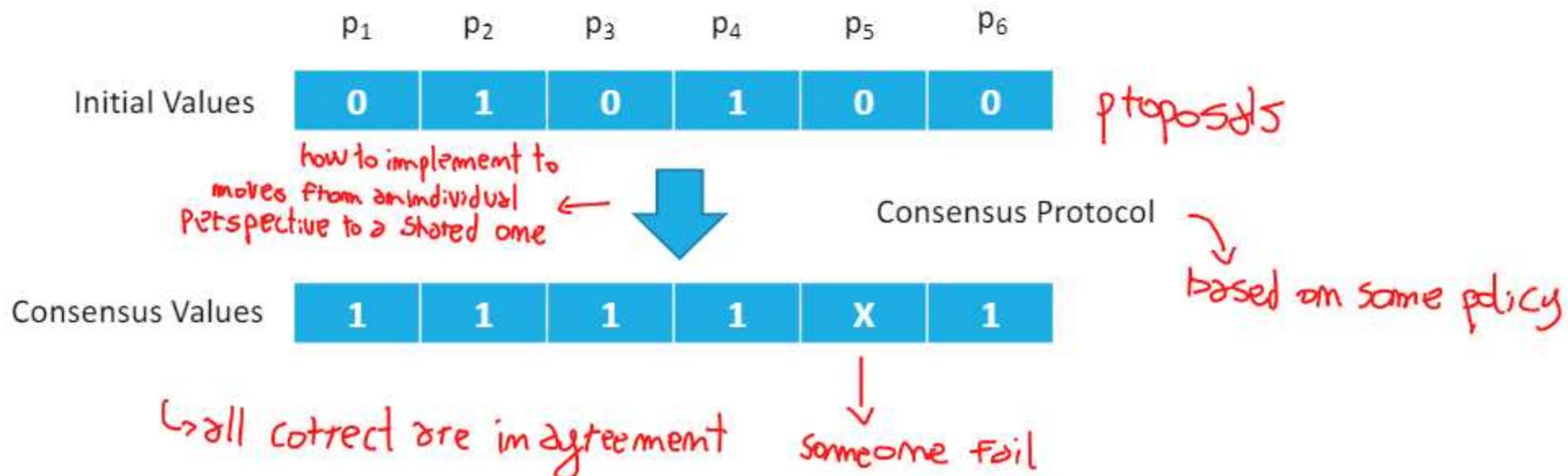


study from 1980,  
there are open problem

Generals have to reach consensus between attack and get back

# Consensus Definition

- Set of initial values  $\in \{0,1\}$ . *→ extended also on multiple values*
- Every process has to decide the same value  $\in \{0,1\}$  based on the initial proposals.



# Consensus Specification

---

Module 5.1: Interface and properties of (regular) consensus

Module:

Name: Consensus, instance  $c$ .

Events:

Request:  $\langle c, \text{Propose} \mid v \rangle$ : Proposes value  $v$  for consensus. *, not propose twice*

Indication:  $\langle c, \text{Decide} \mid v \rangle$ : Outputs a decided value  $v$  of consensus.

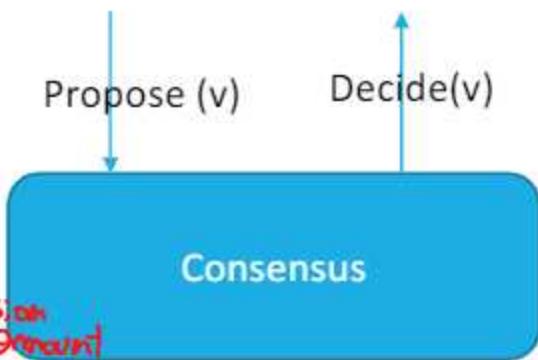
Properties:

C1: *Termination*: Every correct process eventually decides some value. *, reach a decision in a finite amount of time*

C2: *Validity*: If a process decides  $v$ , then  $v$  was proposed by some process.

C3: *Integrity*: No process decides twice.

C4: *Agreement*: No two correct processes decide differently.



↑  
people that prove it

# FLP Impossibility Result



no algorithm can guarantee to reach  
consensus in an asynchronous system, even  
with one process crash failure.

i'm not able distinguish if i miss some proposal because  
processes are faulty or very slow

Fisher, Lynch e Patterson (FLP result)

Ref: Journal of the ACM, Vol. 32, No. 2, April 1985.

→ could happen that  
decide without consider  
the false crash and  
violate agreement

i think is slow, but is crashed  
and violates liveness

# A Consensus Implementation in Synchronous Systems: Flooding Consensus

*no particular process in the system*

Basic Idea:

- Processes exchange their values
- when all the proposals from correct processes are available a one value can be chosen

*↳ all opinion have the same importance*



Problem:

- due to failures, some values can be lost

*deterministic policy,  
most voted and so on...*

Solution:

- A value can be selected only when no failures happen during the communication

# A Consensus Implementation in Synchronous Systems: Flooding Consensus

## Algorithm 5.1: Flooding Consensus

Implements:

Consensus, instance  $c$ .

*# keep track processes that provided a value  
in current round  
# possibility of reiterate in case of failure*

Uses:

BestEffortBroadcast, instance  $beb$ ;

PerfectFailureDetector, instance  $P$ . *~ Fundamental*

upon event  $\langle c, \text{Init} \rangle$  do

$correct := \Pi$ ; *~ processes that i must wait*

$round := 1$ ;

$decision := \perp$ ; *~ store the decided value*

$receivedfrom := \{\emptyset\}^N$ ;

$proposals := \{\emptyset\}^N$ ;

$receivedfrom[0] := \Pi$ ; *~ always store the value proposed by the corresponding process*

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

$correct := correct \setminus \{p\}$ ;

upon event  $\langle c, \text{Propose} \mid v \rangle$  do

$proposals[1] := proposals[1] \cup \{v\}$ ;

    trigger  $\langle beb, \text{Broadcast} \mid [\text{PROPOSAL}, 1, proposals[1]] \rangle$ ;

*3 set of values proposed*

upon event  $\langle beb, \text{Deliver} \mid p, [\text{PROPOSAL}, r, ps] \rangle$  do

$receivedfrom[r] := receivedfrom[r] \cup \{p\}$ ;

$proposals[r] := proposals[r] \cup ps$ ;

upon  $correct \subseteq receivedfrom[\text{round}] \wedge decision = \perp$  do

    if  $receivedfrom[\text{round}] = receivedfrom[\text{round} - 1]$  then

$decision := \min(proposals[\text{round}])$ ;

        trigger  $\langle beb, \text{Broadcast} \mid [\text{DECIDED}, decision] \rangle$ ;

        trigger  $\langle c, \text{Decide} \mid decision \rangle$ ;

    else *~ someone fail*

$round := round + 1$ ;

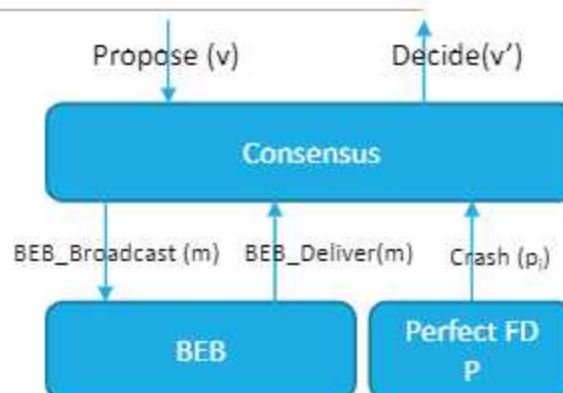
        trigger  $\langle beb, \text{Broadcast} \mid [\text{PROPOSAL}, round, proposals[round - 1]] \rangle$ ;

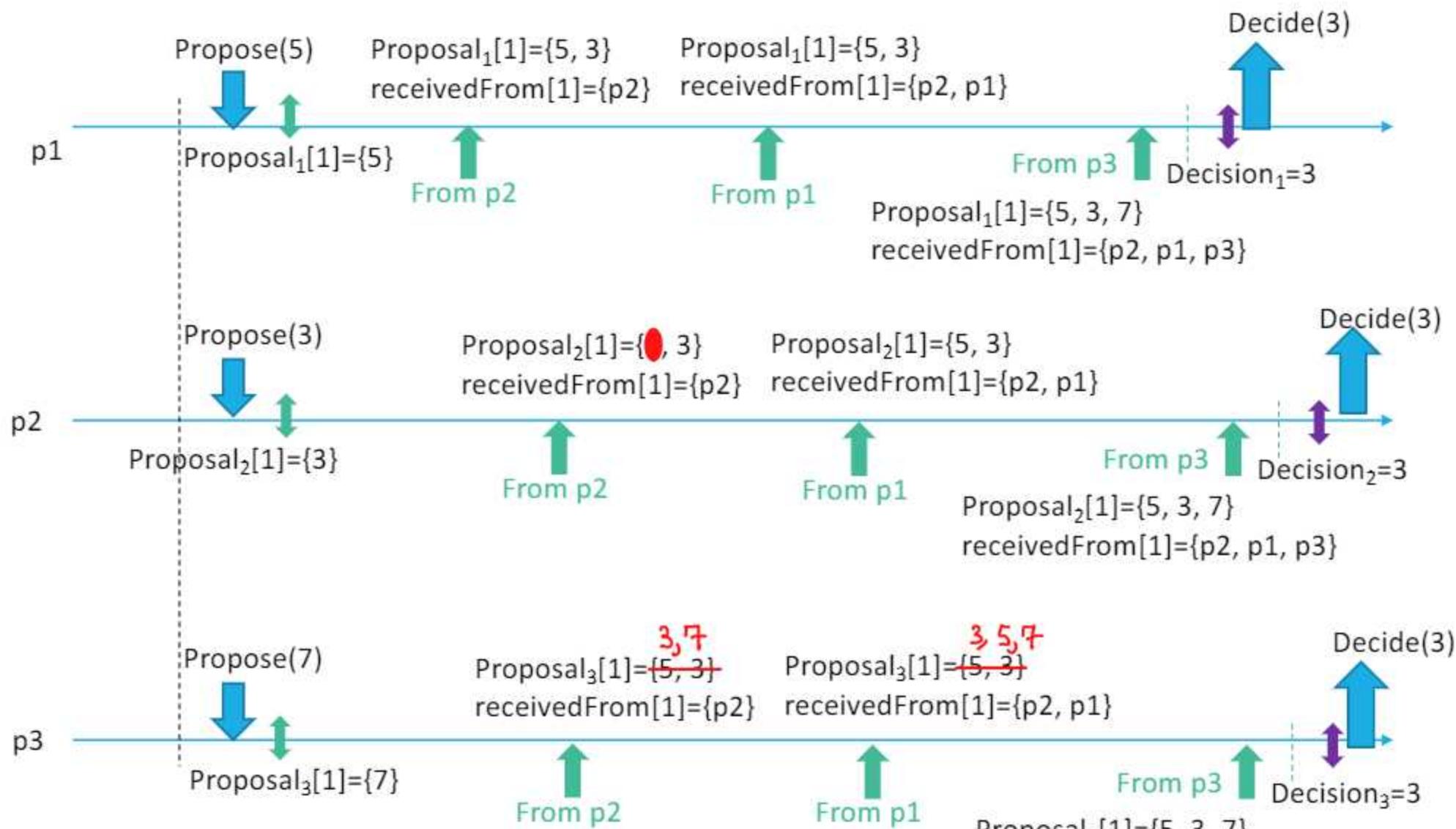
upon event  $\langle beb, \text{Deliver} \mid p, [\text{DECIDED}, v] \rangle$  such that  $p \in correct \wedge decision = \perp$  do

$decision := v$ ;

    trigger  $\langle beb, \text{Broadcast} \mid [\text{DECIDED}, decision] \rangle$ ;

    trigger  $\langle c, \text{Decide} \mid decision \rangle$ ;





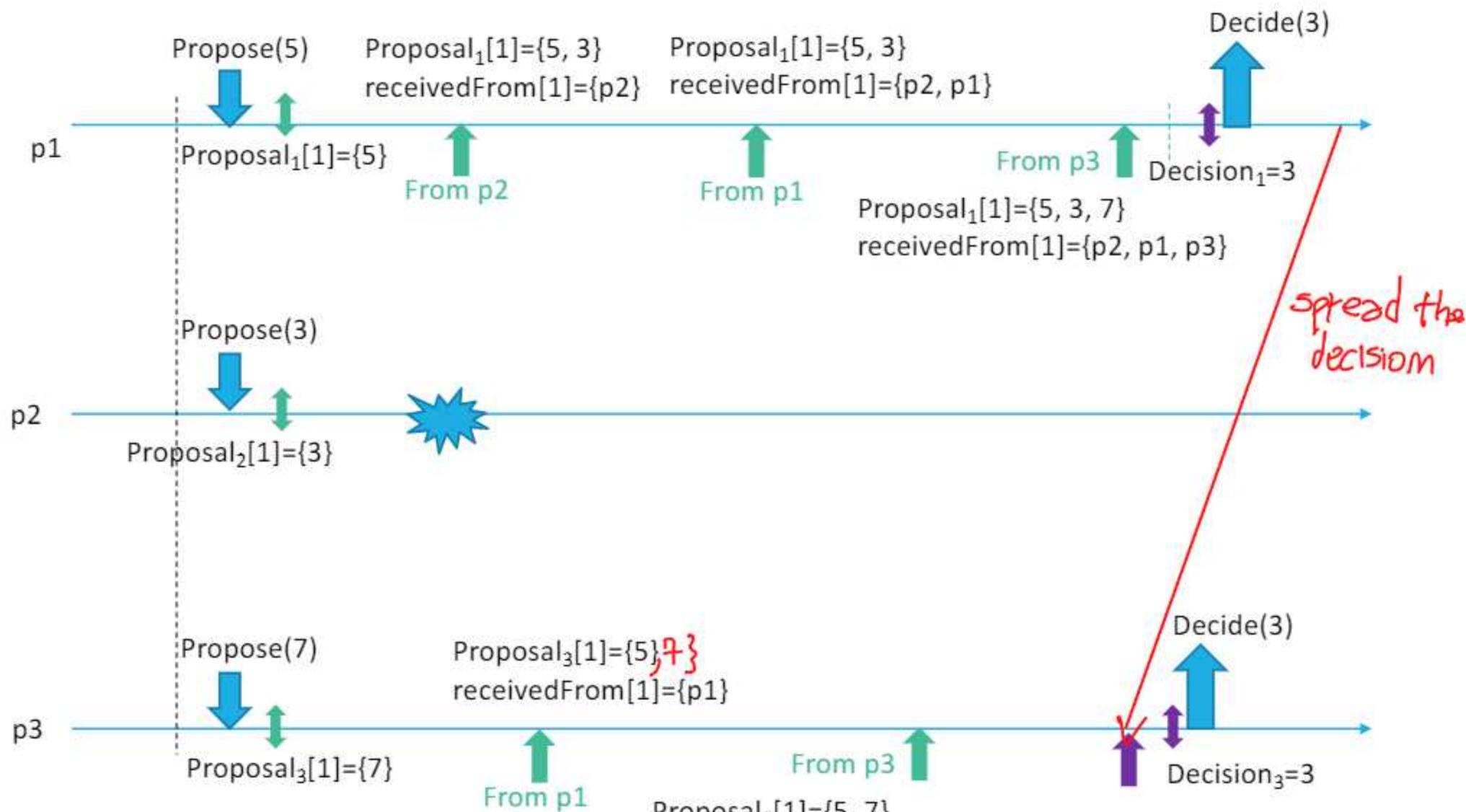
Correct={p1, p2, p3}

Round=1

Decision=⊥

receivedFrom[0]={p1, p2, p3}

Proposals[0]={}



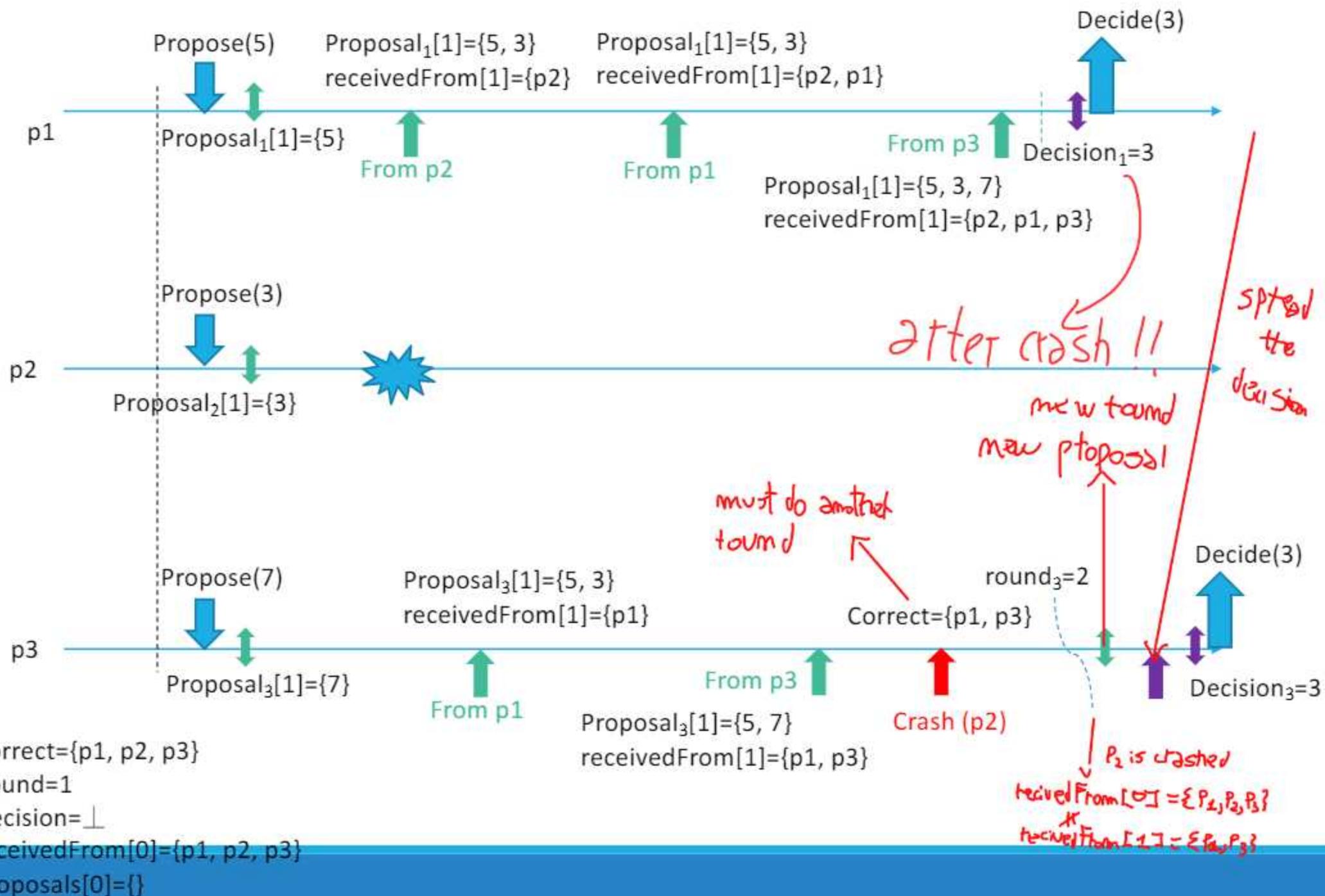
Correct= $\{p1, p2, p3\}$

Round=1

Decision= $\perp$

$receivedFrom[0]=\{p1, p2, p3\}$

$Proposals[0]=\{\}$



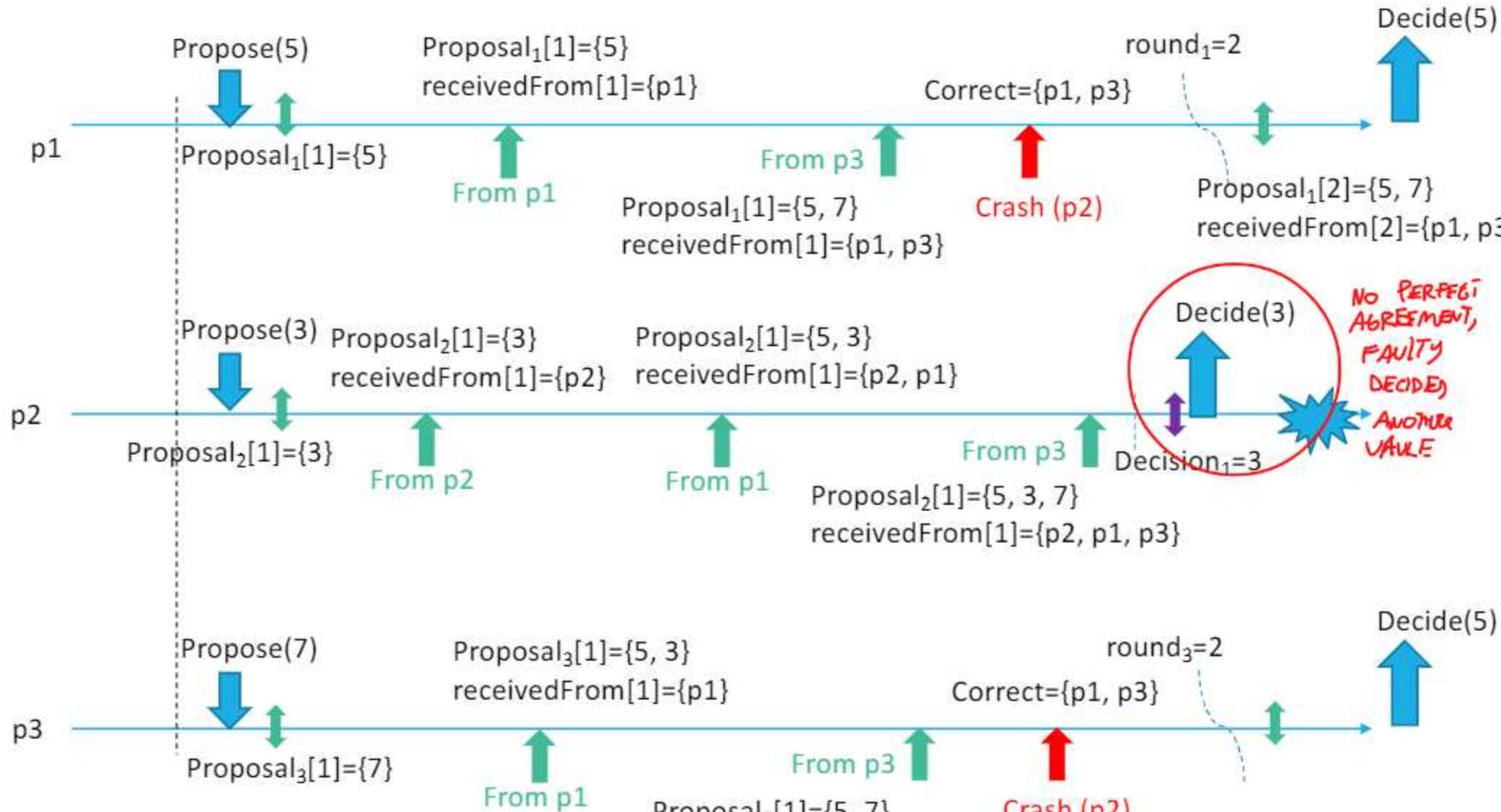
# Correctness and Performance

## Correctness

- Validity and integrity follow from the properties on the communication channels  $P_2P$
- Max N-1 fails  $\leftrightarrow$  Termination. At most after  $N$  rounds processes decide, thanks to PerfectFailureDetector
- Agreement. The same deterministic function is applied to the same values by correct processes

## Performance

- Best Case (No failures). One communication round ( $2N^2$  messages)
  - Worst case ( $n-1$  failures).  $N^2$  messages exchanged for each communication step and at most  $N$  rounds  $\Rightarrow N^3$  messages
- $N^2$  BEB messages for proposal and  $N^2$  BEB messages for decide
- 



Final State (Round 3):

- $Correct=\{p1, p2, p3\}$
- $Round=1$
- $Decision=\perp$
- $ReceivedFrom[0]=\{p1, p2, p3\}$
- $Proposals[0]=\{\}$

# Uniform Consensus Specification

---

## Module 5.2: Interface and properties of uniform consensus

---

Module:

Name: UniformConsensus, instance  $uc$ .

Events:

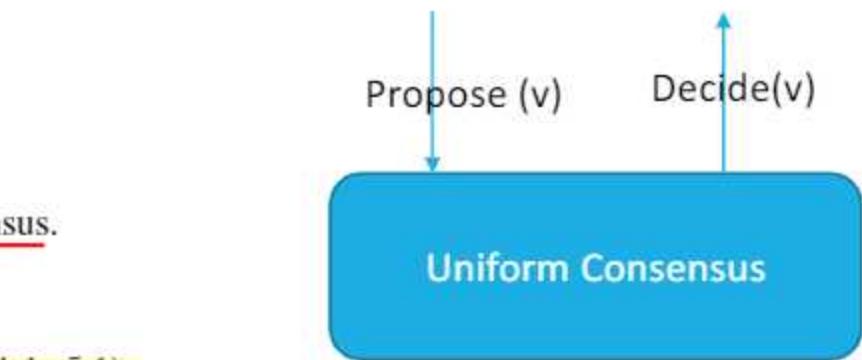
Request:  $\langle uc, \text{Propose} \mid v \rangle$ : Proposes value  $v$  for consensus.

Indication:  $\langle uc, \text{Decide} \mid v \rangle$ : Outputs a decided value  $v$  of consensus.

Properties:

UC1–UC3: Same as properties C1–C3 in (regular) consensus (Module 5.1).

UC4: *Uniform agreement*: No two processes decide differently.



*~> also faulty one*

# Uniform Consensus and Flooding Consensus algorithm

---

Does the flooding consensus algorithm (described in the previous slides) satisfy the Uniform Consensus specification? *NO, Example of before*

# Uniform Consensus Implementation in Synchronous Systems

*mom decide as soon you can, but ensure all have same decision*

## Algorithm 5.3: Flooding Uniform Consensus

Implements:

UniformConsensus, instance  $uc$ .

Uses:

BestEffortBroadcast, instance  $beb$ ;  
PerfectFailureDetector, instance  $\mathcal{P}$ .

upon event  $\langle uc, Init \rangle$  do

```

correct :=  $\Pi$ ;
round := 1;
decision :=  $\perp$ ;
proposalset :=  $\emptyset$ ;
receivedfrom :=  $\emptyset$ ;
```

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

```
correct :=  $correct \setminus \{p\}$ ;
```

upon event  $\langle uc, Propose \mid v \rangle$  do

```

proposalset := proposalset  $\cup \{v\}$ ;
trigger  $\langle beb, Broadcast \mid [PROPOSAL, 1, proposalset] \rangle$ ;
```

No more related to the round

upon event  $\langle beb, Deliver \mid p, [PROPOSAL, r, ps] \rangle$  such that  $r = round$  do

```

receivedfrom := receivedfrom  $\cup \{p\}$ ;
proposalset := proposalset  $\cup ps$ ;
```

upon  $correct \subseteq receivedfrom \wedge decision = \perp$  do

```

if  $round = N$  then
  decision :=  $\min(proposalset)$ ;
  trigger  $\langle uc, Decide \mid decision \rangle$ ;
```

else

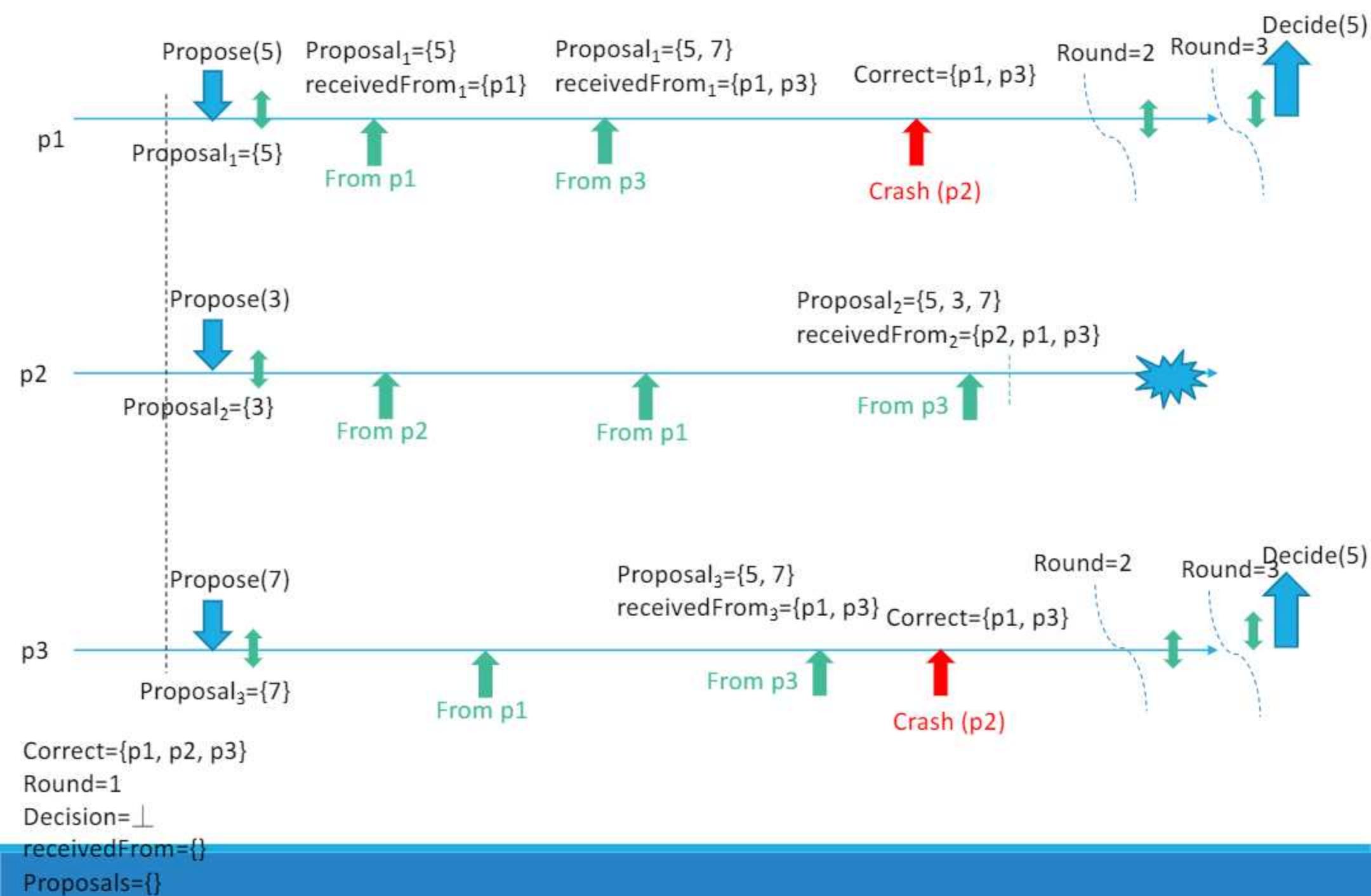
```

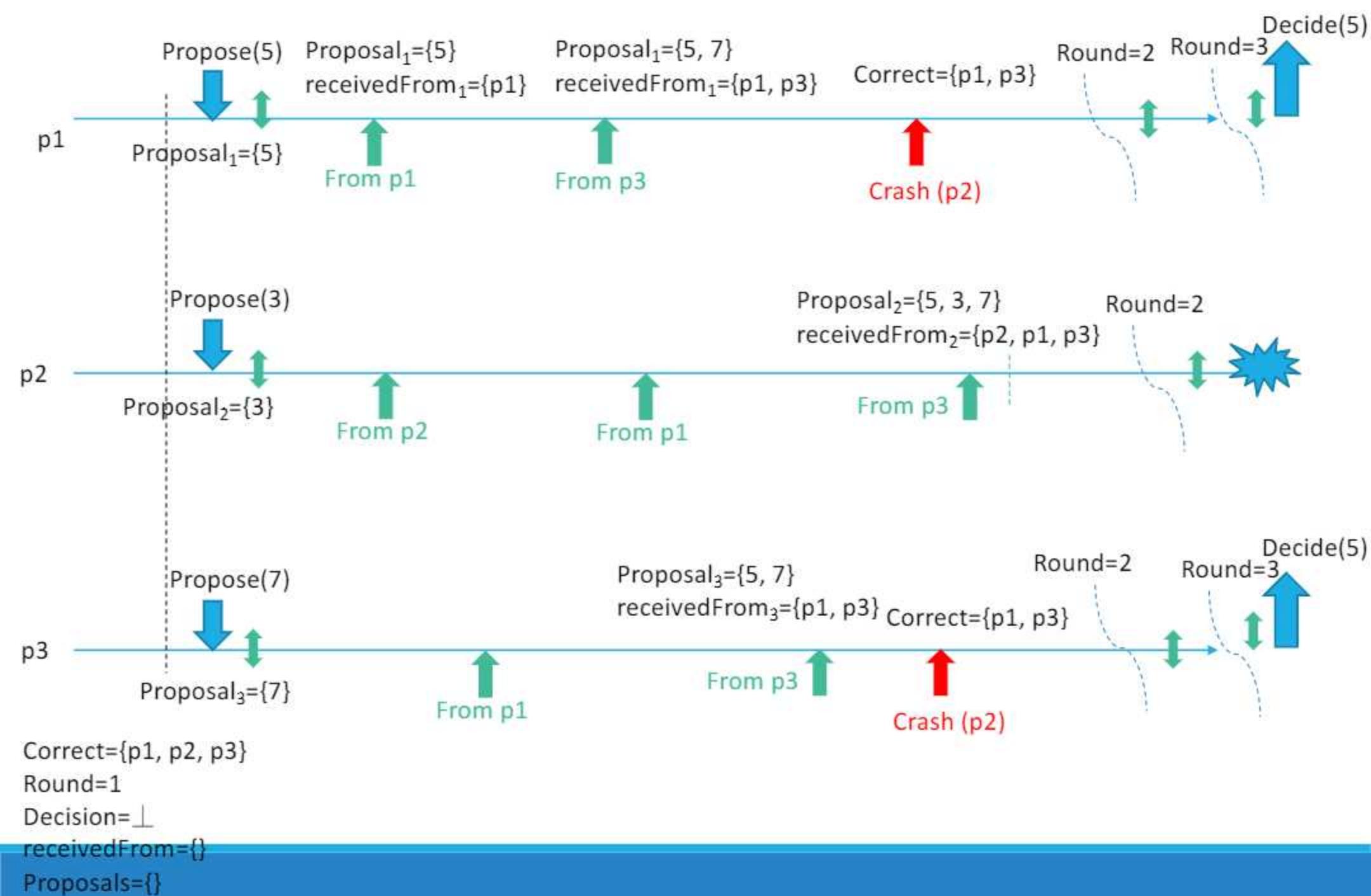
  round := round + 1;
  receivedfrom :=  $\emptyset$ ;
  trigger  $\langle beb, Broadcast \mid [PROPOSAL, round, proposalset] \rangle$ ;
```

Decision only at the end

Cleaned at the beginning of each round







If we don't have a perfect failure detector, we have eventually perfect accuracy then we violate agreement in the two algorithms. If we have eventually strong completeness we could not terminate.

# Correctness and Performance

---

## Correctness

- The validity and integrity properties follow from the algorithm and from the properties of best-effort broadcast
- The termination property is ensured as all correct processes reach round N and decide in that round
  - the strong completeness property of the failure detector implies that no correct process waits indefinitely for a message from a process that has crashed, as the crashed process is eventually removed from correct.
- The uniform agreement holds because all processes that reach round N have the same set of values in their variable proposalset.

## Performance

- N communication steps and  $O(N^3)$  messages for all correct processes to decide

# References

---

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

- Chapter 5, Sections 5.1.1, 5.1.2, 5.2.1, 5.2.2

Dependable Distributed Systems  
Master of Science in Engineering in  
Computer Science

AA 2022/2023

---

LECTURE 11: CONSENSUS - PAXOS

# Impossibility Result

Consensus in asynchronous system



**Impossibility of Consensus in asynchronous  
systems in the presence also of a single  
crash**

to be solved deterministically, but we  
can solve with paxos in partially asynchronous

Fisher, Lynch e Patterson (FLP result).

Ref: Journal of the ACM, Vol. 32, No. 2, April 1985.

Consider eventually synchronous system, at some point the synchronous  
bound solve the problem termination

## Paxos

The Paxos family of algorithms was introduced in 1999 to provide a viable solution to consensus in asynchronous settings:

- Safety is always guaranteed, if you decide then you decide in agreement on a value proposed by someone
- The algorithm makes some progress (liveness) only when the network behaves in a "good way" for long enough periods of time (partial synchrony).
  - you can use fairless link and also recovery is accepted
  - not guarantee liveness if system fully asynchronous

# Paxos – recap about properties

---

Let us think about what the protocol should do. *satisfy these properties*

- Only a value that has been proposed may be chosen, *don't invent some value*
- Only a single value is chosen
- A process never learns that a value has been chosen unless it actually has been

*decision should follow agreement of other*

# Paxos – System model and basic assumptions

Processes behave on their own speed without any type of synchronization

- Agents operate at arbitrary speed, may fail by stopping (and may restart).

Correctly accepted

- Observation: Since all agents may fail after a value is chosen and then restart, a solution is impossible unless some information can be remembered by an agent that has failed and restarted.

→ some component state information

- Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted.

→ 11KB 3 fiteloss

only assumption is if deliver something, someone else deliver it

# Paxos - Actors *three roles*

## Actors in the basic Paxos protocol

- Proposers: propose a value *~on start of algorithm*
- Acceptors: processes that must commit on a final decided value *actively participate in discussion*
- Learners: passively assist to the decision and obtain the final decided value

*each actor have the code for each role, are not fixed*

*processes became very slow*

# Paxos: choosing a value

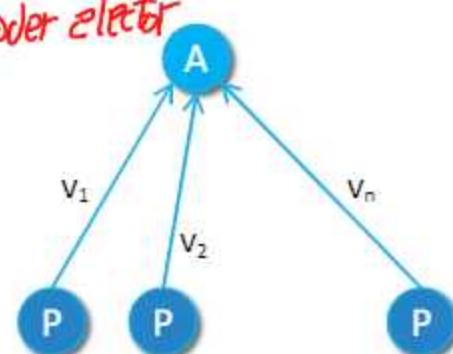
The easiest way to solve the problem is to have a single acceptor

- A proposer sends a proposal to the acceptor
- The acceptor chooses the first proposed value it receives and informs everybody about its choice

*linear complexity*      *but we don't have leader elector*

What if the only acceptor fails ? *block computation* ↑

=> We must have more than one acceptor

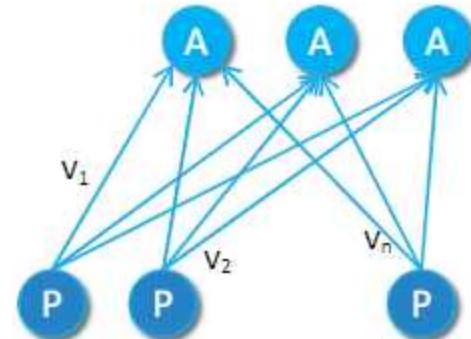


*Coordinator based approach, without fail is perfect, must handle failure, multiple acceptors*

# Paxos: choosing a value

## Having Multiple acceptors

- Each proposer sends a proposal to a set of acceptors
- Each acceptor needs to chose a value

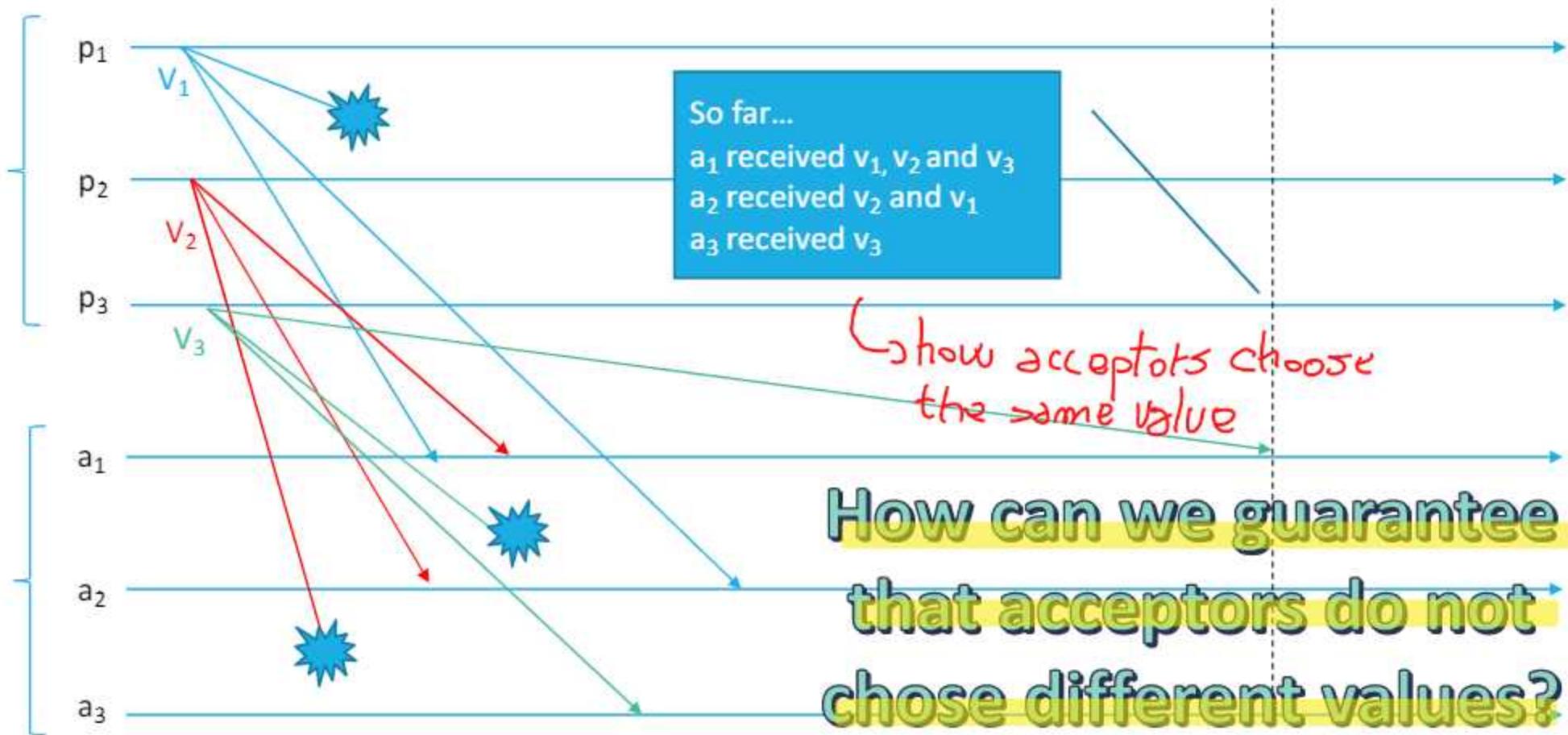


Each Acceptor may receive a different set of proposals

# Paxos: choosing a value

PROPOSERS

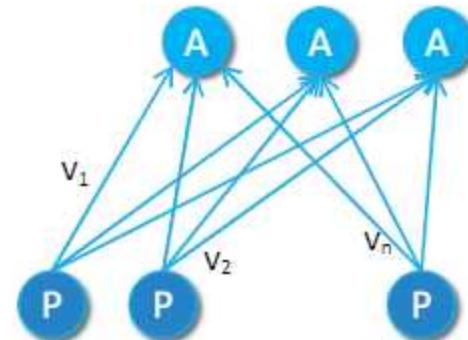
ACCEPTORS



# Paxos: choosing a value

A possible solution....

- An acceptor may accept at most one value
- A value is chosen when a majority of acceptors accept it



The majority is needed to guarantee that only a value is accepted

# Paxos: choosing a value

---

**Which value should be accepted by every acceptor?**

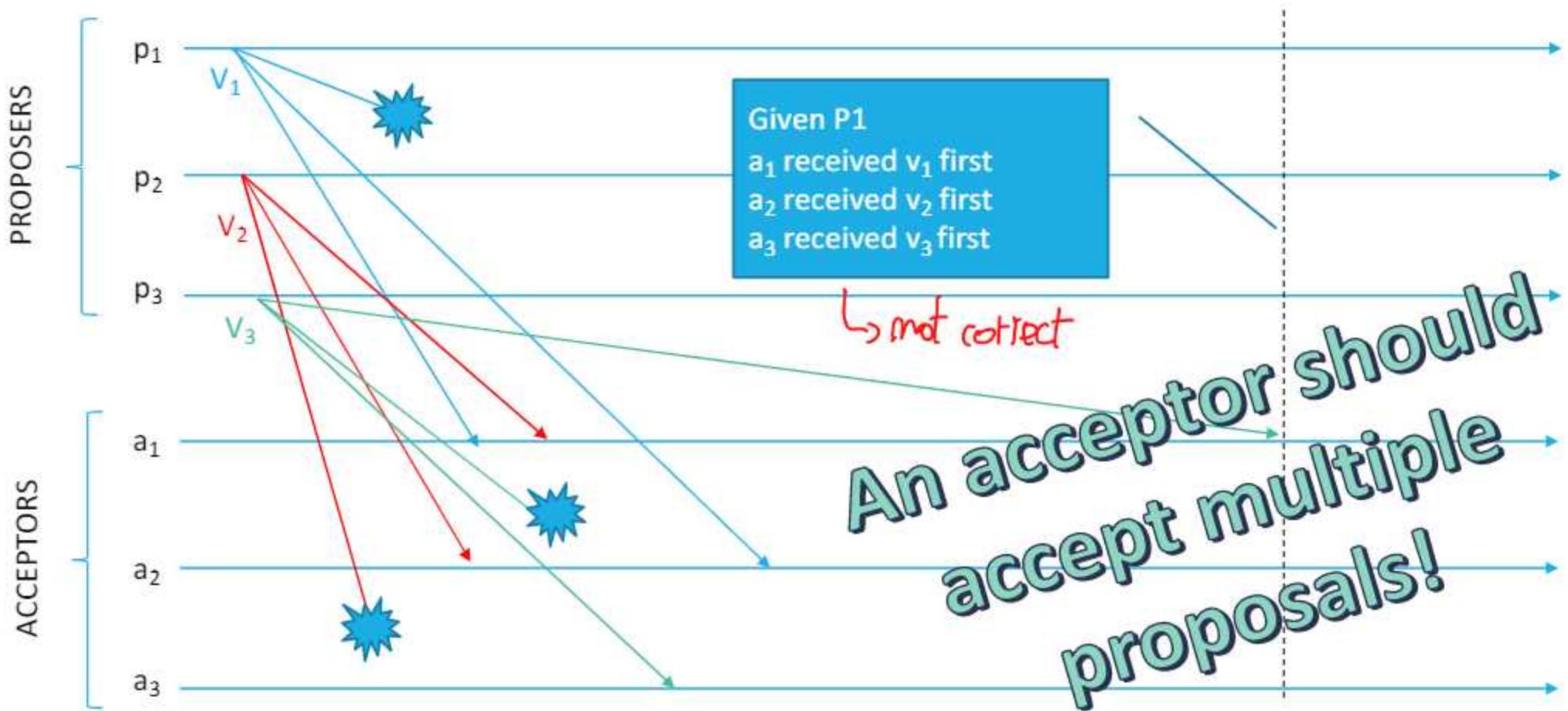
**P1: An acceptor must accept the first proposal it receives** No!

---

*Any bound on latency*

Problem: if several values are concurrently proposed by different proposers, none of them could reach the needed majority. We have a sort of deadlock.

# Paxos: choosing a value



# Paxos: choosing a value

---

**Solution** *same mechanism of mutual exclusion*

- We keep track of different proposal assigning them a proposal number that is unique (total order of proposals)
- A value is chosen when a single proposal with that value has been accepted by a majority of the acceptors (chosen proposal).
- We can allow multiple proposal to be accepted, but all accepted proposals must have the same value.

**P2:** if a proposal with value  $v$  is chosen, every higher-numbered proposal that is chosen has value  $v$

# Paxos: choosing a value

For a value  $v$  to be chosen, a proposal containing it must have been accepted by at least one acceptor...

↳ need to have the highest timestamp

...thus:

P2a: if a proposal with value  $v$  is accepted, every higher-numbered proposal that is accepted by any acceptor has value  $v$

One more problem: What if a proposal with value  $v$  is accepted while an acceptor  $c$  never saw it ?

A new proposer could wake up and propose to  $c$  a different value  $v'$  that  $c$  must accept due to P1

we guarantee a majority to reach a decision, a consistency in the view

for have a chance need the greatest timestamp

## Paxos: choosing a value

---

We must slightly modify P2a to take into account this case:

---

P2b: if a proposal with value  $v$  is chosen, every higher-numbered proposal issued by any proposer has value  $v$

## How can we guarantee P2b ?

# Paxos: choosing a value

---

Assume a proposal  $m$  with value  $v$  has been accepted

We should guarantee that any proposal  $n$ , with  $n > m$  has value  $v$

We could prove it by induction on  $n$  assuming that every proposal with number in  $[m, n-1]$  has value  $v$

For  $m$  to be accepted there is a set  $C$  of acceptors (a majority) that accept it

# Paxos: choosing a value

---

Therefore, the assumption that **m** has been accepted implies that:

---

- Every acceptor in  $C$  has accepted a proposal with number in  $[m, n-1]$  and
  - every proposal in this range accepted by an acceptor has value **v**
-

# Paxos: choosing a value

Given the intersection among majorities, we can conclude that a proposal numbered  $n$  has value by ensuring that the following invariant is maintained:

$\begin{matrix} \text{value} & \nearrow \text{timestamp} \\ \downarrow & \end{matrix}$        $\nearrow \text{generated}$

**P2c:** For any  $v$  and  $n$ , if a proposal with value  $v$  and number  $n$  is issued, then there is a set  $S$  consisting of a majority of acceptors such that either (a) no acceptor in  $S$  has accepted any proposal numbered less than  $n$ , or (b)  $v$  is the value of the highest-numbered proposal among all proposals numbered less than  $n$  accepted by the acceptors in  $S$ .

By maintaining P2c we satisfy P2b.

$P2c \Rightarrow P2b \Rightarrow P2b$

# Paxos: how to ensure to choose a value

---

P2c can be maintained by asking to a proposer that wants to propose a value numbered  $n$  to learn the highest-numbered value with number less than  $n$  that

- has been accepted
- or will be accepted

by any acceptor in a majority.

# Paxos: learning about proposed values

---

Learning accepted values is easy

Instead of trying to predict the future, ask acceptors to promise they will not accept proposals numbered less than  $n$

# Paxos: learning about proposed values

---

The protocol has two main phases:

- Phase 1: prepare request ↔ response
- Phase 2: accept request ↔ response

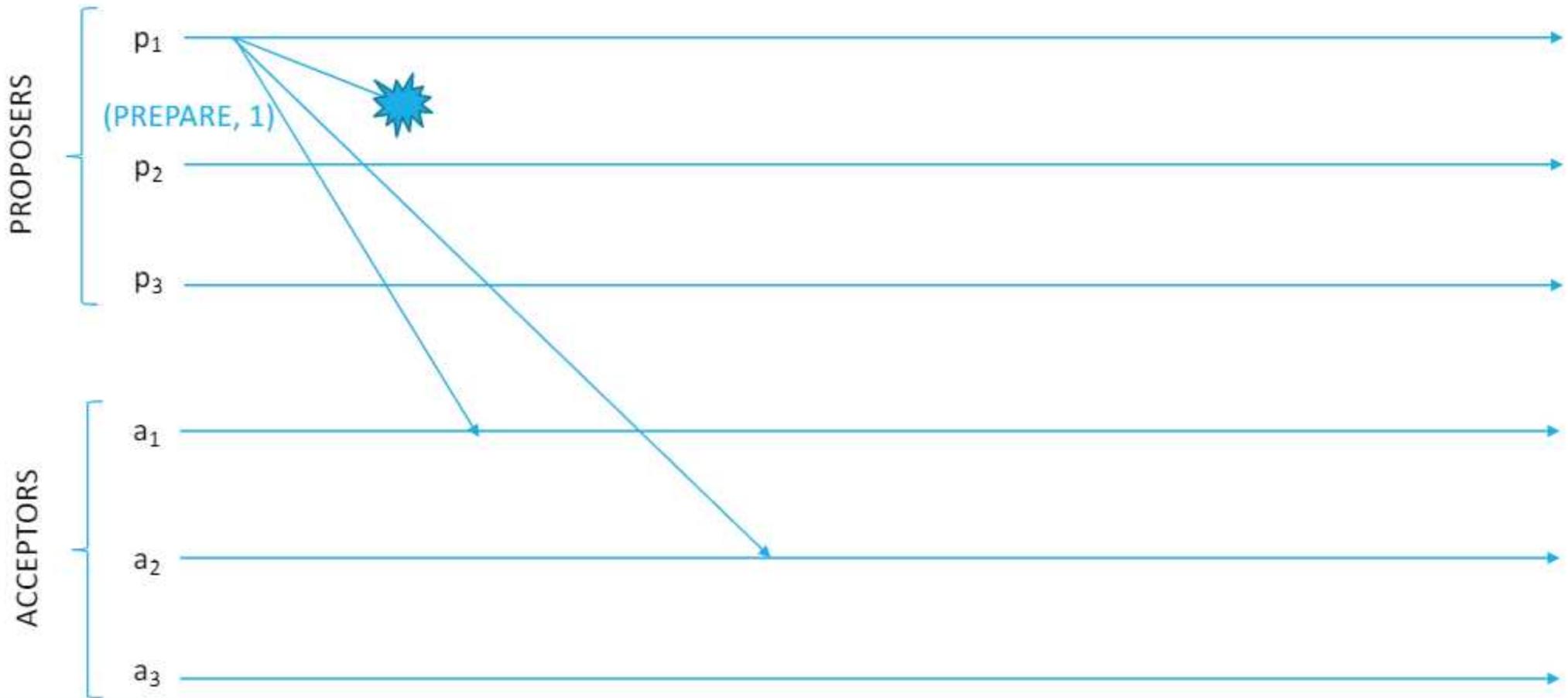
# Paxos

---

## Phase 1

- 1) A proposer chooses a new proposal version number  $n$  , and sends a prepare request (PREPARE, $n$ ) to a majority of acceptors:
  - (a) Can I make a proposal with number  $n$  ?
  - (b) if yes, do you suggest some value for my proposal?

# Paxos: Phase 1



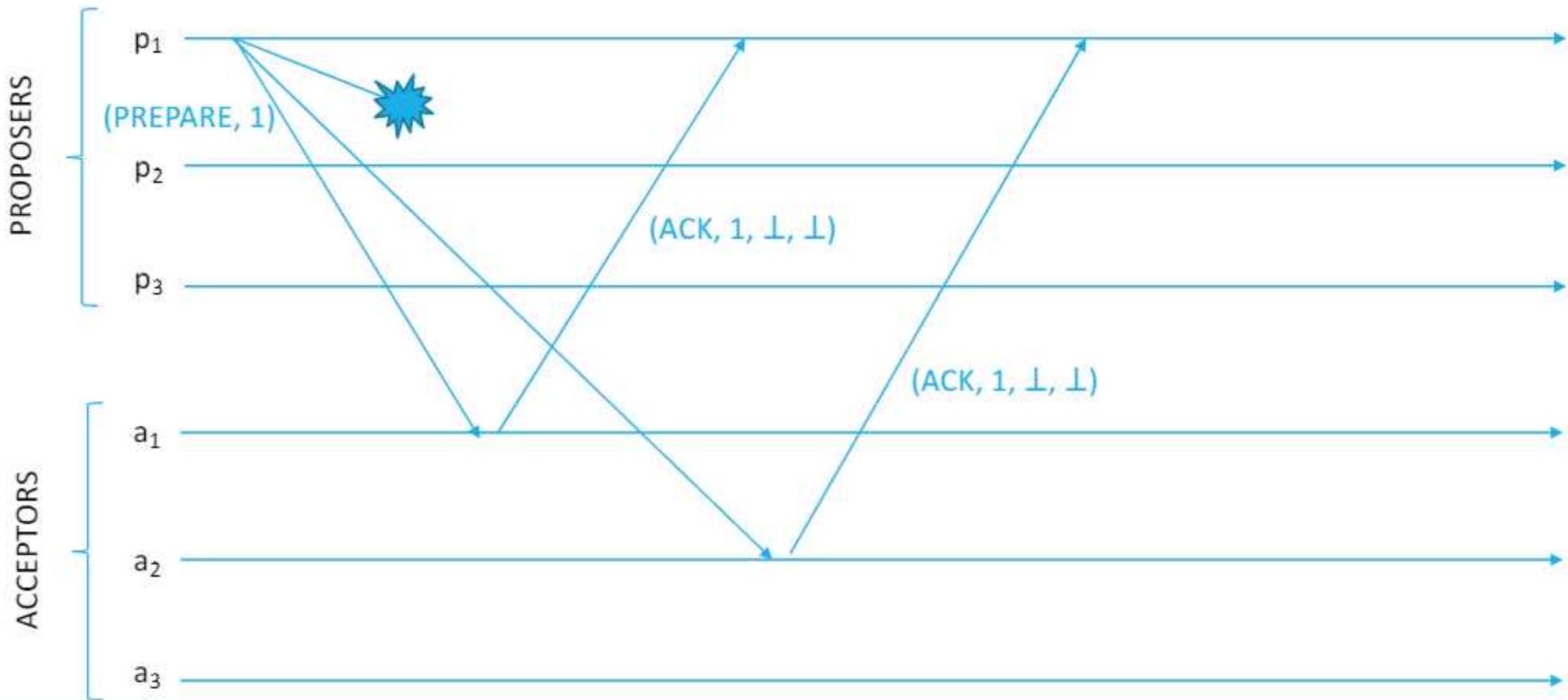
# Paxos

---

## Phase 1

- 2) If an acceptor receives a prepare request (PREPARE,  $n$ ) and it will answer sending:
  - (a) responds with a promise not to accept any more proposals numbered less than  $n$ .
  - (b) suggests the value  $v'$  of the highest-number proposal that it has accepted if any, else  $\perp$
- In particular, it will reply with the highest number less than  $n$  that it has accepted
  - (ACK,  $n$ ,  $n'$ ,  $v'$ ) if it exists
  - or (ACK,  $n$ ,  $\perp$ ,  $\perp$ ) if not

# Paxos: Phase 1



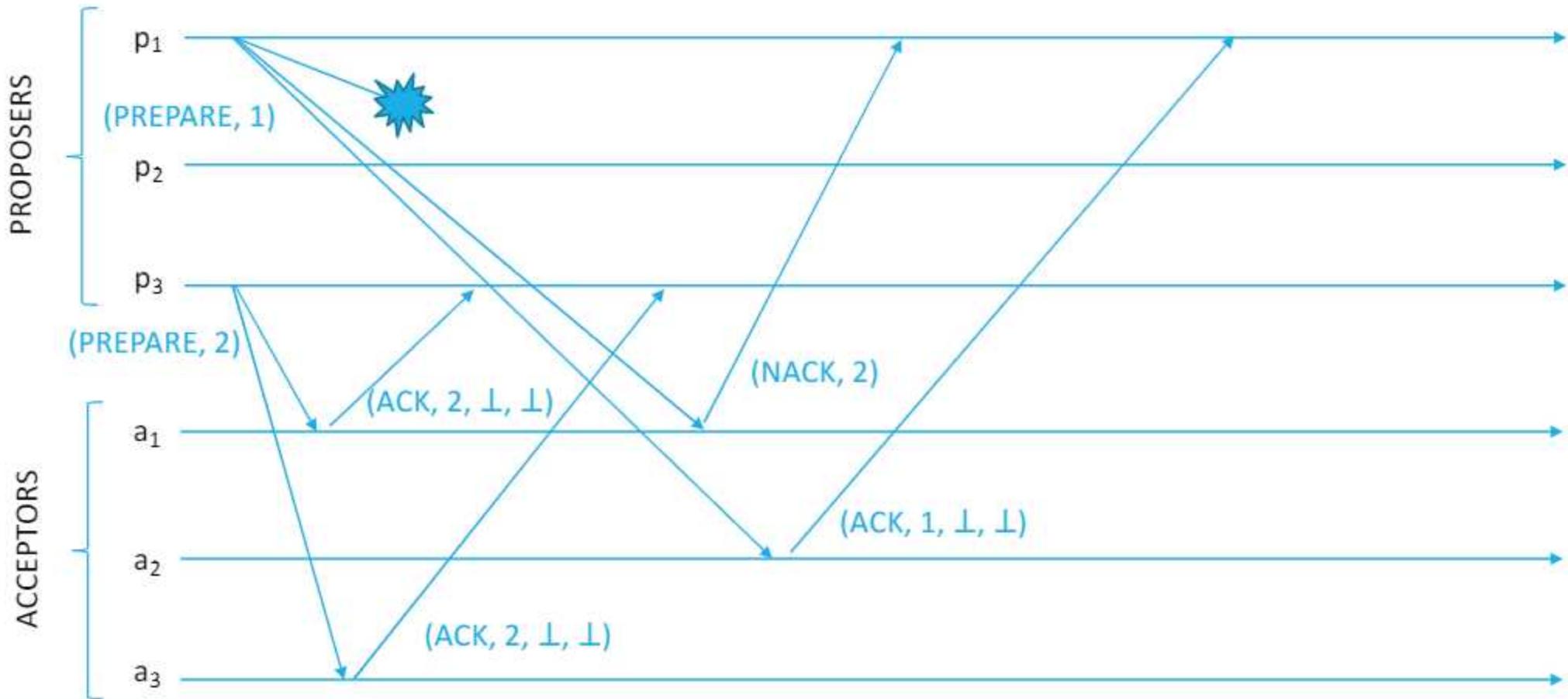
# Paxos

---

## Phase 1

- 2) If an acceptor receives a prepare request (PREPARE,  $n$ ) with  $n$  lower than  $n'$  from any prepare request it has already responded, sends out (NACK,  $n'$ )
  - (a) responds with a denial to proceed with the agreement as the proposal is too old.

# Paxos: Phase 1



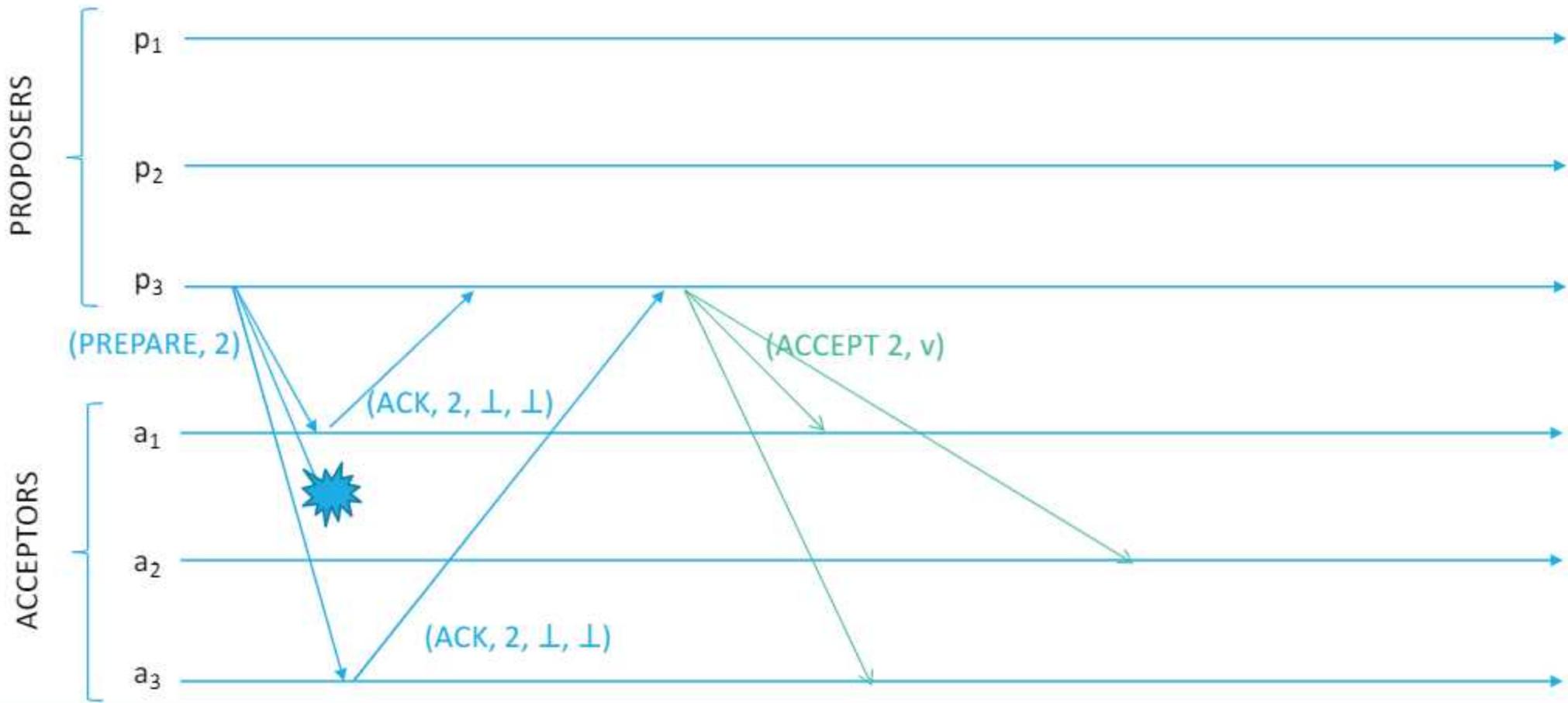
# Paxos

---

## Phase 2

- 3) If the proposer receives responses from a majority of the acceptors, then it can issue an accept request (ACCEPT,  $n$  ,  $v$ ) with number  $n$  and value  $v$ :  
  - (a)  $n$  is the number that appears in the prepare request.
  - (b)  $v$  is the value of the highest-numbered proposal among the responses (or the proposer's own proposal if none was received).

# Paxos: Phase 2



# Paxos

---

## Phase 2

- 4) If the acceptor receives an accept request (ACCEPT,  $n$  ,  $v$ ) , it accepts the proposal unless it has already responded to a prepare request having a number greater than  $n$ .

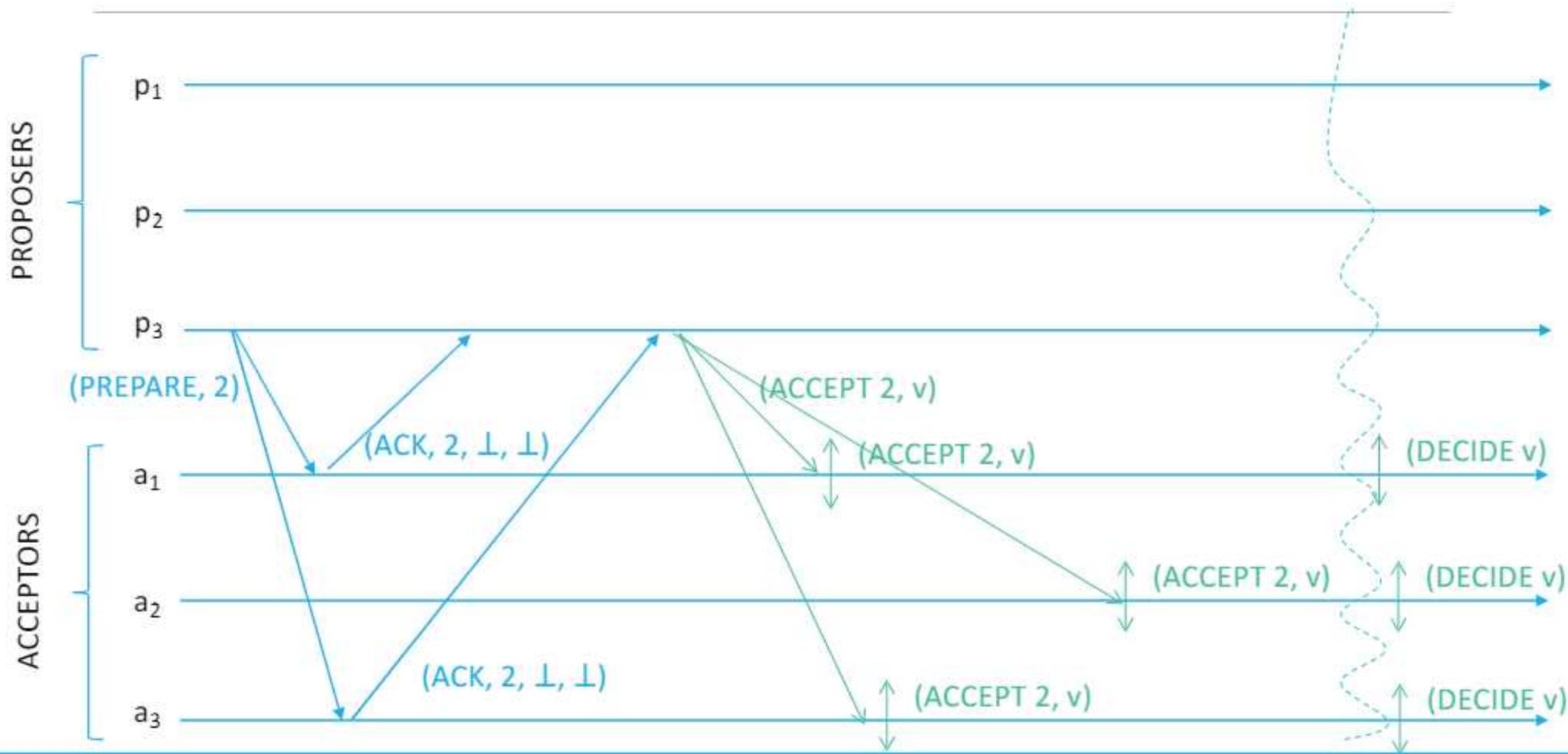
# Paxos

---

## Learning the decided value

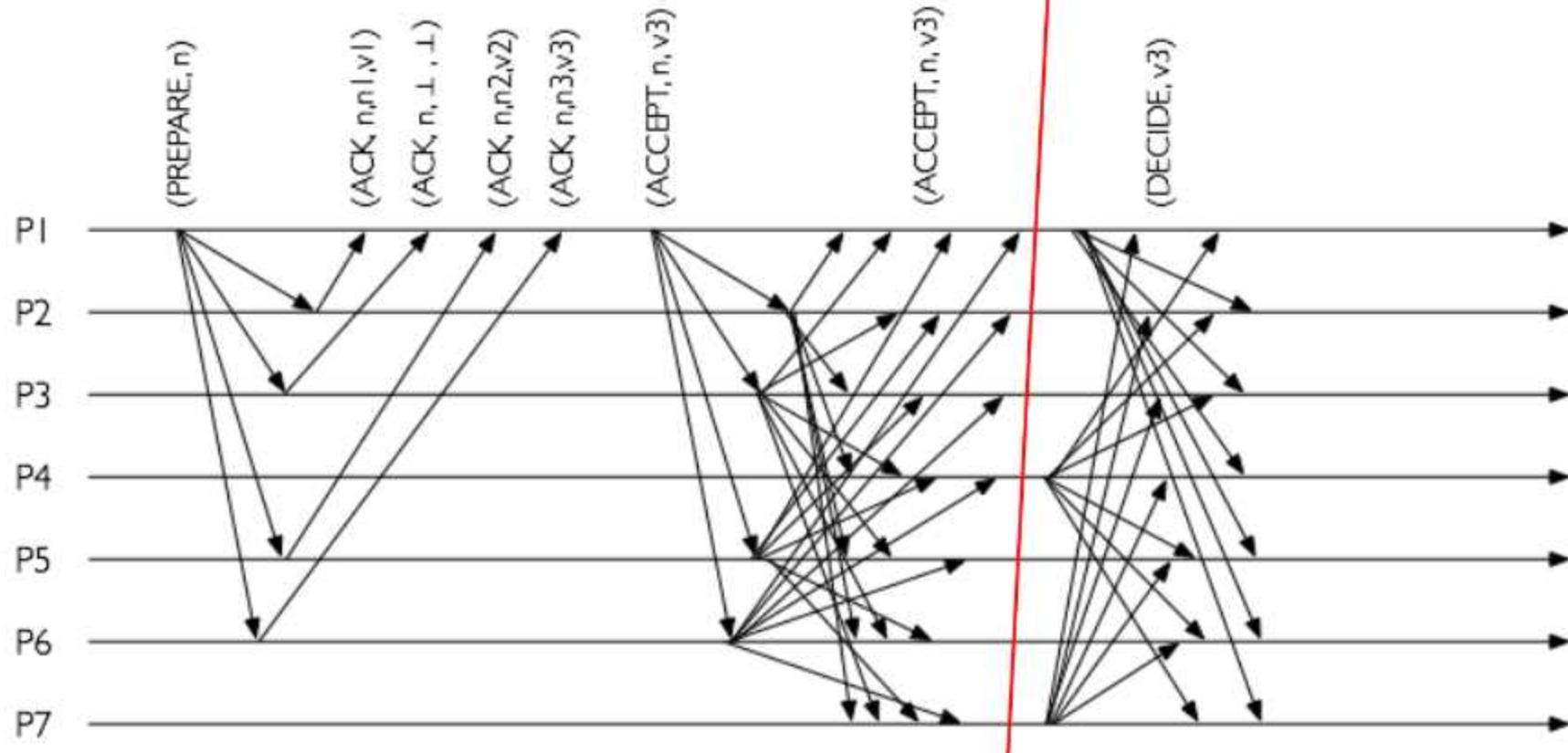
- Whenever acceptor accepts a proposal, respond to all learners (ACCEPT, n, v).
- Learner receives (ACCEPT, n, v) from a majority of acceptors, decides v, and sends (DECIDE, v) to all other learners.
- Learners receive (DECIDE, v), decide v

# Paxos: Phase 2

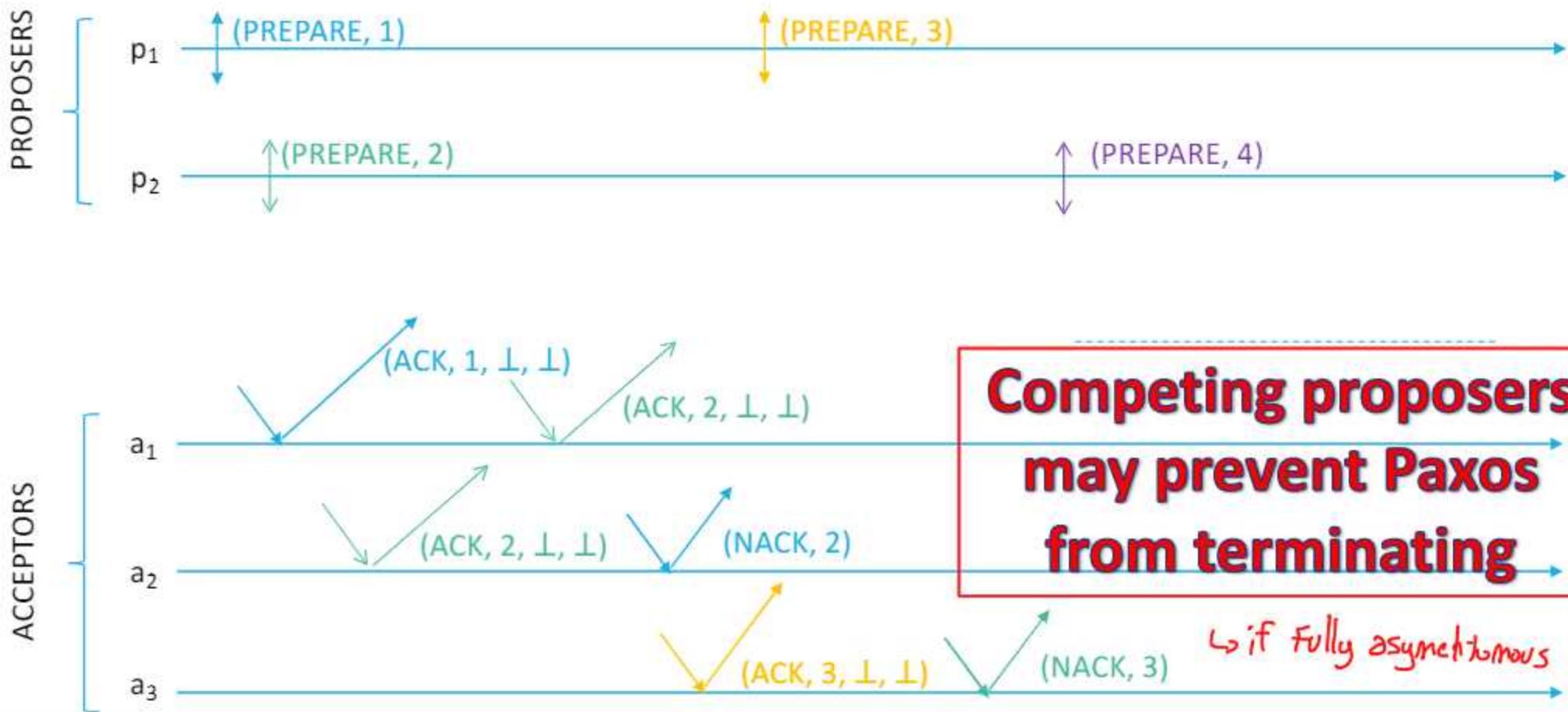


# Paxos

Run example:



# Paxos: liveness



# Paxos

---

Solution: Proposers are “elected” using a leader election protocol

- Protocol becomes a 2-phase commit with a 3- phase commit when leader fails
-

# Paxos

---

Is Paxos just a theoretical exercise ?

- Used but not widely. For example, Google uses Paxos in their lock server. Yahoo for ZooKeeper
- One issue is that Paxos gets complex if we need to reconfigure it to change the set of nodes running the protocol
- Another problem is that other more scalable alternatives are available

# References

---

- [1] L. Lamport “*Paxos Made Simple*”, <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/paxos-simple-Copy.pdf>

And for very brave students... ☺

- [2] L. Lamport. The part-time parliament. ACM Transactions on Computer Systems, 16(2):133–169, May 1998.

Dependable Distributed Systems  
Master of Science in Engineering in  
Computer Science

AA 2022/2023

---

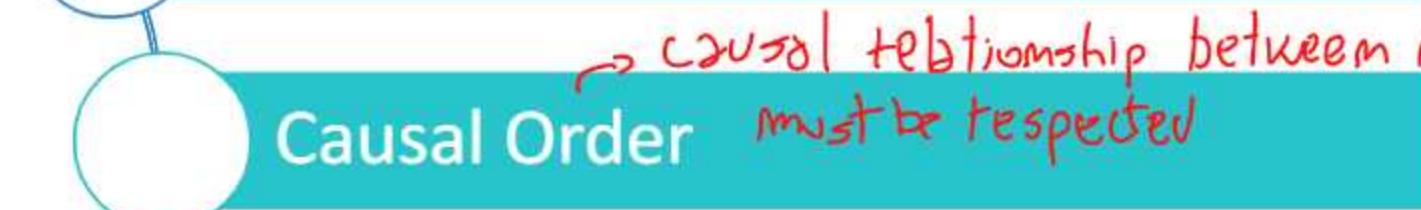
LECTURE 12: ORDERED COMMUNICATIONS

# Ordered Communication

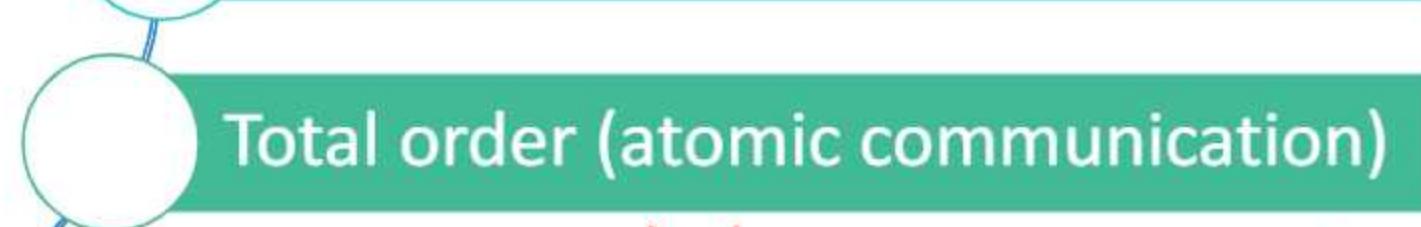
Define guarantees about the order of deliveries inside group of processes



if a source send m and after m', all the other processes will deliver the messages in the same order of source



causal relationship between messages must be respected



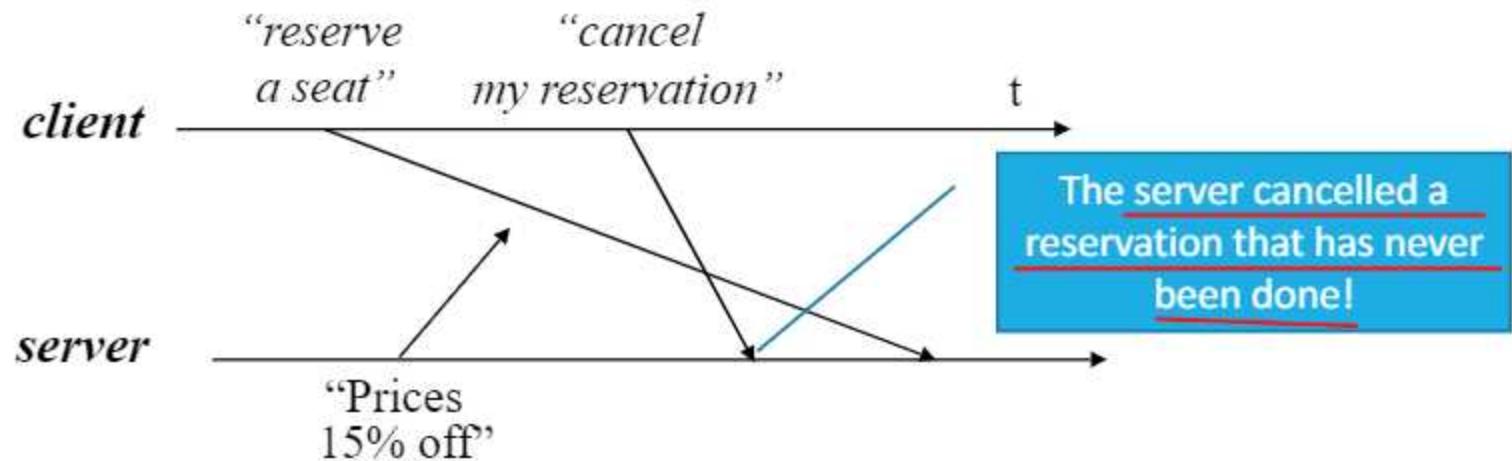
↳ always same order in all processes

# Advantages of ordered communication

Orthogonal wrt reliable communication.

- Reliable broadcast does not have any property on ordering deliveries of messages

This can cause anomalies in many applicative contexts



"Reliable ordered communication" are obtained adding one or more ordering properties to reliable communication

# FIFO Broadcast - Specification

FIFO Broadcast can be uniform/non uniform

---

**Module 3.8:** Interface and properties of FIFO-order (reliable) broadcast

---

**Module:**

**Name:** FIFOReliableBroadcast, **instance** *frb*.

**Events:**

**Request:**  $\langle frb, \text{Broadcast} \mid m \rangle$ : Broadcasts a message *m* to all processes.

**Indication:**  $\langle frb, \text{Deliver} \mid p, m \rangle$ : Delivers a message *m* broadcast by process *p*.

**Properties:**

**FRB1–FRB4:** Same as properties RB1–RB4 in (regular) reliable broadcast (Module 3.2).

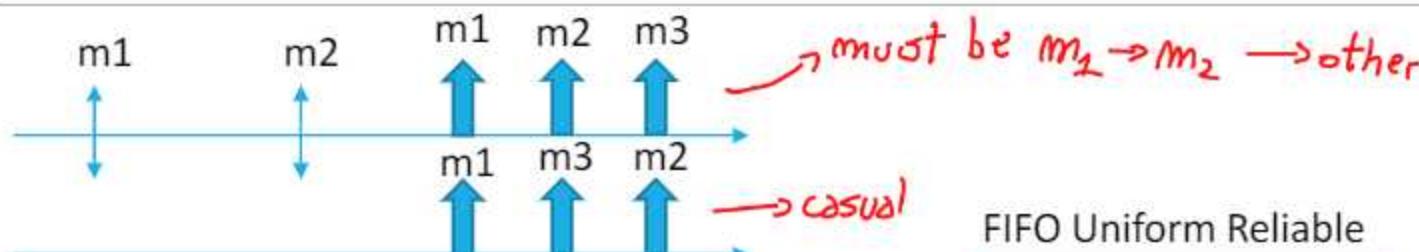
**FRB5: FIFO delivery:** If some process broadcasts message *m*<sub>1</sub> before it broadcasts message *m*<sub>2</sub>, then no correct process delivers *m*<sub>2</sub> unless it has already delivered *m*<sub>1</sub>.

Well correct processes respect order  
of that process

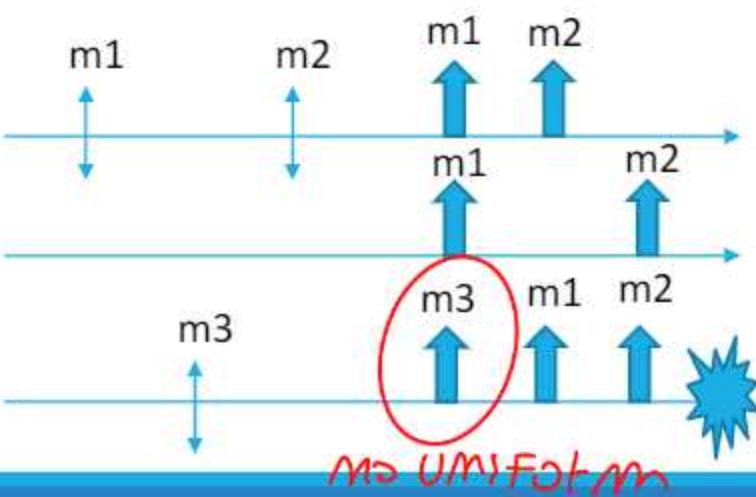
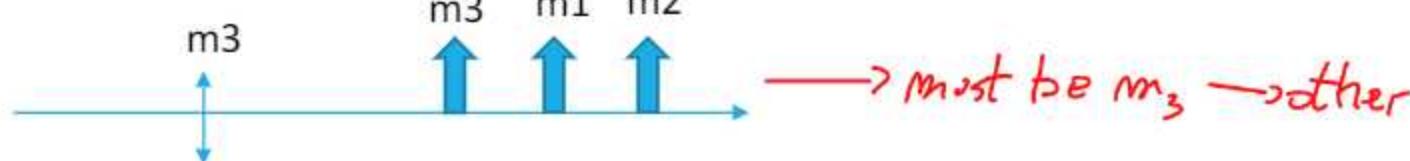


Same as Reliable Broadcast  
(URB for Uniform FIFO Broadcast)  
+  
FIFO Order

# Example



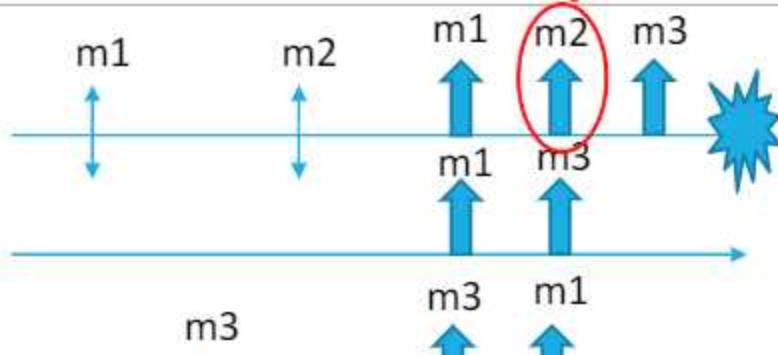
FIFO Uniform Reliable



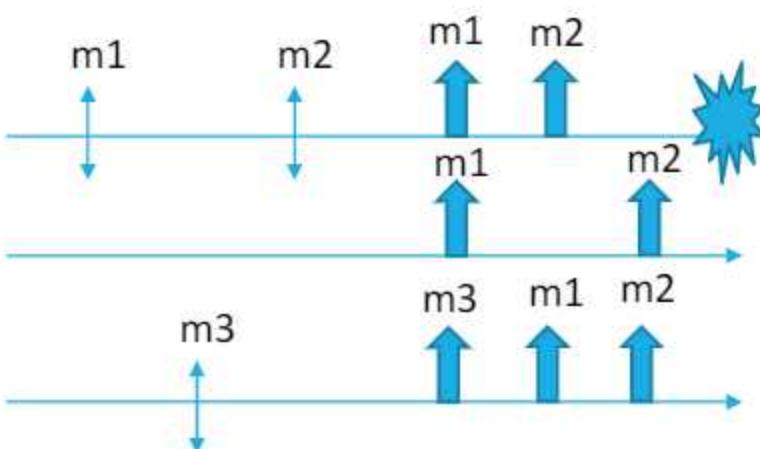
FIFO (Regular) Reliable

↓  
correct processes have  
same set of processes

# Example

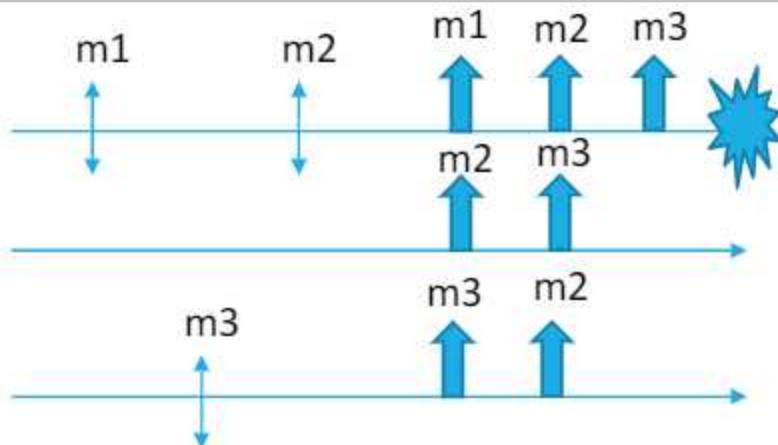


FIFO (Regular) Reliable

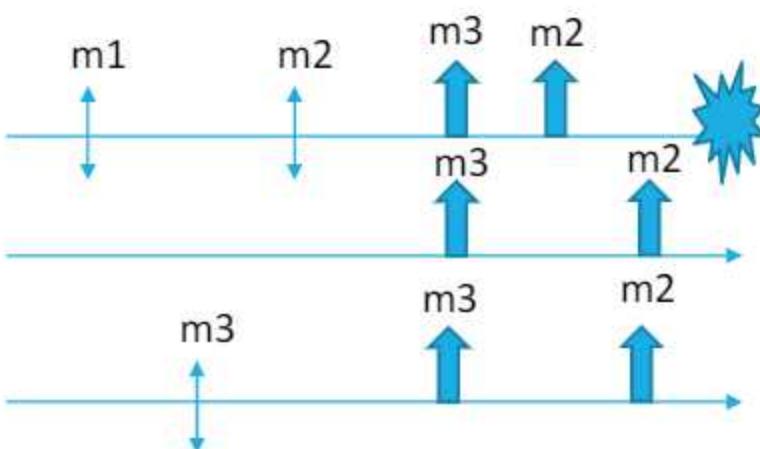


Not Reliable but it satisfies  
FIFO Order

# Example



Reliable Broadcast but not  
FIFO Broadcast



Uniform Reliable Broadcast  
but not FIFO Broadcast

# FIFO Broadcast - Implementation

---

## Algorithm 3.12: Broadcast with Sequence Number

---

### Implements:

FIFOReliableBroadcast, instance *frb*.

### Uses:

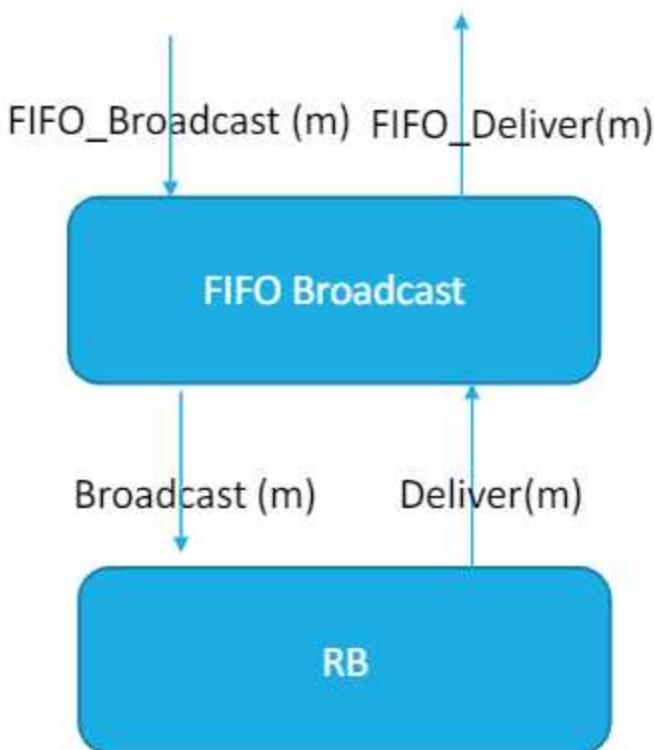
ReliableBroadcast, instance *rb*.

```
upon event (frb, Init ) do
  lsn := 0;
  pending := {};
  next := [1]N; ~ list with size the number of processes, initialize with all 1
  use for check if deliver the
  message of a process
  }

upon event (frb, Broadcast | m ) do
  lsn := lsn + 1;
  trigger (rb, Broadcast | [DATA, self, m, lsn] );
  sequence number
  }

upon event (rb, Deliver | p. [DATA, s, m, sn] ) do
  pending := pending ∪ {(s, m, sn)};
  while exists (s, m', sn') ∈ pending such that sn' = next[s] do
    next[s] := next[s] + 1;
    pending := pending \ {(s, m', sn')};
    trigger (frb, Deliver | s, m' );
```

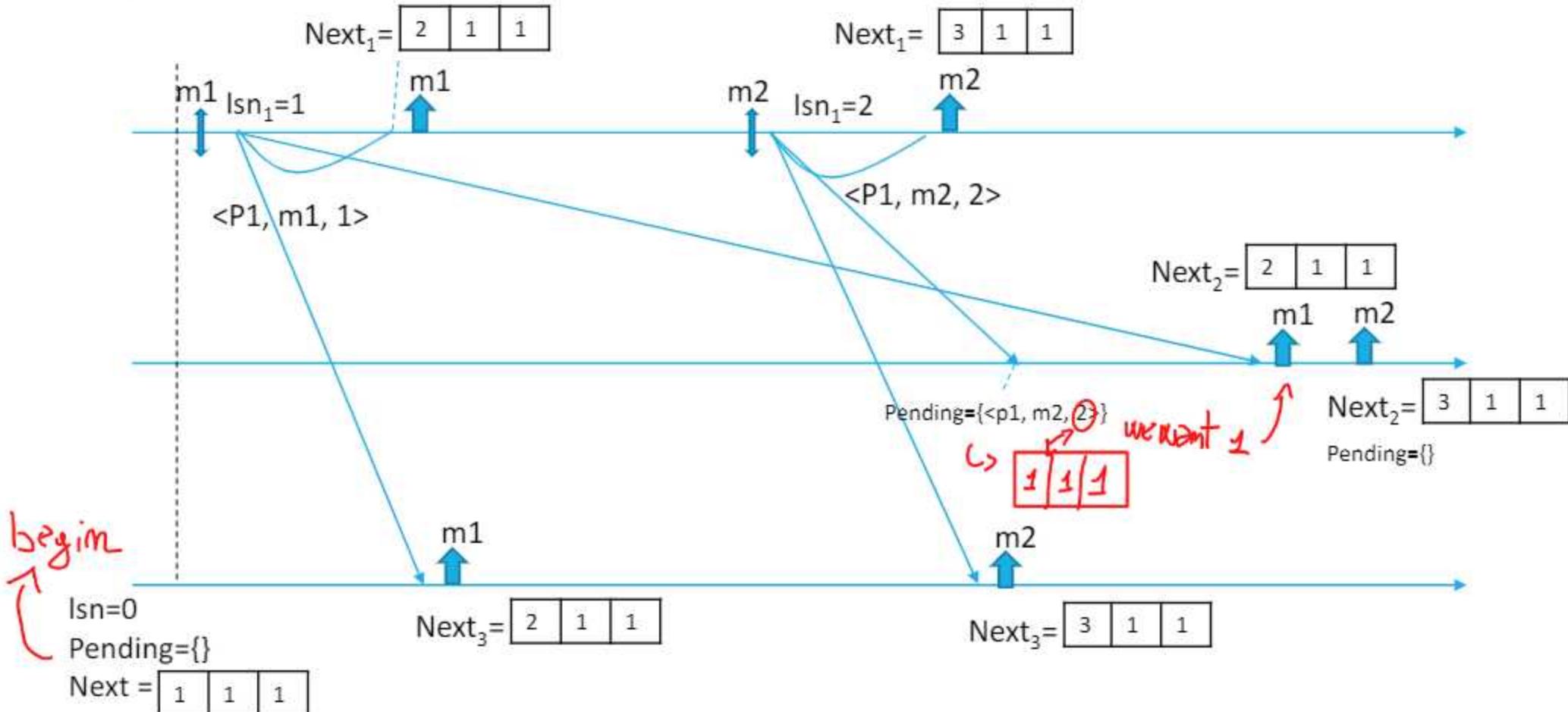
*that must be deliver from s*



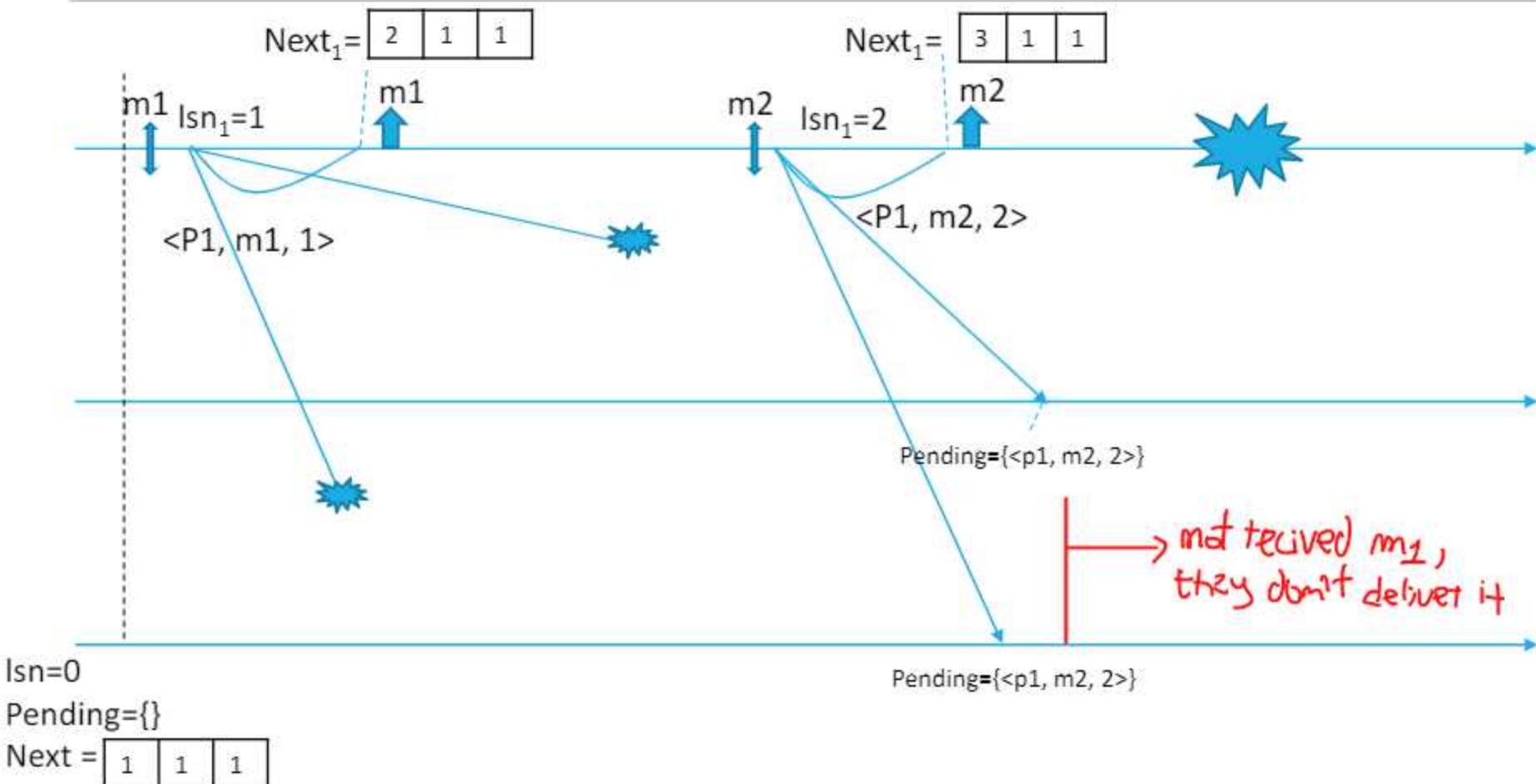
```

upon event ⟨ rb, Deliver | p, [DATA, s, m, sn] ⟩ do
  pending := pending ∪ {(s, m, sn)};
  while exists (s, m', sn') ∈ pending such that sn' = next[s] do
    next[s] := next[s] + 1;
    pending := pending \ {(s, m', sn')};
    trigger ⟨ frb, Deliver | s, m' ⟩;
  
```

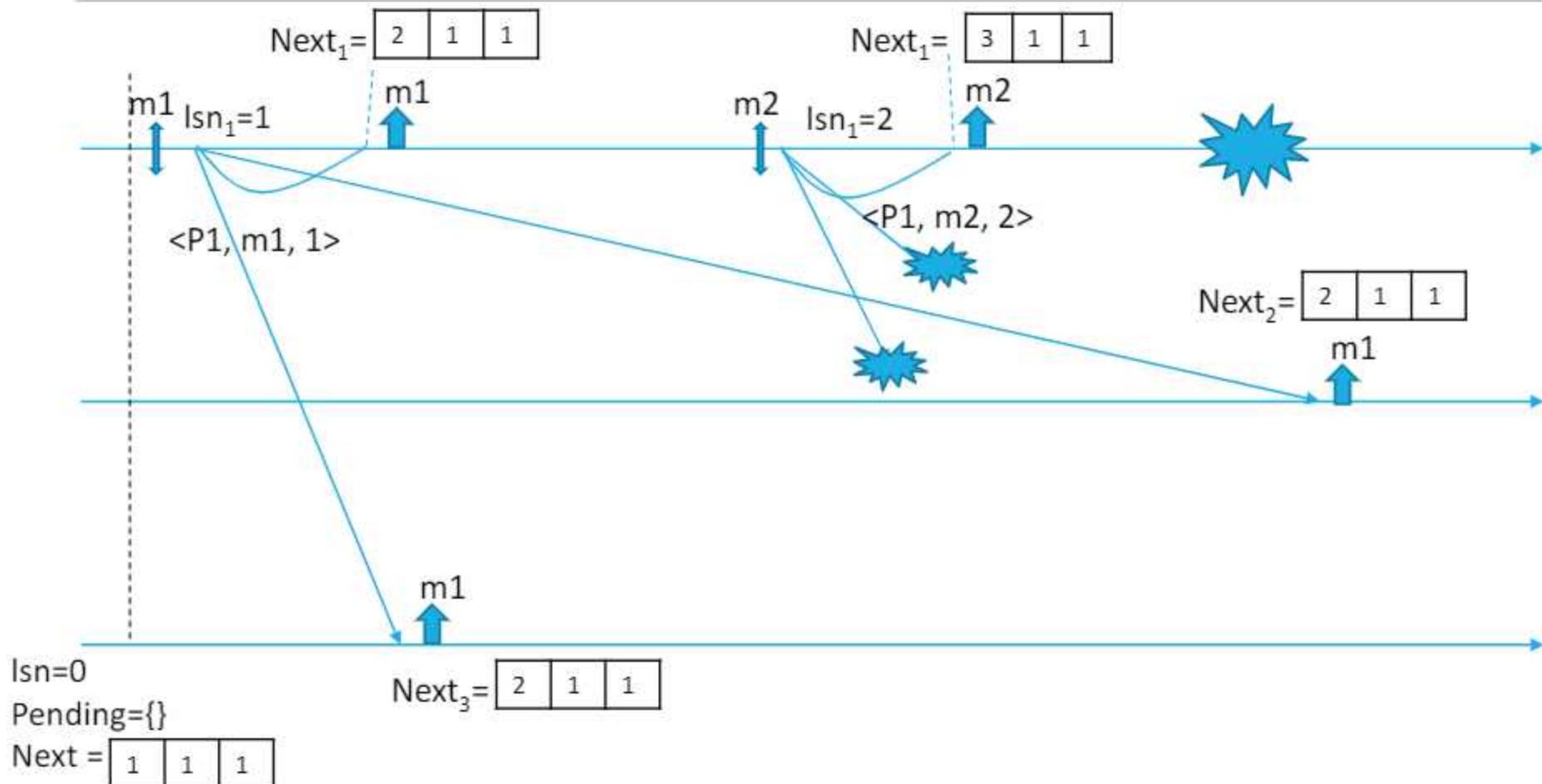
# Example



# Example



# Example



# Causal Order Broadcast

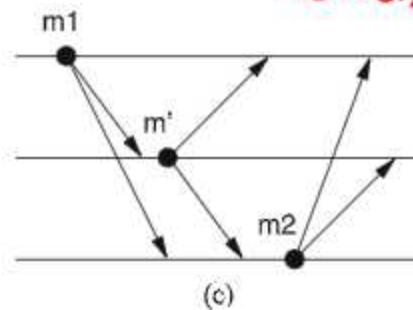
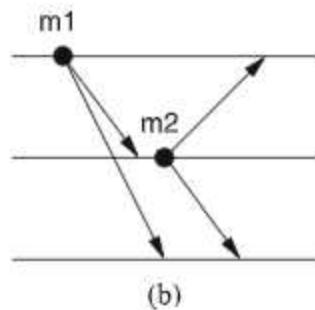
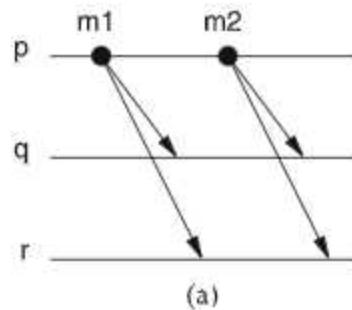
Ensures that messages are delivered according to the cause–effect relationships

- Causal order is an extension of the happened-before relation

A message  $m_1$  may have *potentially caused* another message  $m_2$  (denoted as  $m_1 \rightarrow m_2$ ) if any of the following holds:

- a) some process  $p$  broadcasts  $m_1$  before it broadcasts  $m_2$
- b) some process  $p$  delivers  $m_1$  and subsequently broadcasts  $m_2$
- c) there exists some message  $m'$  such that  $m_1 \rightarrow m'$  and  $m' \rightarrow m_2$

Same properties  
of happen-before-  
relationship



# Causal Order Broadcast Specification

Causal Order Broadcast can be uniform/non uniform

**Module 3.9:** Interface and properties of causal-order (reliable) broadcast

**Module:**

**Name:** CausalOrderReliableBroadcast, **instance** *crb*.

**Events:**

**Request:**  $\langle \text{crb}, \text{Broadcast} \mid m \rangle$ : Broadcasts a message *m* to all processes.

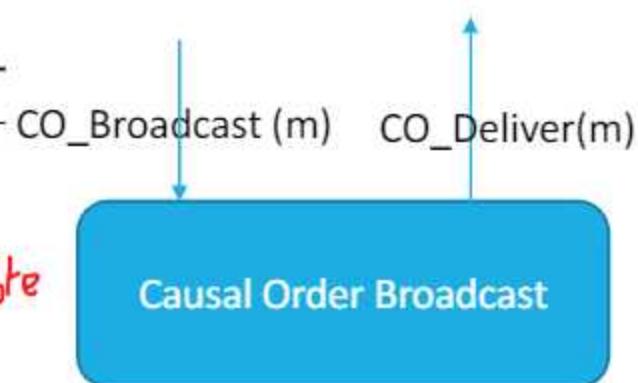
same as before

**Indication:**  $\langle \text{crb}, \text{Deliver} \mid p, m \rangle$ : Delivers a message *m* broadcast by process *p*.

**Properties:**

**CRB1–CRB4:** Same as properties RB1–RB4 in (regular) reliable broadcast (Module 3.2).

**CRB5: Causal delivery:** For any message  $m_1$  that potentially caused a message  $m_2$ , i.e.,  $m_1 \rightarrow m_2$ , no process delivers  $m_2$  unless it has already delivered  $m_1$ .



Same as Reliable Broadcast  
(URB for Uniform FIFO Broadcast)

⊕  
Causal Order

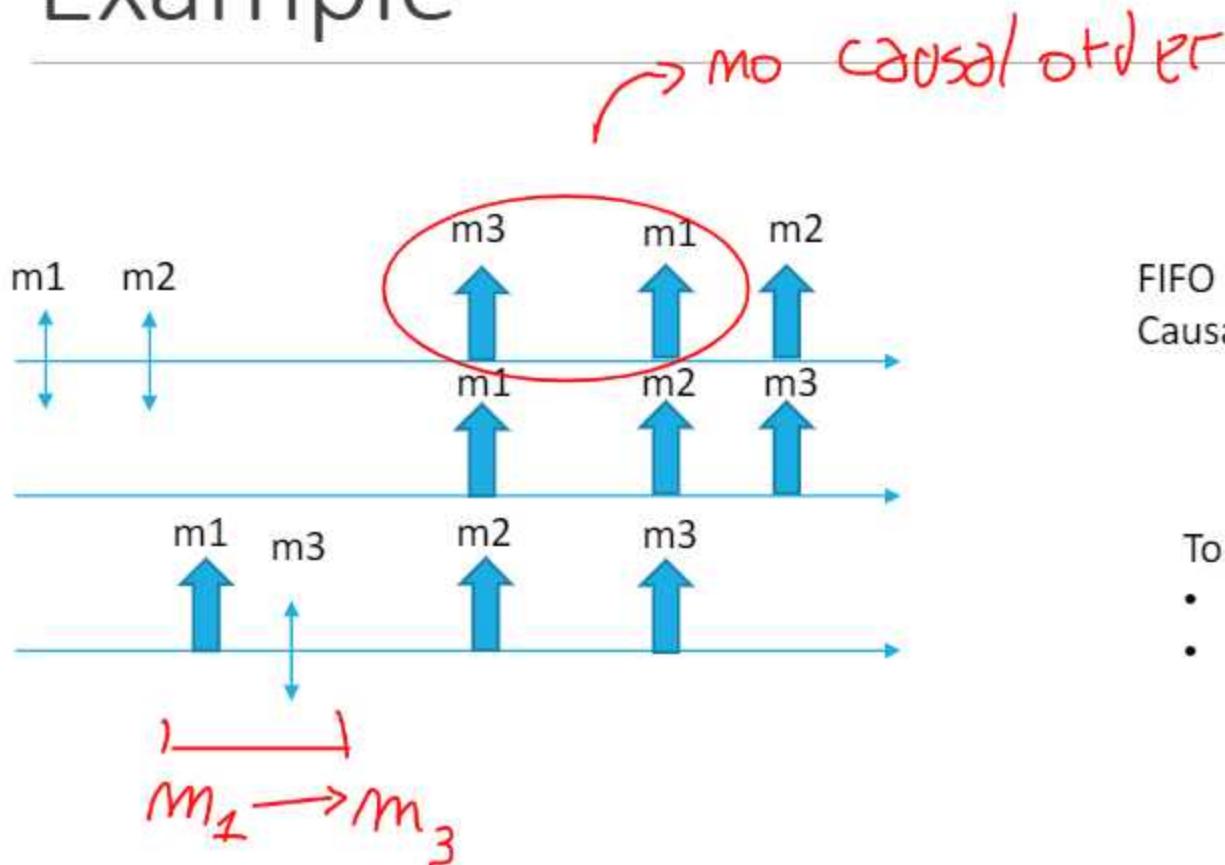
# Causal Order Broadcast

---

## *Observation*

- Causal Broadcast = Reliable Broadcast + Causal Order
  - Causal Order  $\Rightarrow$  FIFO Order
  - But FIFO Order  $\not\Rightarrow$  Causal Order
- Causal Order = FIFO Order + Local Order
  - Local Order: if a process delivers a message  $m$  before sending a message  $m'$ , then no correct process deliver  $m'$  if it has not already delivered  $m$

# Example



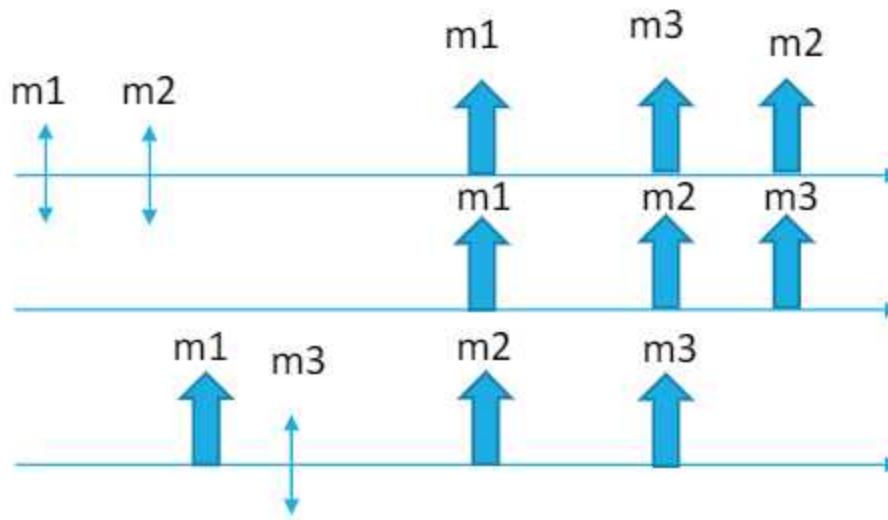
FIFO Reliable but Not Causal

To have causal we need

- $m_1 \rightarrow m_2$  (FIFO)
- $m_1 \rightarrow m_3$  (local Order)

# Example

$m_1 \rightarrow m_2$  and  $m_1 \rightarrow m_3$



Causal Reliable

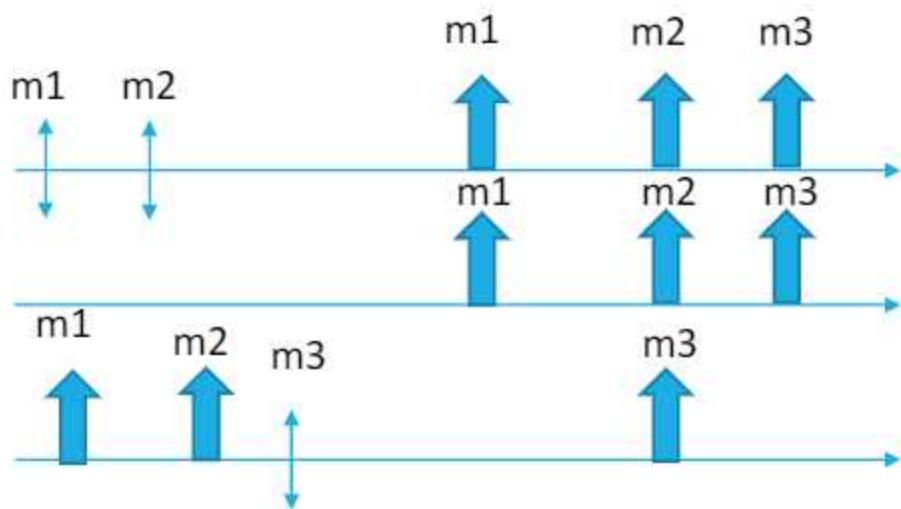
To have causal we need

- $m_1 \rightarrow m_2$  (FIFO)
- $m_1 \rightarrow m_3$  (local Order)

m1, m2, m3  
m1, m3, m2

acceptable  
execution

# Example



$m_1 \rightarrow m_2$  FIFO

$m_1 \rightarrow m_2 \rightarrow m_3$  local order

Causal Reliable

To have causal we need

- $m_1 \rightarrow m_2$  (FIFO)
- $m_2 \rightarrow m_3$  (local Order)



$m_1, m_2, m_3$

only possibilities

# Causal Order Broadcast Implementation

## Algorithm 3.15: Waiting Causal Broadcast

### Implements:

CausalOrderReliableBroadcast, **instance** *crb*.

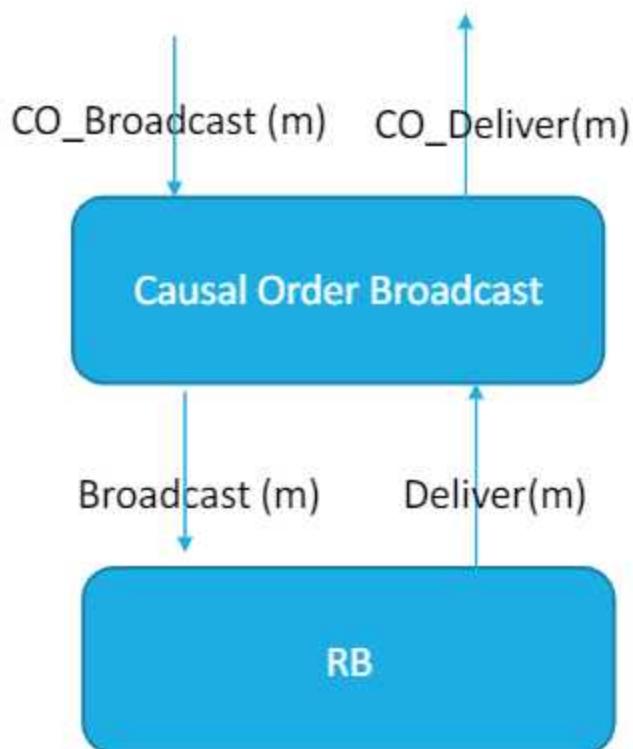
### Uses:

ReliableBroadcast, **instance** *rb*.

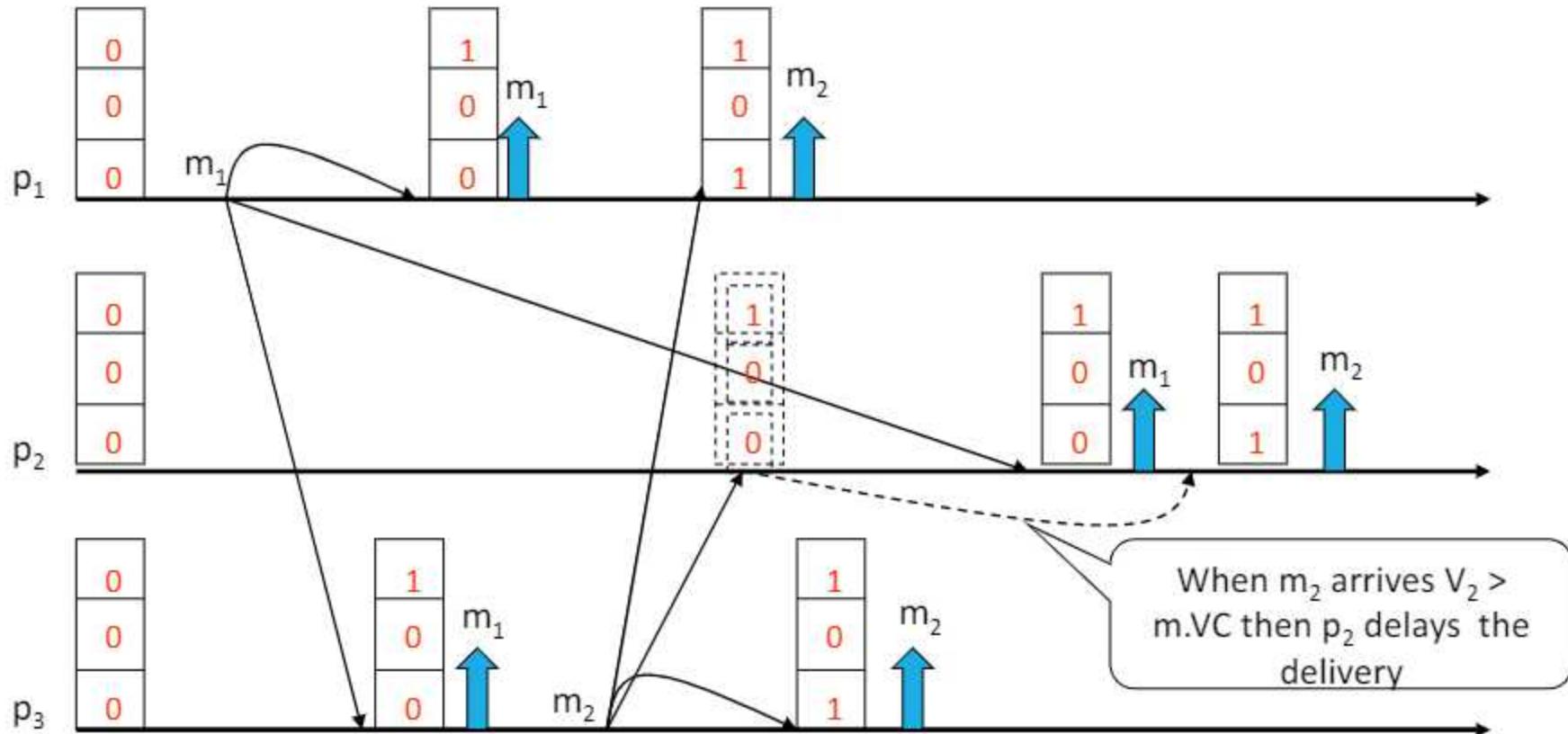
```
upon event ( crb, Init ) do
    V :=  $[0]^N$ ; use a vector clock
    lsn := 0; local sequence number
    pending :=  $\emptyset$ ;

upon event ( crb, Broadcast | m ) do
    W := V;
    W[rank(self)] := lsn;
    lsn := lsn + 1;
    trigger ( rb, Broadcast | [DATA, W, m] );

upon event ( rb, Deliver | p, [DATA, W, m] ) do
    pending := pending  $\cup$  {(p, W, m)};
    while exists ( p', W', m' )  $\in$  pending such that W'  $\leq$  V do
        pending := pending  $\setminus$  {(p', W', m')};
        V[rank(p')] := V[rank(p')] + 1;
        trigger ( crb, Deliver | p', m' );
        comparate vector local clock
        for causal relationship
```



# Waiting Causal Broadcast example



# Causal Order Broadcast: Safety

---

## Property:

- Let two broadcast messages  $m$  and  $m'$  such that  $\text{broadcast}(m) \rightarrow \text{broadcast}(m')$  then each process have to deliver  $m$  before  $m'$

## Observation:

- if  $m$  is the  $k$ -th message sent by  $p_i$  then  $m.Vc[i] = k-1$

Safety property can be proved by induction using the causal ordering relation among broadcast messages

## Definition:

- Let two broadcast events  $b$  and  $b'$  with  $b \rightarrow b'$ . These events have a **causal distance**  $k$  if  $\exists$  a sequence of  $k$  broadcast events  $b_1 \dots b_k$  such that
  - $\forall i \in \{1 \dots k\} b_i \rightarrow b_{i+1} \wedge (\neg \exists m^* | b_i \rightarrow m^* \rightarrow b_{i+1})$
  - $b \rightarrow b_1$
  - $b_k \rightarrow b'$

## Proof – basic case (K=0)

---

Given two messages  $m, m'$  such that

1.  $\text{broadcast}(m) \rightarrow \text{broadcast}(m')$
2. There does not exist  $\text{broadcast}(m'')$  such that  
 $\text{broadcast}(m) \rightarrow \text{broadcast}(m'') \wedge \text{broadcast}(m'') \rightarrow \text{broadcast}(m')$ .

We can have two distinct cases

1.  $m$  and  $m'$  have been issued by the same process
2.  $m$  and  $m'$  have been issued by distinct processes

# Case 1 – broadcast produced by the same process

---

1.  $p_j$  is the receiver
2. For line 3 in broadcast procedure
  1.  $m'.VC[i] := m.VC[i] + 1$ .  
if  $m$  is the  $h$ -th message sent by  $p_i$ ,  $m.VC[i] = h-1$  and  $m'.VC[i] = h$ .
3. A process  $p_j$  that receives  $m'$  verifies the following delivery condition:
  1.  $\forall x \in \{1, \dots, n\} m'.VC[x] \leq V_j[x]$  and  $m'.VC[i] \leq V_j[i]$
4.  $V_i[x]$  is equals to  $h$  if and only if the  $h$ -th message sent by  $p_x$  was delivered by  $p_i$ .  
(line 3 receive thread).
5. Consequently from 2,3,4,  $m'$  can be delivered only after the deliver of  $m$ .

# Case 1 – broadcast produced by distinct processes

---

$m$  and  $m'$  was been sent by distinct processes, respectively  $p_i$  e  $p_j$ .  $P_k$  is the receiver.

$\text{broadcast}(m) \rightarrow \text{broadcast}(m')$ ,  $m'$  was broadcasted by  $p_j$  after the deliver of  $m$ .

Without loss of generality  $m.\text{VC}[i] = h-1$

- For line 3 of reception thread e for assumption of  $k=0$  we have  $m'.\text{VC}[i] = h$ .

The receiver process  $p_k$  respects the following delivery condition:

- $\forall x \in \{1, \dots, n\} m'.\text{VC}[x] \leq V_k[x]$  and  $m'.\text{VC}[i] \leq V_k[i]$

To deliver the message,  $m'.\text{VC}[i] \leq V_k[i]$ , that is  $V_k[i] \geq h$

$V_k[i]$  is equals to  $h$  if and only if the  $h$ -th message sent by  $p_i$  has been delivered by  $p_k$ .  
(line 3 of reception thread thread).

For 2,3,4,  $p_k$  can deliver  $m'$  only after the deliver of  $m$

## Proof – Inductive step( $k > 0$ )

---

$\exists$  a sequence of  $k$  broadcast events  $b_1, b_2 \dots b_k$  such that

$$b \rightarrow b_1 \rightarrow b_2 \rightarrow \dots \rightarrow b_k \rightarrow b'$$

Inductive hypothesis :  $m$  has been delivered before  $m_k$

We have to prove that  $m_k$  has been delivered before  $m'$ .

- It follows from the basic case.

$m$  has been delivered before  $m'$ .

# Causal Order Broadcast: Liveness

---

## Property:

- Eventually each message will be delivered

Liveness is guaranteed by the following assumptions:

- The number of broadcast events that precedes a certain event is finite
- Channels are reliable

# Causal Order Broadcast Implementation

## Algorithm 3.13: No-Waiting Causal Broadcast

Implements:

CausalOrderReliableBroadcast, instance *crb*.

Uses:

ReliableBroadcast, instance *rb*.

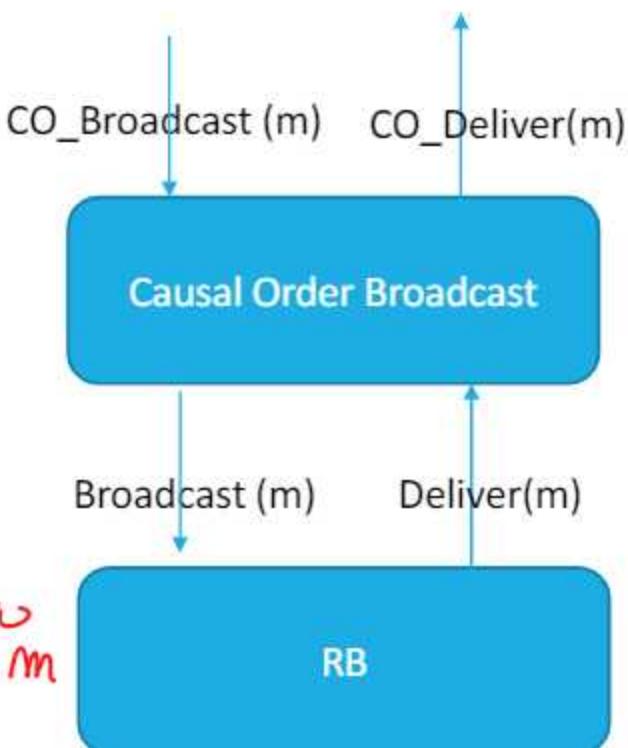
```
upon event { crb, Init } do
  delivered := {};
  past := [];

upon event { crb, Broadcast | m } do
  trigger { rb, Broadcast | [DATA, past, m] };
  append(past, (self, m));

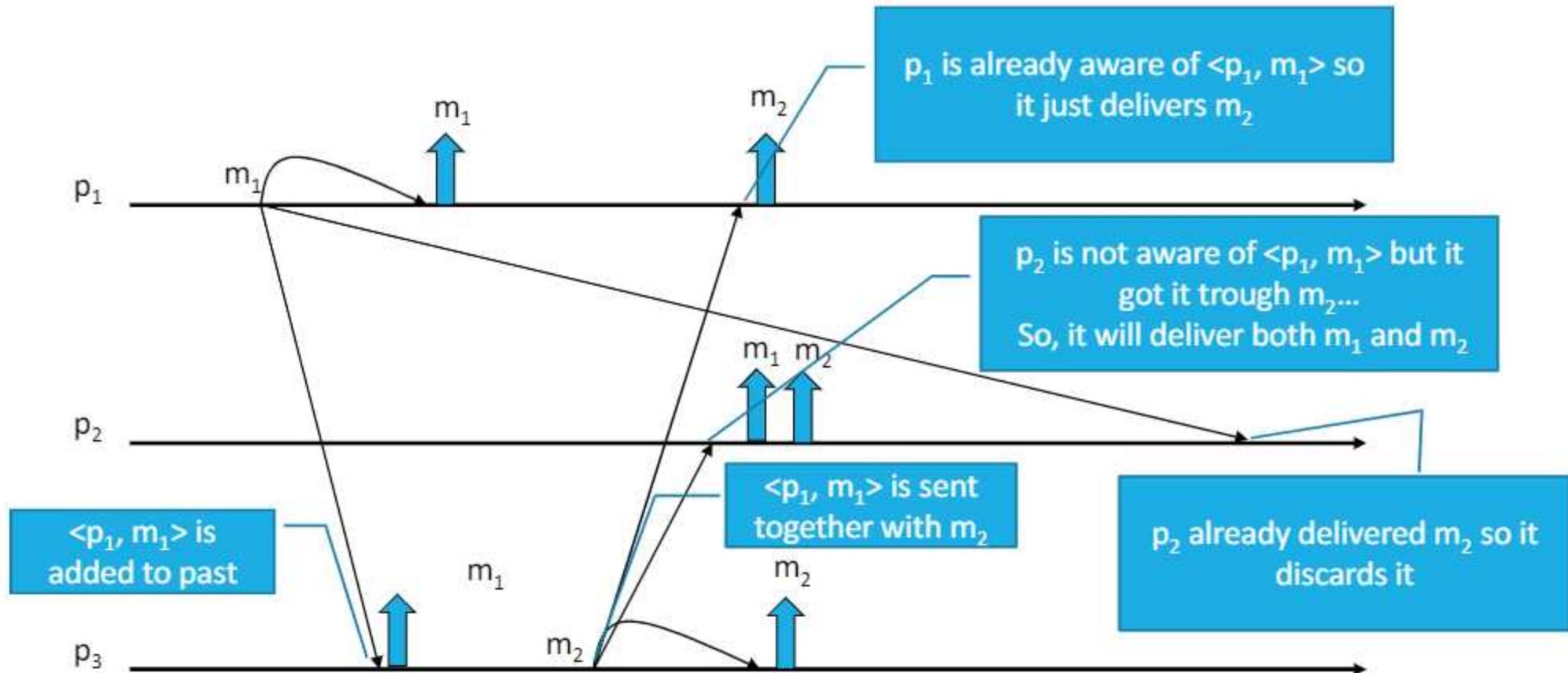
upon event { rb, Deliver | p, [DATA, mpast, m] } do
  if m ∉ delivered then
    forall (s, n) ∈ mpast do
      if n ∉ delivered then
        trigger { crb. Deliver | s, n };
        delivered := delivered ∪ {n};
        if (s, n) ∉ past then
          append(past, (s, n));
  trigger { crb, Deliver | p, m };
  delivered := delivered ∪ {m};
  if (p, m) ∉ past then
    append(past, (p, m));
```

append(L, x) adds an element x at the end of list L

*(implicit assumption) No two processes broadcast same m by the order in the list*

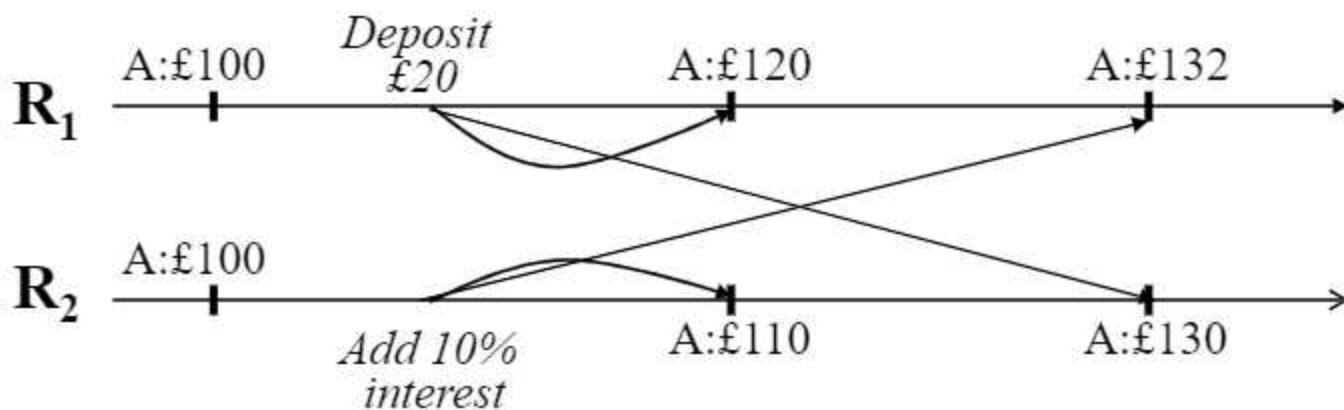


# Non-Waiting Causal Broadcast example



## Advantages of Ordered Communication

- Causal Order is not enough strong to avoid anomalies
- E.g. Bank account replicated on two sites



- same initial state, different final state at the two sites
- To have the same final state we need to ensure that the order of deliveries is the same at each process.
- Note that ensuring the same delivery order at each replicas does not consider the sending order of messages

# Total Order Broadcast

A total-order (reliable) broadcast abstraction orders all messages, even those from different senders and those that are not causally related

Reliable Broadcast + Total Order

processes agree on the same set of messages they deliver

Processes agree on the same sequence of messages

The total-order broadcast abstraction is sometimes also called atomic broadcast

*↳ a single evolution in the system*

message delivery occurs as if the broadcast were an indivisible "atomic" action

the message is delivered to all or to none of the processes and, if the message is delivered, every other message is ordered either before or after this message.

# Total Order Broadcast

none of them guarantee the others

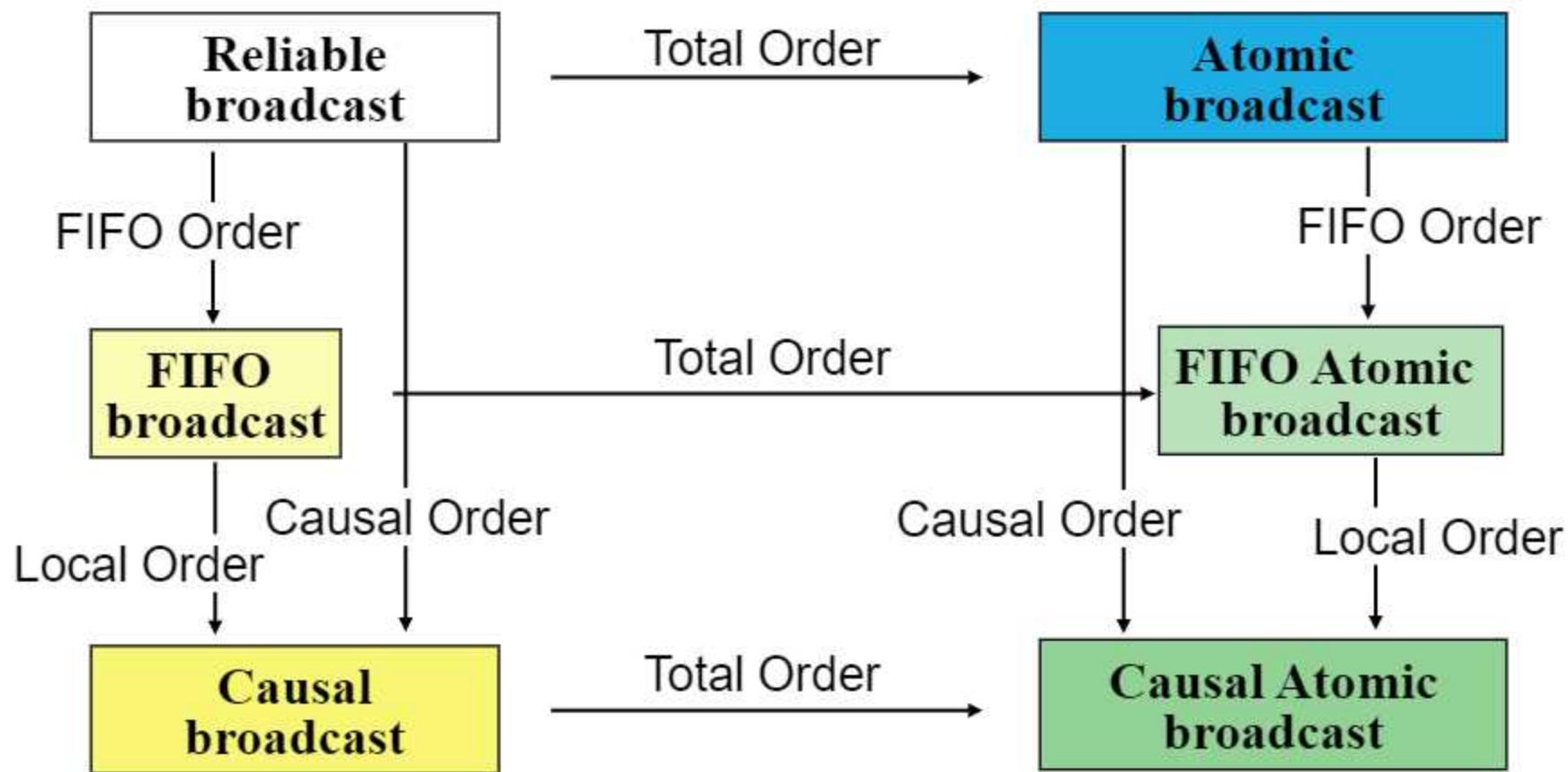


**Total order is orthogonal with respect to FIFO and Causal Order.**

Total order would accept indeed a computation in which a process  $p_i$  sends  $n$  messages to a group, and each of the processes of the group delivers such messages in the reverse order of their sending.

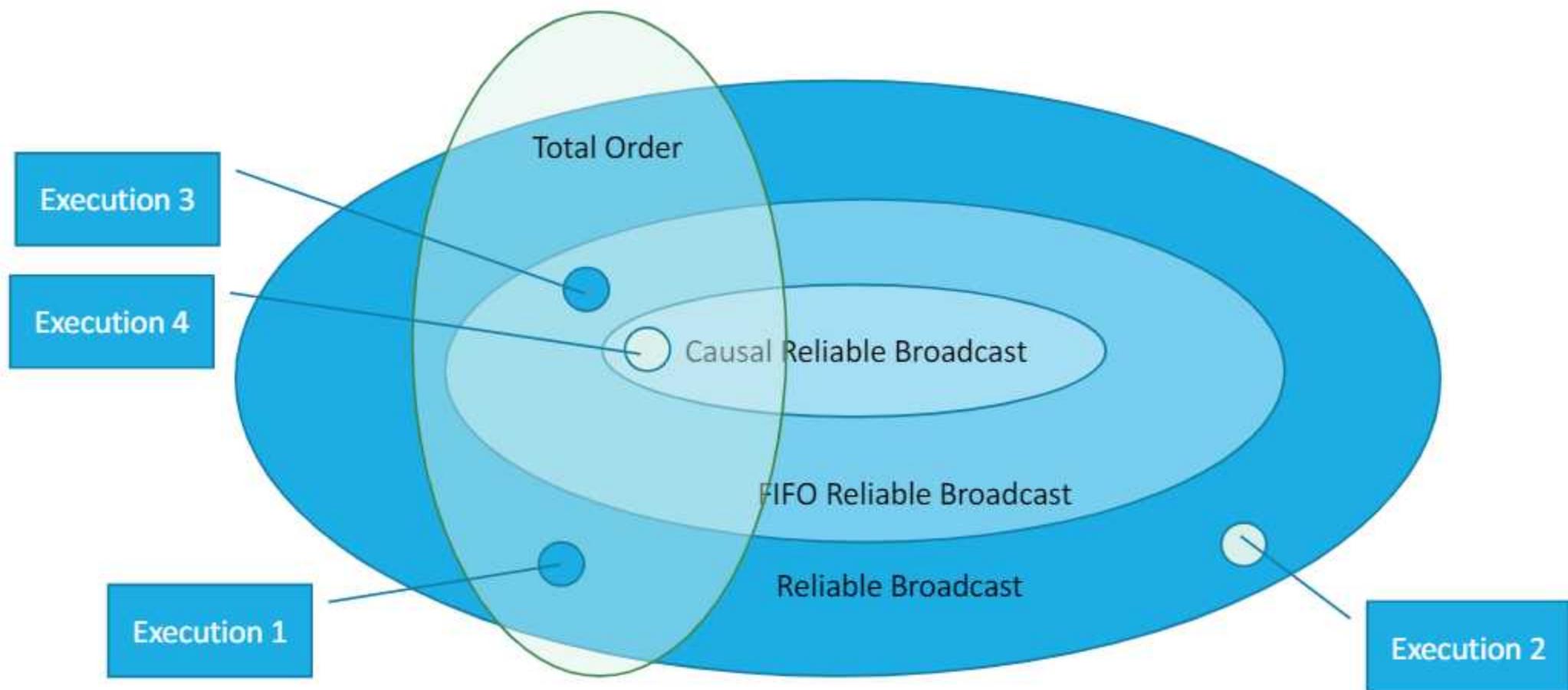
The computation is totally ordered but it is not FIFO.

## Relationship between Broadcast Specifications



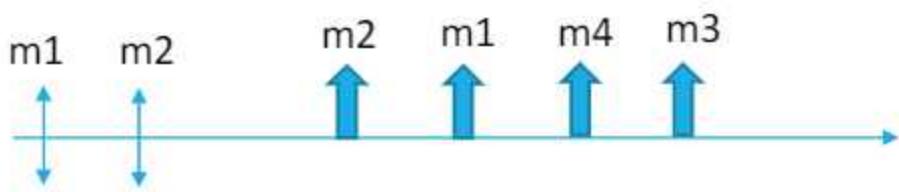
# Let's Identify

---

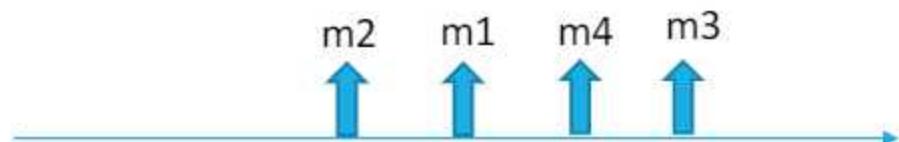


# Execution 1

*Regular, Total but not FIFO*

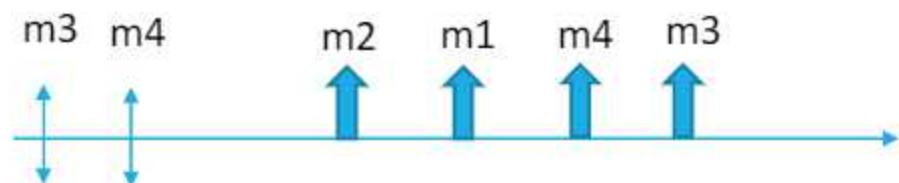


Total but not FIFO



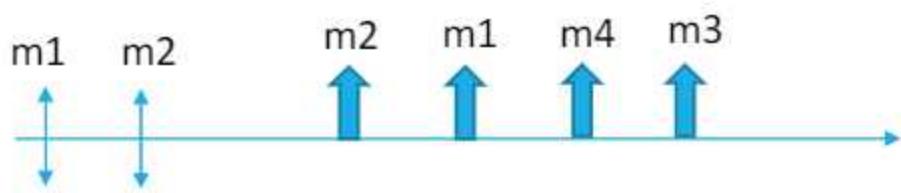
FIFO Order

- $m_1 \rightarrow m_2$
- $m_3 \rightarrow m_4$

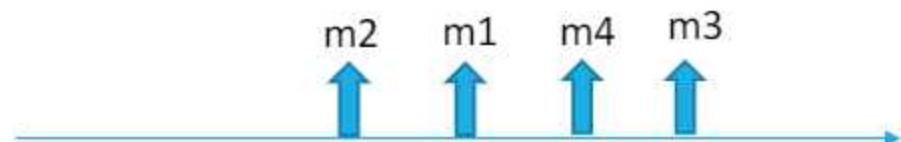


# Execution 2

---

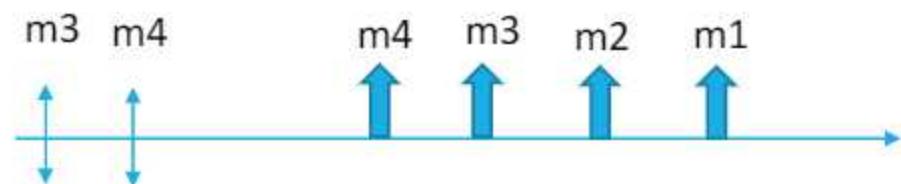


Reliable not total not FIFO

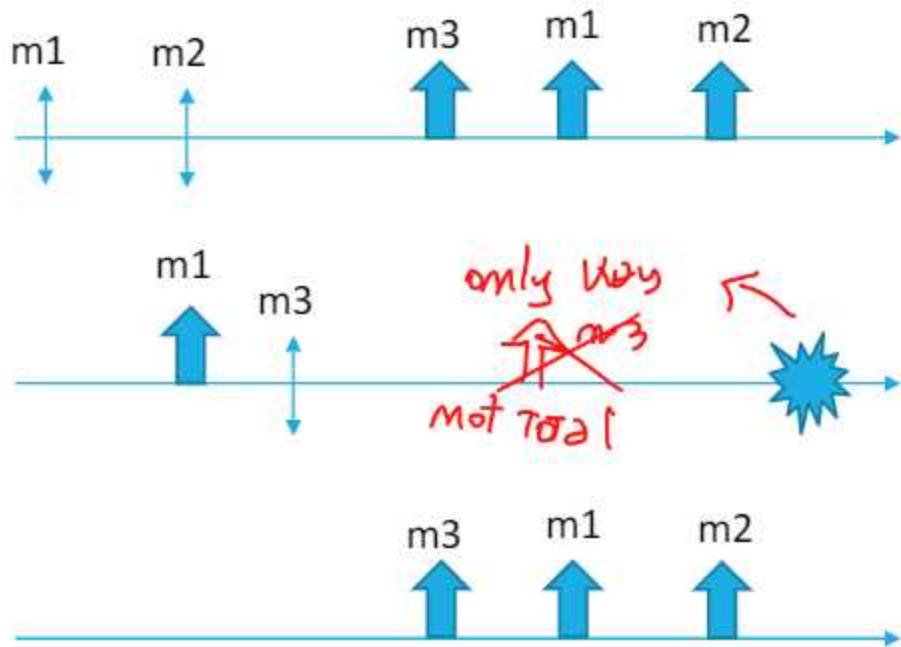


FIFO Order

- m1 -> m2
- m3 -> m4



# Execution 3

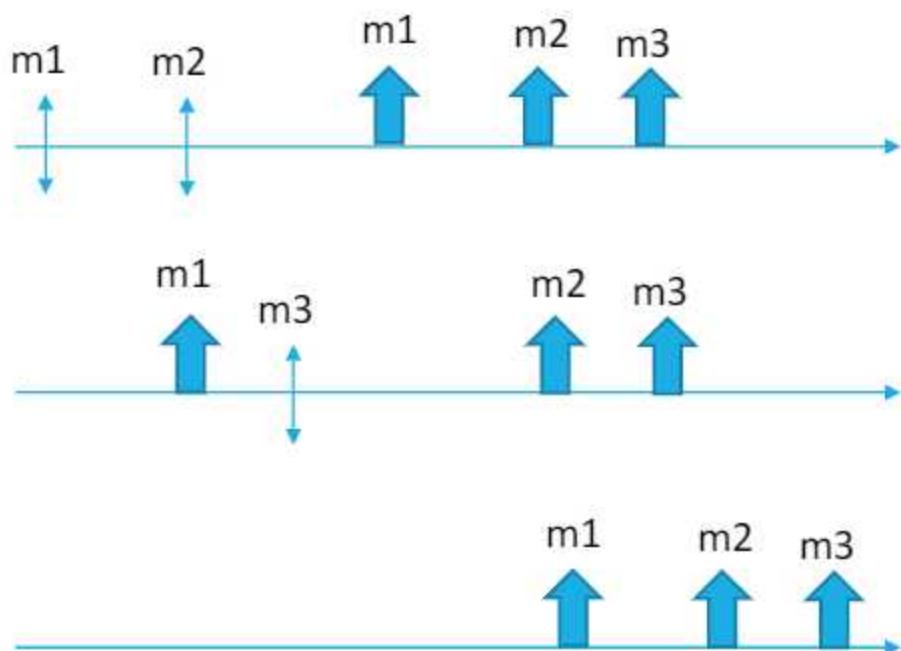


Total, FIFO, Not Causal

Ordering Relationships

- $m1 \rightarrow m2$  (FIFO order)
- $m1 \rightarrow m3$  (local order)

# Execution 4



Total, FIFO, Causal

Ordering Relationships

- $m1 \rightarrow m2$  (FIFO order)
- $m1 \rightarrow m3$  (local order)



- $m1, m2, m3$
- $m1, m3, m2$

# References

---

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

- Chapter 3 - from Section 3.9 (except 3.9.6)

# Dependable Distributed Systems

## Master of Science in Engineering in Computer Science

AA 2022/2023

---

LECTURE 13: TOTAL ORDER BROADCAST

↳ add properties on order of broadcast

# System model

we want sequences the same for all correct process

Static set of processes  $\Pi = \{p_1 \dots p_n\}$

Message passing over perfect channels

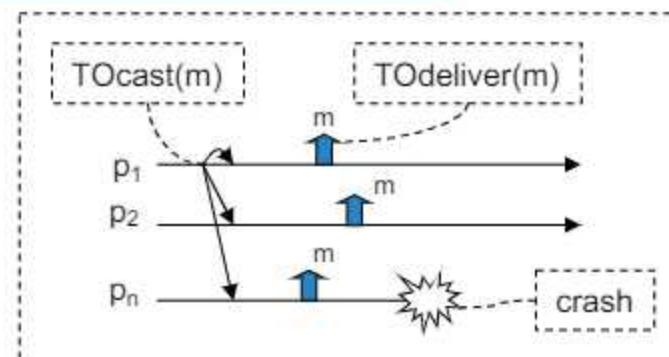
- i.e., message exchanged between correct processes are reliably delivered

Asynchronous, no use timer or synchrony bound, no assumption on time

Crash fault model for processes, crash and not going to recover

We characterize the system in terms of its possible runs  $R$

(> sequence of events that are generated inside the protocol that specify the behaviour of the total system broadcast



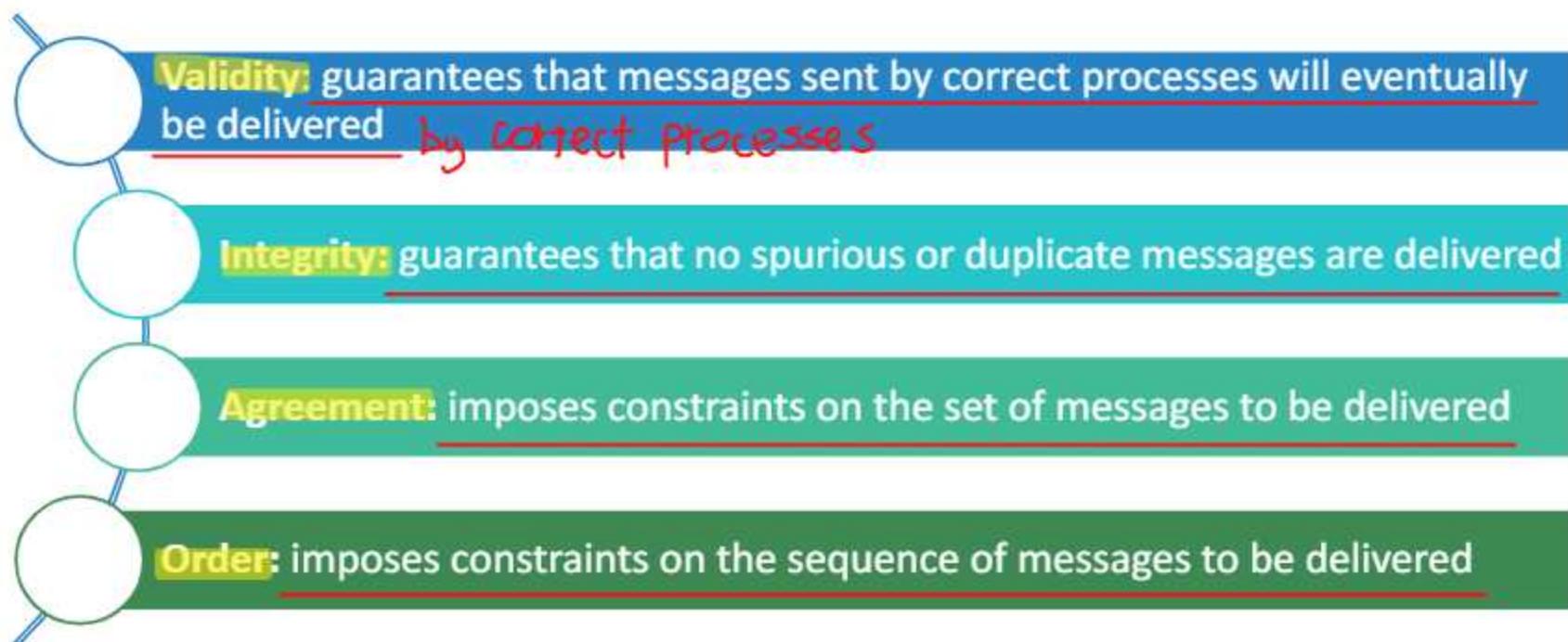
a possible run

Set of all possible combination of deliveries and broadcast messages generated



# TO specifications

Total order specifications are usually composed by four properties

- 
- Validity:** guarantees that messages sent by correct processes will eventually be delivered *by correct processes*
  - Integrity:** guarantees that no spurious or duplicate messages are delivered
  - Agreement:** imposes constraints on the set of messages to be delivered
  - Order:** imposes constraints on the sequence of messages to be delivered

# TO specifications

---

**Total Order Broadcast =  $TO(V, I, A, O)$**

- $V$  = Validity
- $I$  = Integrity *no duplication and no creation*
- $A$  = Agreement
- $O$  = Order

Distinct specifications arise from distinct formulations of each property

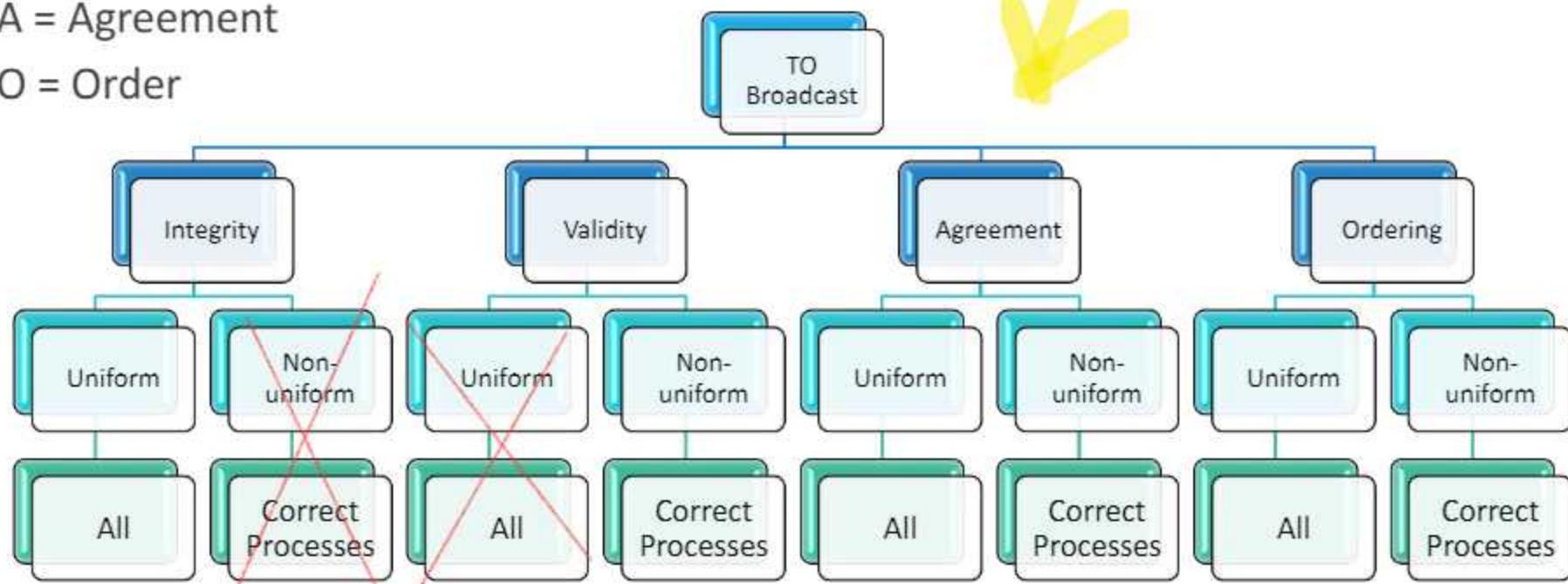
- uniform vs non-uniform
- A uniform property imposes restrictions on the behavior of (at least) correct processes  
*↳ impose constraint also on faulty processes*

# TO specifications

*taxonomy of  
broadcast primitives*

$$\text{Total Order Broadcast} = \text{TO}(V, I, A, O)$$

- V = Validity
- I = Integrity
- A = Agreement
- O = Order



# TO Specifications *observation*

Crash failure + Perfect channels  $\Rightarrow$

only crash, not recovery

PP<sub>2</sub>PlmK

Broadcast

↙

Non Uniform  
Validity

- NUV: if a correct process TOCAST a message m then some correct process will eventually deliver m

$\hookrightarrow$  PP<sub>2</sub>PlmK could lost messages if sender fails  $\hookrightarrow$  faulty can't assume will do it

↙

Uniform  
integrity

- UI: For any message m, every process p delivers m at most once and only if m was previously TOCAST by some (correct or not) process

Broadcast

use of PP<sub>2</sub>PlmK

# TO specifications

Total Order Broadcast =  $TO(V, I, A, O)$

$V = V$ alue  
 $I =$ Integrity

**NUV**  
**UI**

- ~~A~~ = Agreement
- ~~O~~ = Order

→  $TO(A, O)$

→ not fixed  $A$  and  $O$ , can be uniform  
or not uniform

Distinct specifications arise from distinct formulations of each property

- uniform vs non-uniform
- A uniform property imposes restrictions on the behavior of (at least) correct processes

# The Agreement property

## UNIFORM AGREEMENT (UA)

If a process (correct or not) TODelivers a message  $m$ , then all correct processes will eventually TODeliver  $m$ , if; deliver something i constraint other to deliver the same message, also faulty deliver

## NON-UNIFORM AGREEMENT (NUA)

If a correct process TODelivers a message  $m$ , then all correct processes will eventually TODeliver  $m$

only correct processes

## CONSTRAINS THE SET OF DELIVERED MESSAGES

Correct processes always deliver the same set of messages  $M$

Each faulty process  $p$  delivers a set  $M_p$

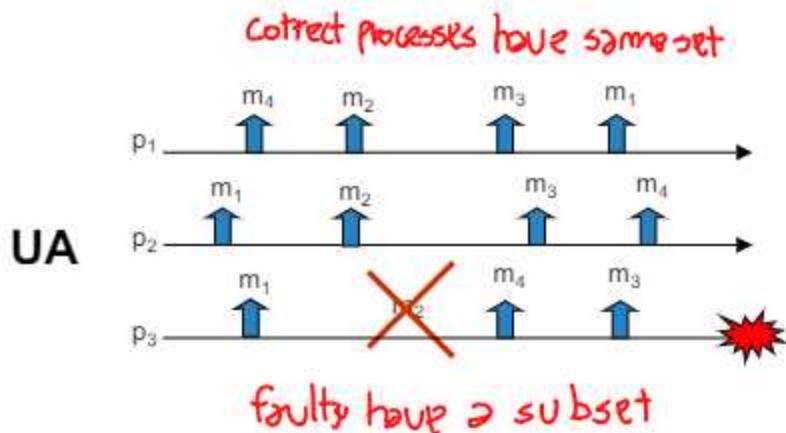
UA:  $M_p \subseteq M$

NUA:  $M_p$  can be s.t.  $M_p - M \neq \emptyset$

# The Agreement property

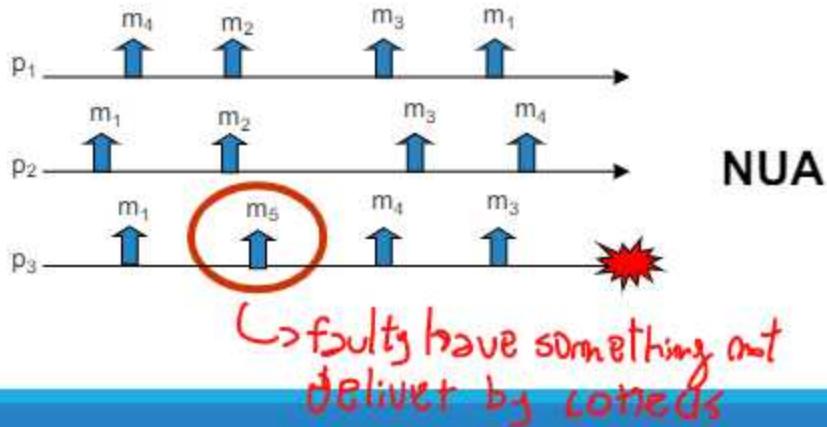
## UNIFORM AGREEMENT (UA)

If a process (correct or not) TOTDelivers a message  $m$ , then all correct processes will eventually TOTDeliver  $m$



## NON-UNIFORM AGREEMENT (NUA)

If a correct process TOTDelivers a message  $m$ , then all correct processes will eventually TOTDeliver  $m$



# The Ordering Property

## STRONG UNIFORM TOTAL ORDER (SUTO)

If some process TODelivers some message  $m$  before message  $m'$ , then a process TODelivers  $m'$  only after it has TODelivered  $m$ .



- same order
- same prefix of the set of delivered messages ↗ fail
- after an omission, disjoint sets of delivered messages

## WEAK UNIFORM TOTAL ORDER (WUTO)

If process  $p$  and process  $q$  both TODeliver messages  $m$  and  $m'$ , then  $p$  TODelivers  $m$  before  $m'$  if and only if  $q$  TODelivers  $m$  before  $m'$ .



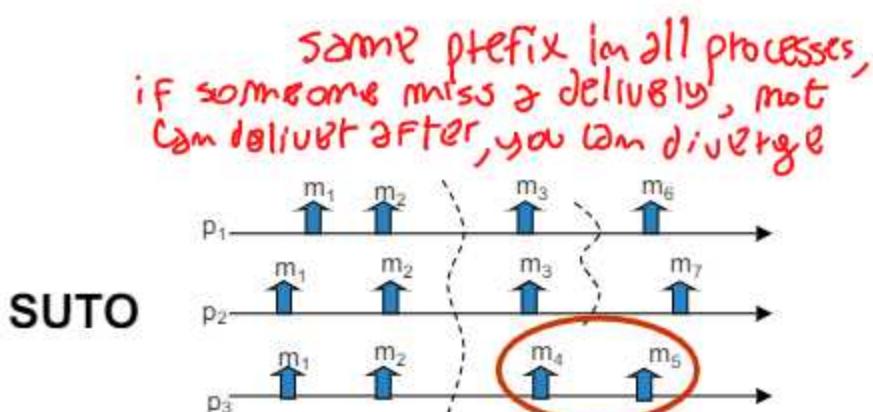
- no restrictions on the set of delivered messages

↳ if you deliver that part you have do the same way

# The Order Property

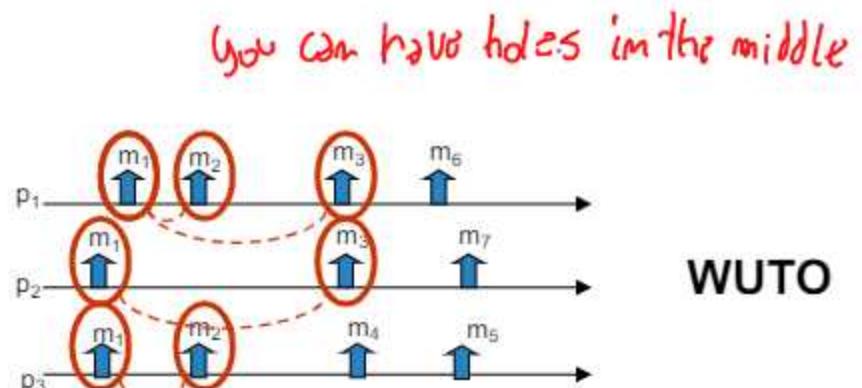
## STRONG UNIFORM TOTAL ORDER (SUTO)

If some process  $\text{TODelivers}$  some message  $m$  before message  $m'$ , then a process  $\text{TODelivers}$   $m'$  only after it has  $\text{TODelivered}$   $m$ .



## WEAK UNIFORM TOTAL ORDER (WUTO)

If process  $p$  and process  $q$  both  $\text{TODeliver}$  messages  $m$  and  $m'$ , then  $p$   $\text{TODelivers}$   $m$  before  $m'$  if and only if  $q$   $\text{TODelivers}$   $m$  before  $m'$ .



# The Order Property

---

SUTO and WUTO are uniform but they both have a non-uniform counterpart

## STRONG NON-UNIFORM TOTAL ORDER (SNUTO)

If some correct process  $\text{TODelivers}$  some message  $m$  before message  $m'$ , then a correct process  $\text{TODelivers}$   $m'$  only after it has  $\text{TODelivered}$   $m$ .

## WEAK NON-UNIFORM TOTAL ORDER (WNUTO)

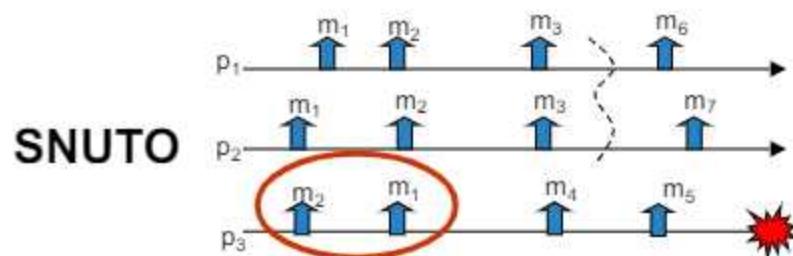
If correct processes  $p$  and  $q$  both  $\text{TODeliver}$  messages  $m$  and  $m'$ , then  $p$   $\text{TODelivers}$   $m$  before  $m'$  if and only if  $q$   $\text{TODelivers}$   $m$  before  $m'$ .

Constraint of before only in correct processes, faulty can do whatever they want

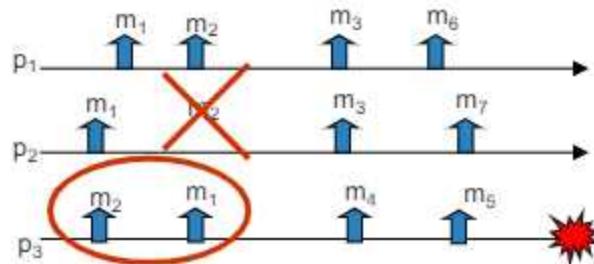
# The Order property (2)

SUTO  $\Rightarrow$  WUTO

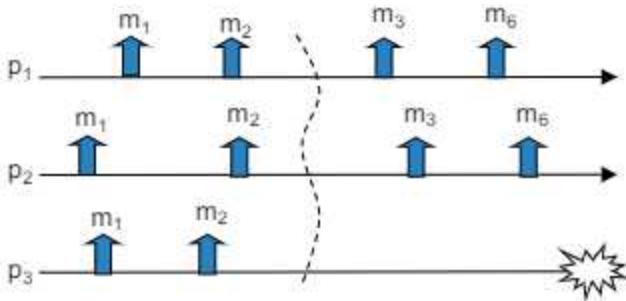
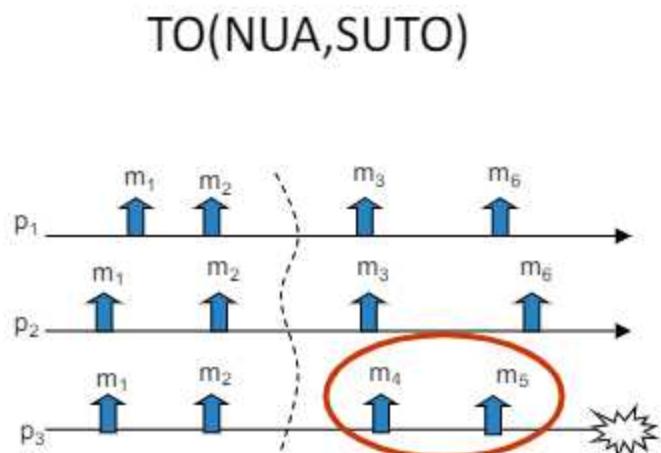
SNUTO  $\Rightarrow$  WNUTO



**WNUTO**

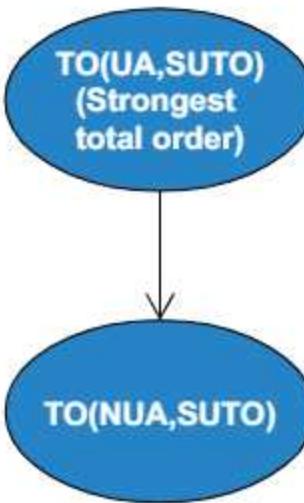


# TO specifications



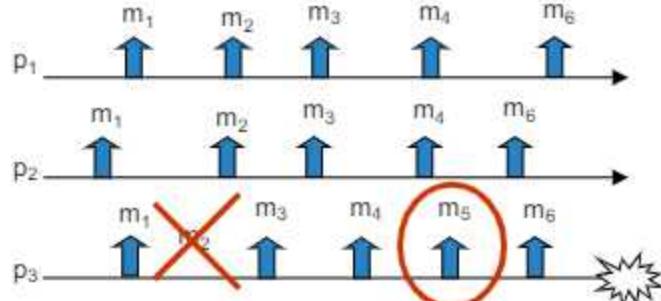
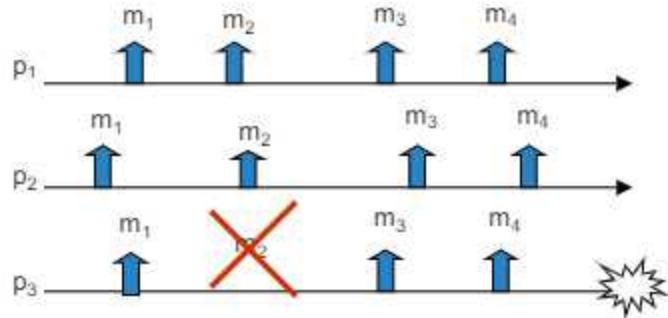
TO(UA,SUTO)

- The strongest TO spec.  
*atomic broadcast*



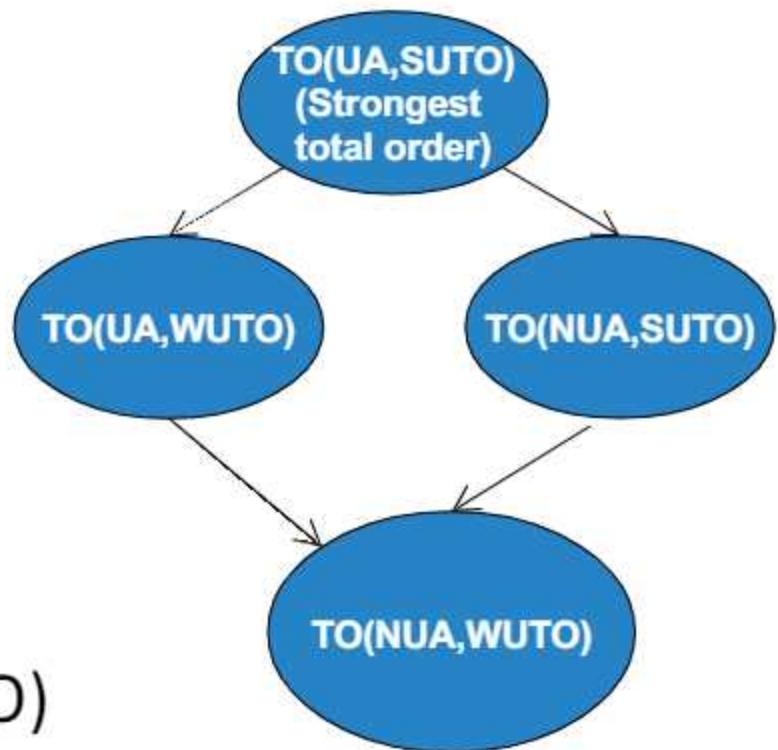
# TO specifications (2)

TO(UA,WUTO)



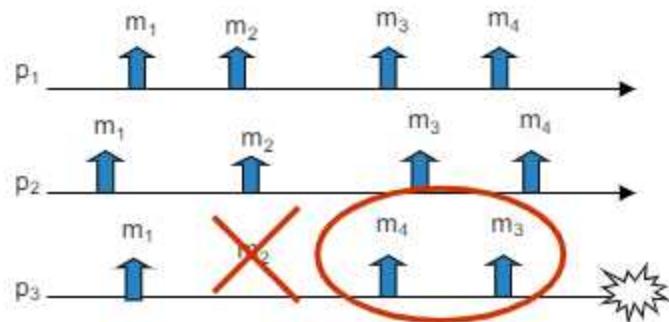
TO(NUA,WUTO)

lose one in the middle

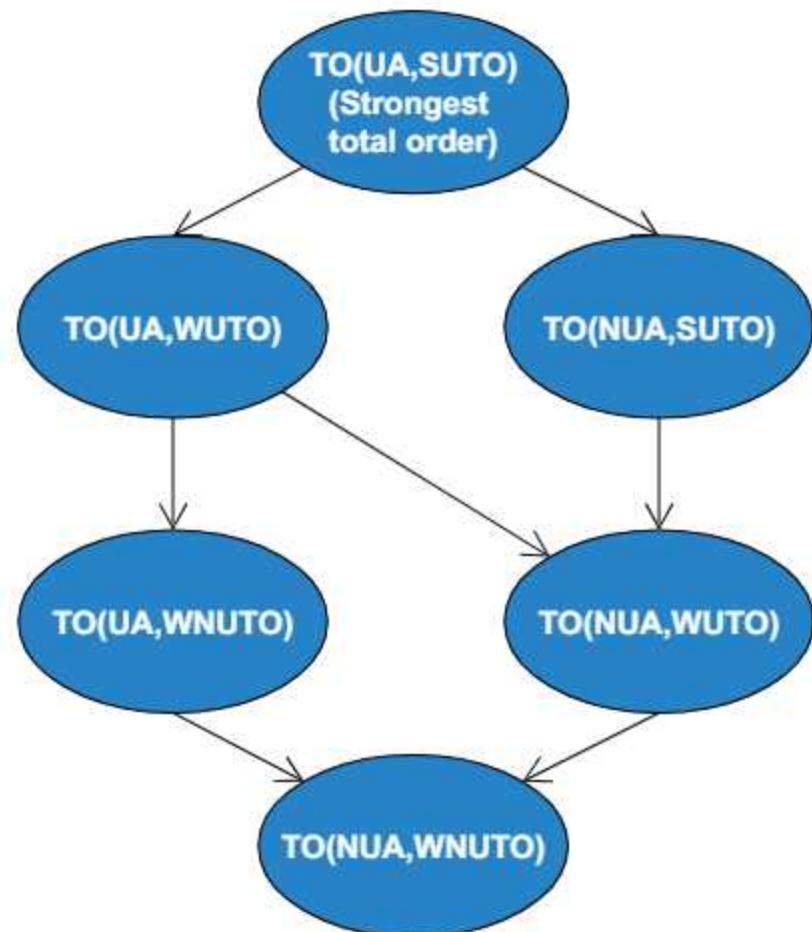
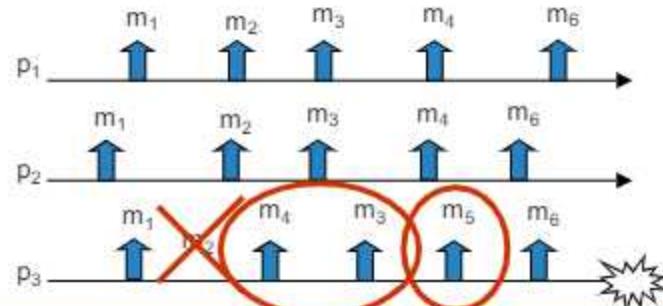


# TO specifications (3)

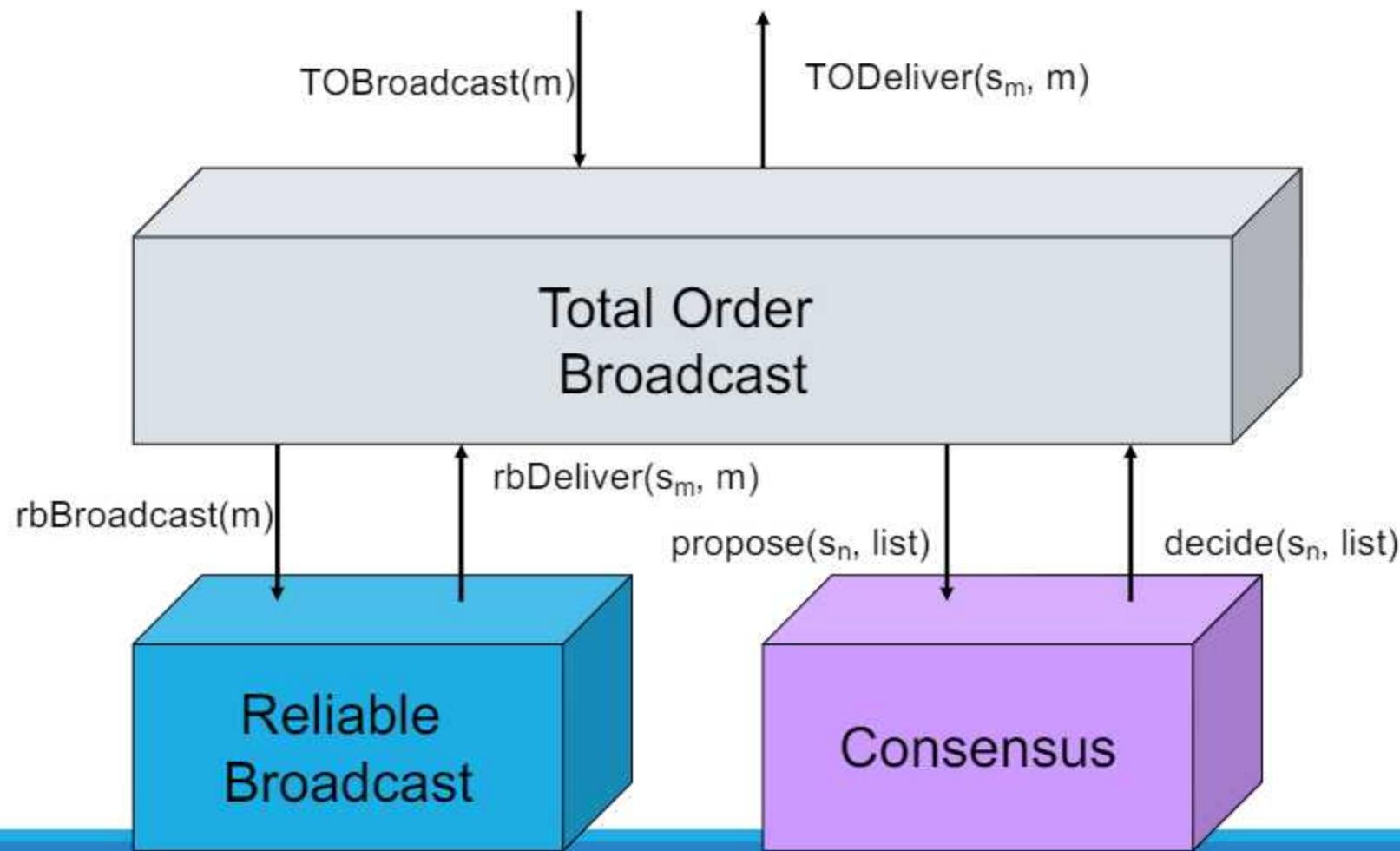
TO(UA,WNUTO)



TO(NUA,WNUTO)



# Total Order **Implementation**



# Total Order Algorithm

---

## Algorithm 6.1: Consensus-Based Total-Order Broadcast

---

### Implements:

TotalOrderBroadcast, instance *tob*.

### Uses:

ReliableBroadcast, instance *rb*;  
Consensus (multiple instances).

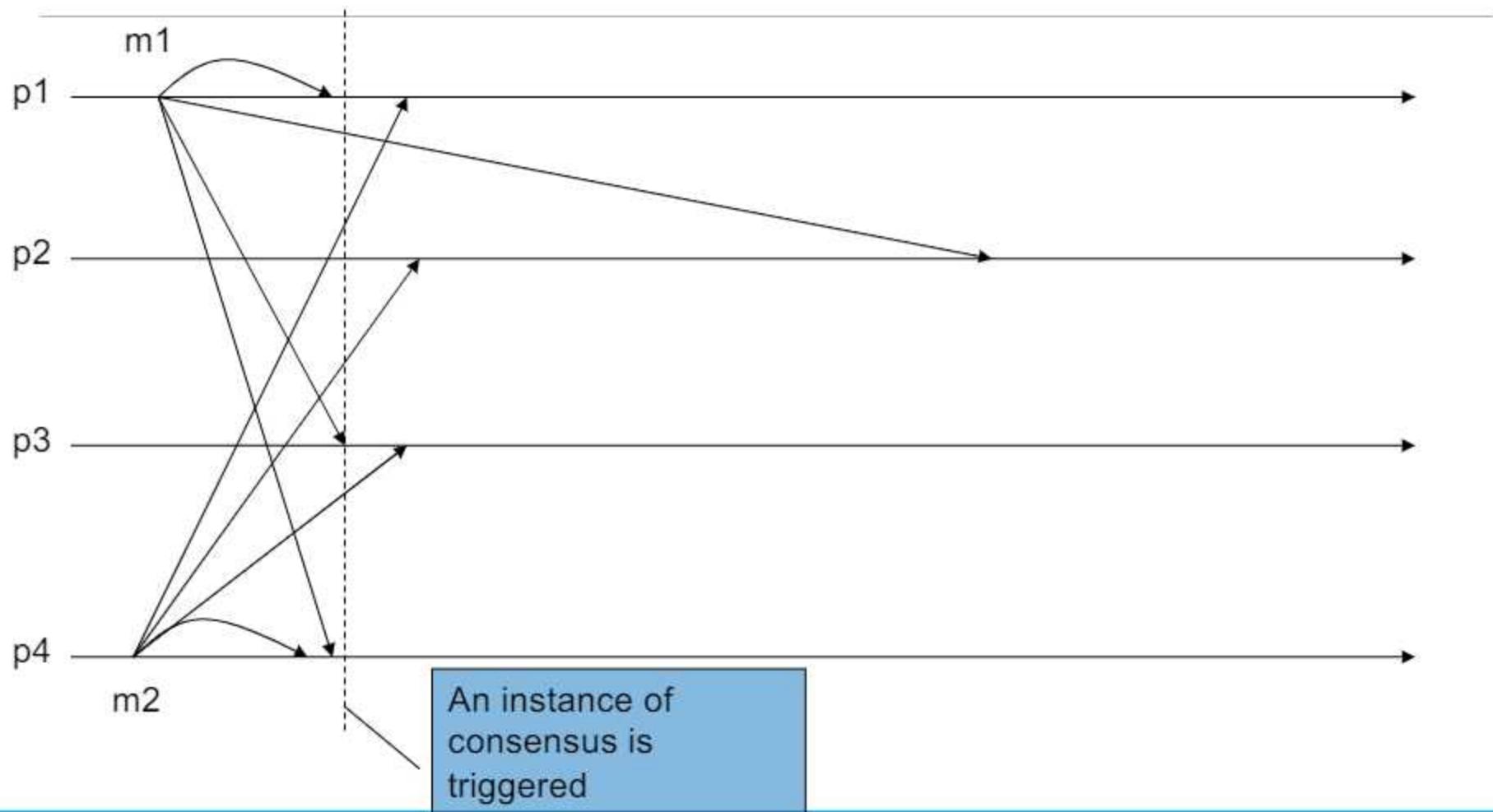
```
upon event < tob, Init > do
  unordered :=  $\emptyset$ ; delivered :=  $\emptyset$ ; round := 1;
  wait := FALSE; to call consensus for call multiple instances of consensus
  unordered :=  $\emptyset$ ; delivered :=  $\emptyset$ 
  round := 1;
  wait := FALSE;
upon event < tob, Broadcast | m > do
  trigger < rb, Broadcast | m >;
upon event < rb, Deliver | p, m > do
  if m  $\notin$  delivered then
    unordered := unordered  $\cup$  {(p, m)};
```

*not do anything*

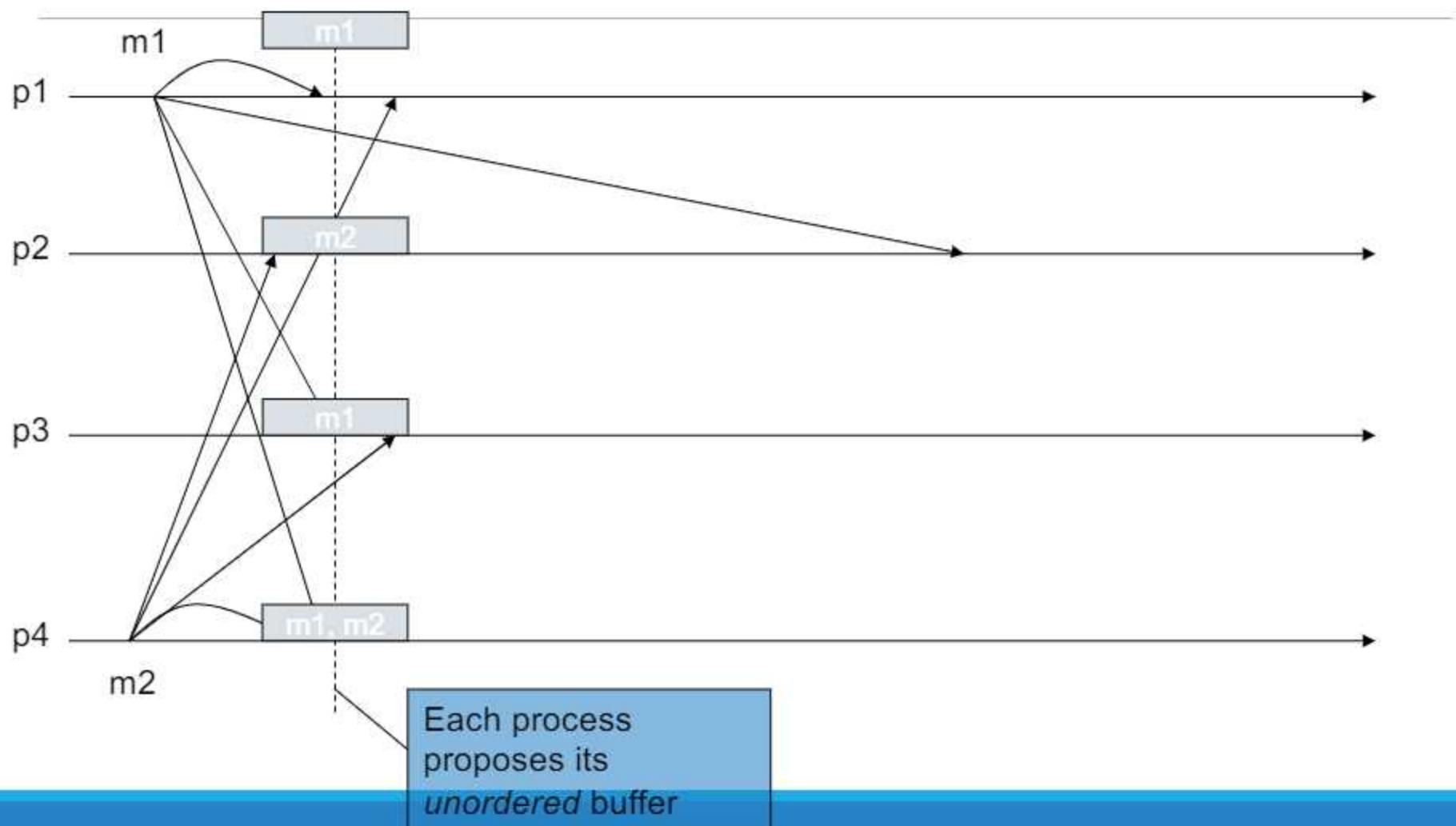
upon *unordered*  $\neq \emptyset$   $\wedge$  *wait* = FALSE do  
 *wait* := TRUE; *to block processing*  
 Initialize a new instance *c.round* of consensus;  
 trigger < *c.round*, Propose | *unordered* >;

upon event < *c.r*, Decide | *decided* > such that *r* = *round* do  
 forall  $(s, m) \in \text{sort}(\text{decided})$  do *by the order in the resulting sorted list*  
 trigger < *tob*, Deliver | *s*, *m* >;  
 *delivered* := *delivered*  $\cup$  *decided*;  
 *unordered* := *unordered*  $\setminus$  *decided*;  
 *round* := *round* + 1;  
 *wait* := FALSE;

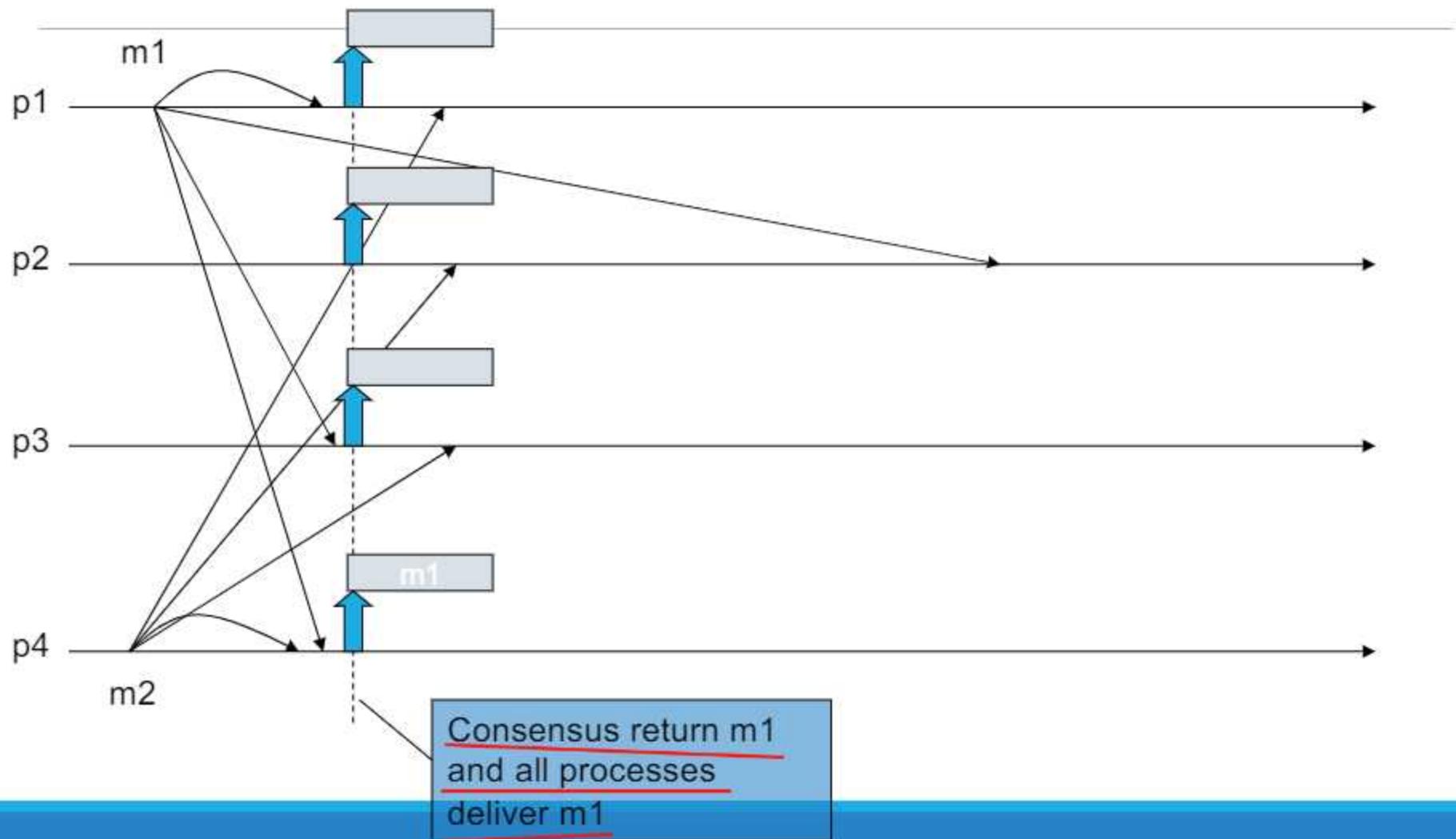
# Example



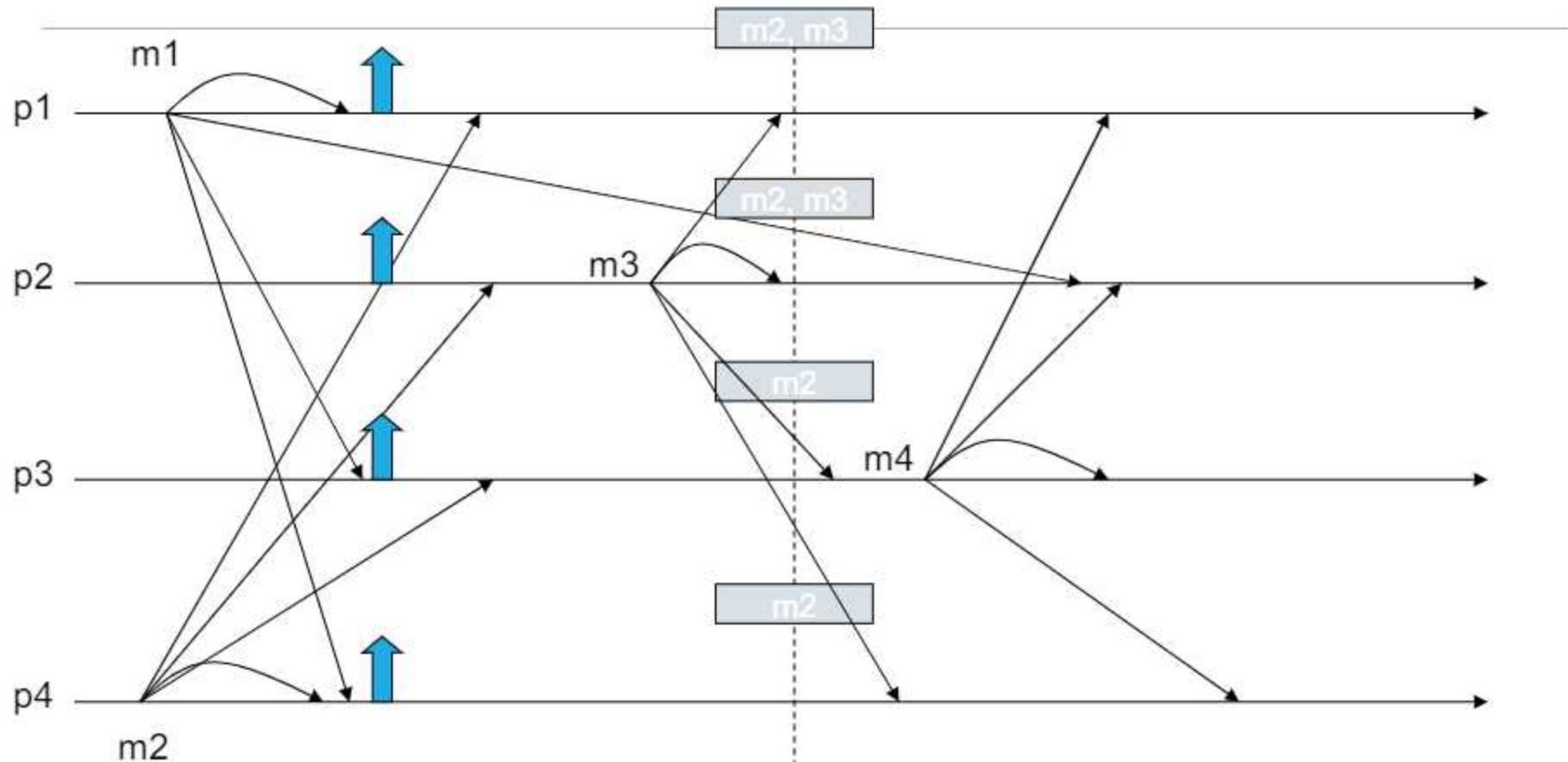
# Example



# Example



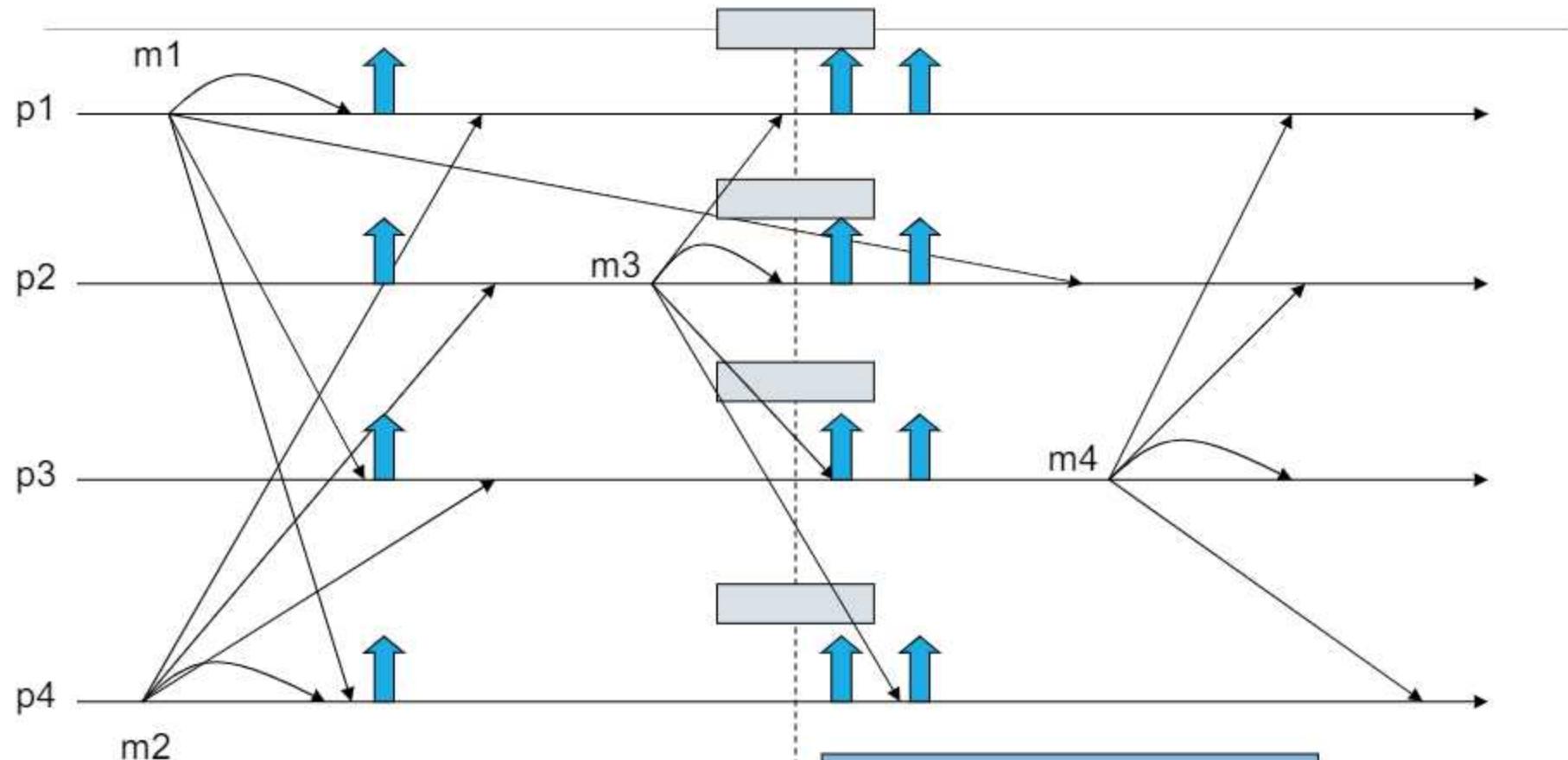
# Example



An instance of  
consensus is  
triggered

Each process  
proposes its  
unordered buffer

# Example



# Exercice

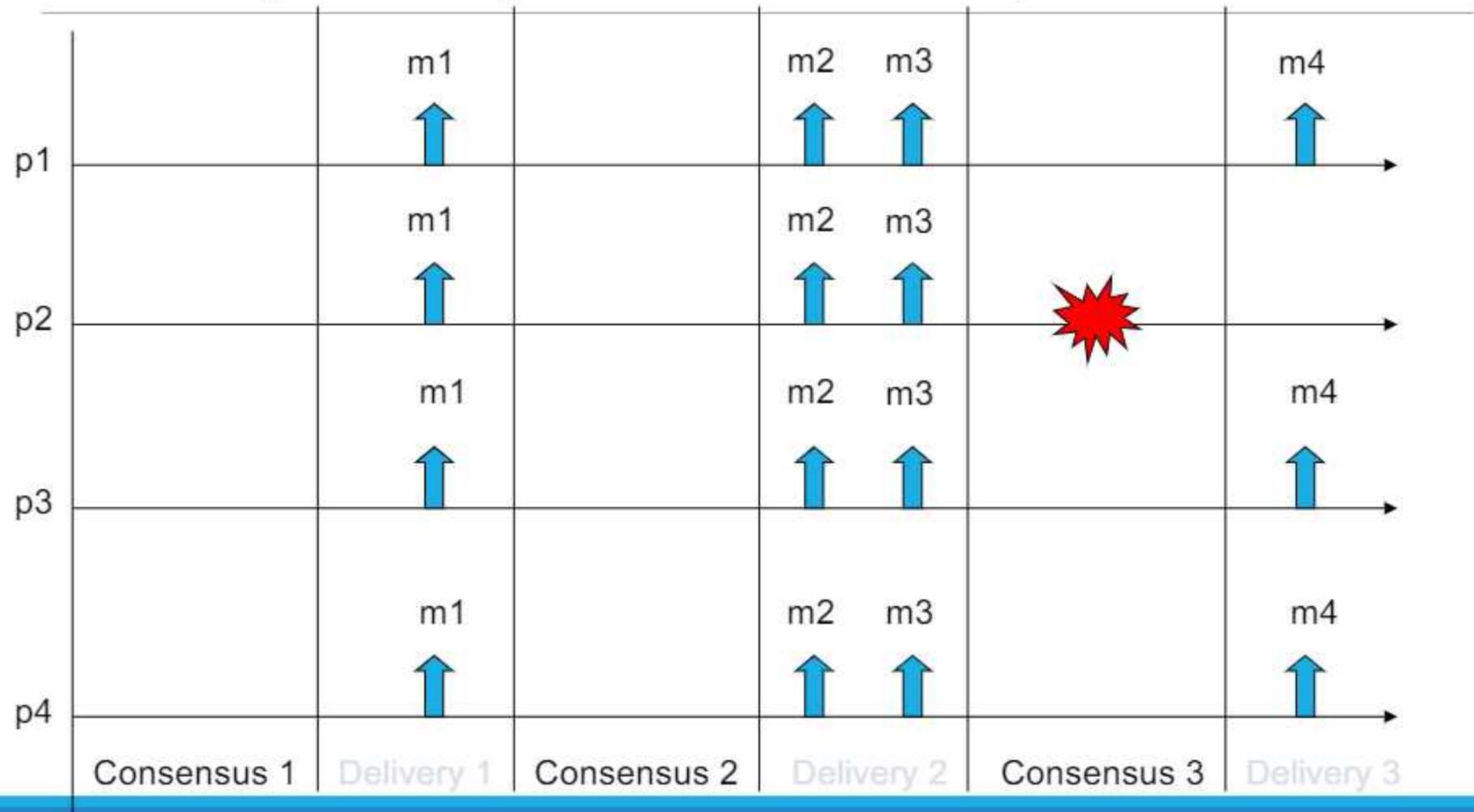
---

Which TO specification is satisfied by this algorithm?

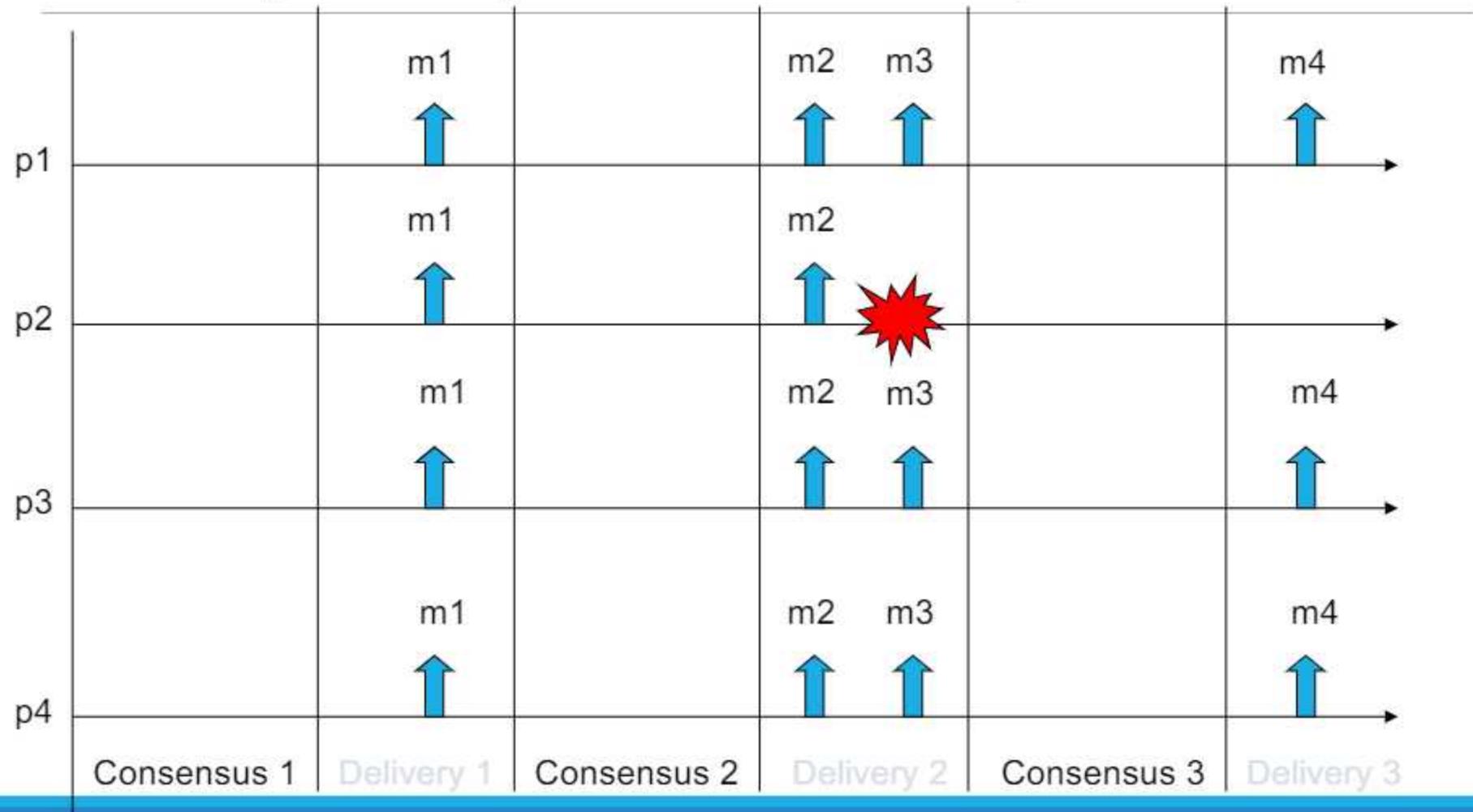
It depends from the assumptions about Reliable Broadcast and Consensus

Consensus \\ Reliable Broadcast	Uniform	Non Uniform
Uniform	UA SUTO	UA WNUTO
Non Uniform	NUA SUTO	NUA WNUTO

# Example 1 (UC and URB)



## Example 2 (UC and URB)



# Uniform Consensus (UC) and Uniform Reliable Broadcast (URB)

---

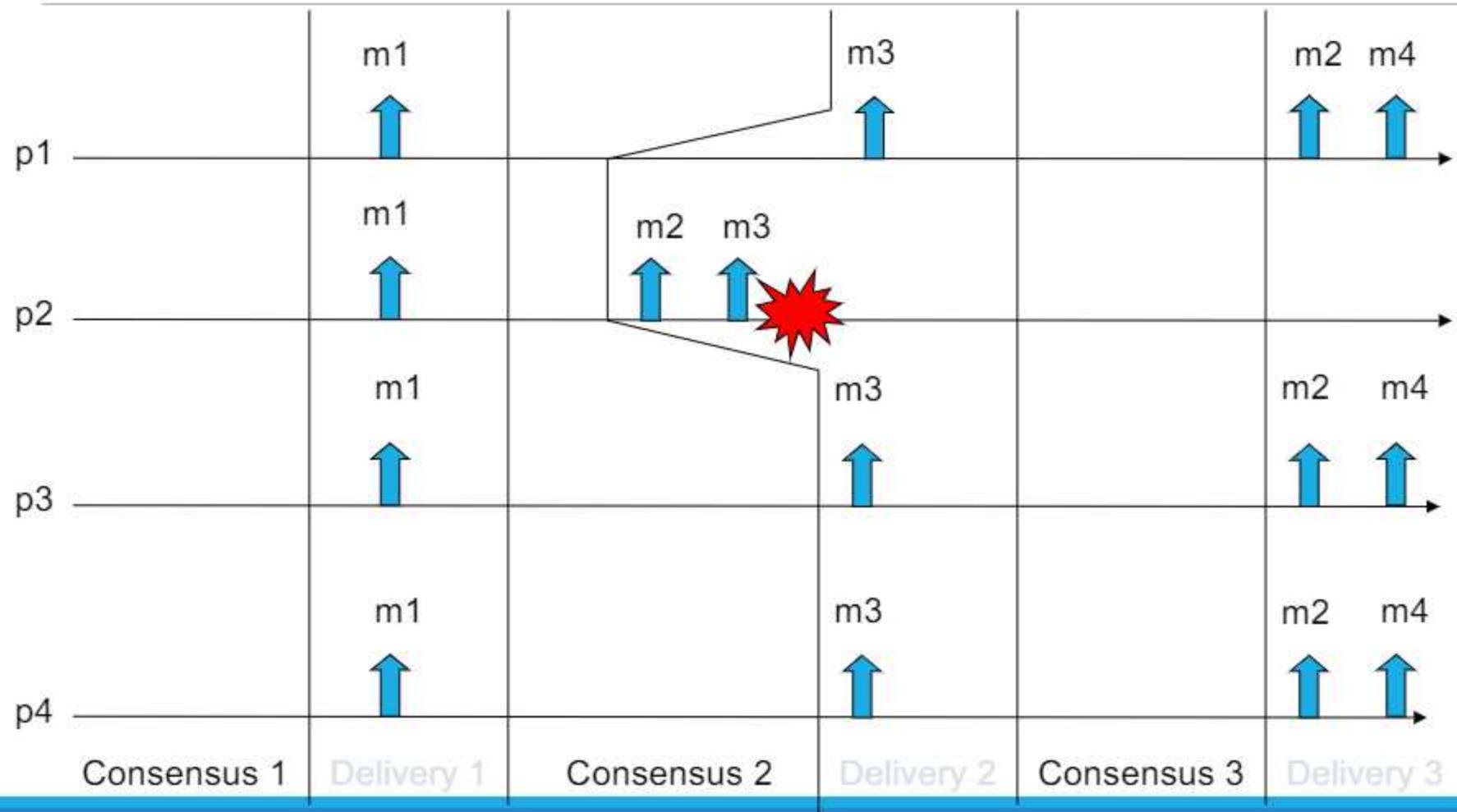
Assuming both Consensus and Reliable Broadcast uniform we have

TO (UA, SUTO)

*Proof.*

- Due to URB all the processes (even the faults) deliver the same set of messages
- The unordered buffer contains the same set of messages for each process
  - All the processes will deliver the same set of messages (UA)
- Due to UC, all processes (even the faults) decide for the same list of messages
- Messages are sorted by a deterministic rule
  - All processes will deliver the messages in the same order

# Example (NUC and URB)



# Non Uniform Consensus (NUC) and Uniform Reliable Broadcast (URB)

---

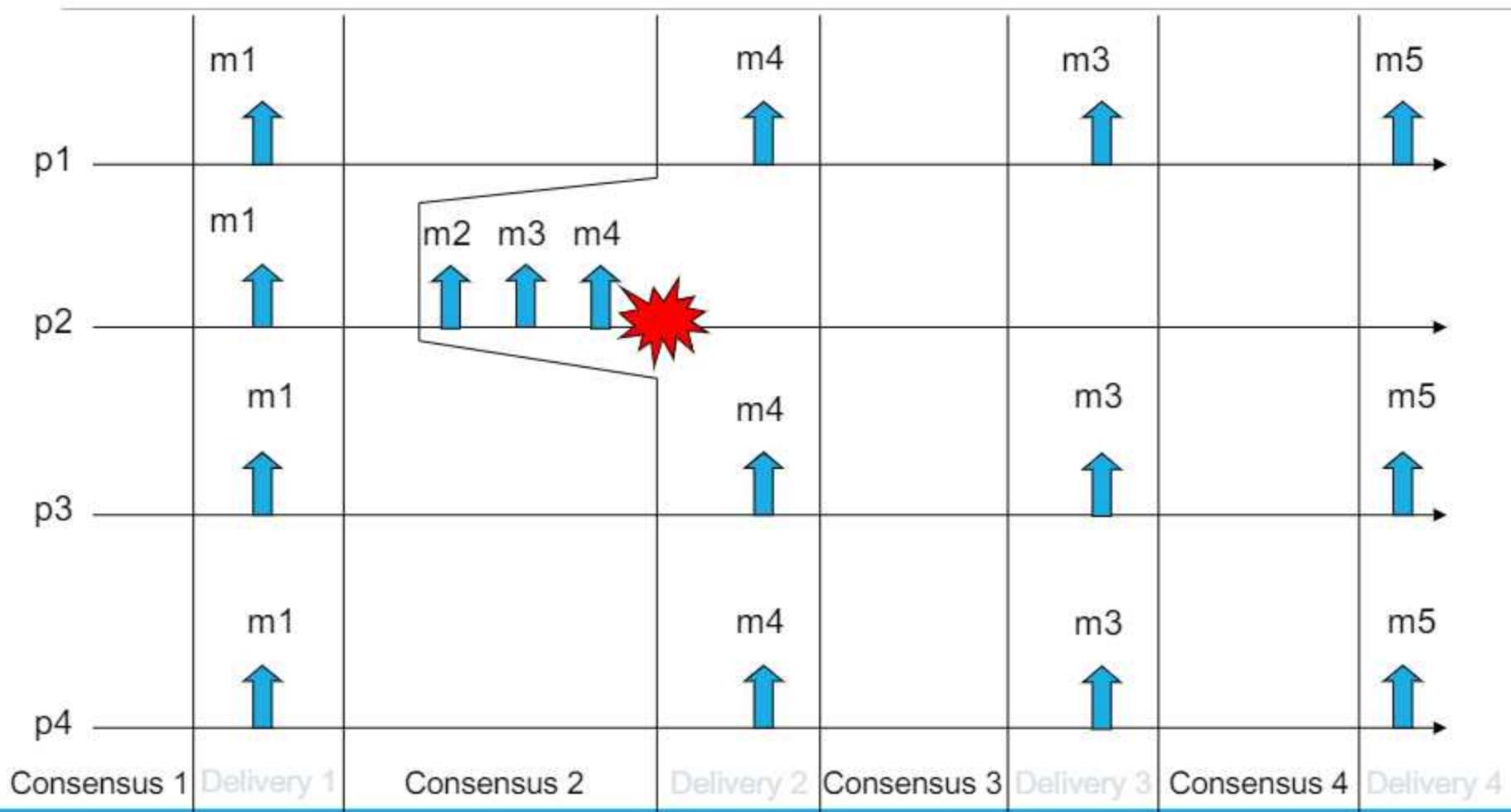
Assuming both Consensus and Reliable Broadcast uniform we have

## TO (UA, WNUTO)

*Proof.*

- Due to URB all the processes (even the faults) deliver the same set of messages
- The unordered buffer contains the same set of messages for each process
  - All the processes will deliver the same set of messages (UA)
- Due to NUC, all correct processes decide for the same list of messages
- Faulty processes can decide differently
  - All correct processes will deliver the messages in the same order
  - Faulty processes will deliver, just before a crash, a different sequence of messages

# Example (NUC and NURB)



# Non Uniform Consensus (NUC) and Non Uniform Reliable Broadcast (NURB)

---

Assuming both Consensus and Reliable Broadcast uniform we have

**TO (NUA, WNUTO)**

*Proof.*

- Due to NURB correct processes deliver the same set of messages
- Faulty processes can deliver other messages
  - Only correct processes will deliver the same set of messages (NUA)
- Due to NUC, all correct processes decide for the same list of messages
- Faulty processes can decide differently
  - All correct processes will deliver the messages in the same order
  - Faulty processes will deliver, just before a crash, a different sequence of messages

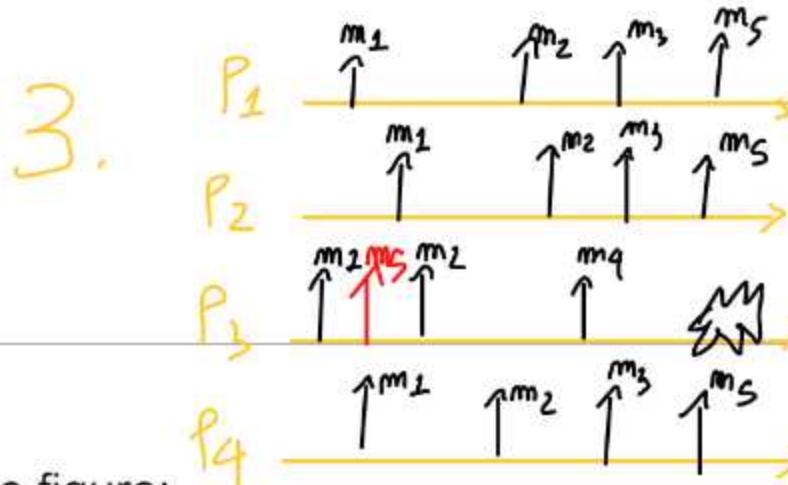
$SNUTO = WNUTO$

Consensus	Uniform	Non Uniform
Reliable Broadcast		
Uniform	UA SUTO	UA WNUTO
Non Uniform	NUA SUTO	NUA WNUTO

## Exercice

Which specification is satisfied assuming UC and NURB?

# Exercice

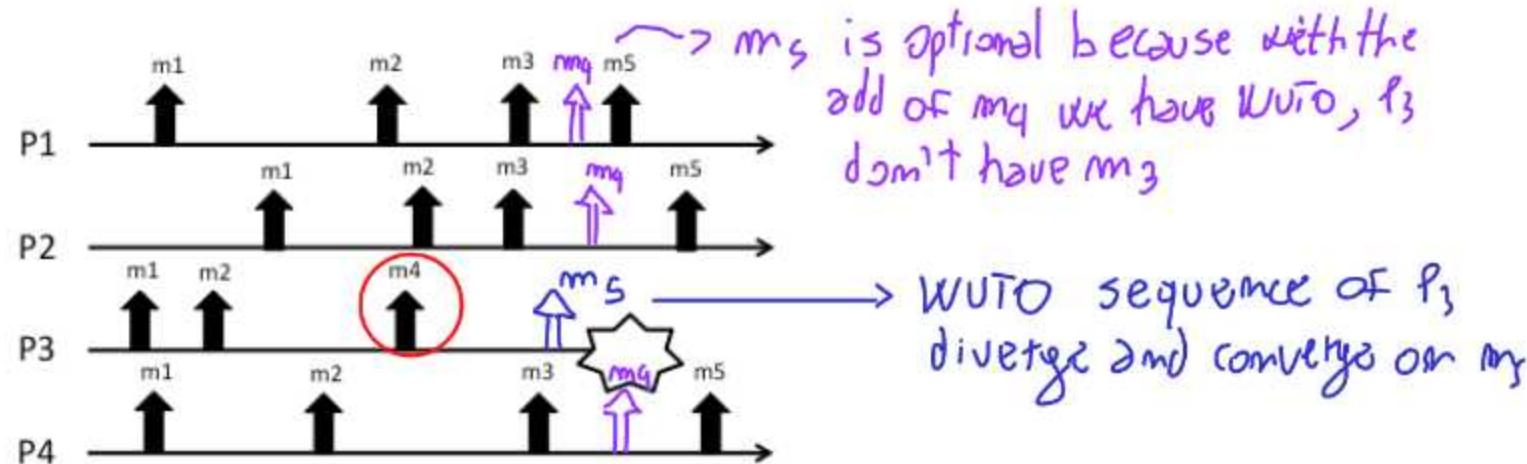


Consider the run depicted in the figure:

TO(CNUA, SUTO)

SUTO only diverges P<sub>3</sub>

NUA for m<sub>4</sub> in P<sub>3</sub>



1. Which type of total ordering is satisfied by the run? Specify both the agreement and the ordering properties.
2. Modify the run in order to satisfy TO(UA, WUTO) but not TO (UA SUTO)
3. Modify the run in order to satisfy TO(NUA, WNUTO) but not TO(NUA, WUTO)

# References

---

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

- Chapter 6 – Section 6.1

Stefano Cimmino, Carlo Marchetti, Roberto Baldoni "*A Guided Tour on Total Order Specifications*" WORDS Fall 2003: 187-194

Dependable Distributed Systems  
Master of Science in Engineering in  
Computer Science

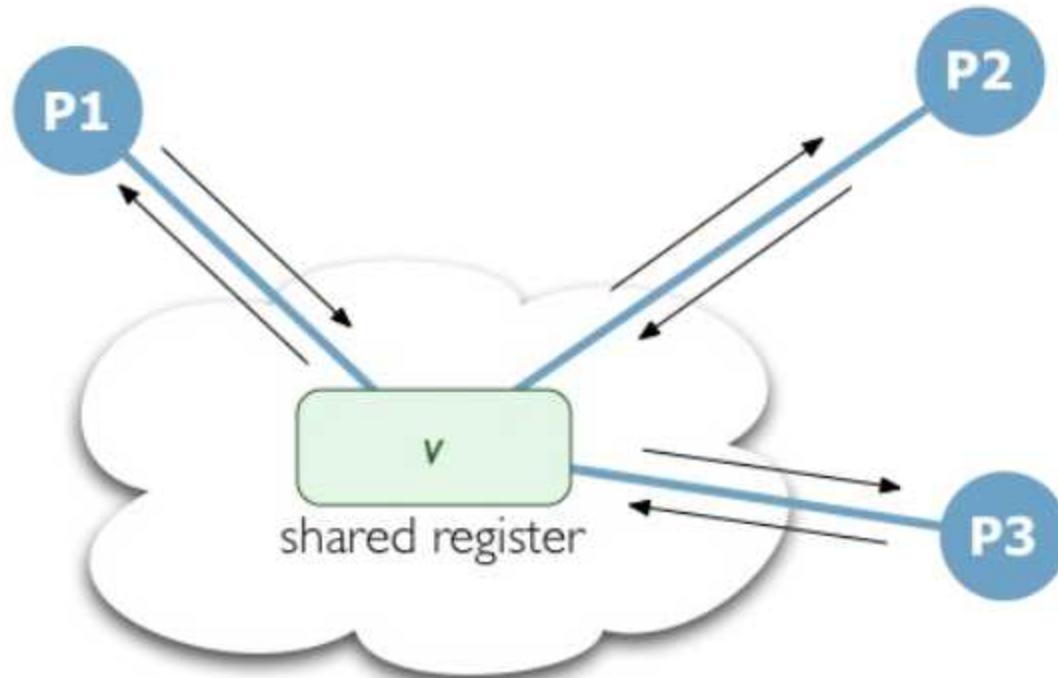
AA 2022/2023

---

LECTURE 14 & 15: REGISTERS

# Register: definition

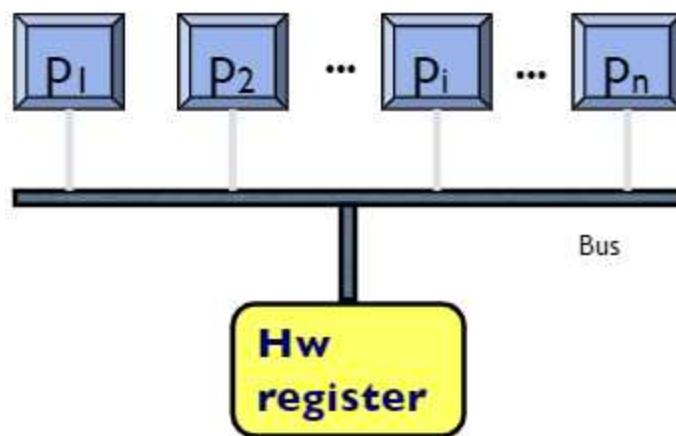
A register is a shared variable accessed by processes through read and write operations



# Distributed Systems: register abstraction

- **Multiprocessor machine:**

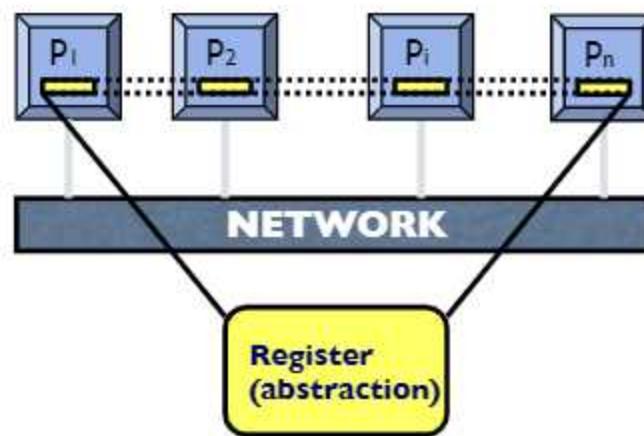
- Processes typically communicate through registers at hardware level



- The set of these registers constitute the physical memory

- **Distributed message passing system:**

- no physical shared memory
- Processes communicate exchanging msg over a network



- Register abstraction support the design of distributed solution, by hiding the complexity of the underlying message passing system and the distribution of the data

# Register operations

---

A process accesses a register through:

---

- **Read operation**, read () → v: it returns the “current” value v of the register; this operation does not modify the content of the register;
- **Write operation**, write (v) : it writes the value v in the register and returns true at the end of the operation, *overwrite old value*

Each operation starts with an invocation and terminates when the corresponding response is received

---

*Operations is not instantaneous, have a latency*

# Register: Assumption

---

*r7 is an integer variable, default is 0*

- A register stores only positive integers and it is initialize to 0
- Each value written is univocally identified *→ not going to write two times of same values*
- Processes are sequential: a process cannot invoke a new operation before the one it previously invoked (if any) returned, *at most one operation in each time unit*

## Register: Notation

each process have a role, can be a writer or a reader.

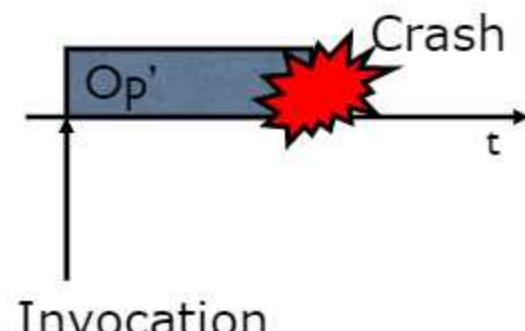
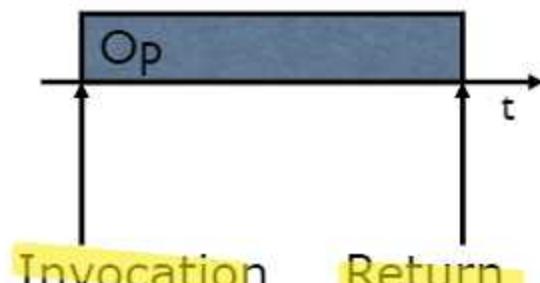
$(X,Y)$  denotes a register where  $X$  processes can write and  $Y$  processes can read

*, globally not at same time, only  $X$  processes have the ability to read, while  $Y$  can read*

- $(1,1)$  denotes a register where only a process can write and only a process can read. It is a priori known which process can write and which can read
- $(1,N)$  denotes a register where a single process, a priori known, can write, and  $N$  processes can read

# Operations

- Every operation is characterized by two events:
  - Invocation
  - Return (Confirmation for the write operation and a value for the read)
- Each of these events occur at a single indivisible point of time
- An operation is complete if both the invocation and the return events are occurred
- A Failed operation is an operation invoked by some process  $p_i$  that crashes before obtaining a return

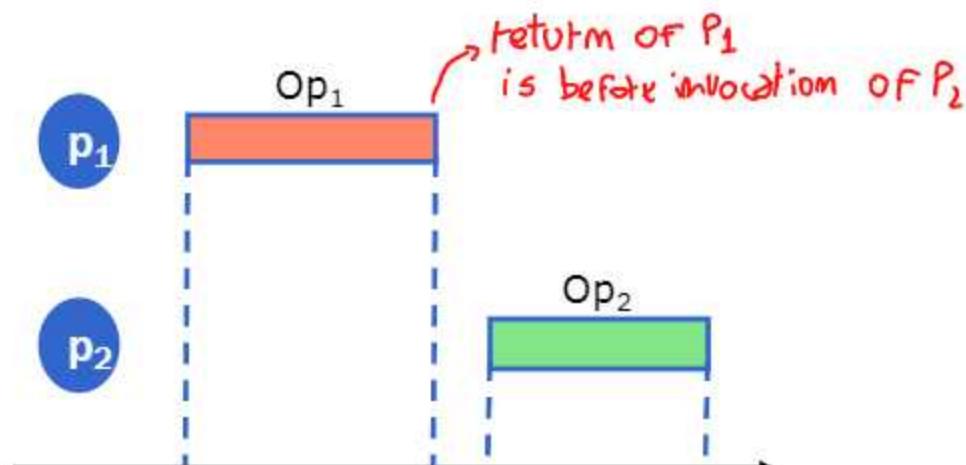


## Precedence between Operations

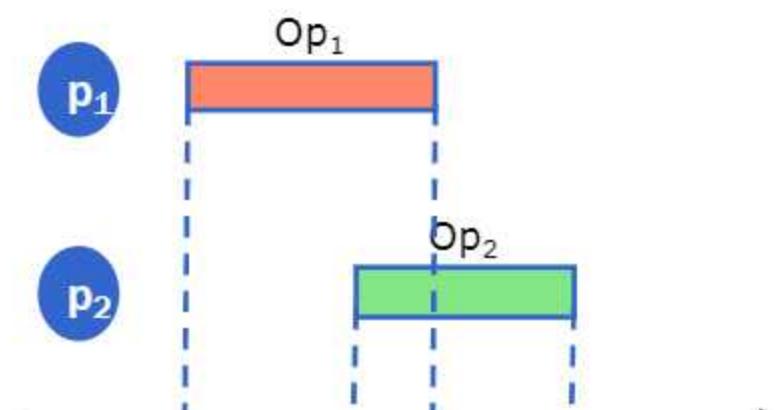
---

- The execution of an operation invoked by a process p, is the time interval defined by the invocation event and the return event
- Given two operations  $o \neq o'$ ,  $o$  precedes  $o'$  if the response event of  $o$  precedes the invocation event of  $o'$
- An operation  $o$  invoked by a process  $p$  may precedes an operation  $o'$  invoked by  $p'$  only if  $o$  completes
- If it is not possible to define a precedence relation between two operations, they are said to be concurrent

# Example



$Op_1$  precedes  $Op_2$



$Op_1$  is concurrent  $Op_2$

# Register Semantics: Serial System, No failures

## Assumptions:

- **serial access**: a process does not invoke an operation on the register if there is another process that previously invoked an operation on it and this latter does not yet complete
- no failures

↳ no concurrency

## Sequential Specification

- **Liveness**. Each operation eventually terminates
- **Safety**. Each read operation returns the last value written

~ we have a sequential object



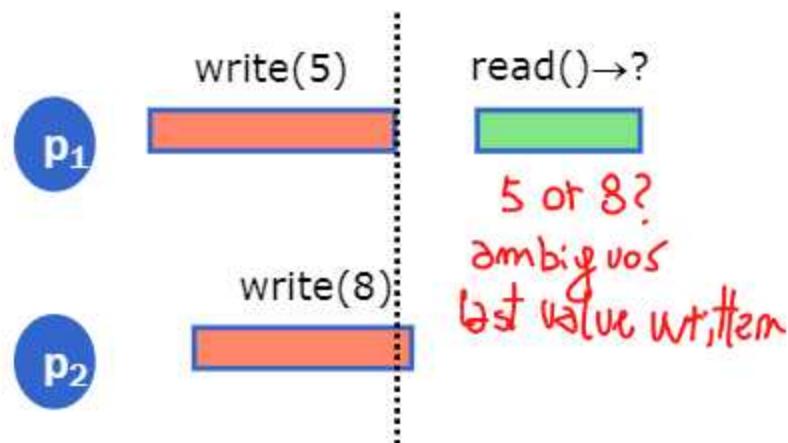
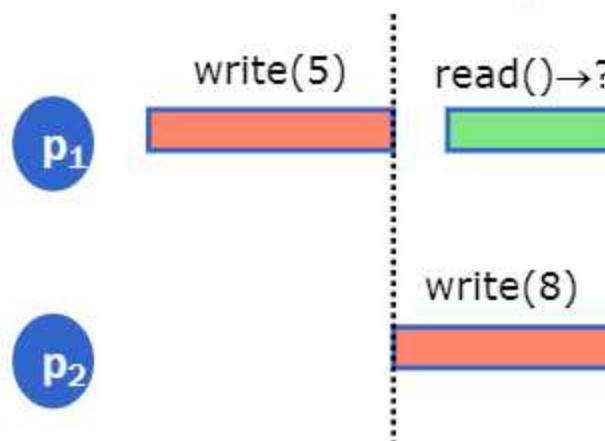
# Register semantics: Concurrency

Assumptions:

- several processes can access the register
- concurrent access
- no failures

↳ because process are located  
in different machines and don't  
know what other are doing

→ concurrent event

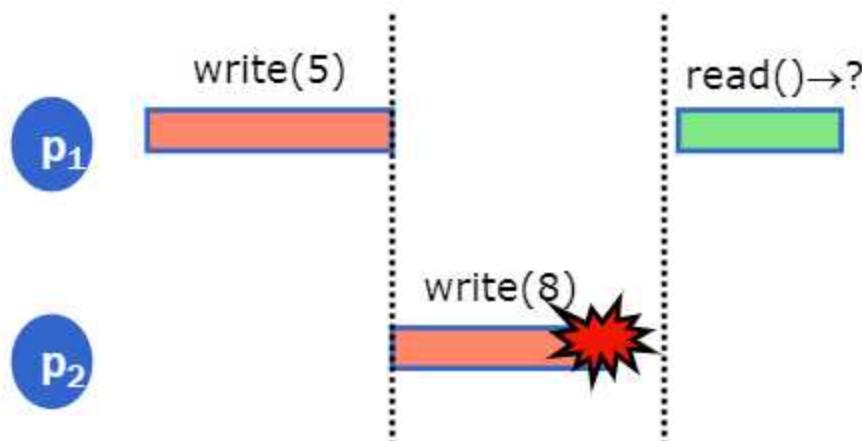


- Which value does the read operation has to return?

# Register semantics: failures

## Assumptions:

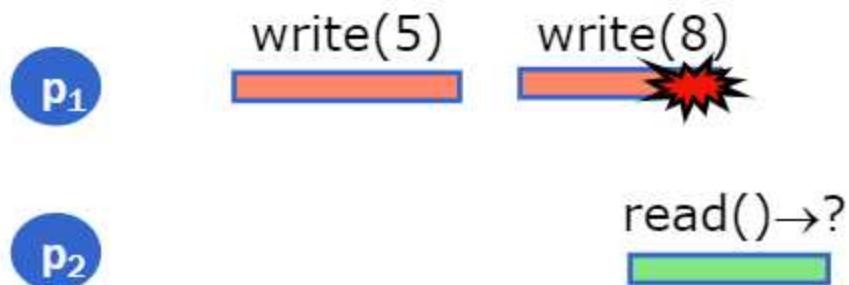
- several processes can access the register
- serial access
- processes can fail by crashing, i.e. after some point in time they stop to run their algorithm forever



**Failed operation:** a process fails at some time in between the invocation and the response of the operation

Which value does the read operation has to return?

## Register Semantics: Concurrency & Failures



A process can invoke a write operation and then crash before the corresponding response event is generated. The write operation could have taken place or not

Register semantics: a read may return both:

- The value written by the last write operation which completes
- The value given as input to the last write operation, even though this operation will fail

# Register Specification

---

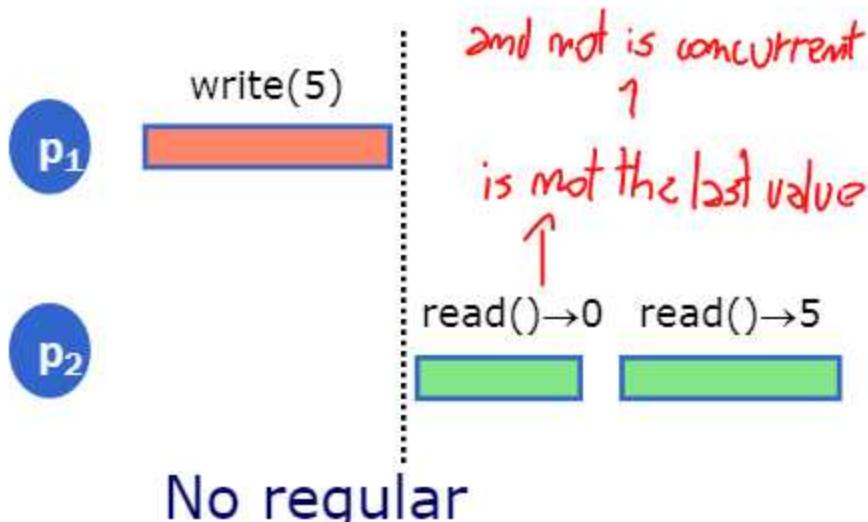
REGULAR AND ATOMIC *two specification*

No multiple write at same time

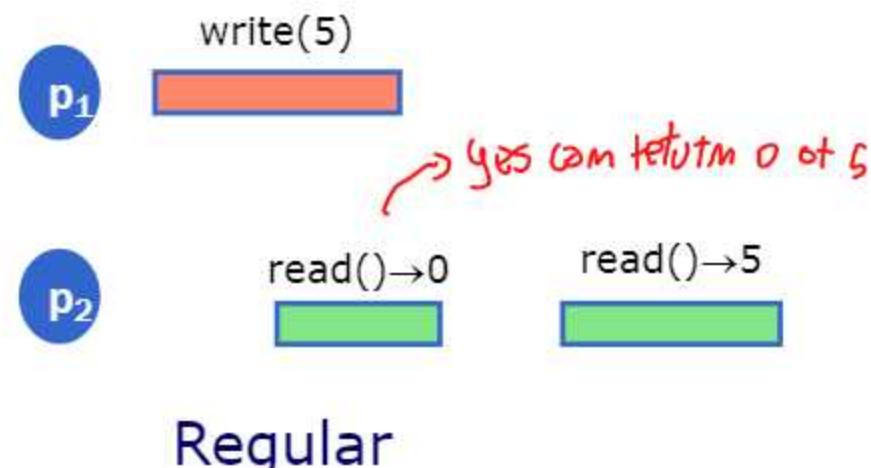


## (1,N) Regular Register: Specification

**Termination.** If a correct process invokes an operation, then the operation eventually receives the corresponding confirmation, correct process have a return **Validity.** A read operation returns the last value written or the value concurrently written ↳ both can return a read

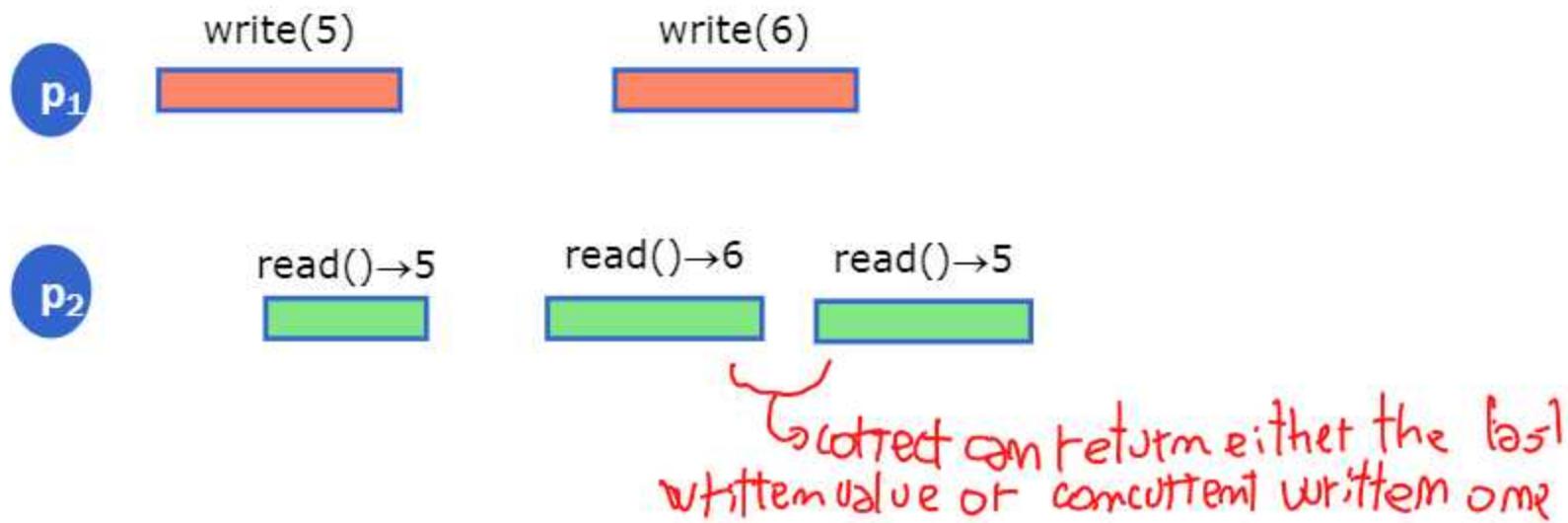


No regular



Regular

# (1,N) Regular Register: Scenario



NOTE: In a regular register, a process can read a value  $v$  and then a value  $v'$ , even if the writer has written  $v'$  and then  $v$ , as long as the write and the read operations are concurrent

This behavior is not allowed in an ATOMIC register

## (1,N) Atomic Register: Specification

---

IDEA: regular register + ordering.

Properties:

**Termination.** If a correct process invokes an operation, then the operation *eventually* receives the corresponding confirmation.

---

**Validity.** A read operation returns the last value written or the value concurrently being written.

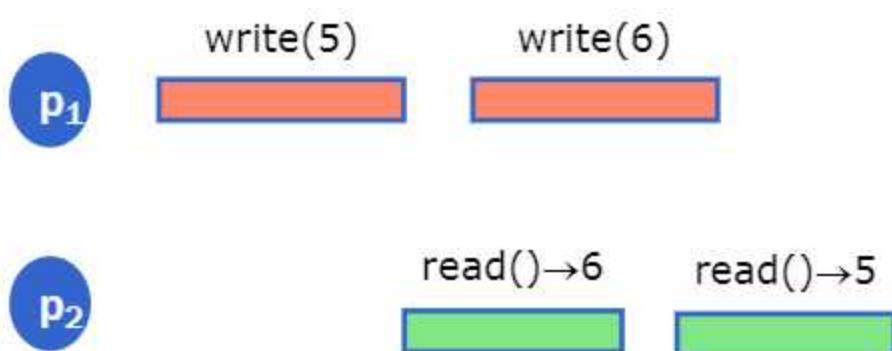
---

**Ordering.** If a read returns  $v_2$  after a read that it precedes it has returned  $v_1$ , then  $v_1$  cannot be written after  $v_2$

↳ second value that you read can't precede the first

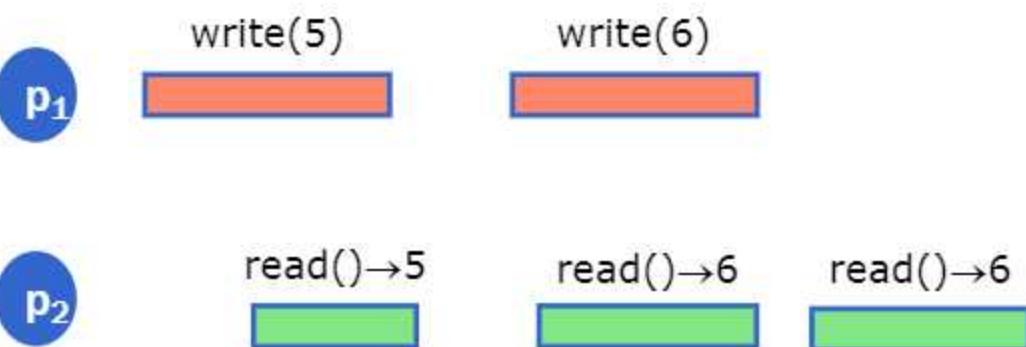
# (1,N) Atomic Register: scenario

es.1



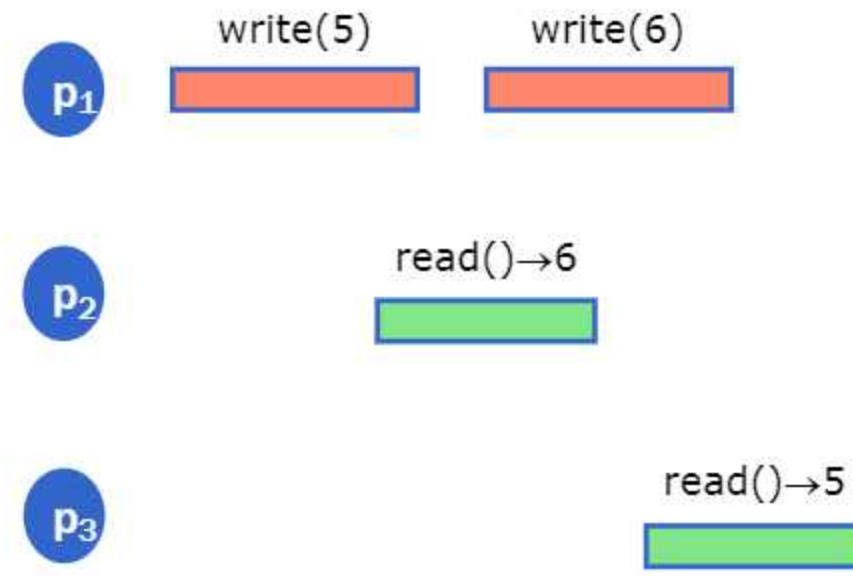
1. Regular but not atomic register: Write(5) precedes write(6). But process  $p_2$  read first the value 6 and then the value 5

es.2



2. The register is atomic

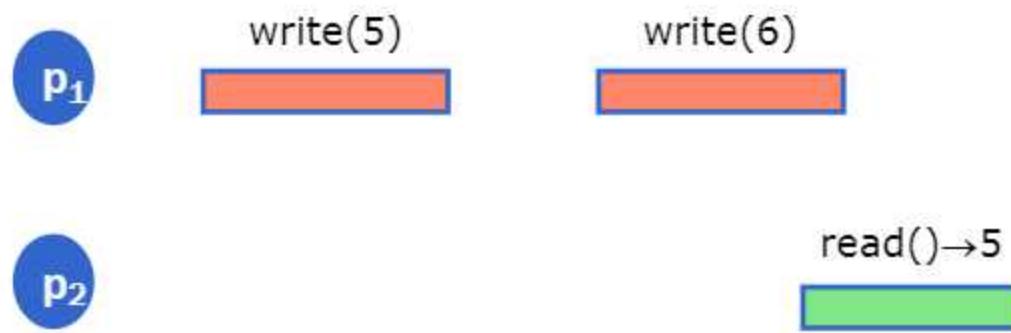
# (1,N) Atomic register: scenario



Not atomic register: the precedence relation also refers to read operations issued by different processes

# Scenario 1

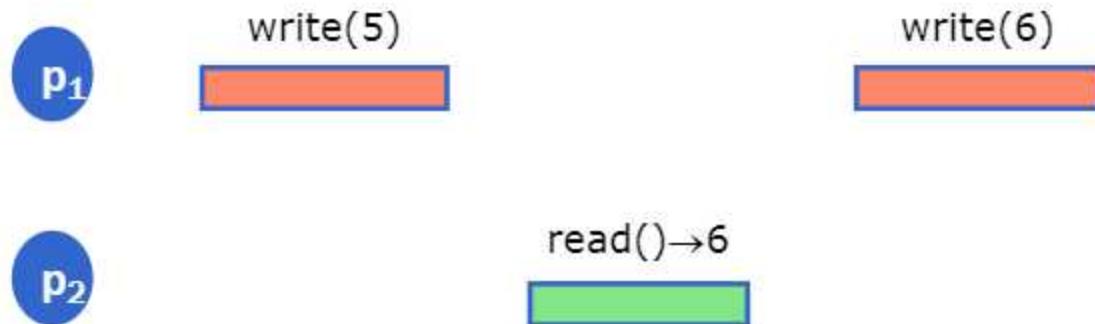
---



**ATOMIC** and **REGULAR**.

## Scenario 2

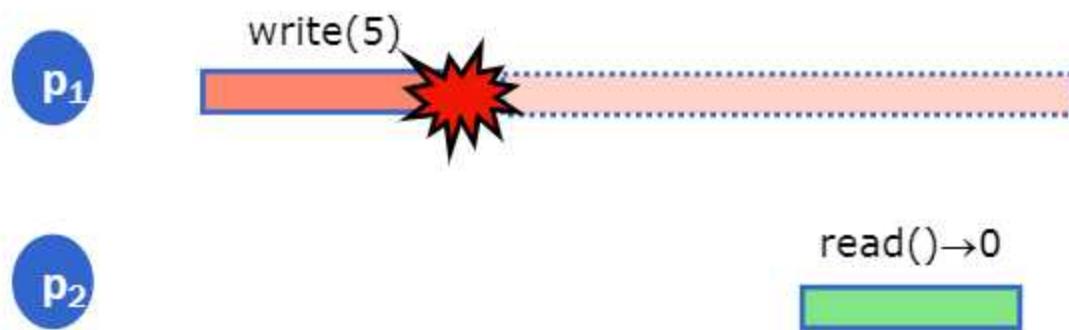
---



**NOT ATOMIC** and **NOT REGULAR**

## Scenario 3

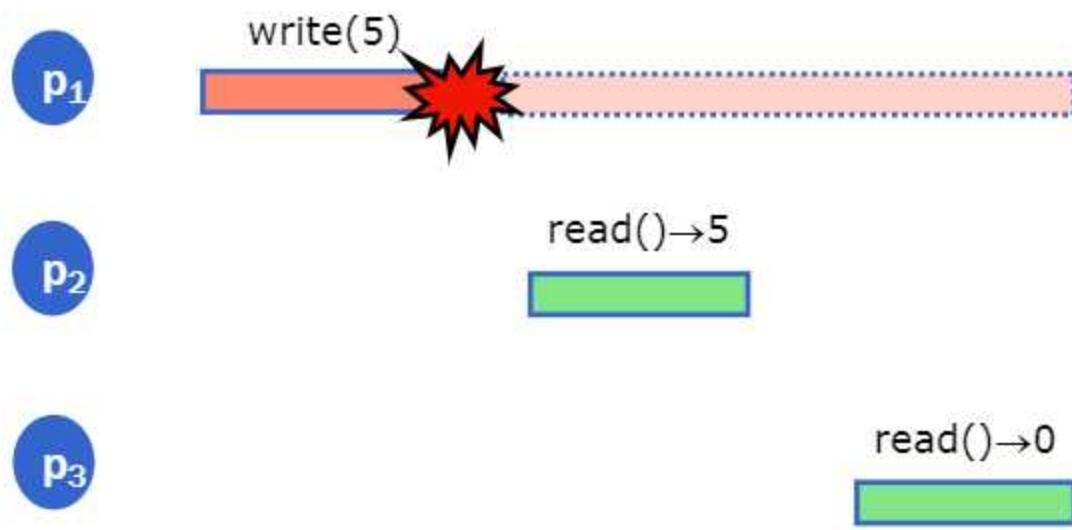
---



**ATOMIC** and so **REGULAR**. write(5) executed by p<sub>1</sub> fails. So, it does not complete and it is concurrent with the read by p<sub>2</sub>. Validity is respected.

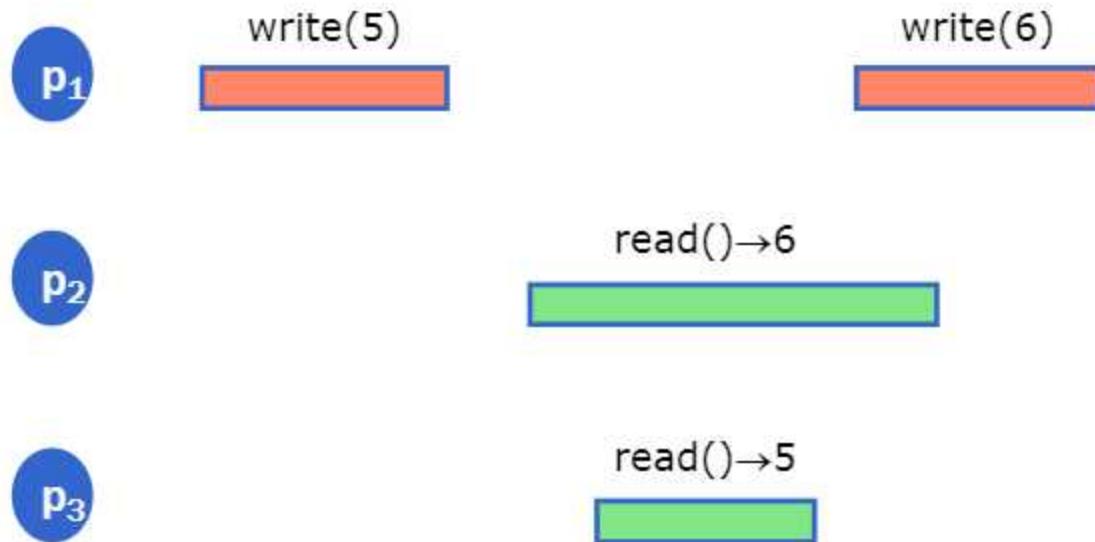
## Scenario 4

---



**REGULAR** but non **ATOMIC**. The ordering property is violated.

# Scenario 5



↪ not valid the order ate  
concurrent

**ATOMIC** and **REGULAR**

# Regular Register

---

IMPLEMENTATION

# (1,N) Regular Register: Interface

## Module 4.1: Interface and properties of a $(1, N)$ regular register

Module:

Name:  $(1, N)$ -RegularRegister, instance  $onrr$ .

Events:

Request:  $\langle onrr, \text{Read} \rangle$ : Invokes a read operation on the register.

Request:  $\langle onrr, \text{Write} \mid v \rangle$ : Invokes a write operation with value  $v$  on the register.

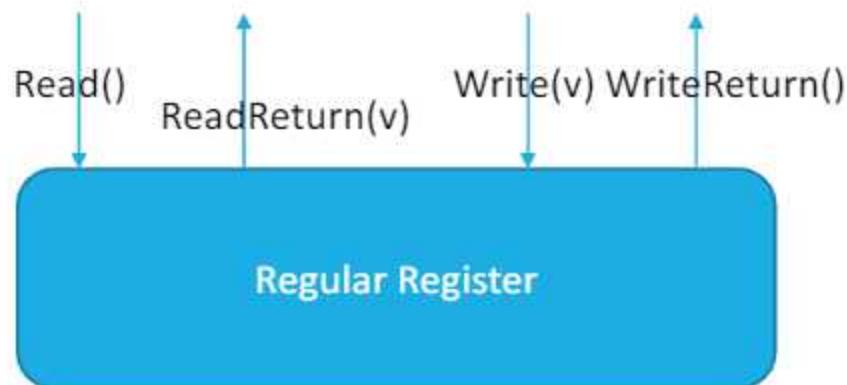
Indication:  $\langle onrr, \text{ReadReturn} \mid v \rangle$ : Completes a read operation on the register with return value  $v$ .

Indication:  $\langle onrr, \text{WriteReturn} \rangle$ : Completes a write operation on the register.

Properties:

**ONRR1: Termination:** If a correct process invokes an operation, then the operation eventually completes.

**ONRR2: Validity:** A read that is not concurrent with a write returns the last value written; a read that is concurrent with a write returns the last value written or the value concurrently written.



## Fail-Stop Algorithm: Read-One-Write-All Regular Register

---

**Fail-Stop Algorithm:** processes can crash but the crashes can be reliably detected by all the other processes

- failure model: crash
- perfect failure detector:
  - **Strong completeness**. The crash of a process is eventually detected by every correct process
  - **Strong accuracy**. No process is detected to have crashed until it has really crashed

## Read-One-Write-All RR: Communication Primitives

---

### Perfect point-point links:

1. **Reliable delivery** - Let  $p_i$  be any process that sends a message  $m$  to a process  $p_j$ . If neither  $p_i$  nor  $p_j$  crashes, then  $p_j$  eventually delivers  $m$ .
2. **No duplication** – No message is delivered by a process more than once.
3. **No creation** – If a message  $m$  is delivered by some process  $p_j$ , then  $m$  was previously sent to  $p_j$  by some process  $p_i$ .

### Best-Effort Broadcast (bebBroadcast):

1. **Best-effort validity** – For any two processes  $p_i$  and  $p_j$ . If  $p_i$  and  $p_j$  are correct, then every message broadcast by  $p_i$  is eventually delivered by  $p_j$ .
2. **No duplication** - No message is delivered more than once
3. **No creation** - If a message  $m$  is delivered by some process  $p_j$ , then  $m$  was previously broadcast by some process  $p_i$

## Fail-Stop Algorithm, Read-One-Write-All: (1,N) Regular Register

---

### Algorithm Idea:

- Each process stores a local copy of the register
- **Read-One**: each read operation returns the value stored in its local copy of the register
- **Write-All**: each write operation updates the value locally stored at each process the writer consider to have not crashed
- A write completes when the writer receives an ack from each process that has not crashed

# (1,N) regular register

---

## Algorithm 4.1: Read-One Write-All

---

Implements:

(1,  $N$ )-RegularRegister, **instance**  $onrr$ .

Uses:

BestEffortBroadcast, **instance**  $beb$ ;

PerfectPointToPointLinks, **instance**  $pl$ ;

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

upon event  $\langle onrr, Init \rangle$  do

$val := \perp$ ; *local value of register*  
*default 1 or 0*

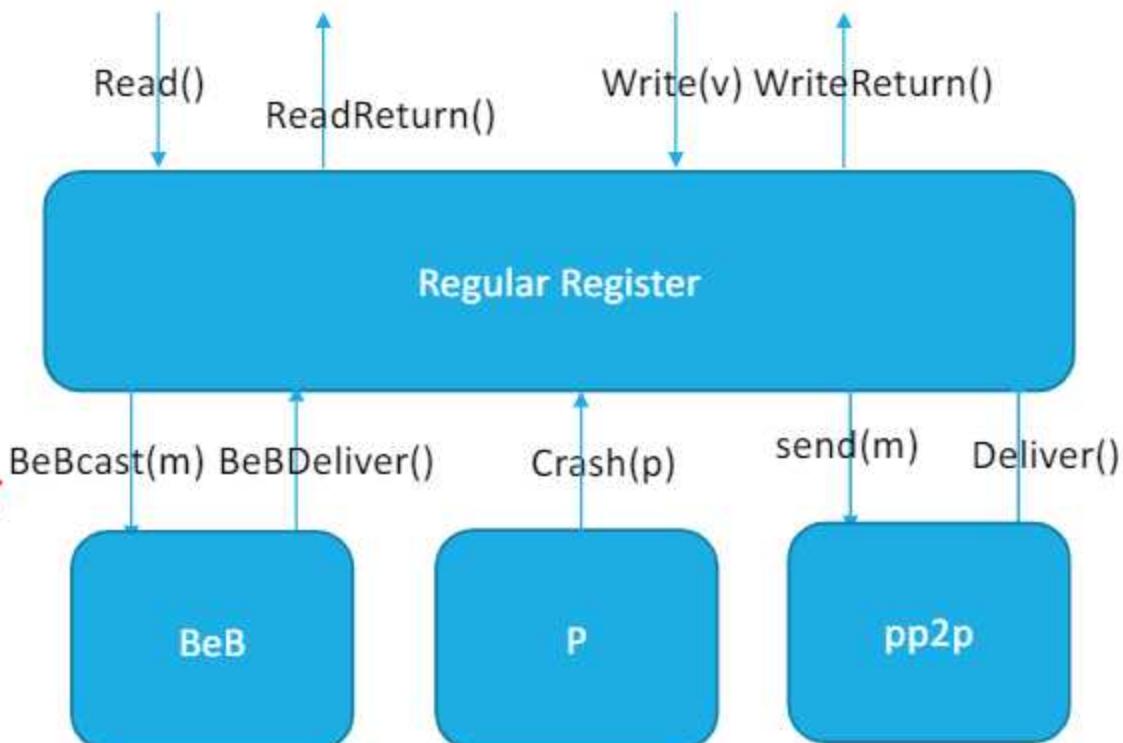
$correct := \Pi$ ;

$writeset := \emptyset$ ;

*set of processes that acknowledge*

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

$correct := correct \setminus \{p\}$ ;



# (1,N) regular register

upon event  $\langle onrr, Read \rangle$  do  
    trigger  $\langle onrr, ReadReturn \mid val \rangle$ ;

*simply read local  
    → value of R*

} Read() operation implementation

upon event  $\langle onrr, Write \mid v \rangle$  do  
    trigger  $\langle beb, Broadcast \mid [WRITE, v] \rangle$ ;

*→ write and communicate to all processes  
    correct*

upon event  $\langle beb, Deliver \mid q, [WRITE, v] \rangle$  do  
     $val := v;$   
    trigger  $\langle pl, Send \mid q, ACK \rangle$ ;

*→ confirm the update  
    to q*

upon event  $\langle pl, Deliver \mid p, ACK \rangle$  do  
     $writeset := writeset \cup \{p\}$ ;

*→ add p to processes that answer to update*

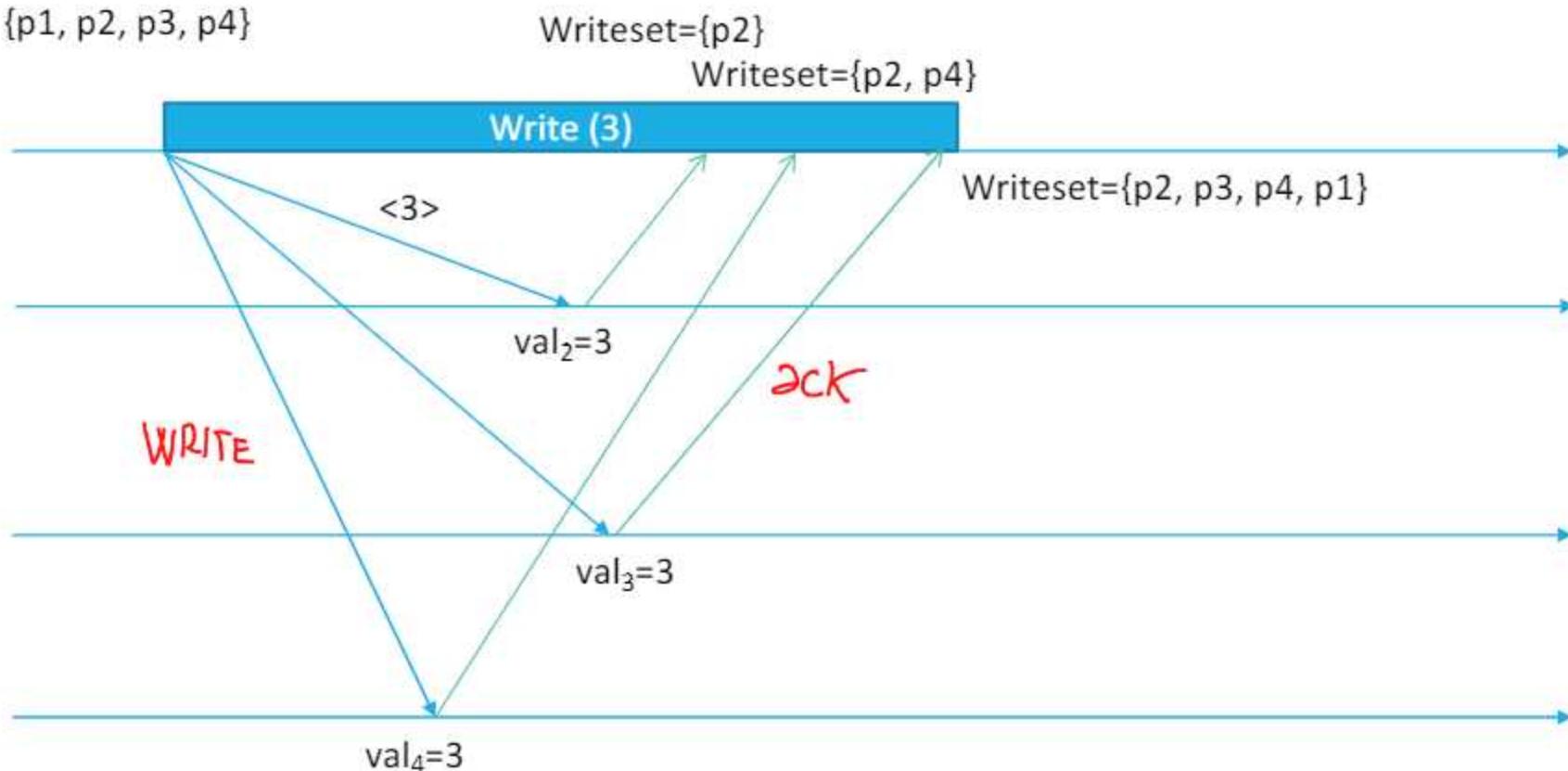
upon  $correct \subseteq writeset$  do  
     $writeset := \emptyset;$   
    trigger  $\langle onrr, WriteReturn \rangle$ ;

*→ all reply*

# Example

Correct = {p1, p2, p3, p4}

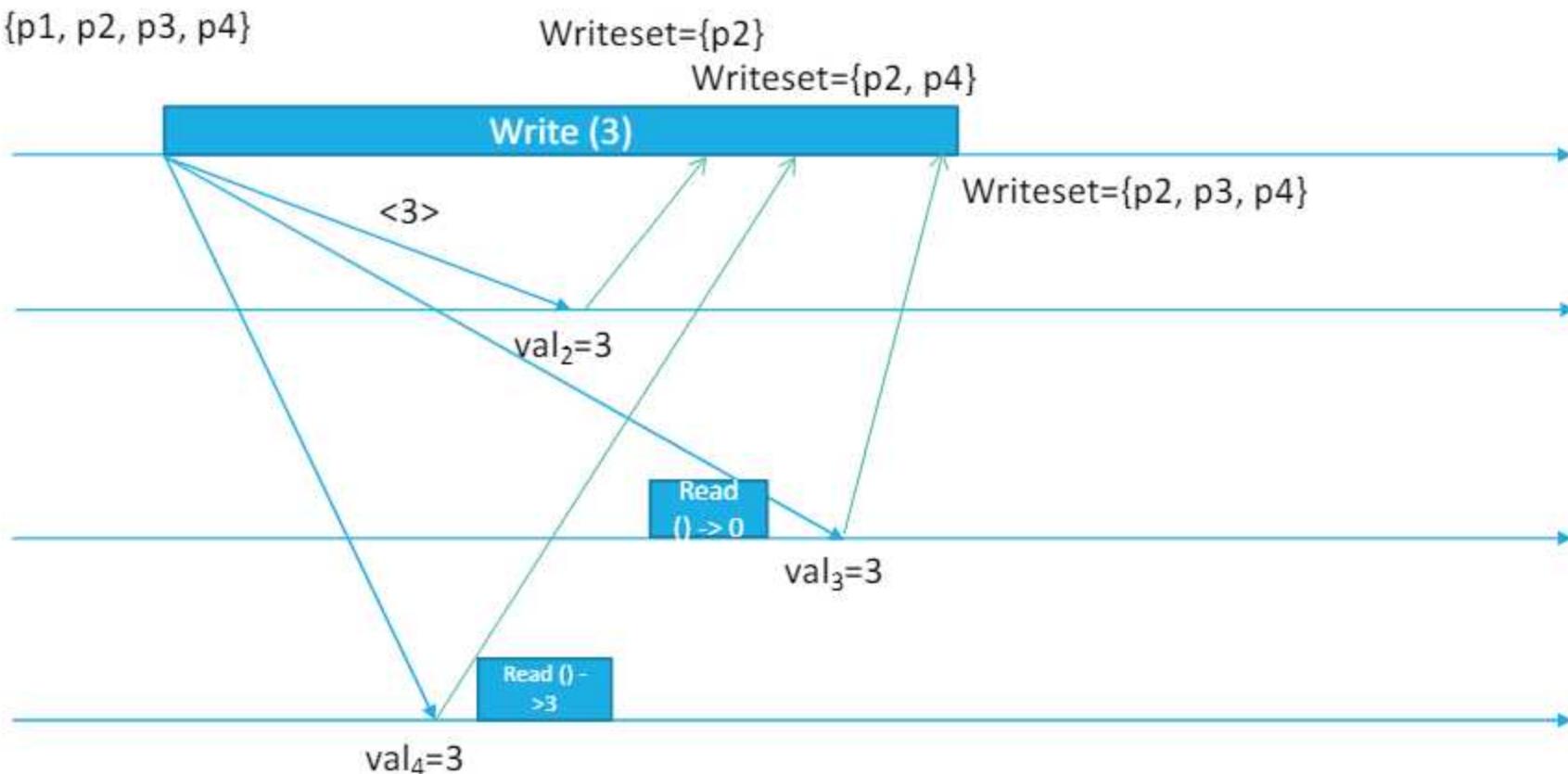
Val = 0



# Example

Correct = {p1, p2, p3, p4}

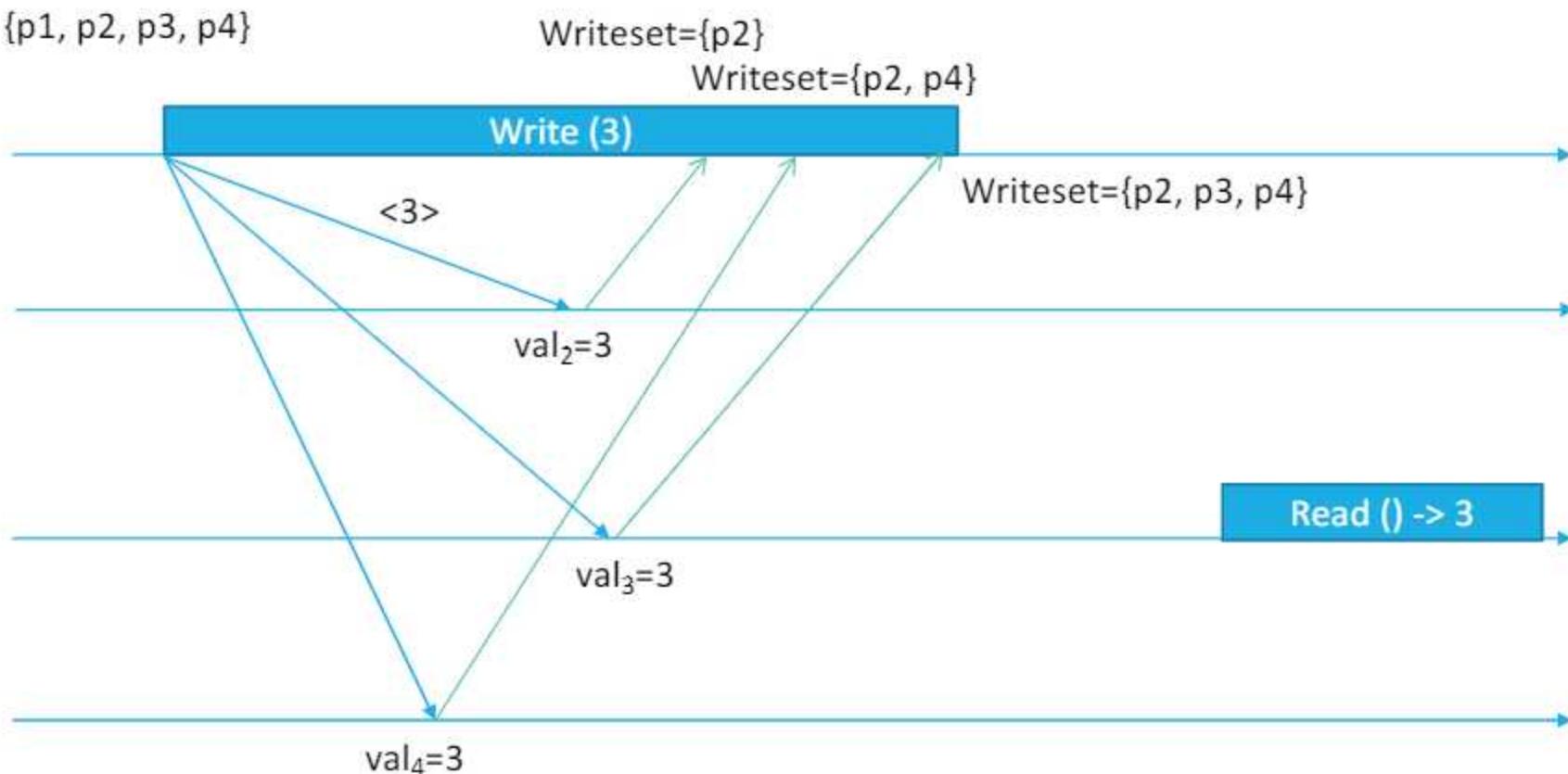
Val = 0



# Example

Correct = {p1, p2, p3, p4}

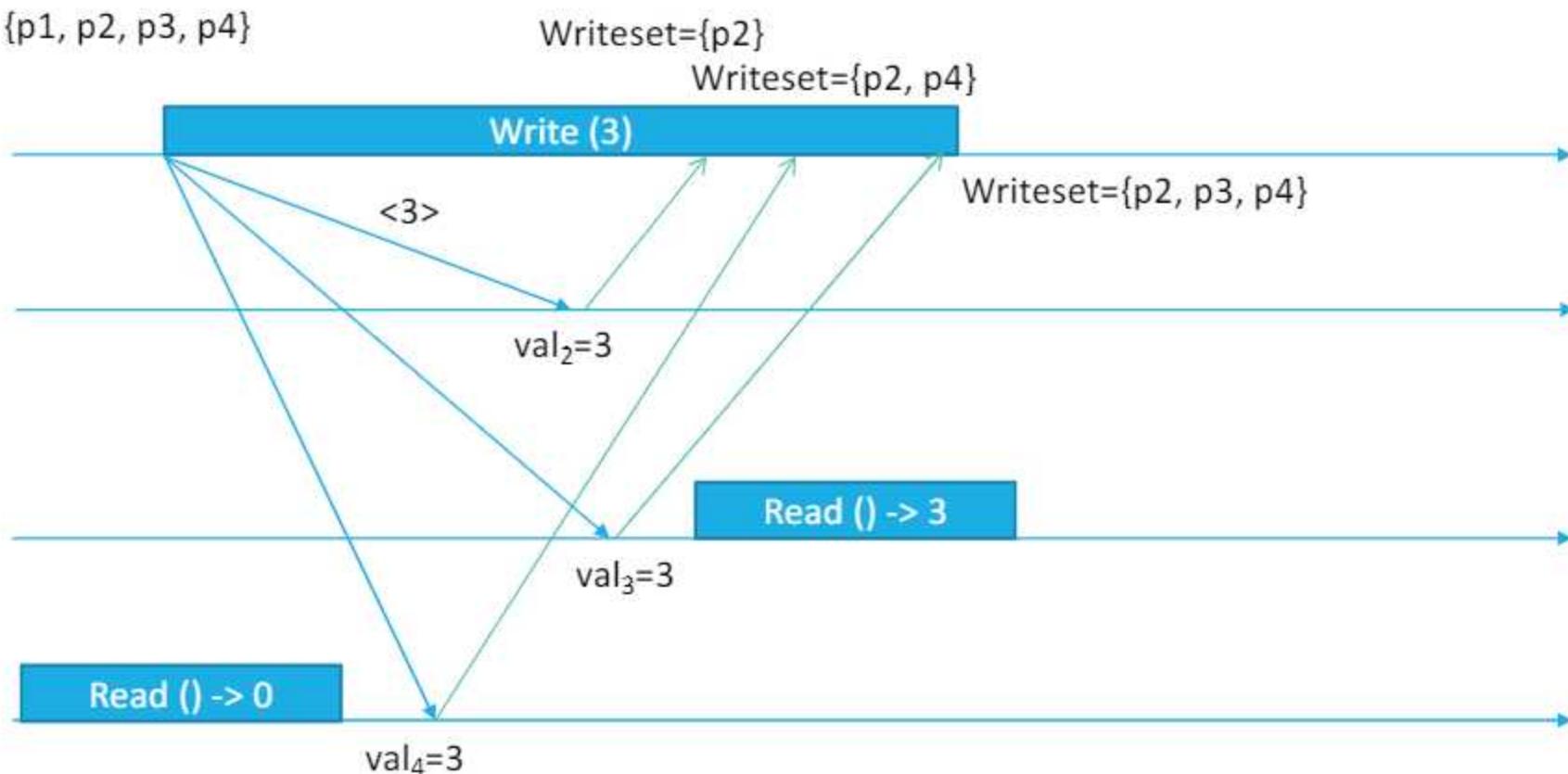
Val = 0



# Example

Correct = {p1, p2, p3, p4}

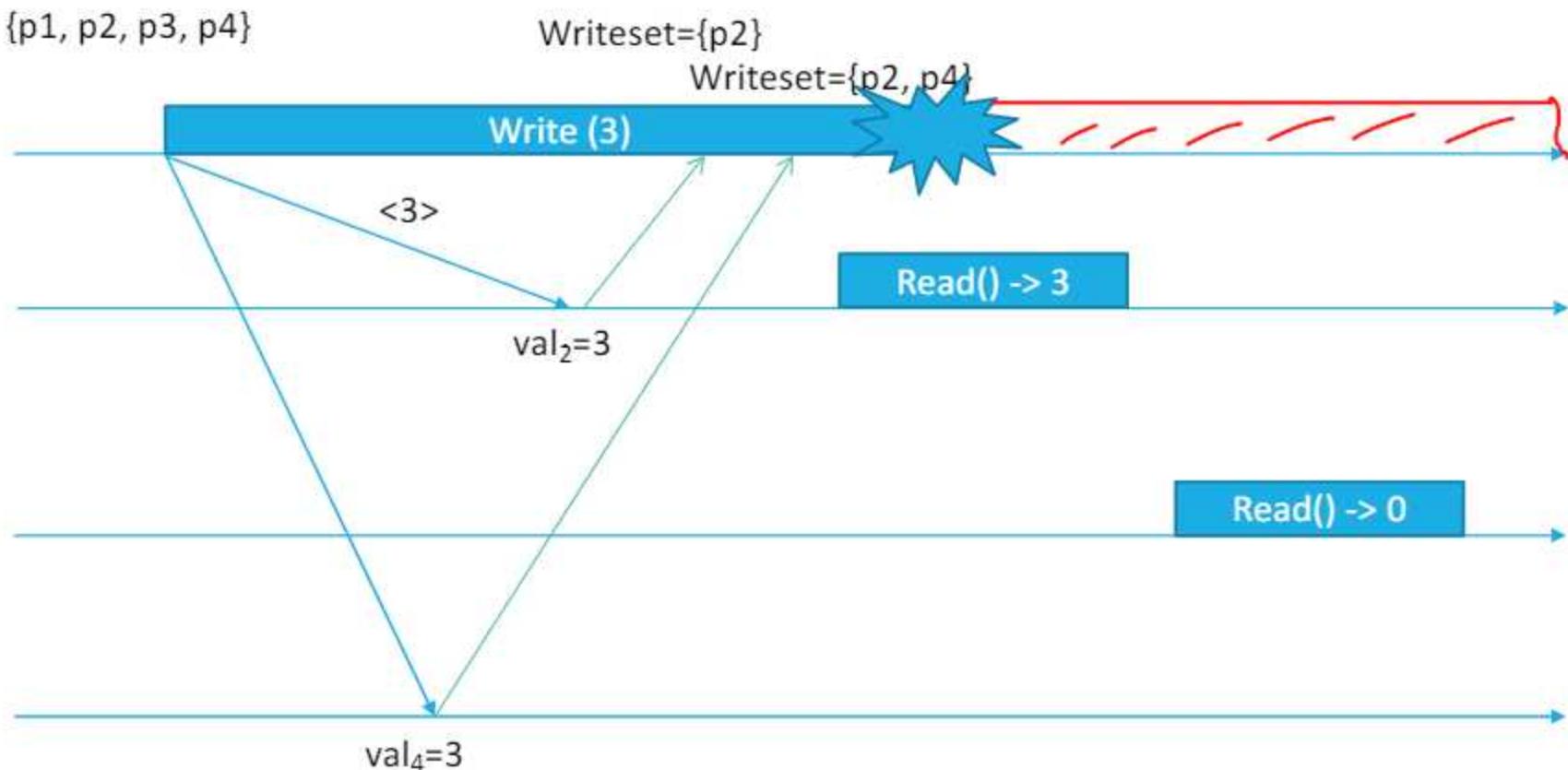
Val = 0



# Example

Correct = {p1, p2, p3, p4}

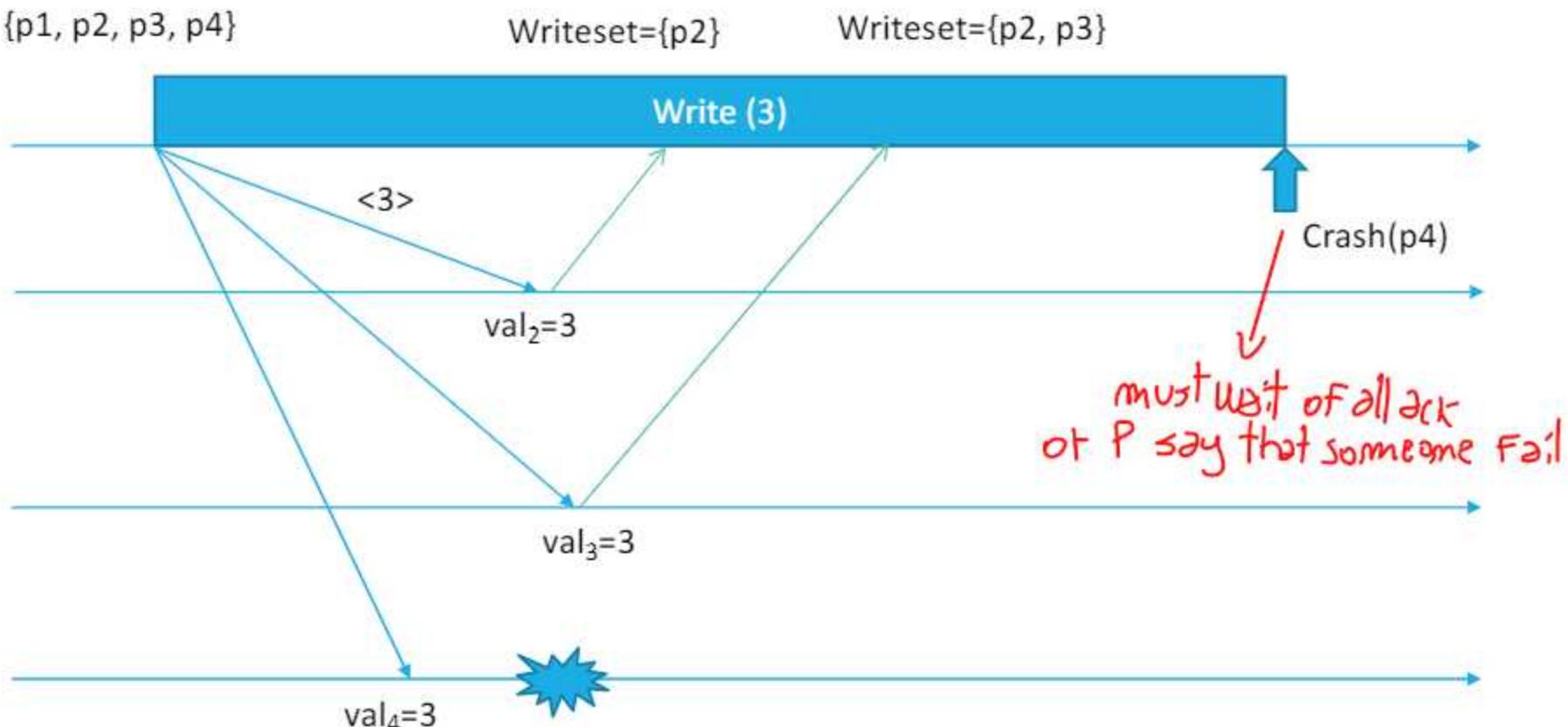
Val = 0



# Example

Correct = {p1, p2, p3, p4}

Val = 0



# (1,N) regular register

---

## Correctness:

### Termination –

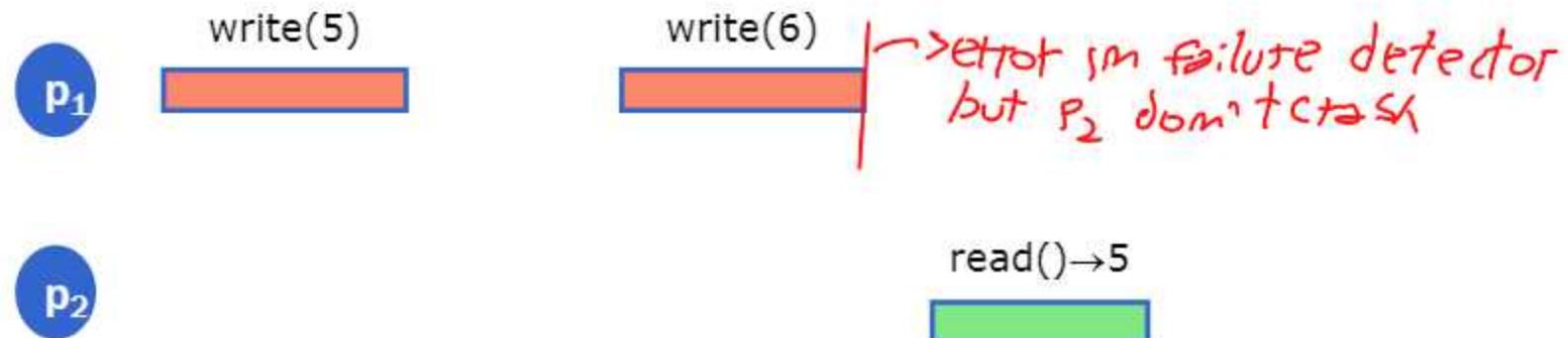
- read: trivial, it is local.
- write: from the properties of the communication primitives and from the completeness property of the perfect failure detector.
- **Validity** – Because of the *strong accuracy* property of the perfect failure detector, each write operation can complete only after all processes that do not crash have updated their local copy of the register. So, the two following cases can hold:
  - The read operation is not concurrent with the last write that has been invoked, the process will read the last value written
  - The read operation is concurrent with the last write. For the *no creation* property of the channels, the value returned is either the last value written or the one being written. This latter is concurrent with the read operation

## Performance:

- Write – At most  $2N$  messages.
- Read – 0 msg, it is local

# Read-One-Write-All RR: problem

- The algorithm does not ensure validity if the failure detector is not perfect. The following scenario could happen:



- P<sub>1</sub> invokes write(6) and then falsely suspects p<sub>2</sub>. Thus, p<sub>1</sub> completes the write operation without waiting for the ack of p<sub>2</sub>, i.e. without being sure that the value 6 has been written in the local copy of the register at p<sub>2</sub>

# Fail-silent algorithm: Majority Voting Regular Register

---

Fail-silent algorithm: "process crashes can never be reliably detected"

- Failure model: crash
- No perfect failure detector

*↪ make an assumption, use a majority*

Assumptions:

- N processes whose 1 writer and N readers
- A majority of correct processes

Communication Primitives:

- Perfect point-to-point link
- Best-effort broadcast

# Fail-silent algorithm: Majority Voting Regular Register

---

## IDEA:

- Each process locally stores a copy of the current value of the register
  - Each written value is univocally associated to a timestamp
  - The writer and the reader processes use a set of **witness processes**, to track the last value written
  - Quorum: the intersection of any two sets of **witness processes** is not empty
  - **“Majority Voting”**: each set is constituted by a majority of processes ↳ that always exists
- header select most recent one with timestamp

# (1,N) regular register

---

## Algorithm 4.2: Majority Voting Regular Register

---

Implements:

**(1, N)-RegularRegister**, instance *onrr*.

Uses:

**BestEffortBroadcast**, instance *beb*;

**PerfectPointToPointLinks**, instance *pl*.

No Perfect Failure Detector!

upon event  $\langle onrr, \text{Init} \rangle$  do

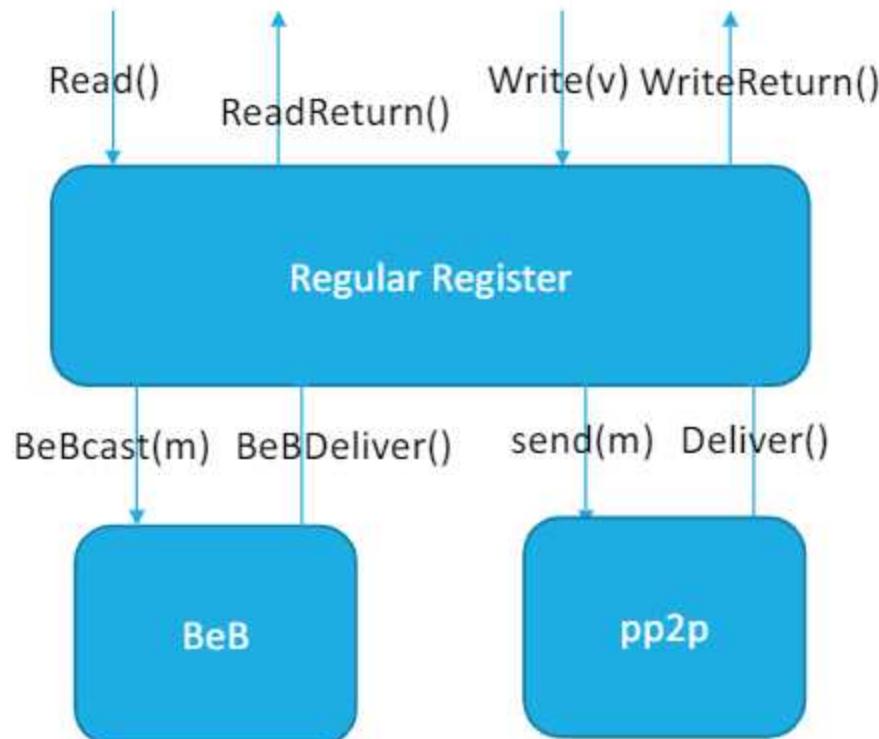
$(ts, val) := (0, \perp)$ ; (timestamp, value)

$wts := 0$ ; (write-timestamp)

$acks := 0$ ;

$rid := 0$ ; ↗ read identifier

$readlist := [\perp]^N$ ; ↗ array where store the  
tally provided



# (1, N) Regular Register: write() operation

upon event  $\langle onrr, Write \mid v \rangle$  do  
     $wts := wts + 1$ ;  $\rightsquigarrow$  increment every writer

$acks := 0$ ;

    trigger  $\langle beb, Broadcast \mid [WRITE, wts, v] \rangle$ ;

don't overwrite  
and fail

↑  
upon event  $\langle beb, Deliver \mid p, [WRITE, ts', v'] \rangle$  do  
    if  $ts' > ts$  then

check that is  
last write

$\rightsquigarrow$  value  
 $\rightsquigarrow$  timestamp

$(ts, val) := (ts', v')$ ;  
        trigger  $\langle pl, Send \mid p, [ACK, ts'] \rangle$ ;  $\rightsquigarrow$  send always ACK

upon event  $\langle pl, Deliver \mid q, [ACK, ts'] \rangle$  such that  $ts' = wts$  do

$acks := acks + 1$ ;

$\rightsquigarrow$  current write

    if  $acks > N/2$  then  $\rightsquigarrow$  we have a majority

$acks := 0$ ;

        trigger  $\langle onrr, WriteReturn \rangle$ ;

# (1,N) regular register: Read() operation

---

upon event  $\langle onrr, Read \rangle$  do

$rid := rid + 1;$

$readlist := [\perp]^N;$

    trigger  $\langle beb, Broadcast \mid [READ, rid] \rangle;$

$\rightarrow$  send read to everybody

upon event  $\langle beb, Deliver \mid p, [READ, r] \rangle$  do

    trigger  $\langle pl, Send \mid p, [VALUE, r, ts, val] \rangle;$

$\rightarrow$  reply to current read

upon event  $\langle pl, Deliver \mid q, [VALUE, r, ts', v'] \rangle$  such that  $r = rid$  do

$readlist[q] := (ts', v');$

    if  $\#(readlist) > N/2$  then  $\rightarrow$  count number of reply

$v := \text{highestval}(readlist); \rightarrow$  highest timestamp pick the value

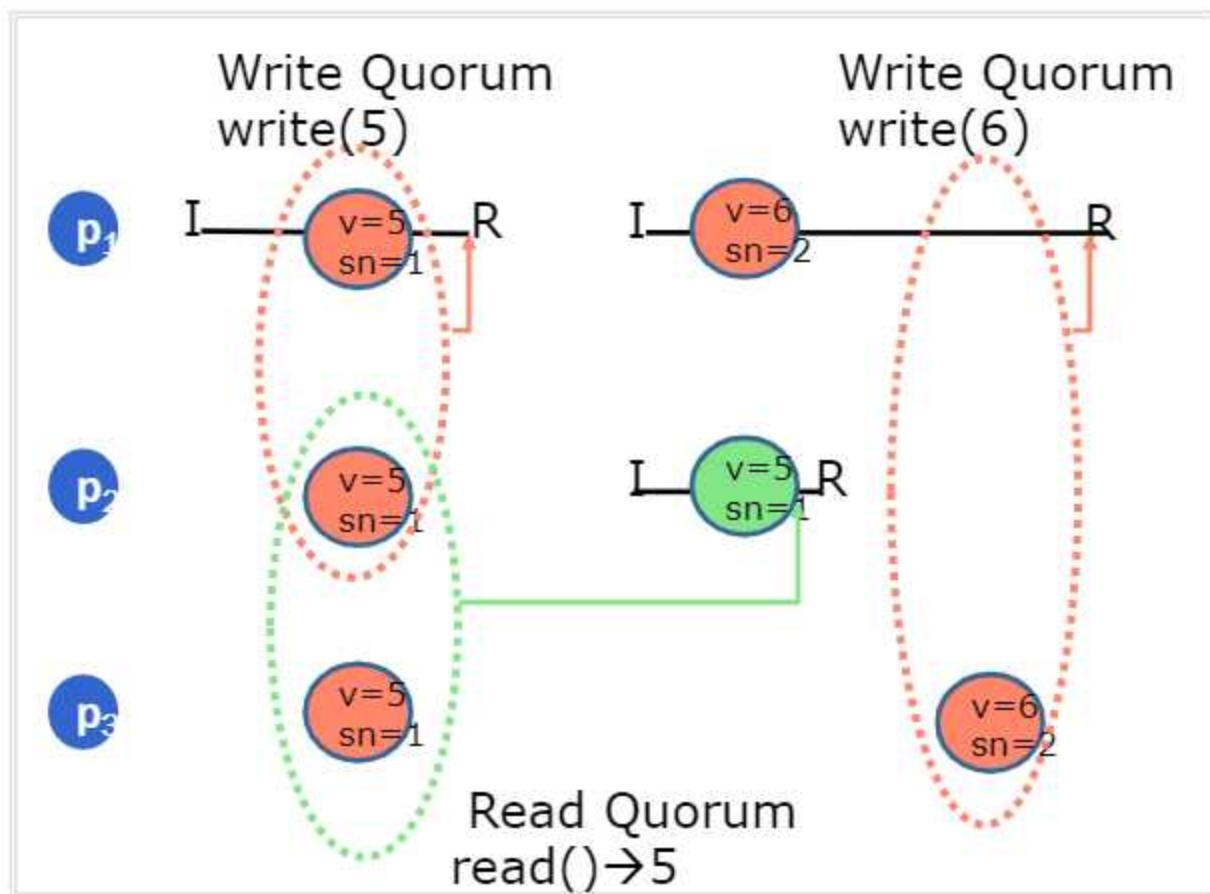
$readlist := [\perp]^N;$

    trigger  $\langle onrr, ReadReturn \mid v \rangle;$

$\rightarrow$  return that value

→ work for the intersection properties  
of majority

# Functioning Scenario

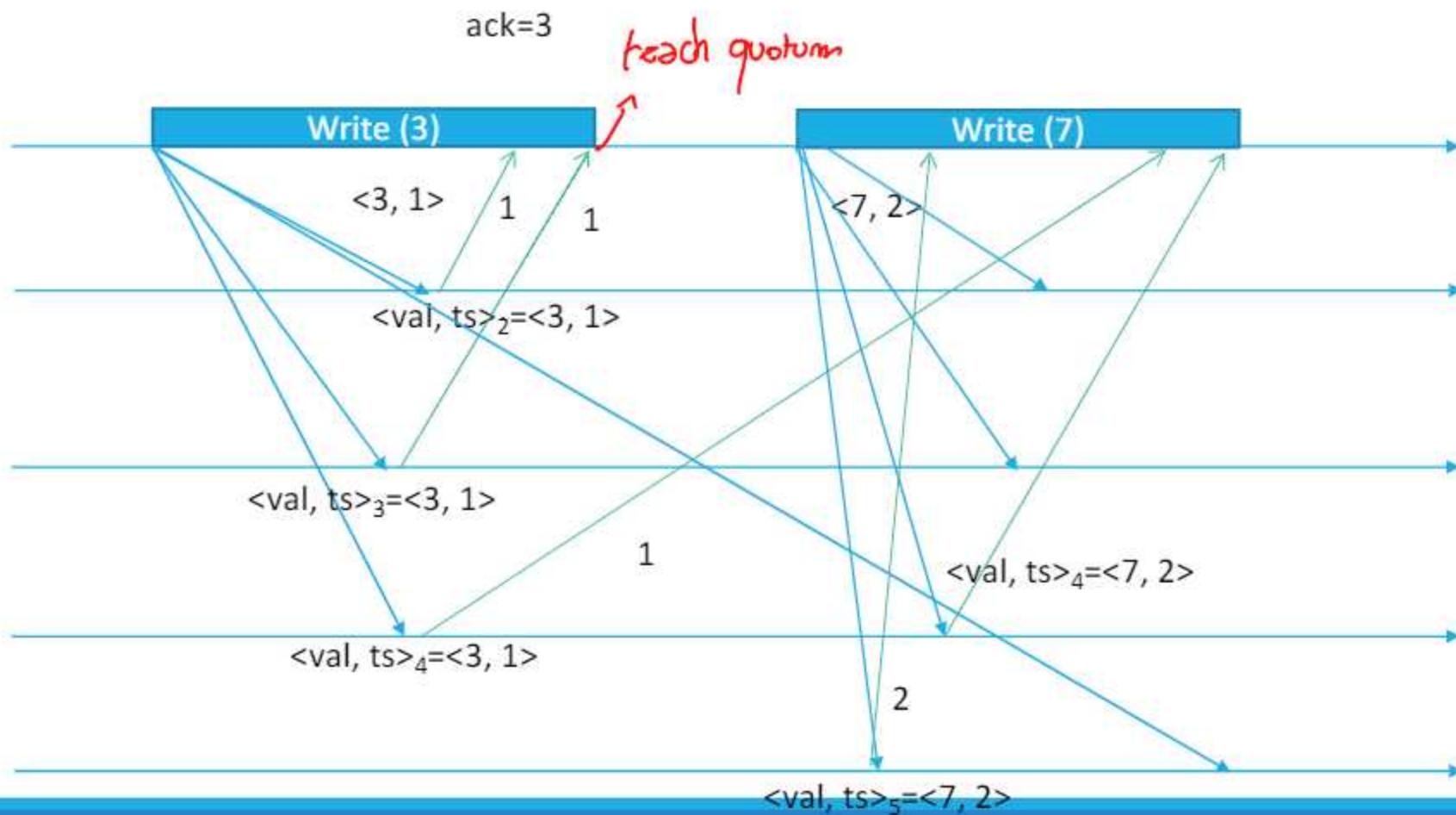


$(1,3)$  regular register

- $\Pi = \{p_1, p_2, p_3\}$
- $p_1$  is the writer
- I = invocation
- R = response

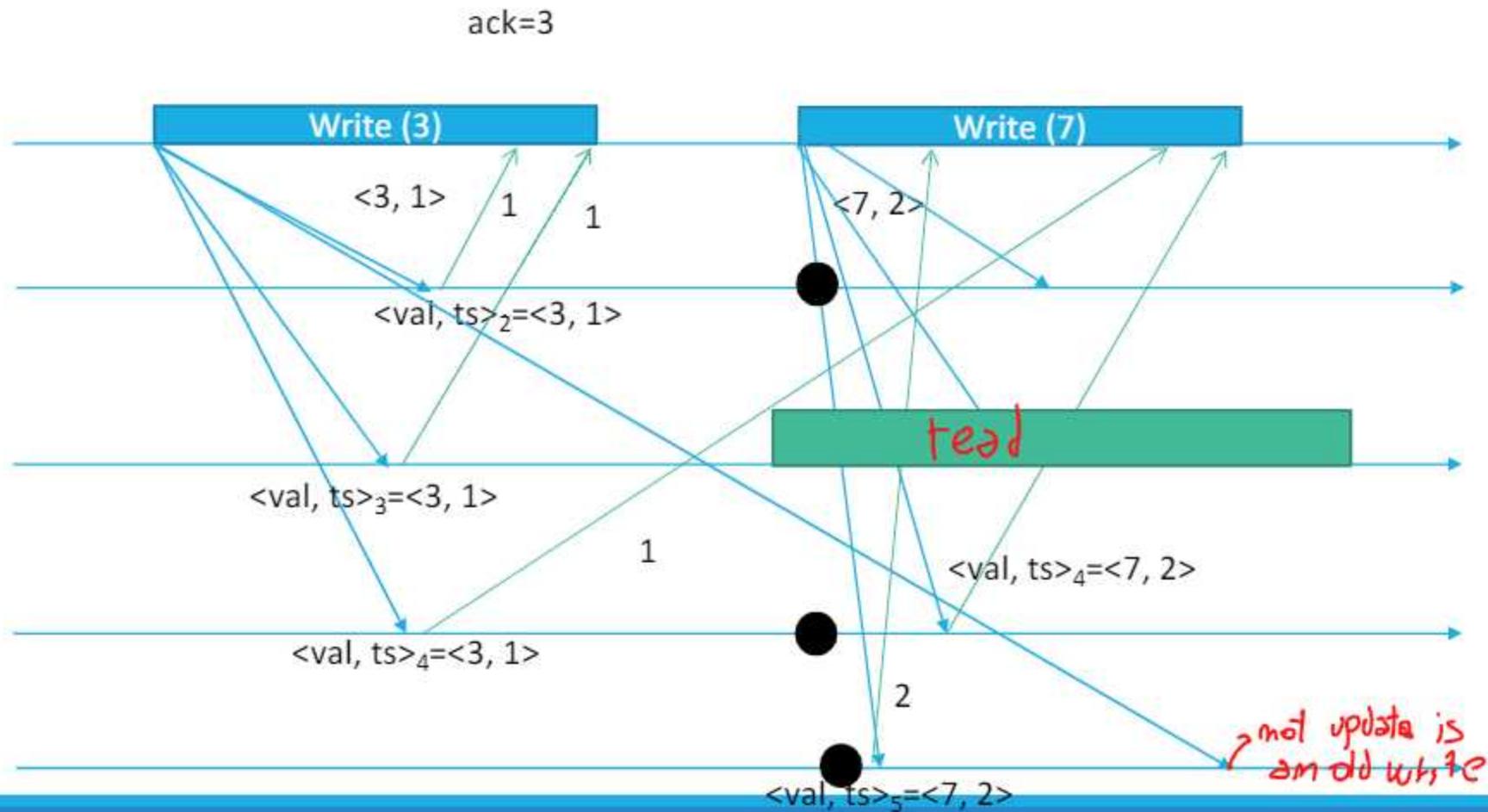
# Example

Val = 0  
Ts = 0



# Example

Val = 0  
Ts = 0



# (1,N) regular register

---

## Correctness:

- *Termination* – from the properties of the communication primitives and the assumption of a majority of correct processes
- *Validity* – from the intersection property of the quorums

## Performance:

- *Write* –at most  $2N$  messages
- *Read* - at most  $2N$  messages

# Atomic Register Implementation

---

# (1,N) Atomic Register: Interface

---

## Module 4.2: Interface and properties of a (1, N) atomic register

---

Module:

Name: (1, N)-AtomicRegister, instance *onar*.

Events:

Request:  $\langle onar, \text{Read} \rangle$ : Invokes a read operation on the register.

Request:  $\langle onar, \text{Write} \mid v \rangle$ : Invokes a write operation with value *v* on the register.

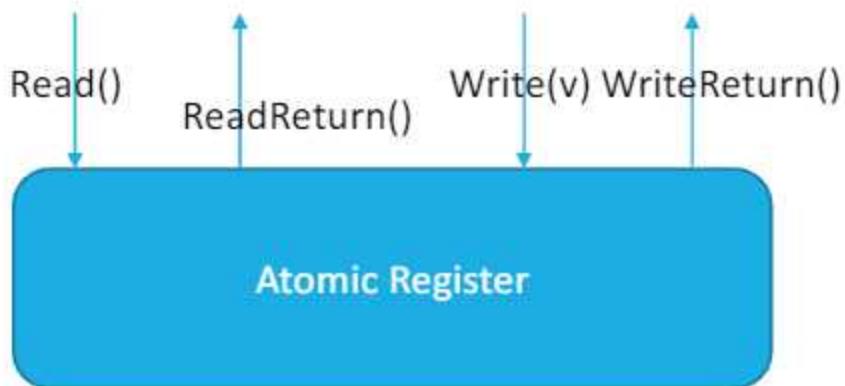
Indication:  $\langle onar, \text{ReadReturn} \mid v \rangle$ : Completes a read operation on the register with return value *v*.

Indication:  $\langle onar, \text{WriteReturn} \rangle$ : Completes a write operation on the register.

Properties:

**ONARI-ONAR2:** Same as properties ONRR1-ONRR2 of a (1, N) regular register (Module 4.1). (validity and termination)

**ONAR3: Ordering:** If a read returns a value *v* and a subsequent read returns a value *w*, then the write of *w* does not precede the write of *v*.



---

From (1,N) Regular to (1,N) Atomic Register

# (1,N) Atomic Register

The algorithm consists of two phases

→ how to manage ordering

PHASE 1. We use a (1,N) regular register to build a (1,1) atomic register

PHASE 2. We use a set of (1,1) atomic registers to build a (1,N) atomic register

NOTATION:

↑  
read execute at regular register level  
, at atomic register level

↳ how to assure that the ordering is preserved  
globally to all processes

Hereafter, rr and ra, will be sometimes used to respectively denote regular register and atomic register

only one process is able to write  
and only one process is able to read

(1,N) Regular Register  $\rightarrow$  (1,1) Atomic Register: **PHASE 1**

### IDEA:

- $p_1$  is the writer and  $p_2$  is the reader of the (1,1) atomic register, we aim to implement
- We use a (1,N) regular register where  $p_1$  is the writer and  $p_2$  is the reader
- Each write operation on the atomic register writes the pair (value, timestamp) into the underlying regular register
- The reader tracks the timestamp of previously read values to avoid to read something old

identify what  
i am returning now is  
oldest or newest on what  
i return in the past

# (1,N)Regular Register → (1,1)Atomic Register

Algorithm 4.3: From (1, N) Regular to (1, 1) Atomic Registers

Implements:

(1, 1)-AtomicRegister, instance *ooar*.

Uses:

(1, N)-RegularRegister, instance *onrr*.

upon event  $\langle ooar, \text{Init} \rangle$  do  
 $(ts, val) := (0, \perp)$ ; *↪ store not only value but also timestamp*  
 $wts := 0$ ; *↪ on writer side i keep track on sequence number*

upon event  $\langle ooar, \text{Write} \mid v \rangle$  do  
 $wts := wts + 1$ ; *↪ increment timestamp*  
trigger  $\langle onrr, \text{Write} \mid (wts, v) \rangle$ ;

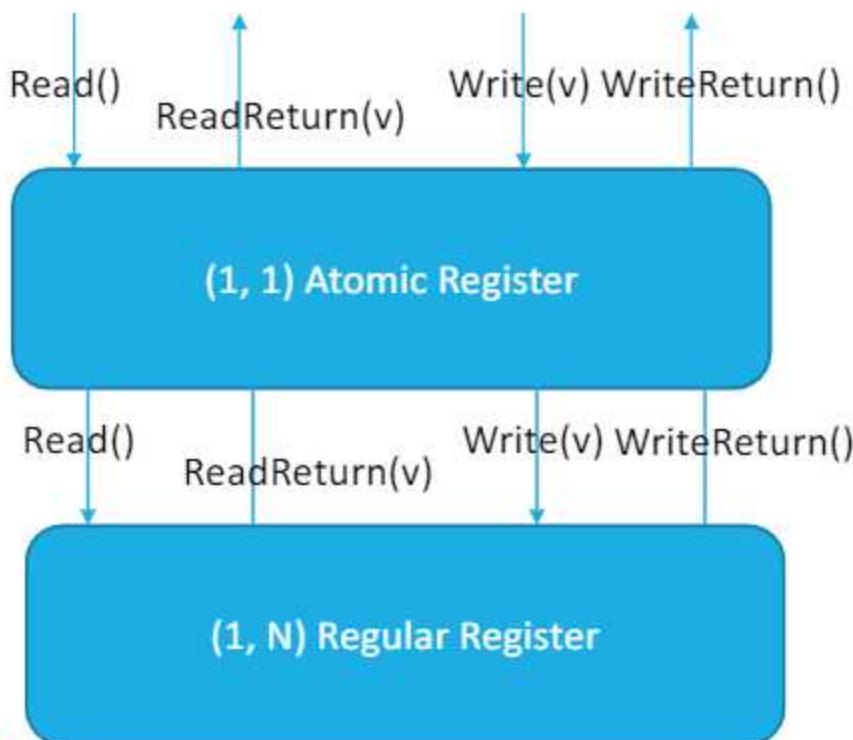
upon event  $\langle onrr, \text{WriteReturn} \rangle$  do  
trigger  $\langle ooar, \text{WriteReturn} \rangle$ ;

upon event  $\langle ooar, \text{Read} \rangle$  do  
trigger  $\langle onrr, \text{Read} \rangle$ ;

upon event  $\langle onrr, \text{ReadReturn} \mid (ts', v') \rangle$  do  
if  $ts' > ts$  then  
 $(ts, val) := (ts', v')$ ;  
trigger  $\langle ooar, \text{ReadReturn} \mid val \rangle$ ;

updated  
my local value

*↪ just check timestamp*



*if is greater it mean that  
the RR is been updated by writer;  
if is smaller either there is a concurrent  
or not match ordering*

## $(1, N)$ Regular Register $\rightarrow$ $(1, 1)$ Atomic Register

---

### Correctness:

- **Termination** – from the termination property of the regular register
- **Validity** – from the validity property of the regular register
- **Ordering** – from the validity property and from the fact that the read tracks the last value read and its timestamp. A read operation always returns a value with a timestamp greater or equal to the one of the previously read value

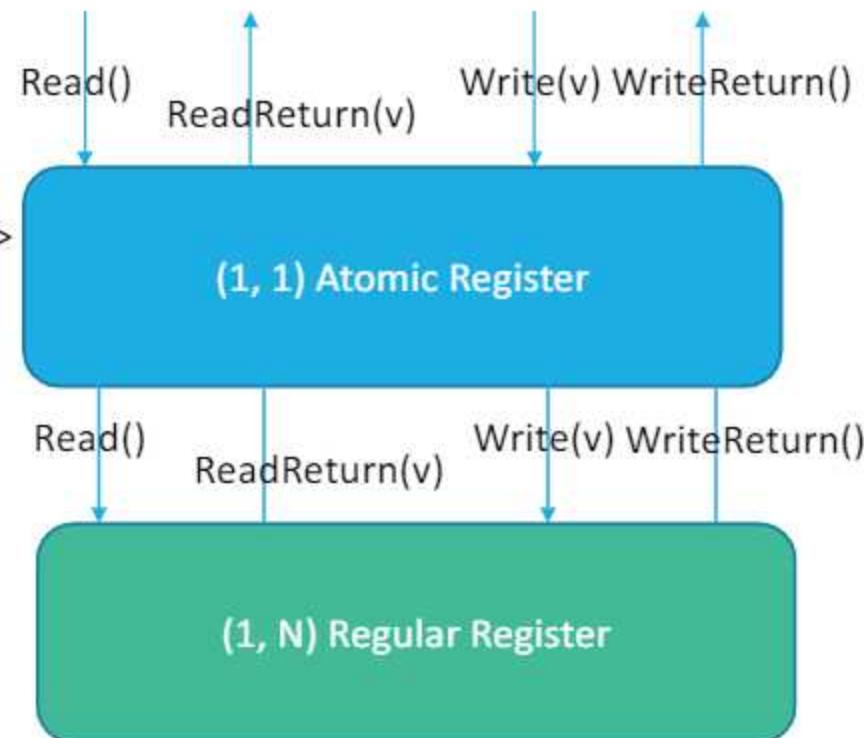
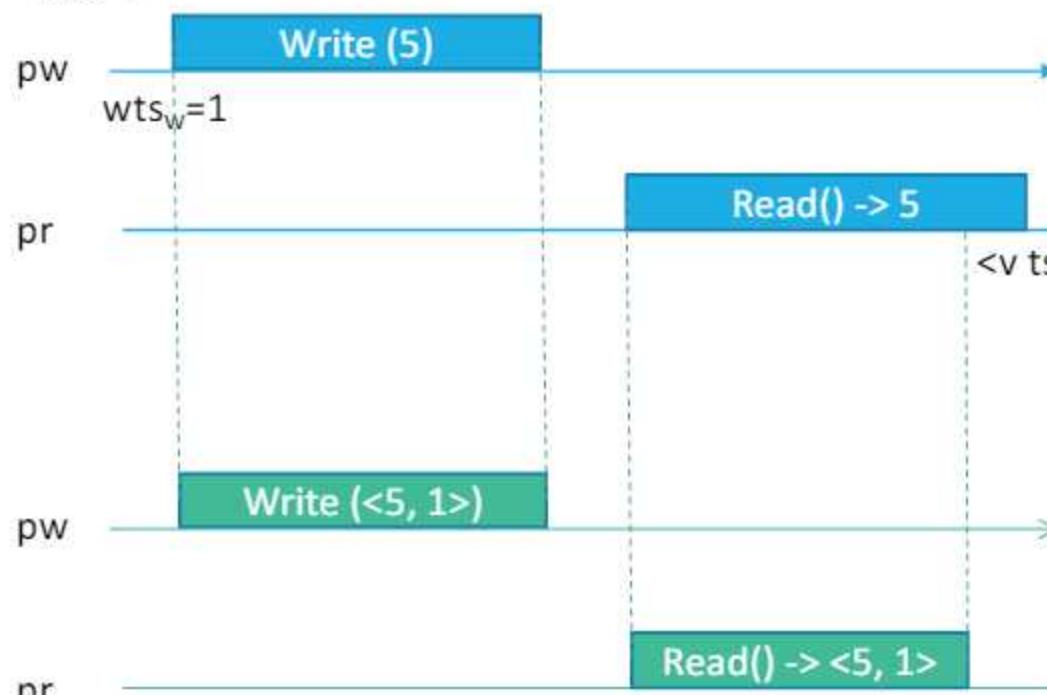
*(↳ single reader that use timestamp)*

### Performance:

- **Write** – Each write operation requests a write on a  $(1, N)$  regular register
- **Read** - Each read operation requests a read on a  $(1, N)$  regular register
- **NOTE:** no more msg w.r.t.  $(1, 1)$  regular register implementation

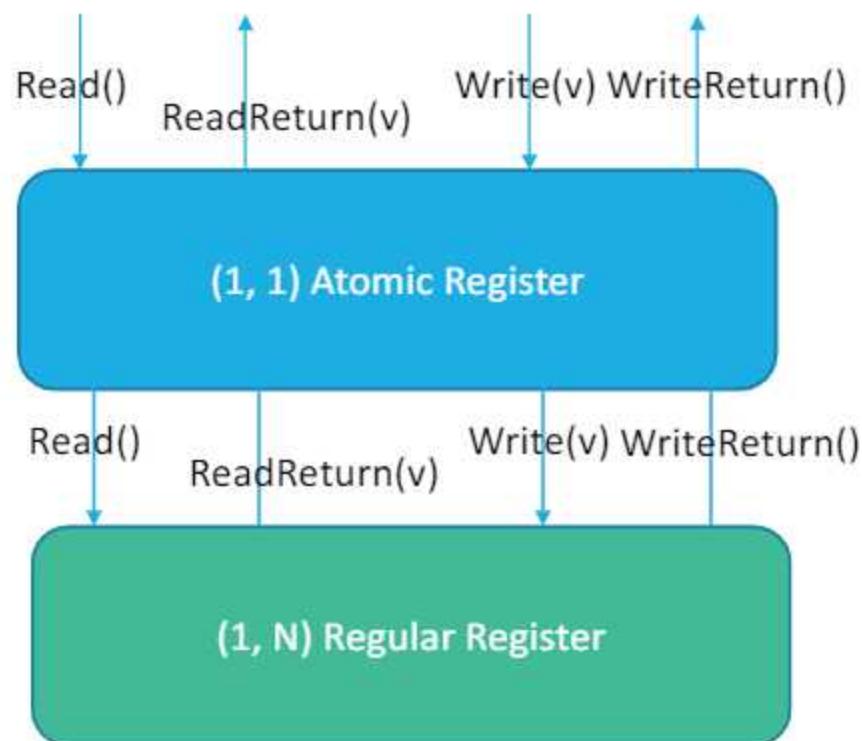
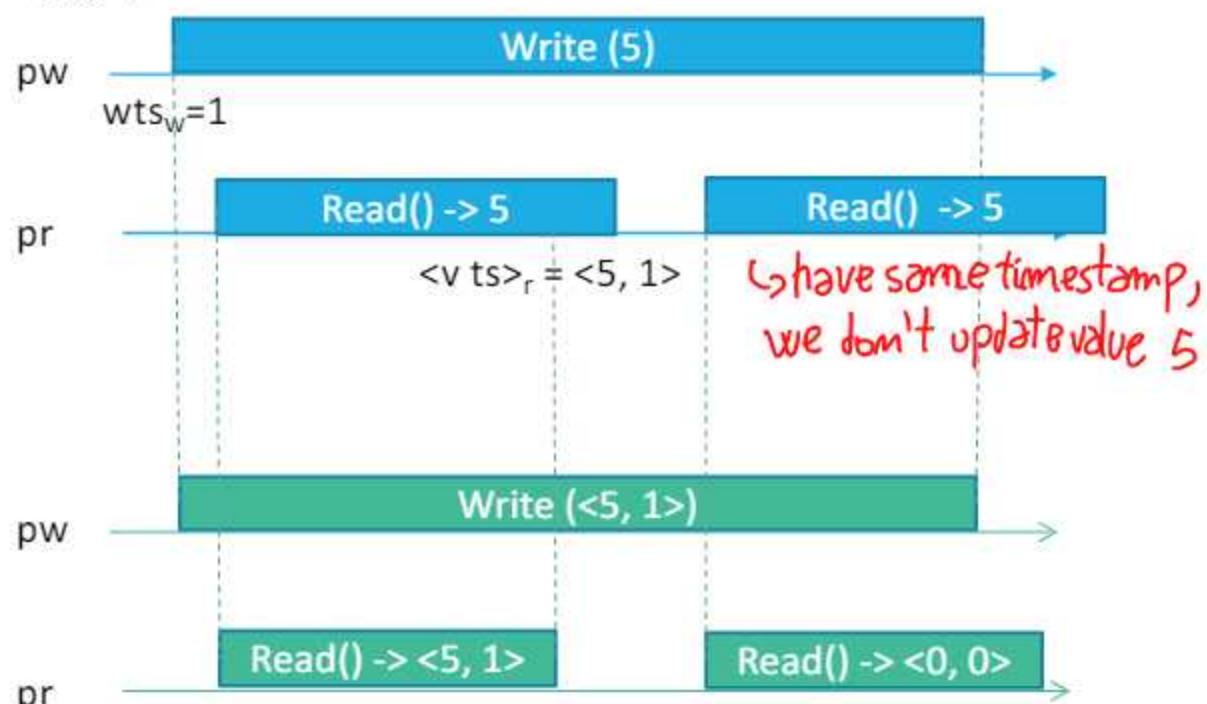
$\langle v \text{ ts} \rangle = \langle 0, 0 \rangle$

wts=0



$\langle v \text{ ts} \rangle = \langle 0, 0 \rangle$

wts=0



# (1,1)Atomic Register $\rightarrow$ (1,N) Atomic Register: Phase 2

## Algorithm 4.4: From (1, 1) Atomic to (1, N) Atomic Registers

Implements:

(1, N)-AtomicRegister, instance *onar*.

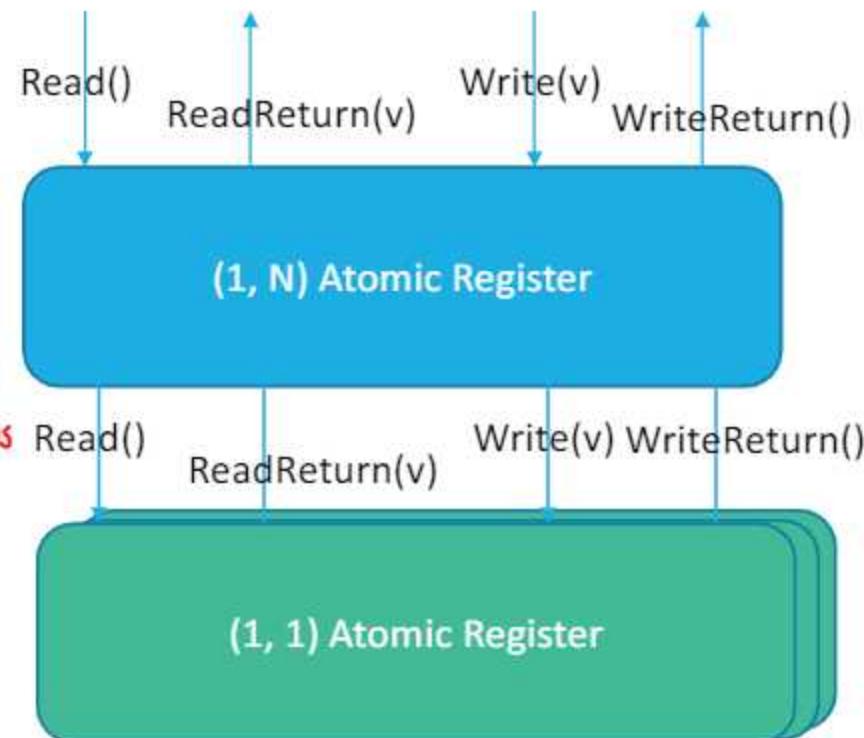
Uses:

(1, 1)-AtomicRegister (multiple instances).

```
upon event < onar, Init > do
    ts := 0; not timestamp at wr. side
    acks := 0;
    writing := FALSE;
    readval :=  $\perp$ ; no last value read
    readlist :=  $[\perp]^N$ ; store the value written by others
    forall  $q \in \Pi, r \in \Pi$  do
```

Initialize a new instance *oar.q.r* of (1, 1)-AtomicRegister  
with writer *r* and reader *q*;

*Every cell of matrix can be write by one  
process and read by one process*



## (1,1)Atomic Register $\rightarrow$ (1,N) Atomic Register: Phase 2

---

```
upon event < onar, Write | v > do
  ts := ts + 1;
  writing := TRUE; now is staking over the rows
  forall q in Π do
    trigger < ooar.q.self, Write | (ts, v) >; move register over i am writer and also
the reader, making q change reader
upon event < ooar.q.self, WriteReturn > do
  acks := acks + 1; count one more register is updated
  if acks = N then
    acks := 0;
    if writing = TRUE then if i am writing
      trigger < onar, WriteReturn >;
      writing := FALSE;
    else
      trigger < onar, ReadReturn | readval >;
```

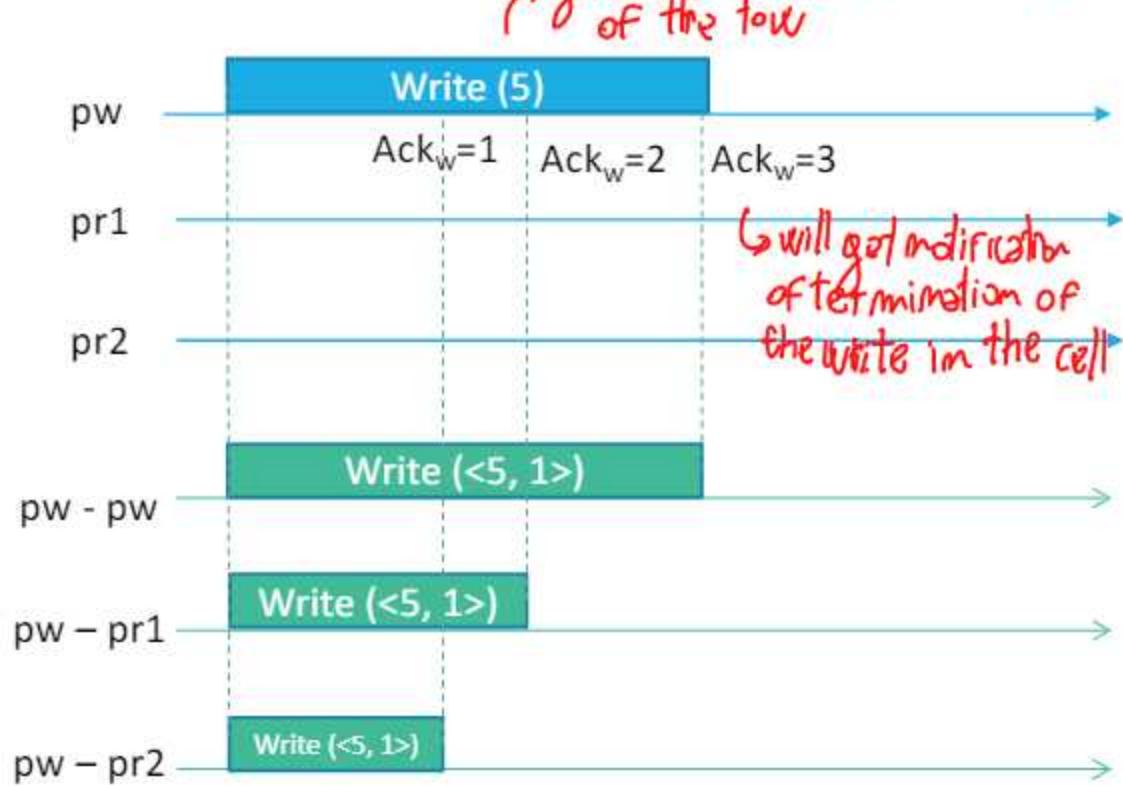
## (1,1)Atomic Register $\rightarrow$ (1,N) Atomic Register: Phase 2

---

```
upon event < onar, Read > do
  forall  $r \in \Pi$  do
    trigger < ooar.self.r, Read >; → scanning the matrix and read all the column

upon event < ooar.self.r, ReadReturn |  $(ts', v')$  > do
  readlist[ $r$ ] :=  $(ts', v')$ ; → for every item
  if #(readlist) =  $N$  then
     $(maxts, readval) := \text{highest}(readlist)$ ; → deterministic choice
    readlist :=  $[\perp]^N$ ;
    forall  $q \in \Pi$  do
      trigger < ooar.q.self, Write |  $(maxts, readval)$  >; → need to updated the new value
```

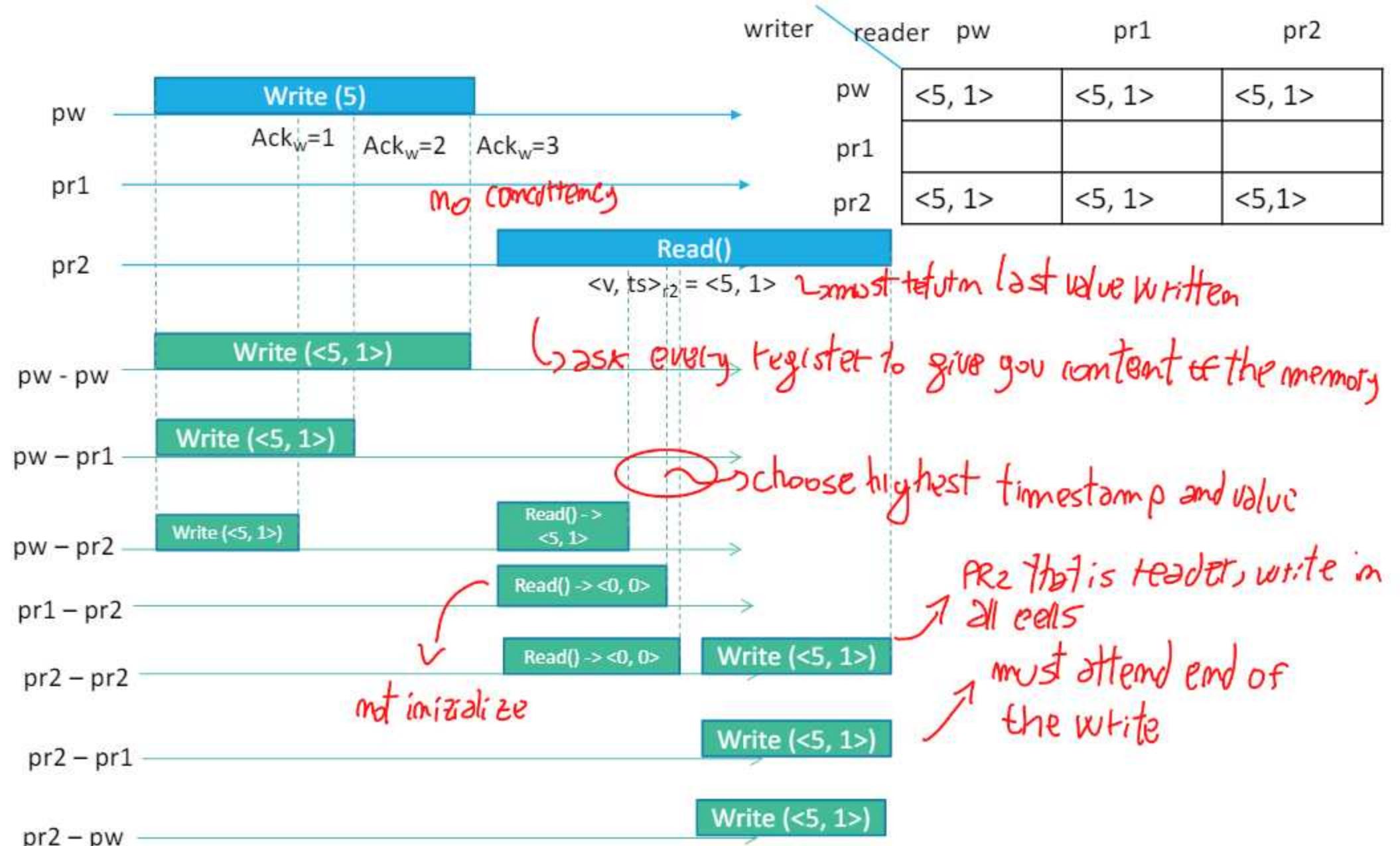
generate one write in every element  
of the row

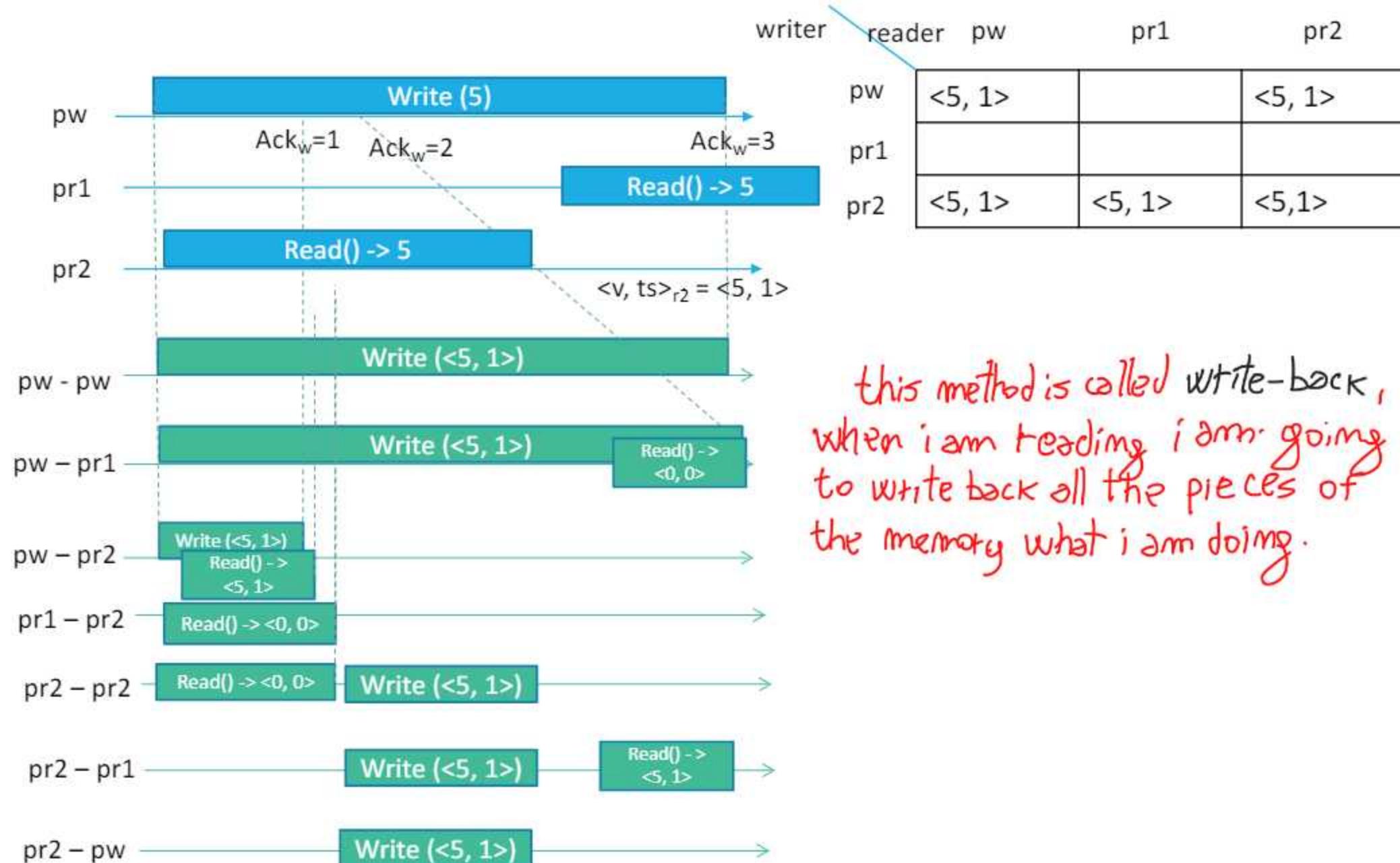


	reader	writer		
pw			pw	pr1
pr1				
pr2				pr2

Cell contents:

<5, 1>	<5, 1>	<5, 1>





this method is called write-back, when i am reading i am going to write back all the pieces of the memory what i am doing.

## (1,1) Atomic Register $\rightarrow$ (1,N) Atomic Register

---

### Correctness:

Termination – from the termination of the (1,1) atomic register

Validity – from the validity of the (1,1) atomic register.

Ordering - Consider a write operation  $w_1$  which writes value  $v_1$  with timestamp  $s_1$ . Let  $w_2$  be a write which precedes  $w_1$ . Let  $v_2$  and  $s_2$  ( $s_1 < s_2$ ) be the value and the timestamp corresponding to  $w_2$ .

Let assume that a read returns  $v_2$ : by the algorithm, for each  $j$  in  $[1:N]$ ,  $p_i$  has written  $(s_2, v_2)$  in  $readers[r; i; j]$ .

For the ordering property of the underlying (1,1) atomic registers, each successive read will return a value with timestamp greater or equal to  $s_2$ . Then  $s_1$  cannot be returned.

### Performance:

- Write – each write operation on a (1,N) atomic register requests N write operations on the (1,1) atomic registers.
- Read – Each read operation on a (1,N) atomic register requests to read N (1,1) atomic registers and to write N (1,1) atomic registers.

---

→ we don't use a shared memory, but use message passing

### (1,N) Atomic Register: Fail-Stop Algorithm

## Read-Impose Write-All Algorithm (1,N) Atomic Register

*synchronous with a perfect failure detector*

The algorithm is a modified version of the Read-One Write-All (1,N) Regular Register

IDEA: “the read operation writes”

The algorithm is called “Read-Impose Write-All” because a read operation imposes to all correct processes to update their local copy of the register with the value read, unless they store a more recent value

## Read-Impose Write-All (1,N) Atomic Register

### Algorithm 4.5: Read-Impose Write-All

Implements:

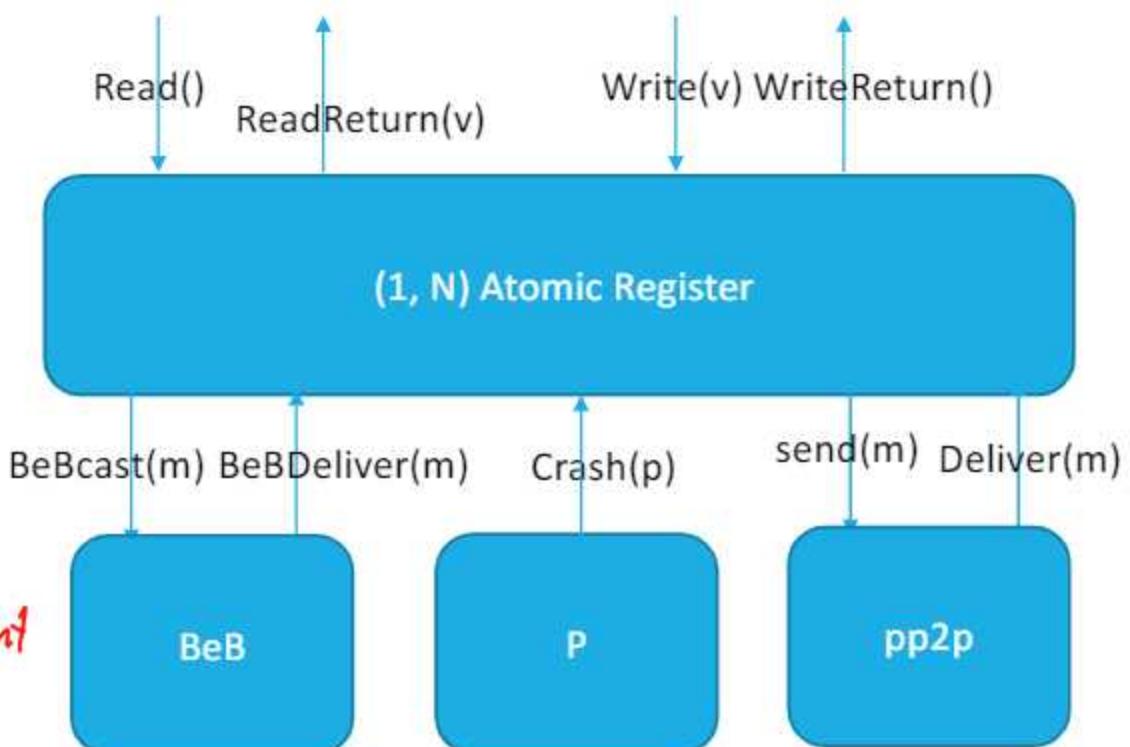
$(1, N)$ -AtomicRegister, **instance onar**.

Uses:

BestEffortBroadcast, **instance beb**;  
PerfectPointToPointLinks, **instance pl**;  
PerfectFailureDetector, **instance P**.

```
upon event < onar, Init > do
  (ts, val) := (0, ⊥);
  correct := Π;
  writeset := ∅;
  readval := ⊥;
  reading := FALSE;
  ↗ need timestamp for manage ordering
  ↗ keep track of acknowledgement
```

```
upon event < P, Crash | p > do
  correct := correct \ {p};
```



# Read-Impose Write-All (1,N) Atomic Register

```
upon event < onar, Read > do
    reading := TRUE;
    store locally the value
    readval := val;
    trigger < beb, Broadcast | [WRITE, ts, val] >;
```

Read() operation  
Implementation

```
upon event < onar, Write | v > do
    trigger < beb, Broadcast | [WRITE, ts + 1, v] >;
```

```
upon event < beb, Deliver | p, [WRITE, ts', v'] > do
    if  $ts' > ts$  then
        check if timestamp p is greater or local
         $(ts, val) := (ts', v');$ 
    trigger < pl, Send | p, [ACK] >;
```

Write() operation  
Implementation

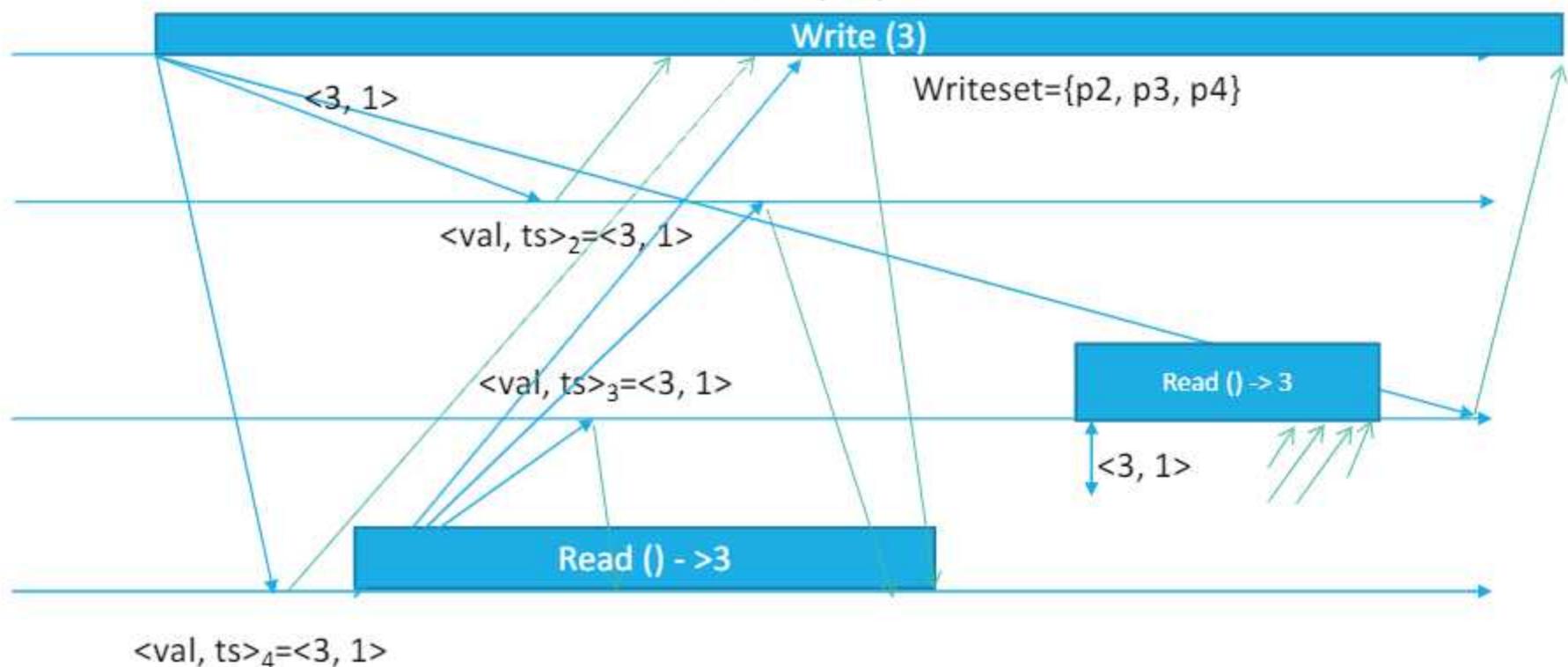
```
upon event < pl, Deliver | p, [ACK] > then
    writeset := writeset  $\cup \{p\}$ ;
```

```
upon correct  $\subseteq$  writeset do receive ACK from all correct
    writeset :=  $\emptyset$ ;
    if reading = TRUE then
        reading := FALSE;
        trigger < onar, ReadReturn | readval >;
    else
        trigger < onar, WriteReturn >;
```

Correct = {p1, p2, p3, p4}  
 $\langle \text{Val}, \text{ts} \rangle = \langle 0, 0 \rangle$

Writeset={p2}  
Writeset={p2, p4}

Writeset={p2, p3, p4}



# Read-Impose Write-All (1,N) Atomic Register

---

## Correctness:

- *Termination* – as for the Read-One Write-All (1,N) Regular Register.
- *Validity* - as for Read-One Write-All (1,N) Regular Register.
- *Ordering* – to complete a read operation, the reader process has to be sure that every other process has in its local copy of the register a value with timestamp bigger or equal of the timestamp of the value read. In this way, any successive read could not return an older value.

## Performance:

- *Write* - a write requests at most  $2N$  messages
- *Read* - a read requests at most  $2N$  messages

send and acks

---

## (1,N) Atomic Register: Fail-Silent Algorithm

↳ asynchronous, don't have perfect failure detector  
use majority!

## Read-Impose Write-Majority (1,N) Atomic Register

---

Failure model: crash

A majority of correct processes is assumed.

The algorithm is a variation of the Majority Voting (1,N) Regular Register

IDEA: A read imposes to a majority of processes to have the value read

## Read-Impose Write-Majority (1,N) Atomic Register

### Algorithm 4.6: Read-Impose Write-Majority (part 1, read)

Implements:

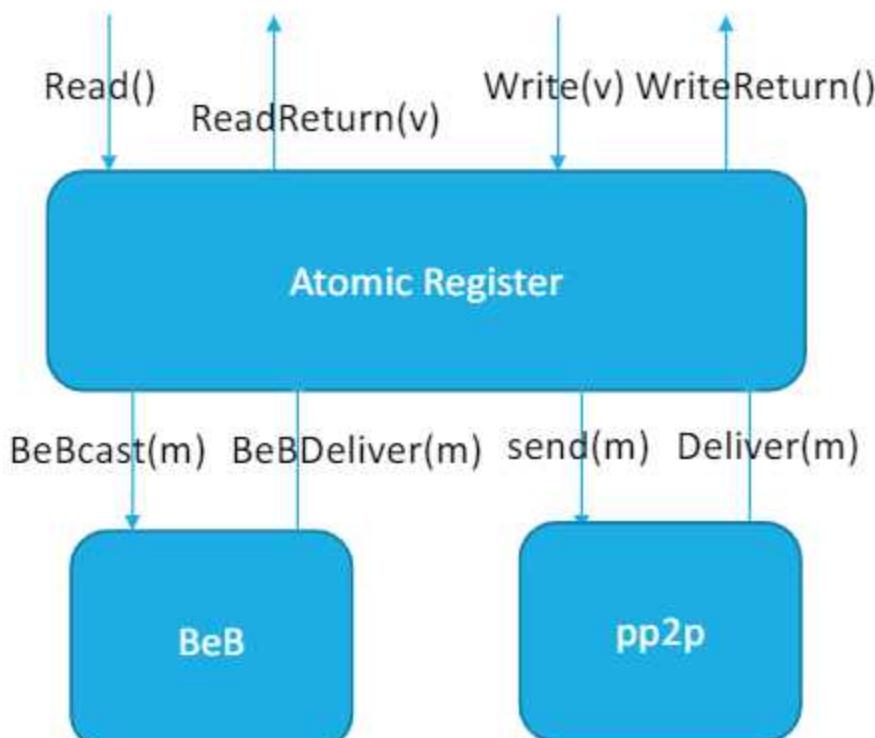
(1, N)-AtomicRegister, **instance onar**.

Uses:

BestEffortBroadcast, **instance beb**;

PerfectPointToPointLinks, **instance pl**.

upon event  $\langle onar, Init \rangle$  do  
     $(ts, val) := (0, \perp)$ ;  
     $wts := 0$ ;  
     $acks := 0$ ; *~to store acknowledgement to the write*  
     $rid := 0$ ;  
     $readlist := [\perp]^N$ ; *~majority*  
     $readval := \perp$ ;  
     $reading := \text{FALSE}$ ; *~if a read is waiting or acting like the writer*



## Read-Impose Write-Majority (1,N) Atomic Register

---

**upon event**  $\langle onar, Read \rangle$  **do**

$rid := rid + 1;$

$acks := 0;$

$readlist := [\perp]^N;$

$reading := \text{TRUE};$

**trigger**  $\langle beb, Broadcast \mid [\text{READ}, rid] \rangle;$

**upon event**  $\langle beb, Deliver \mid p, [\text{READ}, r] \rangle$  **do**

**trigger**  $\langle pl, Send \mid p, [\text{VALUE}, r, ts, val] \rangle;$

**upon event**  $\langle pl, Deliver \mid q, [\text{VALUE}, r, ts', v'] \rangle$  **such that**  $r = rid$  **do**

$readlist[q] := (ts', v');$

**if**  $\#(readlist) > N/2$  **then** *a majority have the value*

$(maxts, readval) := \text{highest}(readlist);$

$readlist := [\perp]^N;$

**trigger**  $\langle beb, Broadcast \mid [\text{WRITE}, rid, maxts, readval] \rangle;$

*wrk back  
to everybody*

## Read-Impose Write-Majority (1,N) Atomic Register

---

```
upon event < onar, Write | v > do
    rid := rid + 1;
    wts := wts + 1;
    acks := 0;
    trigger < beb, Broadcast | [WRITE, rid, wts, v] >; → to have a sequence of write

upon event < beb, Deliver | p, [WRITE, r, ts', v'] > do
    if ts' > ts then
        (ts, val) := (ts', v');
    trigger < pl, Send | p, [ACK, r] >;
```

```
upon event < pl, Deliver | q, [ACK, r] > such that r = rid do
    acks := acks + 1;
    if acks > N/2 then
        acks := 0;
        if reading = TRUE then
            reading := FALSE;
            trigger < onar, ReadReturn | readval >;
        else
            trigger < onar, WriteReturn >;
```

## Read-Impose Write-Majority (1,N) Atomic Register

---

### Correctness:

- Termination – as Majority Voting (1,N) Regular Register
- Validity – as Majority Voting (1,N) Regular Register.
- Ordering – due to the fact that the read imposes the write of the value read to a majority of processes and to the property of intersection of quorums.

### Performance: $\rightarrow$ still linear

- Write – at most  $2N$  messages
- Read – at most  $4N$  messages

$\hookrightarrow$  twice of price i paying before

# References

---

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

- Chapter 4 - until Section 4.3

# Dependable Distributed Systems

## Master of Science in Engineering in Computer Science

AA 2022/2023

LECTURES 16: SOFTWARE REPLICATION

how to replicate software components  
that has an internal state  
to maintain

↳ give more strength in terms  
of reliability and dependability

# Motivation

improving availability service we provide

- **Fault Tolerance**

- Guarantee the availability of a service (also called object) despite failures

like a data state

- Assuming  $p$  the failure probability of an object  $O$ .  $O$ 's availability is  $1-p$ .

↳ based on log, benchmark

- **Replicating an object  $O$  on  $n$  nodes and assuming  $p$  the failure probability of each replica,**  $O$ 's availability is  $1- p^n$  (considering independent failures probability)

# System Model

The system is composed of a set of processes (clients)

- Processes are connected through Perfect point-to-point links
- Processes may fail by crash

entities that generate requests to the system.

objects have an interface

is an abstraction of a web service, exposing several interfaces, that can be invoked

Processes interacts with a set of objects  $X$  located at different sites managed by processes

that have an effect on the state of the object

- Each object has a state accessed through a set of operations
- An operation by a process  $p_i$  on an object  $x \in X$  is a pair invocation/response
  - The operation invocation is noted  $[x \text{ op(arg)} p_i]$  where arg are the arguments of the operation op
  - The operation response is noted  $[x \text{ ok(res)} p_i]$  where res is the result returned
  - The pair invocation/response is noted  $[x \text{ op(arg)}/\text{ok(res)} p_i]$
- After issuing an invocation a process is blocked until it receives the matching response

# Replication: requirements

In order to tolerate process crash failures a logical object must have several physical replicas located at different sites of the distributed system

- replicas of an object  $x$  are noted  $x^1, x^2, \dots x^l$
- Invocation of replica  $x^j$  located on site  $s$  is handled by a process  $p_j$  also located on  $s$
- We assume that  $p_j$  crashes exactly when  $x^j$  crashes

for every object have  
in different copies that  
are running on n  
different  
processes/  
machines

Replication is transparent to the client processes

↳ Should not know that the object is replicated, and how many replicas there are. There is only an interface.

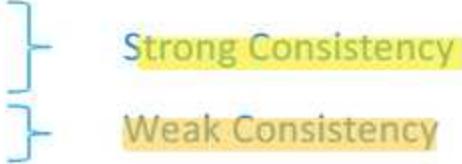
# Consistency criteria

→ tuples must act like a single object

A consistency criterion defines the result returned by an operation

- It can be seen as a contract between the programmer and the system implementing replication

Three main consistency criteria are defined in literature

- Linearizability (strongest)
  - Sequential consistency
  - Causal consistency
- 

↳ a partial order

The three consistency criteria differ however in the definition of the most recent state

# Linearizability

Let us consider the precedence relation ( denoted  $\prec$  ) and the concurrency relation (denoted  $\parallel$  ) defined between two operations.

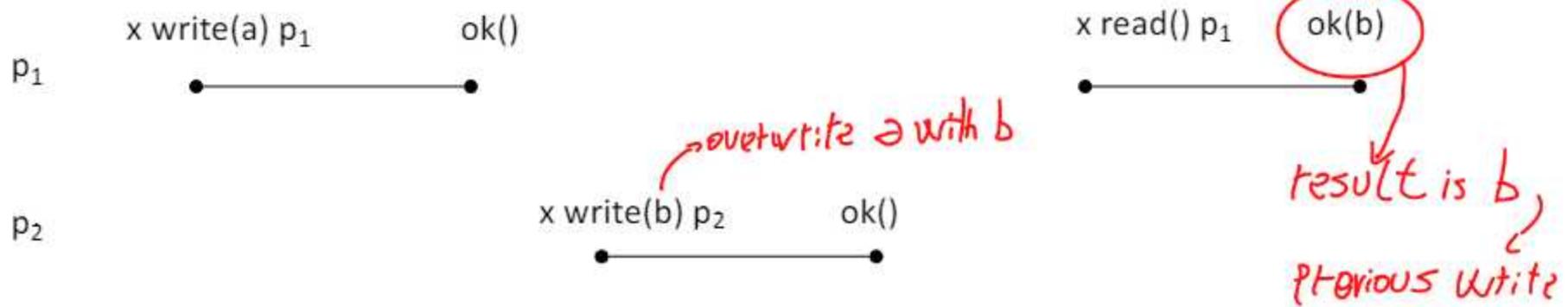
An execution E is linearizable if there exists a sequence S including all operations of E such that the following two conditions hold:

1. for any two operations  $O_1$  and  $O_2$  such that  $O_1 \prec O_2$ ,  $O_1$  appears before  $O_2$  in the sequence S, preserve causality of operations, concurrency operation are in usual order
2. the sequence S is legal i.e., for every object x the subsequence of S of which operations are on x belongs to the sequential specification of x.

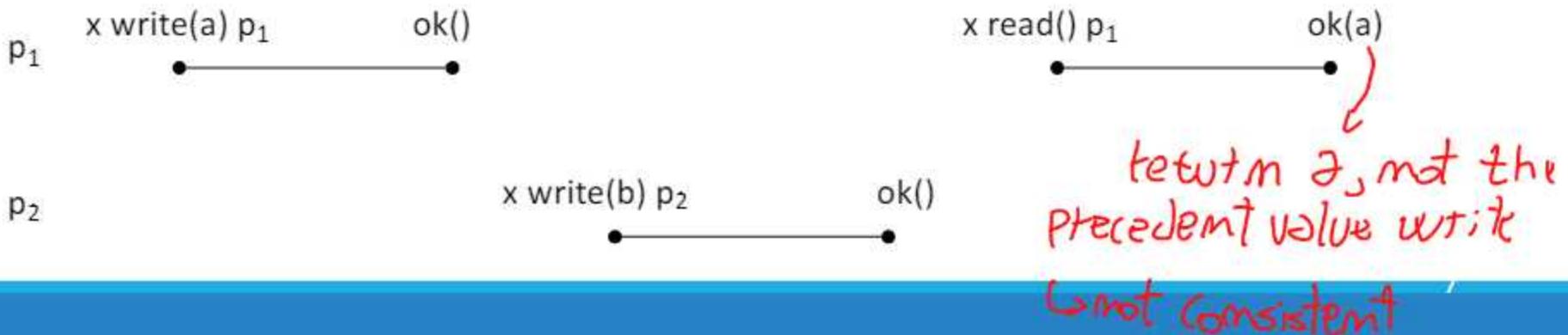
→ outcome is consistent with sequential specification of the object

# Example: simple variable

## 1. LINEARIZZABILE



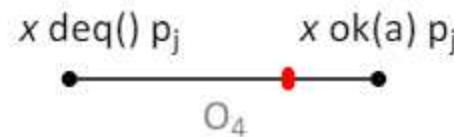
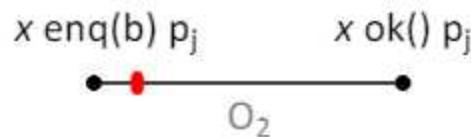
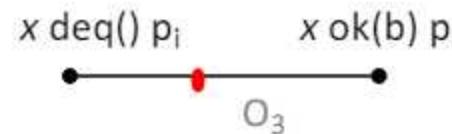
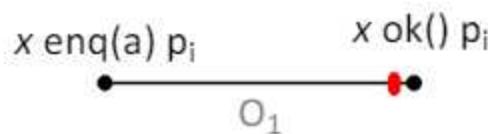
## 2. NON LINEARIZZABILE



First in First out

02: p  
01: b  
03: a  
04: l

# Example: FIFO Queue



Linearizable

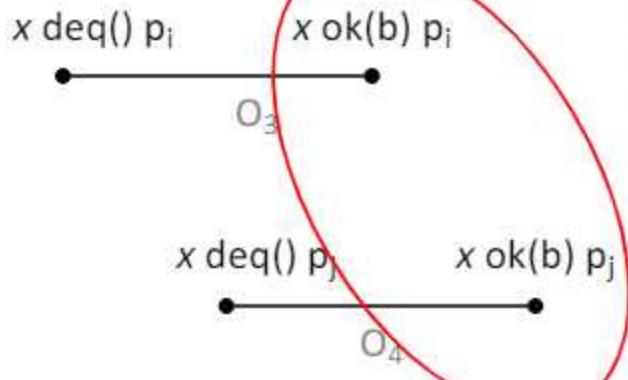
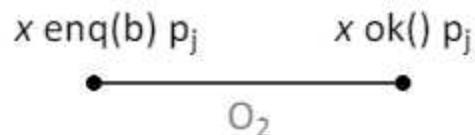
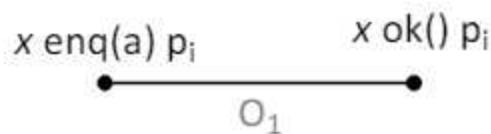
O<sub>1</sub> - O<sub>2</sub> and O<sub>3</sub> - O<sub>4</sub> are concurrent in part, i can decide the part order!

S = {O<sub>2</sub>, O<sub>1</sub>, O<sub>3</sub>, O<sub>4</sub>}

↳ can treat this sequence

↳ not have a causal order  
the concurrent operation

# Example: FIFO Queue



Not Legal wrt  
the sequential  
specification of  
a FIFO queue

NON-Linearizable  
(> not exist  
a consistent sequence)

there is always a and b  
not can be two deq(b)

# A Sufficient Condition for Linearizability

*, two sufficient condition but not necessary  
we can have linearizability without it*

Replicas must agree on the set of invocations they handle and on the order according to which they handle these invocations

**Atomicity:** Given an invocation  $[x \text{ op(arg)} p_i]$ , if one replica of the object  $x$  handles this invocation, then every correct replica of  $x$  also handles the invocation  $[x \text{ op(arg)} p_i]$ . *(like agreement in RB)*

**Ordering:** Given two invocations  $[x \text{ op(arg)} p_i]$  and  $[x \text{ op(arg)} p_j]$  if two replicas handle both the invocations, they handle them in the same order

# Replication Techniques

---

Two main techniques implementing linearizability:

---

- Primary Backup
- Active Replication

# Passive Replication

---

PRIMARY-BACKUP

# Primary Backup

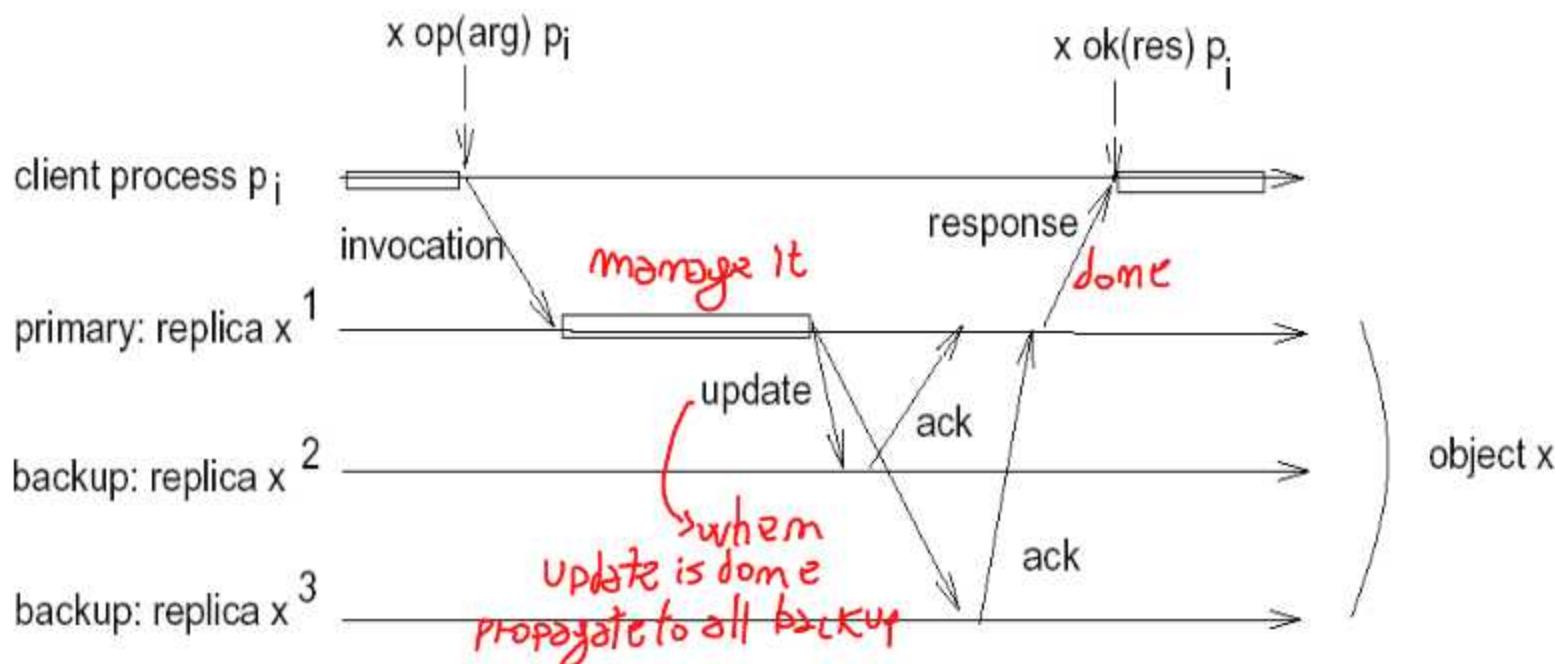
divide the topics in two set.  
Primary is a single replica at like  
a coordinator.

- Primary: is a single point of failure
  - Receives invocations from clients and sends back the answers.
  - Given an object  $x$ ,  $prim(x)$  returns the primary of  $x$ .
- Backup:
  - Interacts with  $prim(x)$
  - is used to guarantee fault tolerance by replacing a primary when crashes

↳ need a leader election module  
in case of failure

# Primary Backup Scenario

(we modify the state)



# Primary Backup: the case of no crash

---

1. When update messages are received by backups, they update their state and send back the ack to  $\text{prim}(x)$ .
2.  $\text{prim}(x)$  waits for an ack message from each correct backup and then sends back the answer,  $\text{res}$ , to the client.

How to guarantee Linearizability: the order in which  $\text{prim}(x)$  receive clients' invocations define the order of the operation on the object.

---

Failure of the client not impact consistency of the object state

# Primary Backup: Presence of Crash

---

Three scenarios :

*before sending update response*

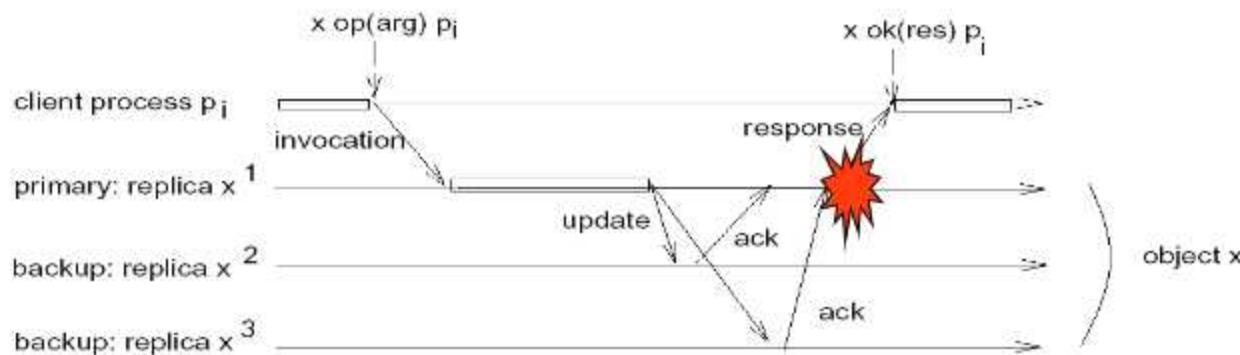
- **Scenario 1:** Primary fails after the client receives the answer.
- **Scenario 2:** Primary fails before sending update messages
- **Scenario 3:** Primary fails after sending update messages and before receiving all the ack messages.

In all cases there is the need of electing a new leader.

---

## Scenario 1 easiest

Primary fails after sending the answer



Two cases

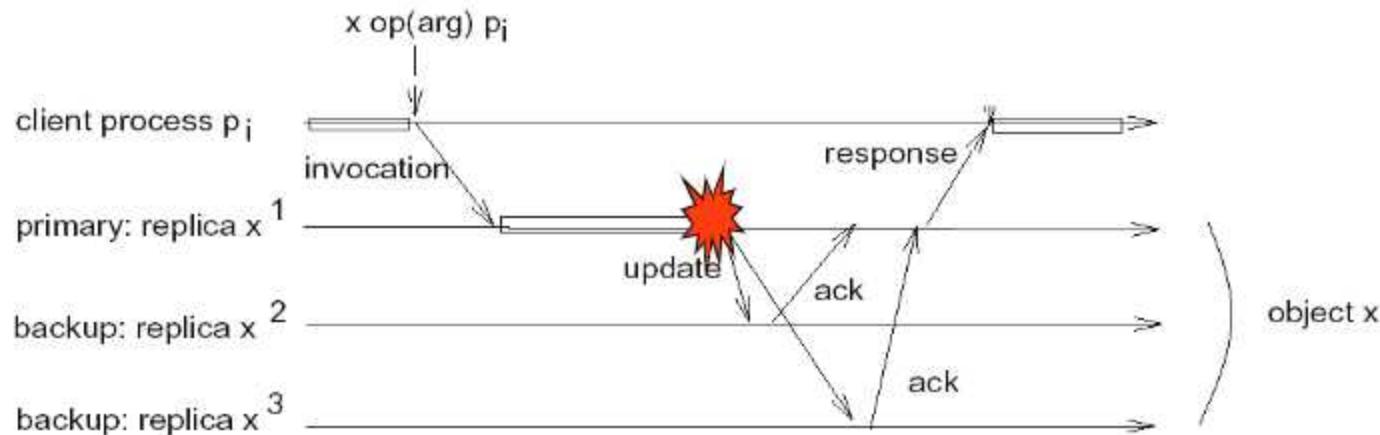
1. Client does not receive the response due to perfect point-to-point link. If the response is lost, client retransmits the request after a timeout
2. Client receives the answer, everybody is happy (but the primary 😊)

The new primary will recognize the request re-issued by the client as already processed and sends back the result without updating the replicas

i must have at least one R/B primitive

## Scenario 2

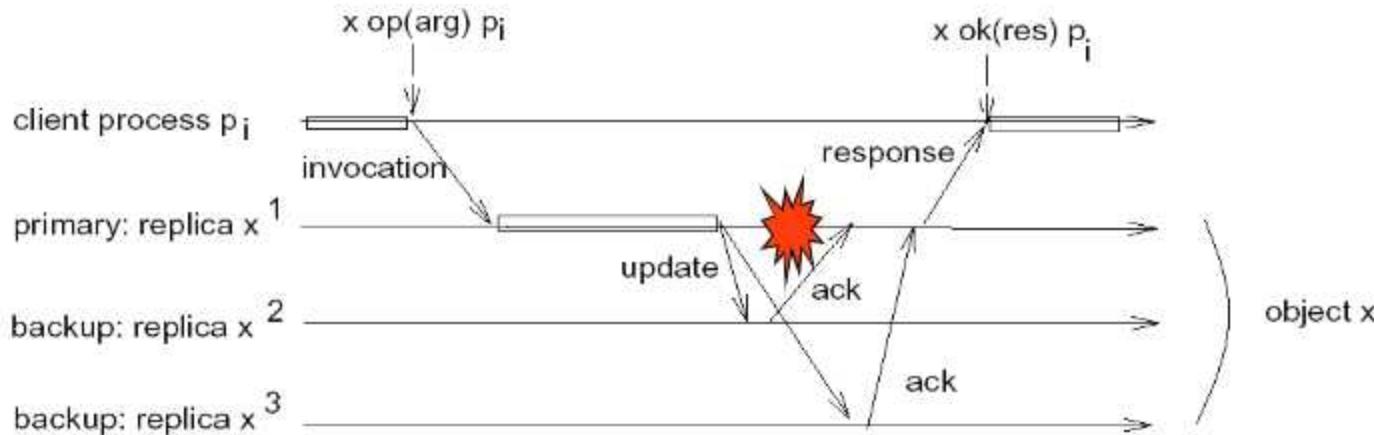
### Primary fails before sending update messages



Client does not get an answer and resends the requests after a timeout

The new primary will handle the request as new

and a leader election not eventually  
Scenario 3 most problematic, implies we have RB not good BeB  
Primary fails after sending update messages and  
before receiving all the ack messages  $P, RB, L$  easy to achieve



**How to Guarantee atomicity?** update it is received either by all or by no one.

When a primary fails there is the need to elect another primary among the correct replicas

## Active Replication

---

# Active Replication

lightway method, use  $2N$  messages  
for each operation, round trip time is  
given by the slowest replica

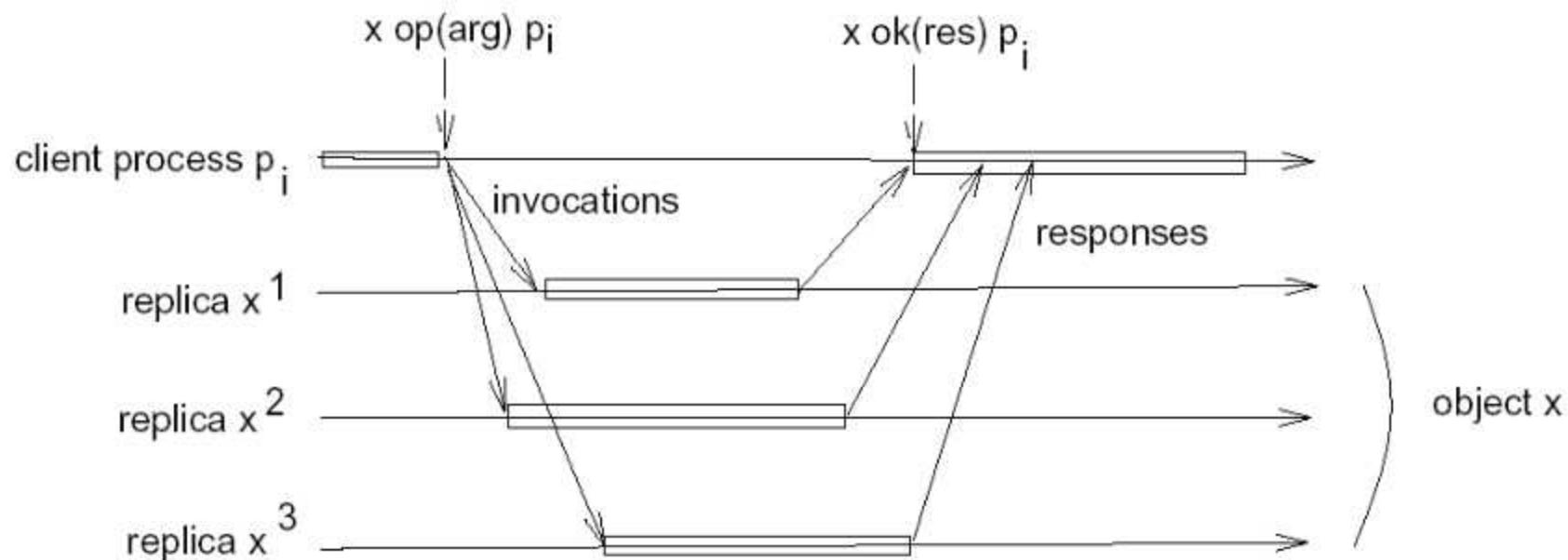
There is no coordinator all replicas have the same role

Each replica is deterministic. If any replica starts from the same state and receives the same input, they will produce the same output

As a matter of fact clients will receive the same response one from each replica

no distinction in replica and client

# Active Replication



# Active Replication

## How to Guarantee Linearizability

---

To ensure linearizability we need to preserve:

- Atomicity: if a replica executes an invocation, all correct replicas execute the same invocation.
- Ordering: (at least) no two correct replicas have to execute two invocations in different order.

We need: TOTAL ORDER Broadcast

- INCLUDING THE CLIENTS!

# Active Replication

## Crash

---

Active Replication does not need recovery action upon the failure of a replica

---

↳ is sufficient one reply, a failure is not important

# References

---

Rachid Guerraoui and André Schiper: “*Fault-Tolerance by Replication in Distributed Systems*”. In *Proceedings of the 1996 Ada-Europe International Conference on Reliable Software Technologies* (Ada-Europe '96).

# Dependable Distributed Systems

## Master of Science in Engineering in Computer Science

AA 2022/2023

---

LECTURE 17 : OVERVIEW ON CAPACITY PLANNING

# Recap dependability

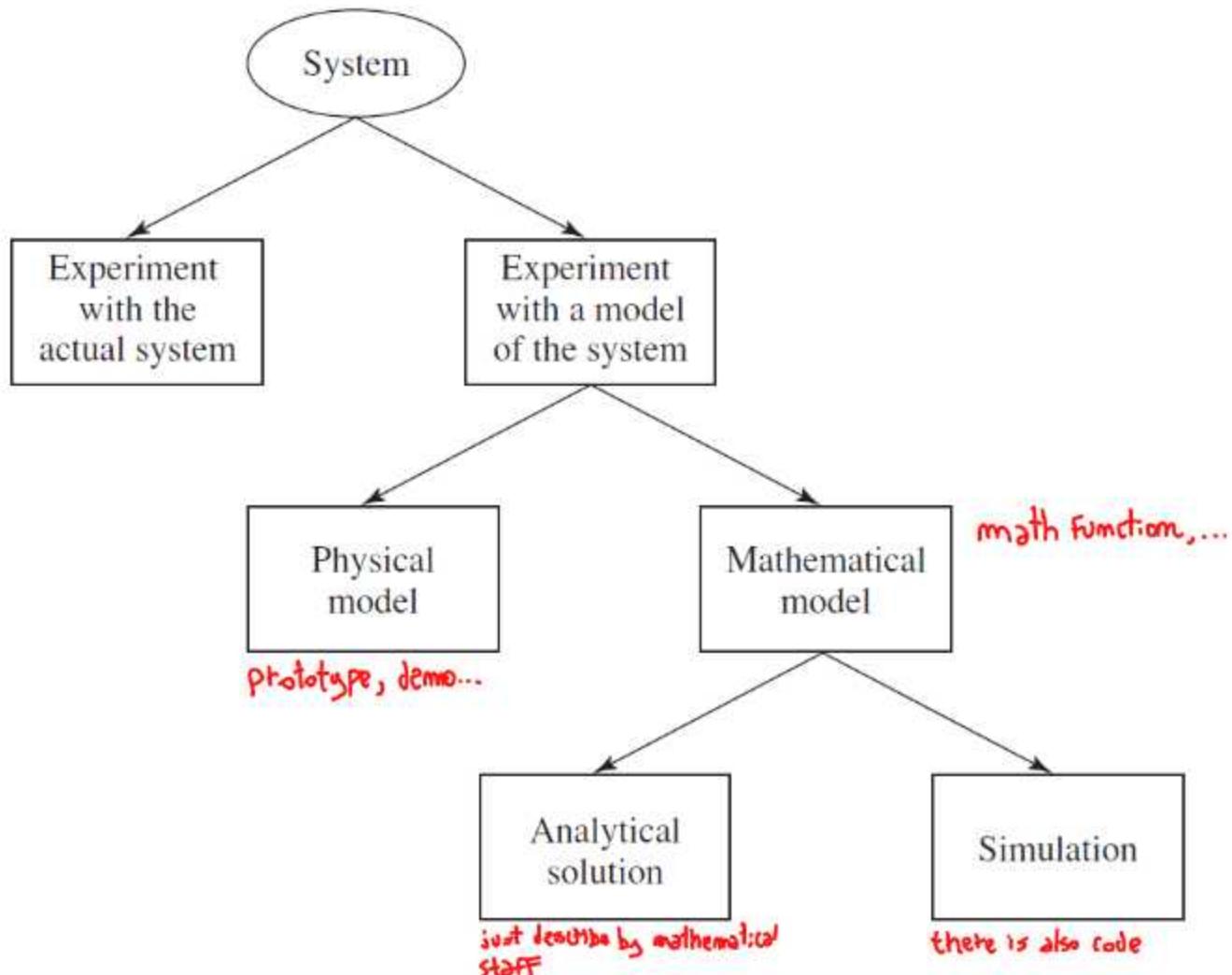
**Dependability** is the ability of a system to deliver a service that can justifiably be trusted,  
it is the ability to avoid service failures that are more frequent and more severe than is acceptable

A **service failure** is an event that occurs when the delivered service deviates from correct service

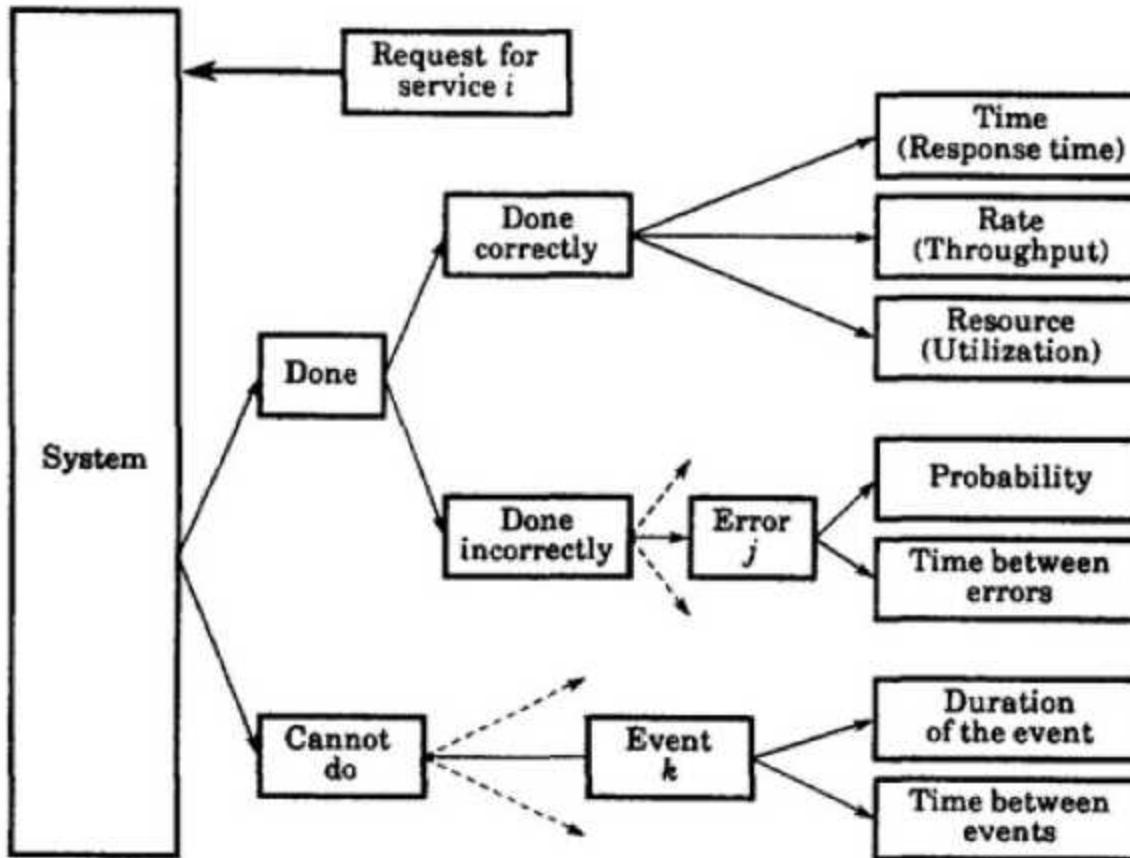
A **correct service** is delivered when the service implements its functional specifications in terms of

- **functionality**
- **performance**

# Ways to study a system



# Very Basics for Dependability Evaluation



# Why do performance affect correct service?

**SLA** (Service level agreement)

**It defines how a service should operate within agreed-upon boundaries**

---

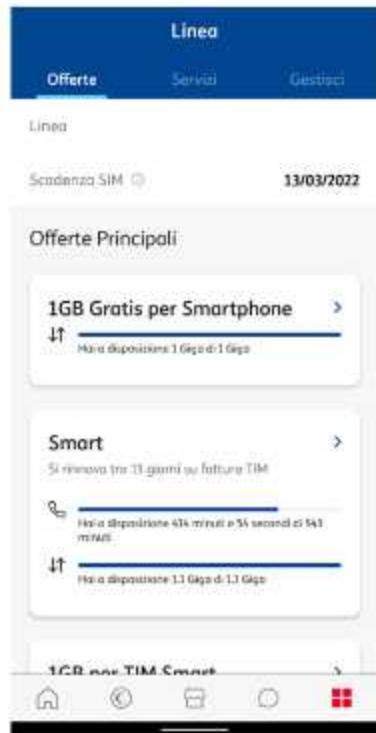
**SLAs determine what a user of an application can expect** in terms of response time, throughput, system availability, reliability, etc.

---

- focus on metrics that users can understand
- set easy-to-measure goals
- tie IT costs to your SLAs

# Why do performance affect correct service?

## Users expectation



The screenshot shows a mobile application interface. At the top, there is a blue header bar with the word 'Linea' in white. Below it, a secondary header bar has three tabs: 'Offerte' (selected), 'Servizi', and 'Gestisci'. The main content area is titled 'Linea' and shows a 'Scadenza SIM' of 13/03/2022. Below this, there is a section titled 'Offerte Principali' with two items: '1GB Gratis per Smartphone' and 'Smart'. Each item has a description and a 'Dettagli' button. At the bottom of the screen, there is a navigation bar with icons for home, back, forward, and search.



Users expectation **varies**  
**depending on what type**  
**of application** they are  
**using** and even what  
portion of the application  
they are interacting with

# How do we analyze performance?

## Benchmarking the system:

- limited number of testable scenarios
- potential expensive

## Building models

- is it possible to characterize the system and its load through models?
- cheaper

**Main requirement:** collect measure

# The need of metrics

*"In physical science the first essential step in the direction of learning any subject is to find principles of numerical reckoning and practicable methods for measuring some quality connected with it.*

*I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the state of Science, whatever the matter may be."*

*[PLA, vol. 1, "Electrical Units of Measurement", 1883-05-03]*



Lord Kelvin

**You cannot manage what you cannot measure!**

# How are certain levels of performance achieved? **Capacity planning** For understand limit of the system

**IT capacity planning** consists in **estimating** the storage, hardware, software and connection infrastructure **resources required over some future period of time to correctly support service provisioning.**

Alternatively

IT capacity planning is the process of **predicting when the service levels will be violated as a function of the workload evolution**, as well as the **determination of the most cost-effective** way of delaying system saturation.

\_> **Adequate capacity**

Properly handle peaks and average behavior

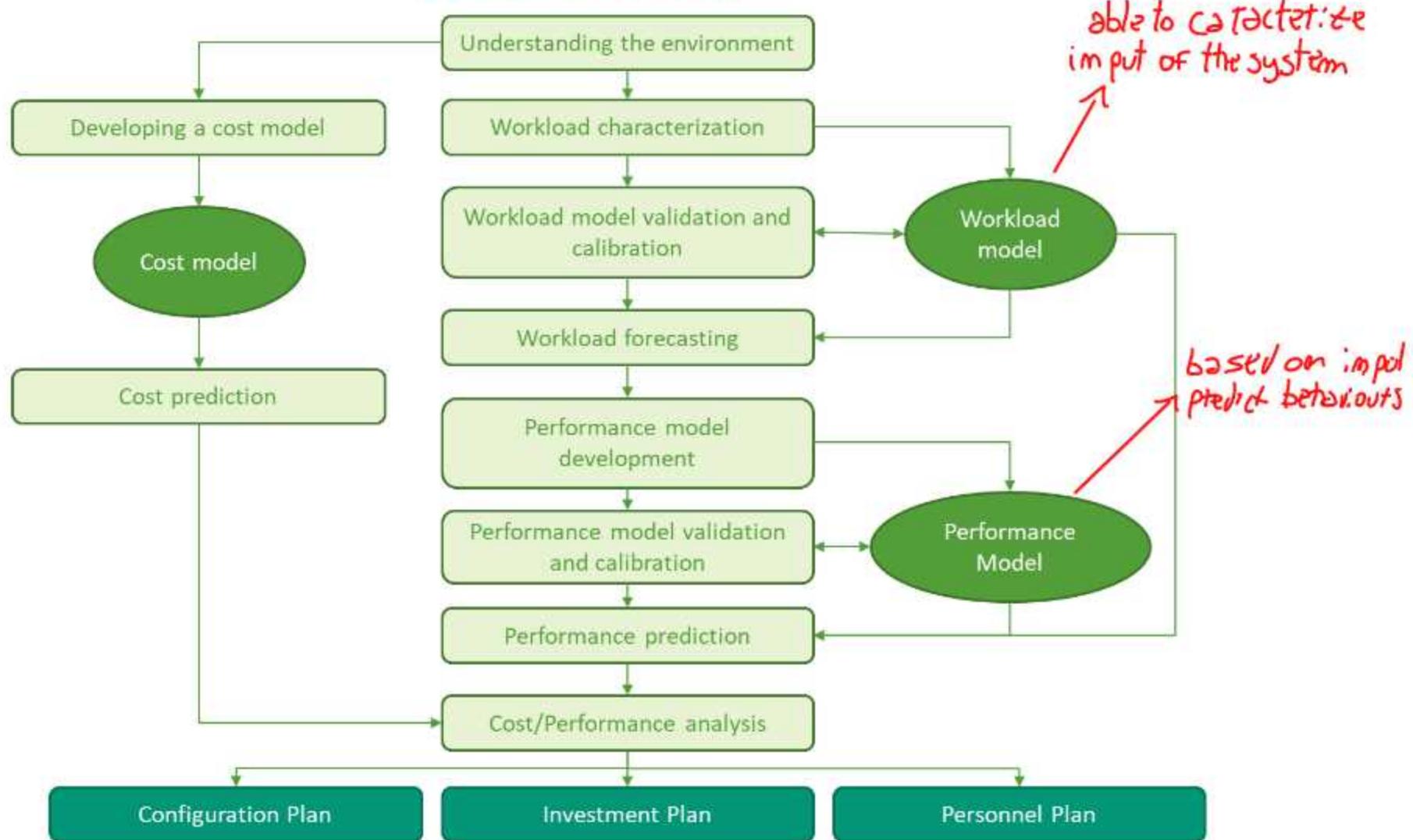
# Why perform capacity planning?

if don't provide correct services:

- Avoid financial losses
- Ensure customer satisfaction
- Preserve company's external image
- **Capacity planning problem cannot be solved instantaneously**

# A methodology for capacity planning

(create three model)



# Understanding the environment

The goal is to learn what kind of

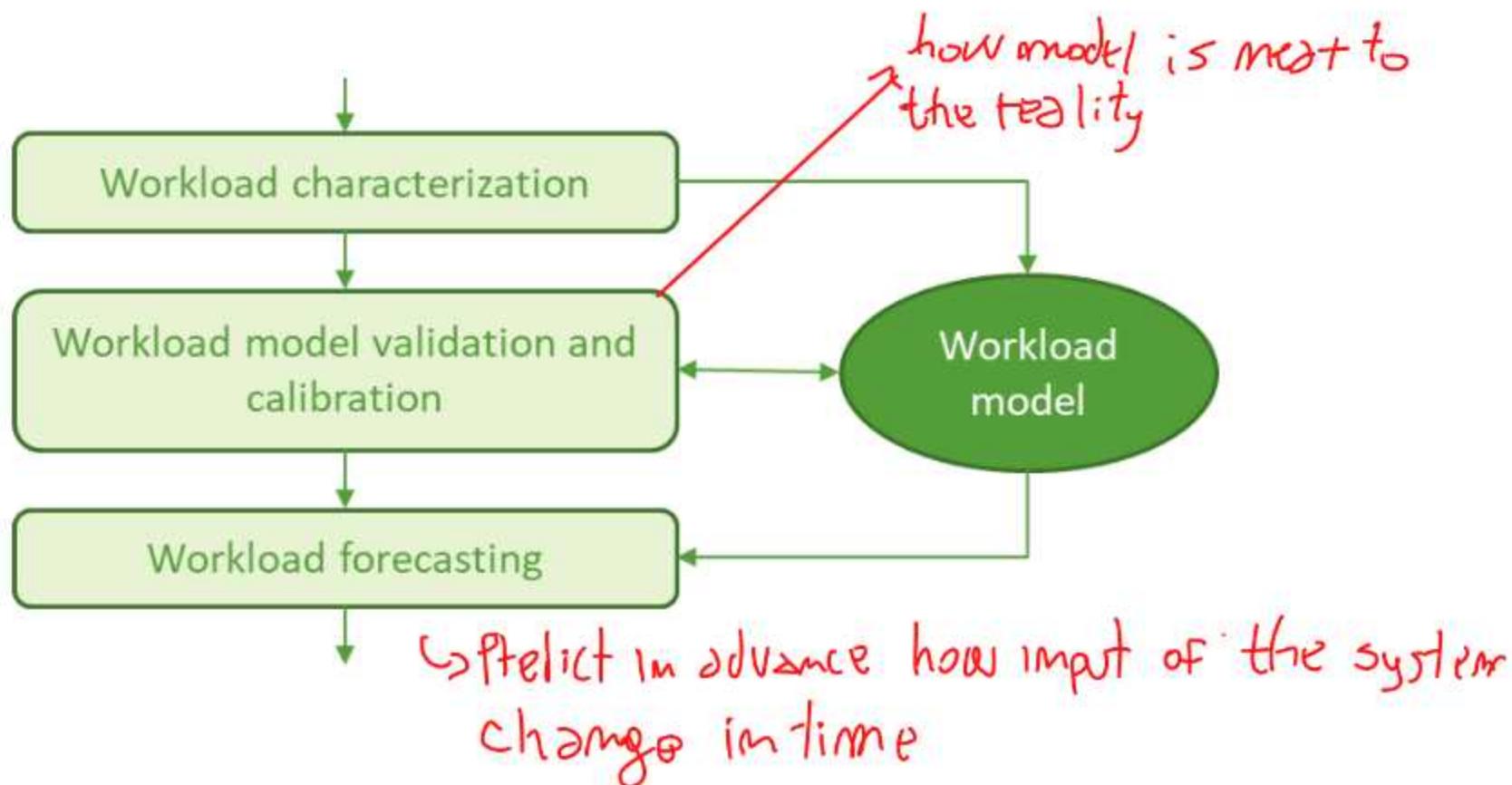
- **hardware (clients and servers)**
- **software (OS, middleware, applications)**
- **network** connectivity and protocols
- **SLA**
- ... (whatever may have an impact on the considered performance metrics)

are present in the environment

---

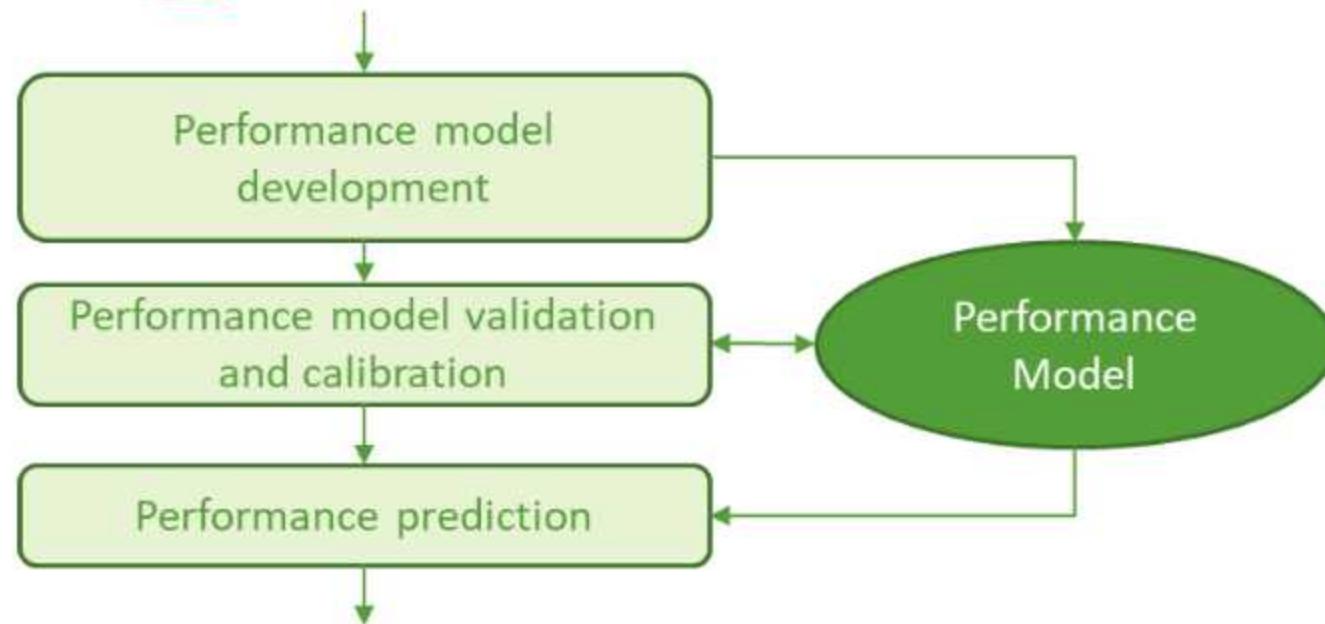
# Workload model

The **workload** of a system is **the sets of all the inputs** that the system receives from its environment during any given period of time



# Performance model

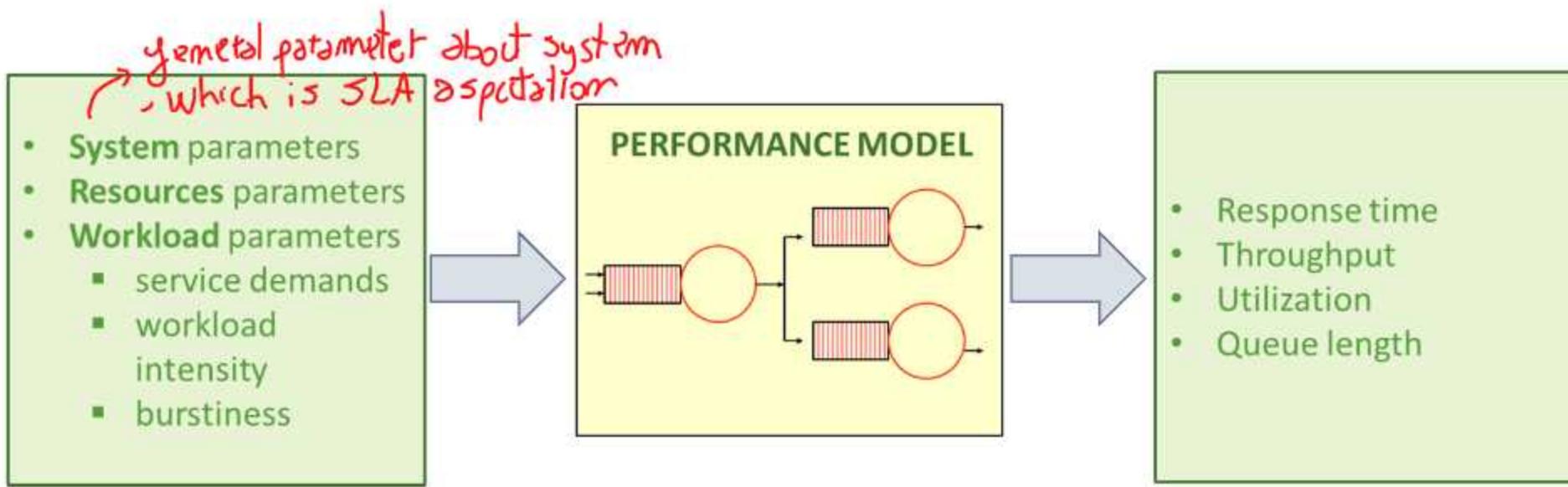
Used to predict performance as function of system description and workload parameters



Estimates performance measures of a computer system for a given set of parameters

Outputs: response times, throughputs, system resources utilizations, queue lengths, etc.

# Estimating performance measures



# Parameters affecting performance metrics

## System parameters examples:

- load-balancing disciplines
- network protocols
- max. num of connections supported
- ...

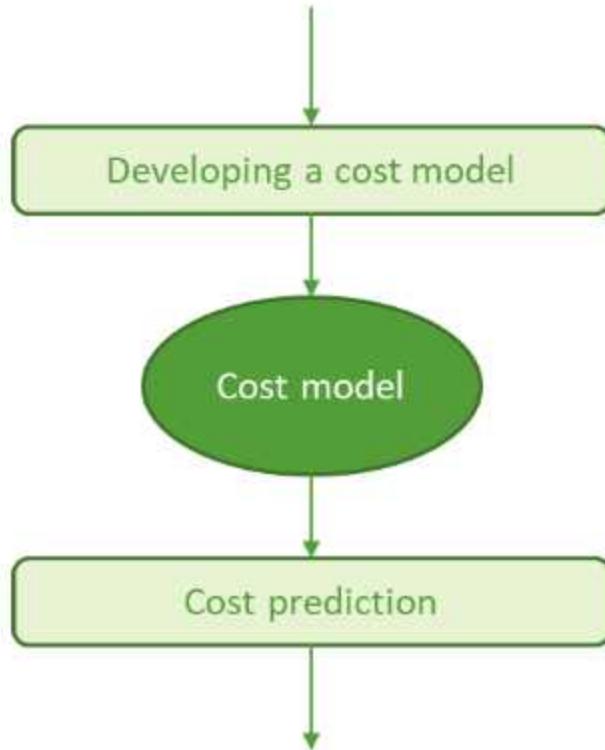
## Resource parameters examples:

- disk latency, transfer rate
- network bandwidth;
- CPU speed
- ...

## Workload parameters examples

- WL intensity parameters:
  - num. of requests
  - num. of clients running an application
  - Burstiness
  - ...
- WL service demand parameters:
  - CPU time per request
  - Disk usage per request
  - ...

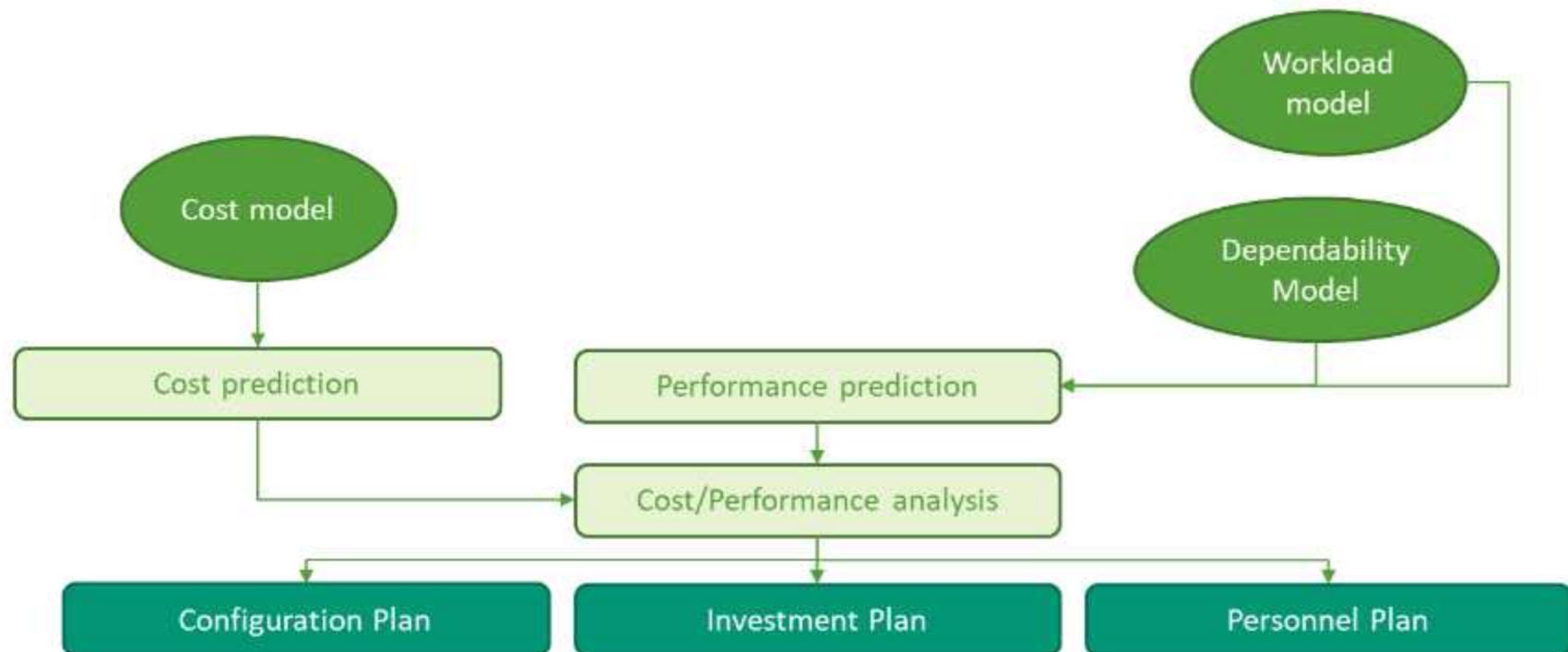
# Cost model



## Categories:

- Hardware cost: machines, disks, routers, etc.
- Software cost: operating systems, middleware, etc.
- Telecommunication cost
- ...

# Cost/performance analysis



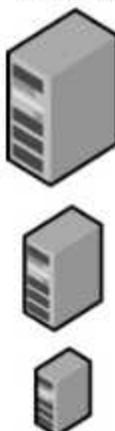
- Assess possible scenarios
- For each scenario, predicts performance metrics and costs
- Comparing scenarios, get configuration, investment and personnel plans
- Assess payback: ROI (return of investment), company's image, etc.

# Scalability: Vertical VS Horizontal

It is the ability of a computer application or product (hardware or software) to continue to function well when it (or its context) is changed in size or volume in order to meet a user need.

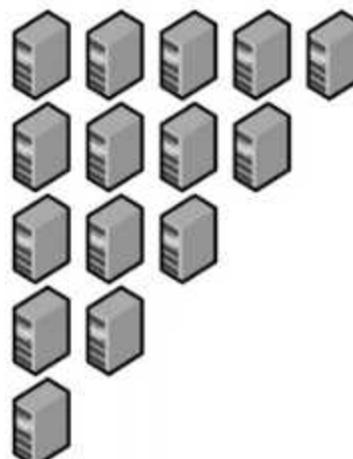
- Less complex
- Upgrade limitations
- Single point of failure

Vertical



vs.

Horizontal



- Increased complexity
- No limit to the number of processes
- Increased resilience and fault tolerance
- Horizontal scale  $\neq$  Increase in performance metrics

# Reference

- Chapter 5 - D. A. Menascé, V. A. F. Almeida: *Capacity Planning for Web Services: metrics, models and methods*. Prentice Hall, PTR  
(Available in the library inside Dipartimento di Ingegneria informatica, automatica e gestionale Antonio Ruberti)

# Dependable Distributed Systems

## Master of Science in Engineering in Computer Science

AA 2022/2023

---

LECTURE 18 – MODELING THE WORKLOAD OF A SYSTEM

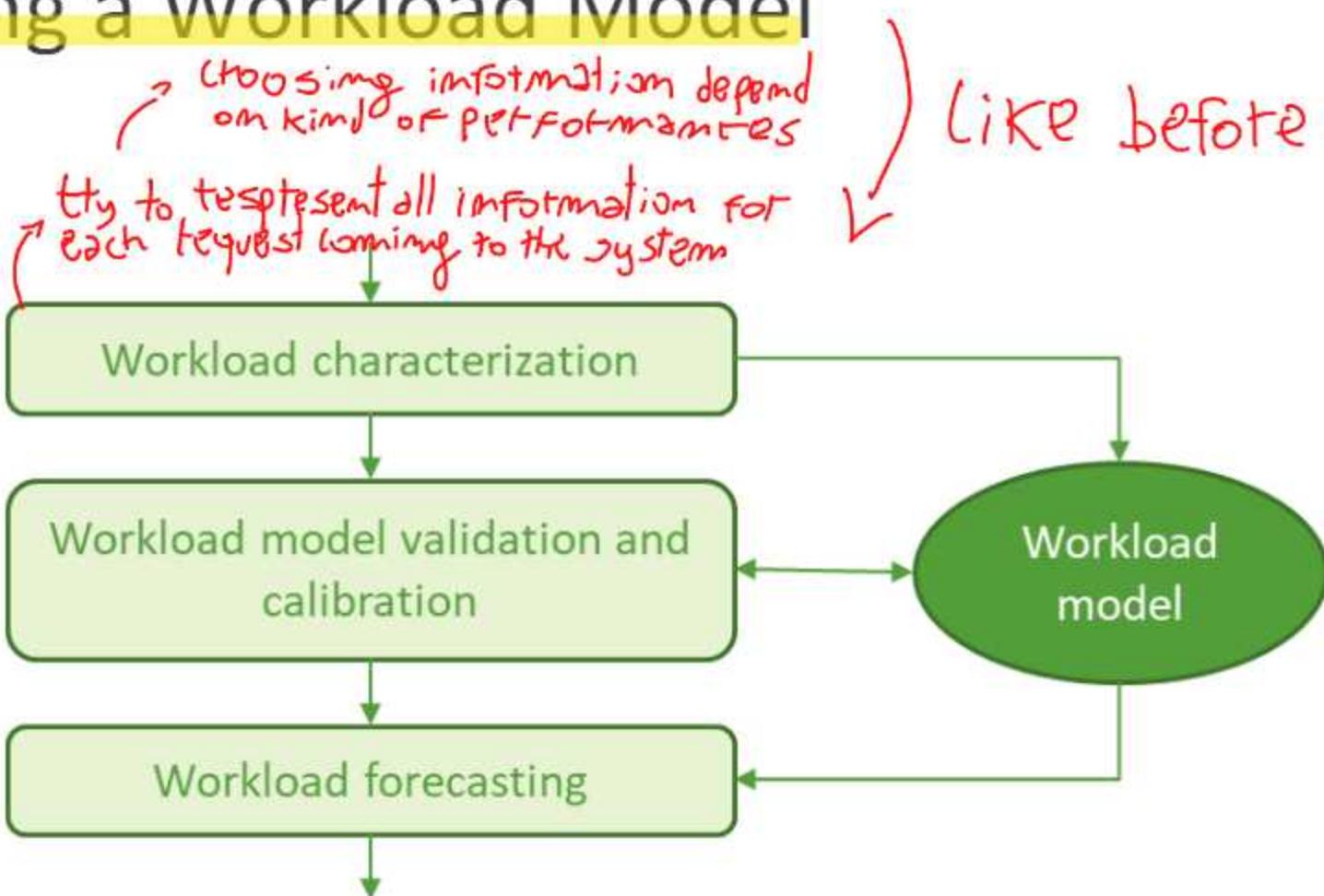
## Workload

The performance of a system depends heavily on the characteristics of its load

→ understand and characterize the workload

The workload of a system is the sets of all the inputs that the system receives from its environment during any given period.

# Building a Workload Model



# Workload Characterization

Since a **real-user environment is generally not repeatable**, it is necessary to study the real-user environments, **observe the key characteristics**, and develop a workload model that can be used repeatedly.

This process is called workload characterization.

The **workload characteristics** are represented by a **set of information** (e.g. arrival and completion times, CPU time, number of I/O operations, size of the requested object, etc.) **for each request**.

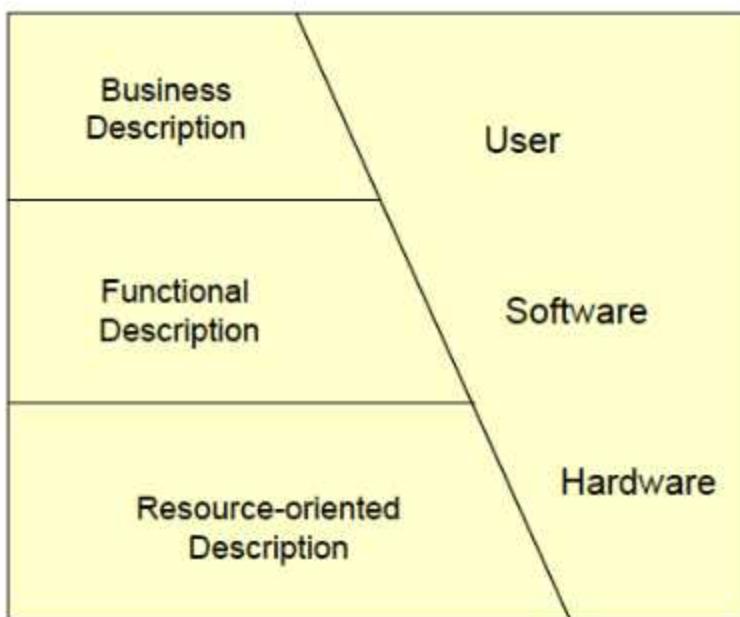
The **choice of the characteristics** and parameters that will describe the workload **depends on the purpose** of the study.

# Workload Model Construction: **Common Steps**

1. **Specification of a point of view** from which the workload will be analyzed (inside or outside the system?)
2. **Choice of the set of parameters** that capture the most relevant characteristics of the workload for the purpose of the study
3. **Monitoring** the system to obtain the raw performance data
4. **Analysis** and reduction of the performance **data**
5. **Construction** of a workload **model**
6. **Verification and Validation** that the characterization captures all the important performance information

# Workload Description

The workload of a computer system can be described at different levels:



- **Business characterization:** a user-oriented description that describes the load in terms such as number of employees, invoices per customer, etc.
- **Functional characterization:** describes programs, commands and requests that make up the workload
- **Resource-oriented characterization:** describes the consumption of system resources by the workload, such as processor time, disk operations, memory, etc.

## Workload Components

The requests arriving to a system could be heterogeneous

---

Identify the workload components, namely the different (and relevant) units of work that arrive at the system from external sources (e.g. read requests, transactions, etc.)

Each workload components must be characterized

# Workload Parameters

Each workload component is characterized by a set of parameters

The kind of parameters strongly depends on the kind of service

The performance provided by a system are mostly influenced by:

- The arrival pattern ↗ how request is distributed over the time
- The service demands ↗ what are the single component interested by a single request

The parameter can be separated into two groups:

- Workload intensity (e.g. arrival rate, number of clients, think time, etc.)
- Service demands, i.e. the service demand of a workload component at every involved resource

# Specific Workload Parameter Examples [2]

## *Web Workloads:*

- page properties,
- traffic properties,
- access patterns,
- user behaviour.

## *Video Service Workloads:*

- media properties,
- traffic properties,
- user behaviour,
- social sharing properties.

## *Shopping Service Workloads:*

- business level,
- session level,
- function level,
- protocol level.

## *Mobile Device Workloads*

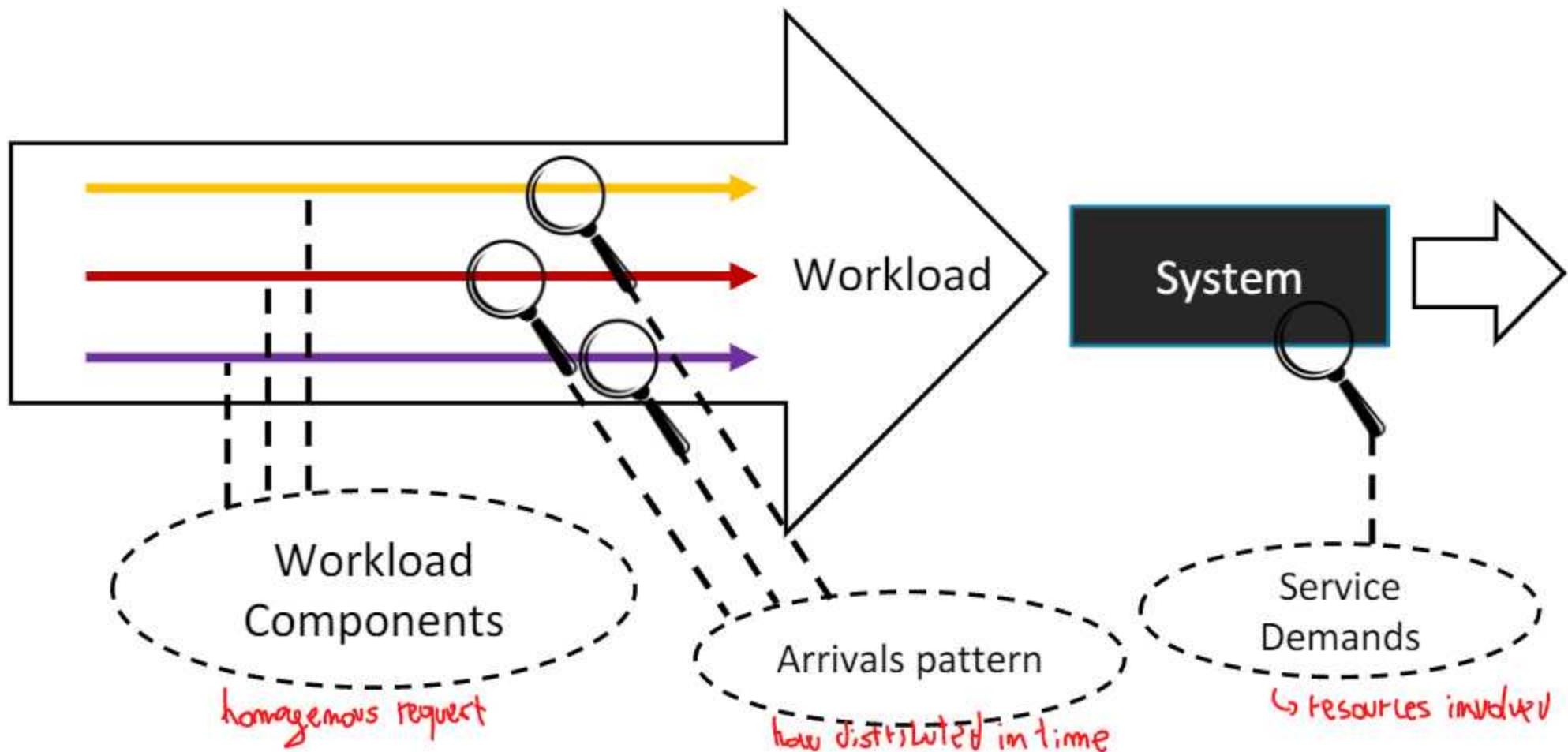
- traffic volume,
- access time,
- unique subscribers,
- Locations.

## *Online Social Network Workloads:*

- user behaviour,
- network structure and evolution,
- content propagation.

There exists dedicated literatures about the characterization of these and other specific parameters characterizing the workloads

# Workload Characterization Recap Picture



# Consideration Characterizing Workload

1. What are the **exercised services**? (e.g., in a distributed software application, which software component are involved? Which are the involved resources?)
2. What is the **level of detail**?
  1. Most frequent request
  2. Frequency of request types
  3. Time-stamped sequence of requests
  4. Average resource demand
  5. Distribution of resource demands
3. **A workload should be representative of the real application:**
  1. **Arrival Rate:** The arrival rate of requests should be the same or proportional to that of the application.
  2. **Resource Demands:** The total demands on each of the key resources should be the same or proportional to that of the application.
  3. **Resource Usage Profile:** Resource usage profile relates to the sequence and the amounts in which different resources are used in an application.
4. **Timeliness**
  1. Workloads should follow the changes in usage pattern in a timely fashion

## Collect Measures

Workload characterization relies on **experimental approaches** based on the analysis of **measurements collected** on the technological infrastructures while they are operating (i.e., under their real workloads).

Measurements provide qualitative and quantitative **information about the individual workload components**.

NOTE: take into account that **monitors may introduce overhead!**

↳ Software component that try to log events in the system

## Collect Measures: **too many data?**

In general, the amount of **data being collected can become quite large** and sometimes even intractable.

→ **Identify the time window**

Appropriate **sampling techniques** may need to be applied. Since there might be the danger of ignoring events referring to rare significant workload components, it is very important to ensure the representativeness of the data sample being considered.

# Analysis

**Statistical Analysis Techniques:** application of statistical and visualization techniques.

**Descriptive statistics and measures of dispersions** (e.g. mean, range, variance, coefficient of variation, skewness, median, percentiles) are useful to summarize the properties of each attribute.

# Analysis

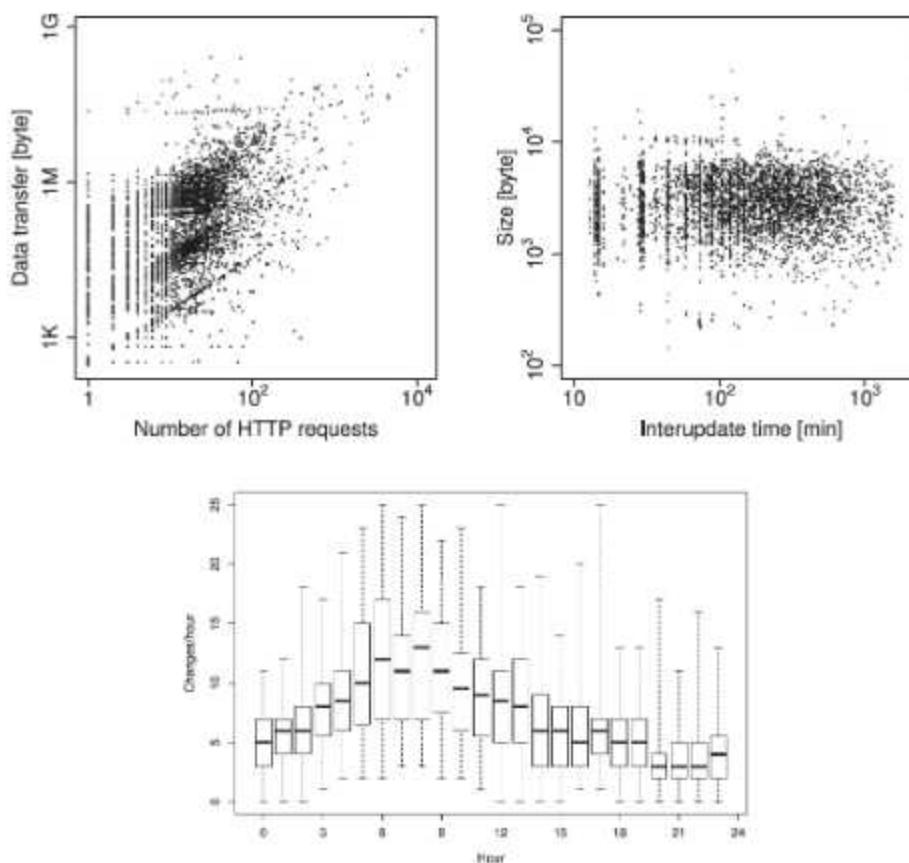
Real workload can be viewed as a **collection of heterogeneous components**  
→ **Partition the workload**, i.e. divide it into a series of **classes** such that  
their population are formed by quite **homogeneous components**.

For analysis purposes, it is **useful** to classify these components into a **small**  
**number of classes or clusters** such that **the components within a cluster**  
**are very similar to each other**.

# Analysis

**Diagrams**, such as histogram, scatter plot or box plots, may provide initial hints to interpret collected data

Scatter plots highlight the **correlation** between attributed, whereas box plots summarize their **distribution**



# Analysis

The **term outlier** denotes the workload components characterized by an atypical behaviour of one or more attributes.

It is critical to take the right approach toward outliers because of their potential effects on the workload models.

Outliers could indicate phenomena or properties previously unknown, thus worth exploring.

On the contrary, they could correspond to anomalous operating conditions of the infrastructures or even errors in the measurements, thus worth discarding.

## Analysis: Identify Components, Clustering

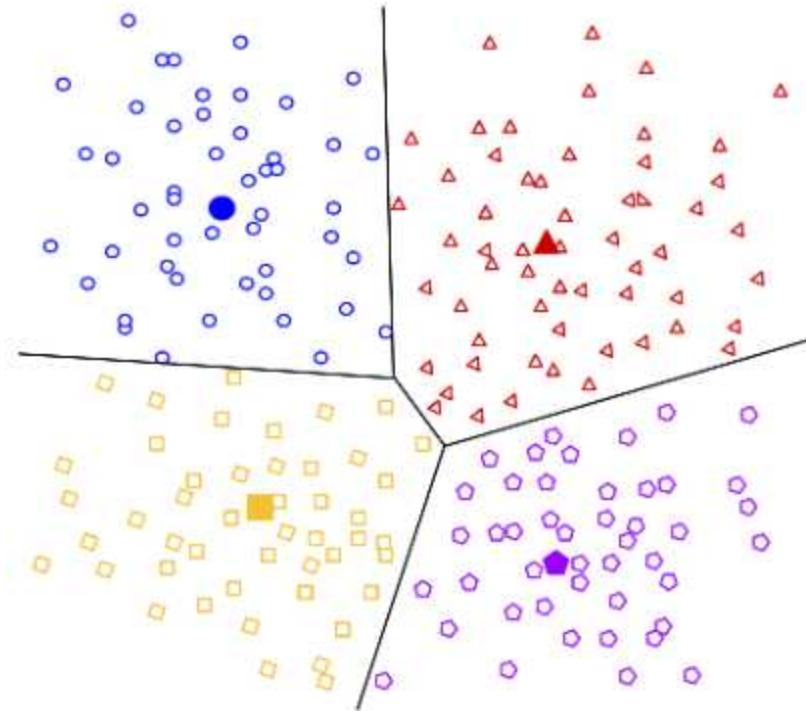
Clustering is an **unsupervised** process that **subdivides a set of observations** (i.e., workload components) **into homogeneous groups** (i.e., clusters).

The **components of each group are very similar**, whereas the **components across groups are quite distinct**.

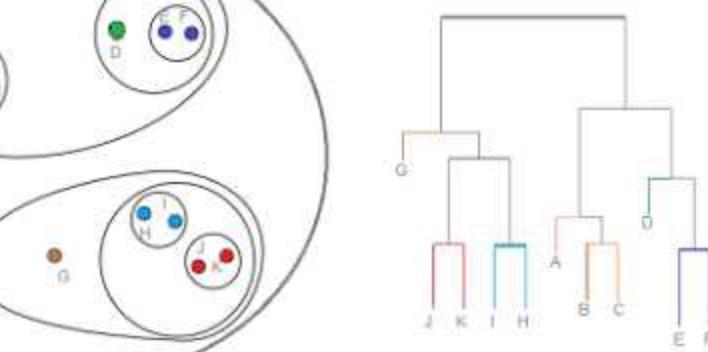
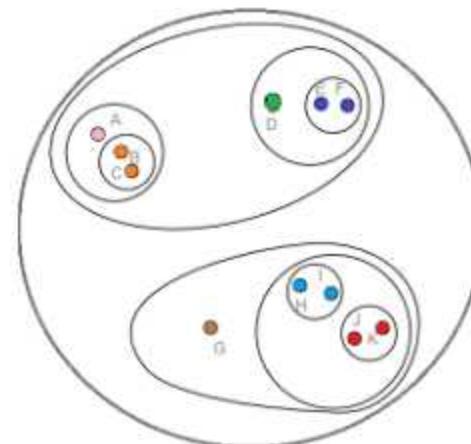
The **centroids** (i.e., the **geometric center of the clusters**) are often used as representatives of the groups.

Distance-based clustering techniques **differ** for the **algorithms applied** (e.g., hierarchical, iterative) **and** their **similarity measures** (e.g., Euclidean distance, Manhattan distance).

# Some Clustering Algorithms

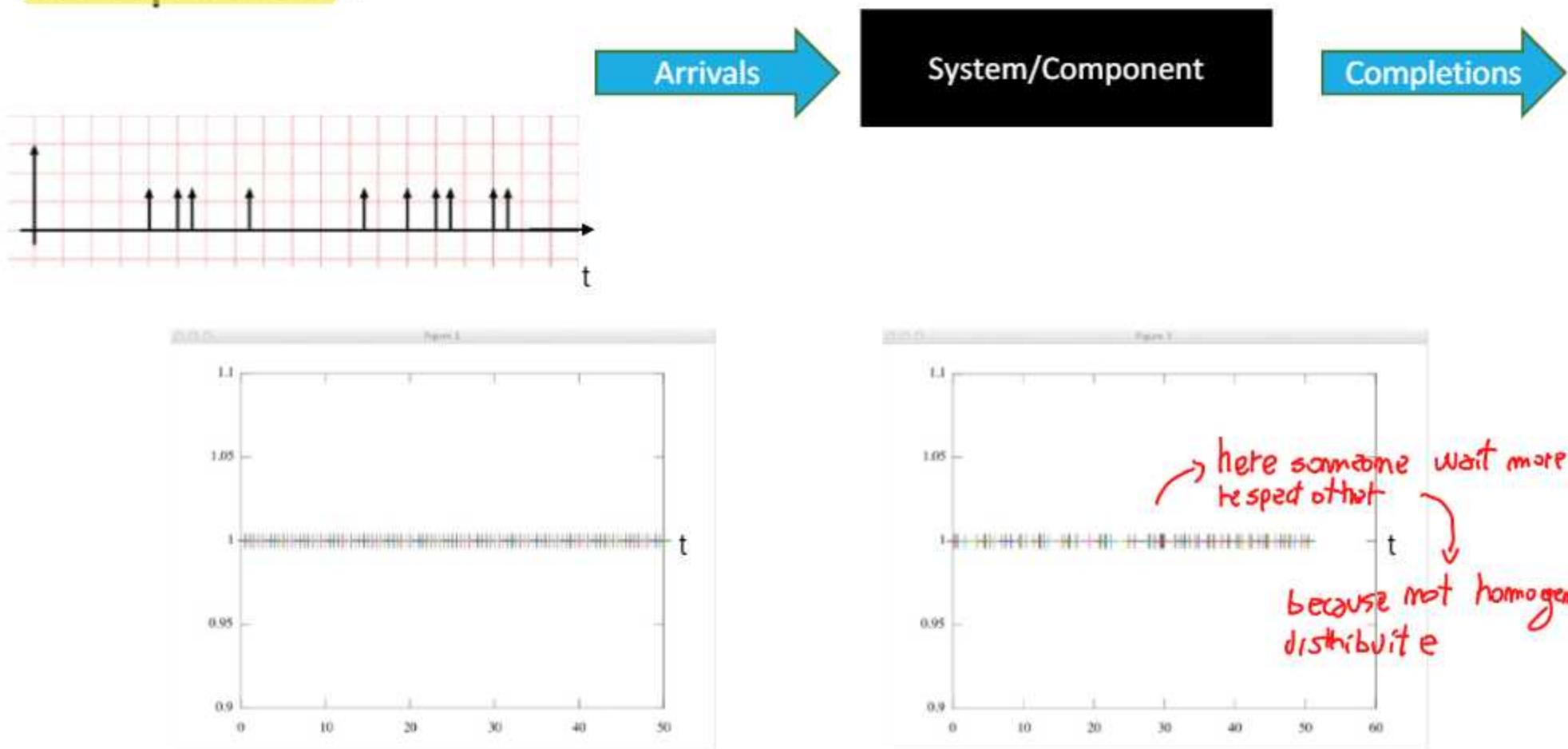


Centroid-based Clustering  
(e.g. K-means)



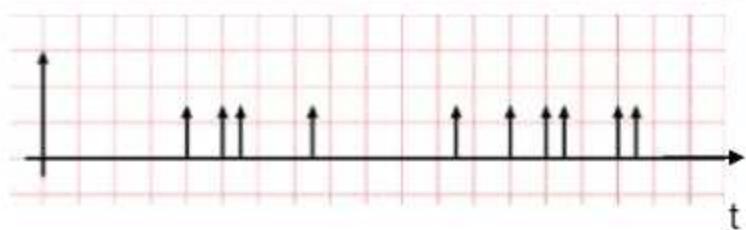
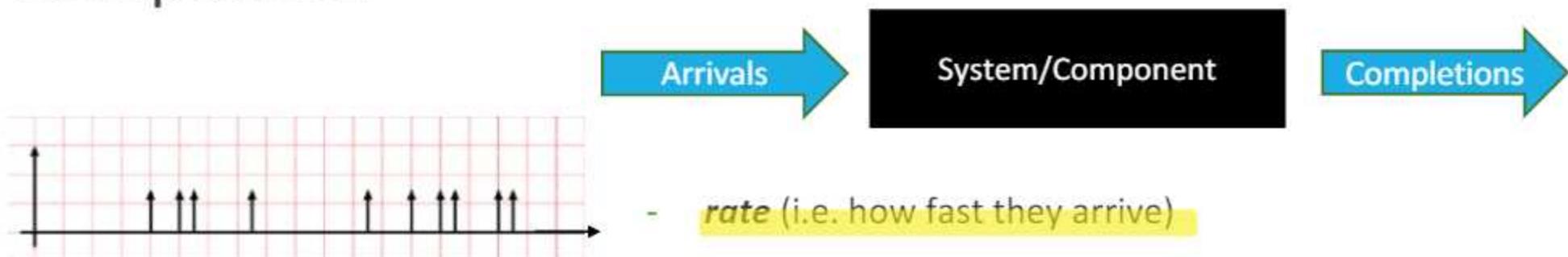
Hierarchical Clustering

# Characterizing the Arrival Pattern of a Single Component

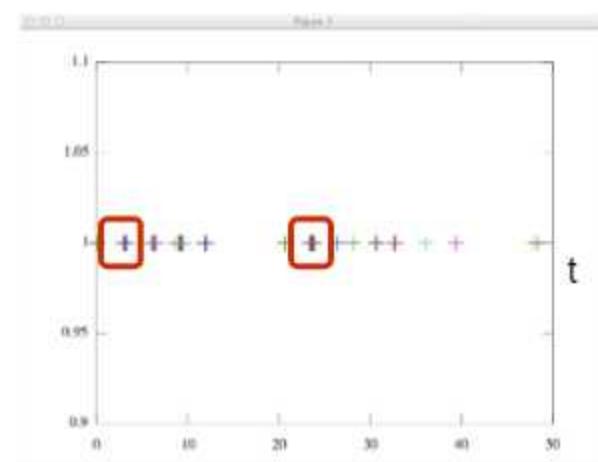
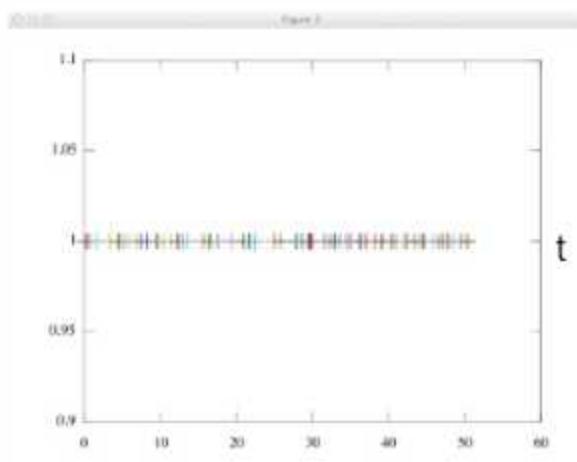
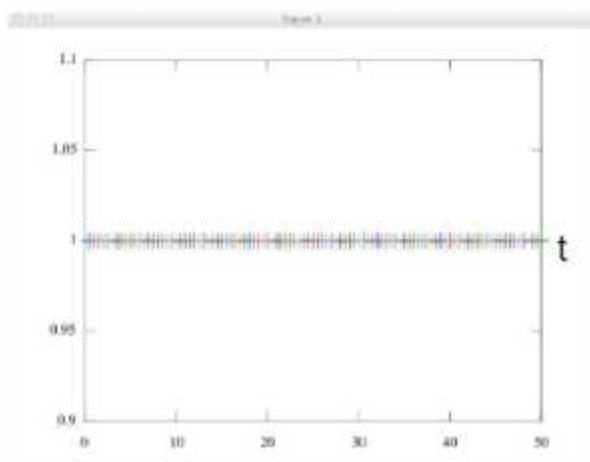


The mean arrival rate  $\lambda$  of the two workloads is the same

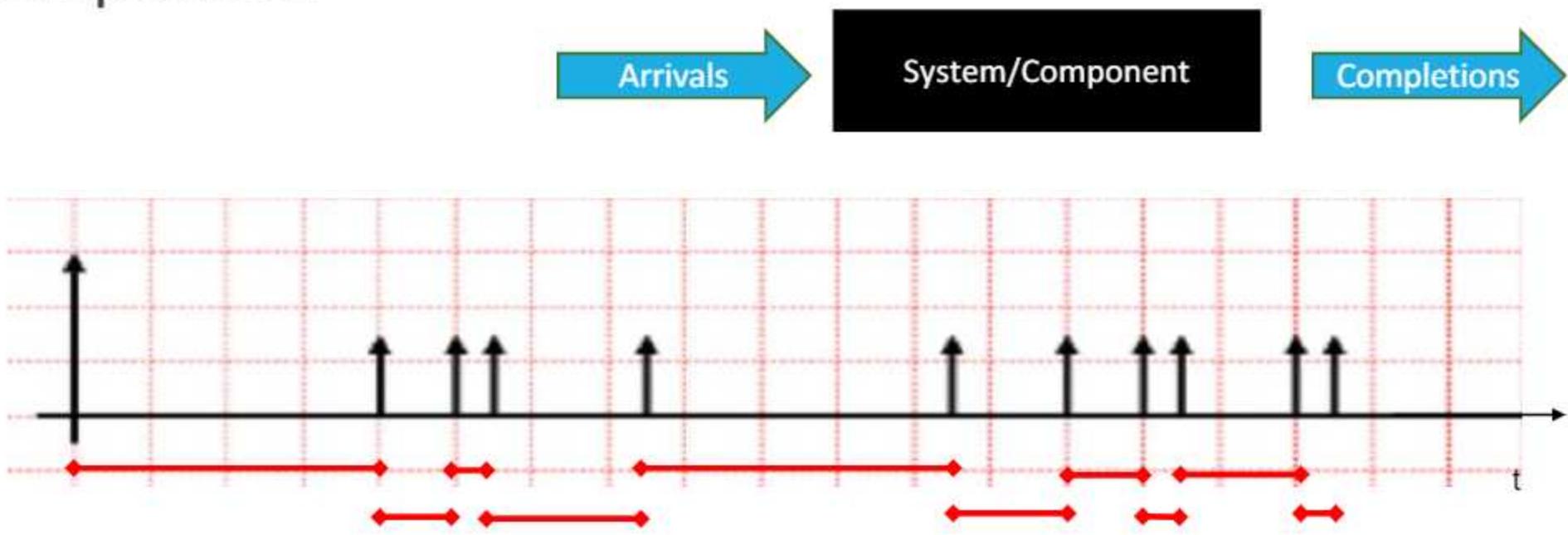
# Characterizing the Arrival Pattern of a Single Component



- **rate** (i.e. how fast they arrive)
- **regularity** (i.e. the time that passes between two occurrences)
- **correlation** (informally, inter-arrival are independent or there is a correlation?)

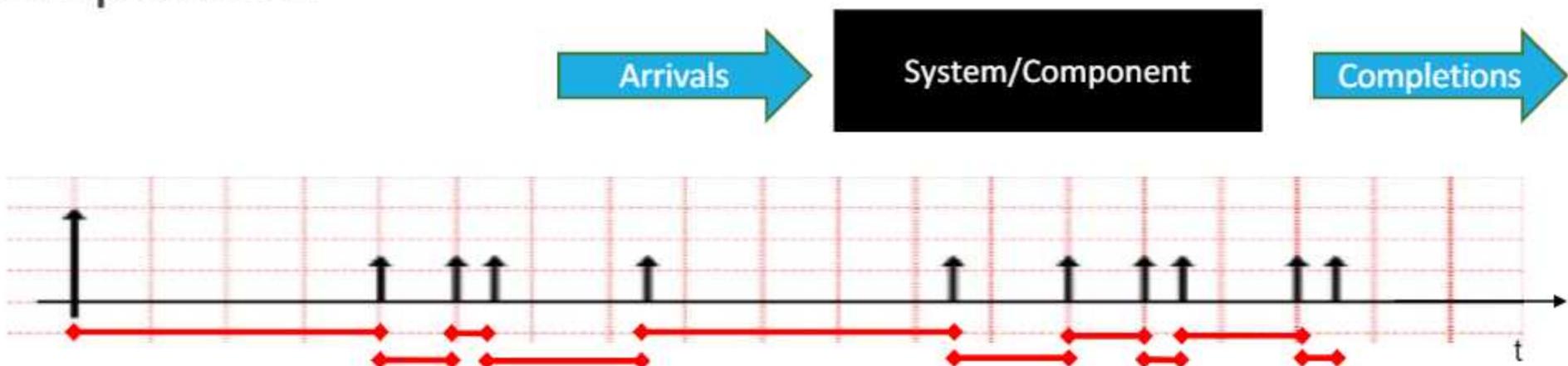


# Characterizing the Arrival Pattern of a Single Component



**Inter-arrival time:** time that passes between an arrival and the following one

# Characterizing the Arrival Pattern of a Single Component



**Most of the times, the arrivals to a system are not deterministic**, namely they do no occur exactly with the same pattern (same arrivals with the same inter-arrival times). **They are random.**

e.g., let us assume we deployed an e-commerce; we cannot know at what times requests will arrive to the service (search an item, add to cart, etc.) but, looking at the log of previous request (or asking to some expert) we may predict the possible target of such request (what is the mean inter-arrival time of all the request, if there are bursts at specific hours, etc.)

We model the arrivals of our system through a **random variable**

## Random Variable in a Nutshell

Informally: a variable is called a random variable if **it takes one of a specified set of values with a specified probability.**

---

It can either be continuous or discrete.

---

The description of **how likely** a random variable takes one of its **possible values** can be given by a **probability distribution.**

---

The **probability distribution** is a mathematical function that gives the probabilities of different outcomes for an experiment.

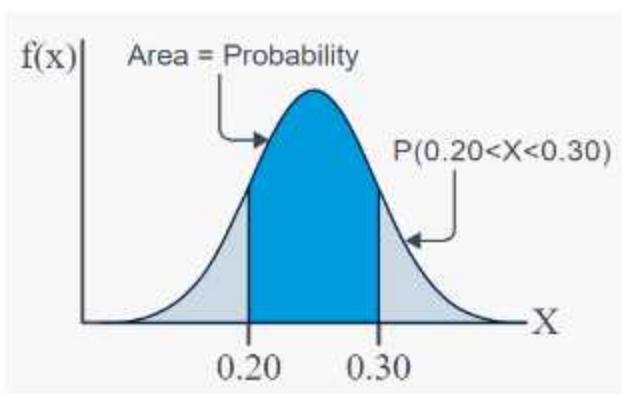
---

# Random Variable in a Nutshell

Continuous Random Variable, Probability Distribution

## \_> Probability Density Function (PDF)

PDF is used to **specify the probability of the random variable falling within a particular range of values** (a continuous random variable takes on an uncountably infinite number of possible values, the probability that the variable takes on any particular value is 0)

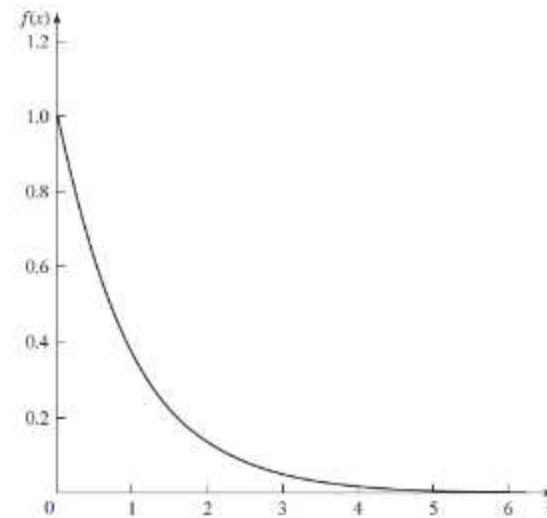
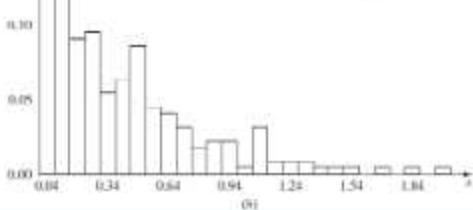
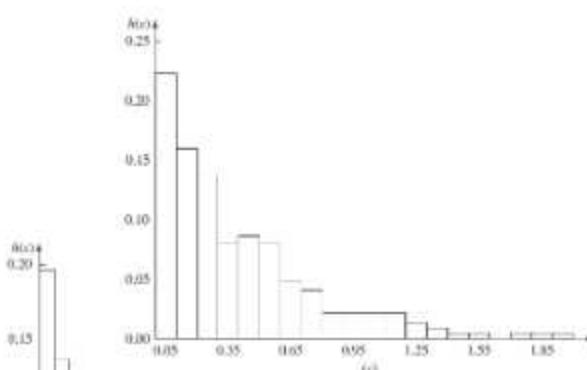
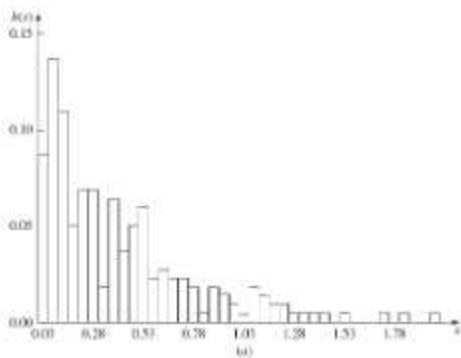


<https://online.stat.psu.edu/stat414/lesson/14/14.1>

<https://towardsdatascience.com/understanding-random-variables-and-probability-distributions-1ed1daf2e66>

# From Data to a Random Variable (simplified)

1. Take the measures (e.g., inter-arrival times between arrivals) and order them in increasing order.
2. Partition data in  $k$  adjacent intervals (bins) of the same size, then count the number of occurrences in every bin and plot  $\rightarrow$  make a histogram
3. Get a graphical estimate of the PDF



PDF exponential distribution

# Exponential Distribution

We say that a random variable  $X$  is distributed Exponentially with rate  $\lambda$ ,

$$X \sim \text{Exp}(\lambda)$$

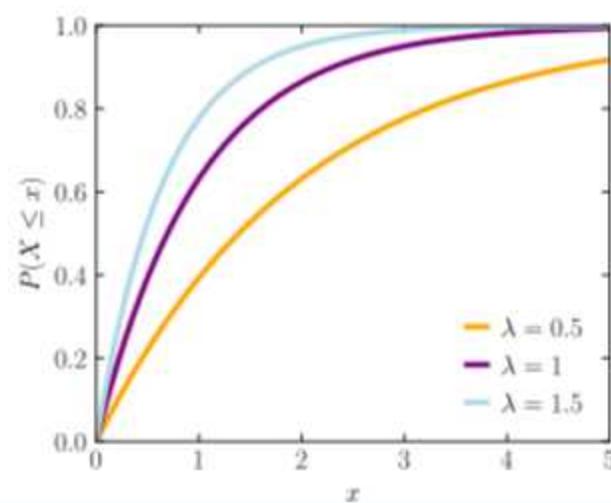
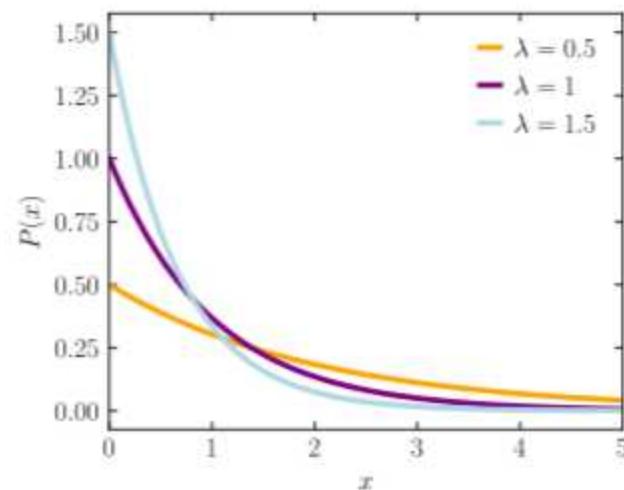
if  $X$  has the probability density function:

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0. \\ 0 & x < 0. \end{cases}$$

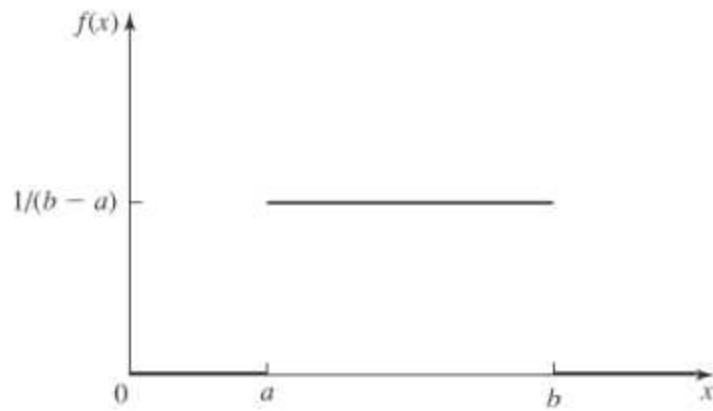
The cumulative distribution function,  $F(x) = \mathbf{P}\{X \leq x\}$ , is given by

$$F(x) = \int_{-\infty}^x f(y) dy = \begin{cases} 1 - e^{-\lambda x} & x \geq 0. \\ 0 & x < 0. \end{cases}$$

$$\bar{F}(x) = e^{-\lambda x}, \quad x \geq 0.$$

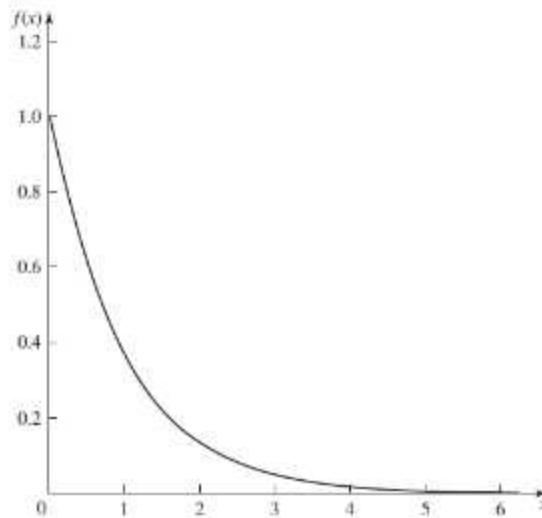


# Some Common Distributions (PDF)



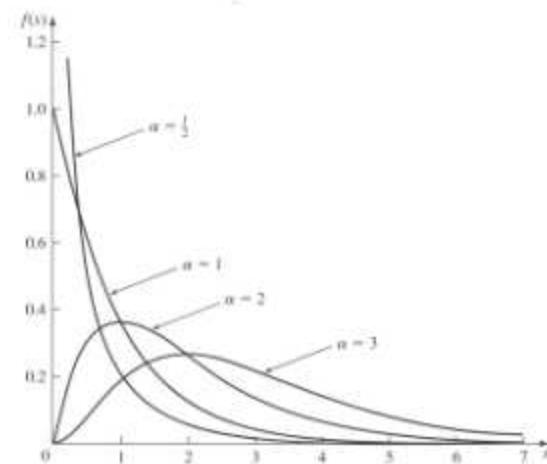
## Uniform

Used as a “first” model for a quantity that is felt to be randomly varying between  $a$  and  $b$  but about which little else is known



## Exponential

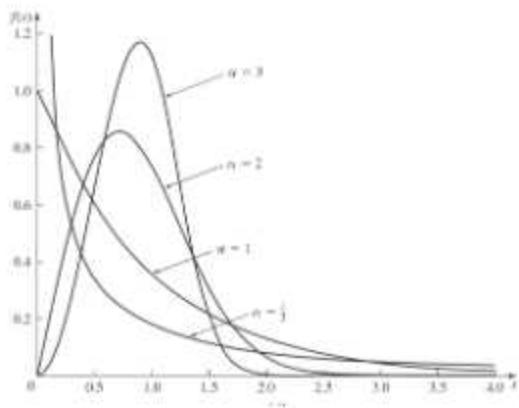
Interarrival times of “customers” to a system that occur at a constant rate, time to failure of a piece of equipment



## Gamma

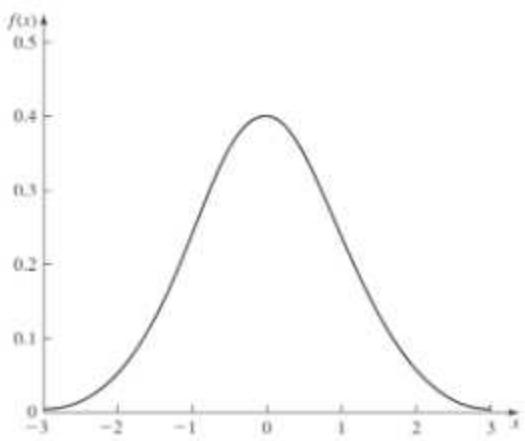
Time to complete some task, e.g., customer service or machine repair

# Some Common Distributions (PDF)



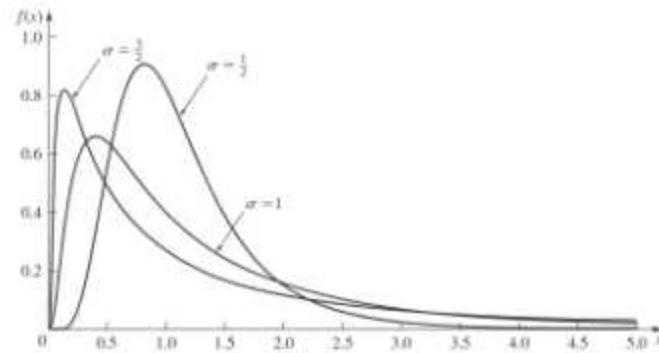
## Weibul

Time to complete some task, time to failure of a piece of equipment; used as a applications rough model in the absence of data



## Normal

Errors of various types



## Lognormal

Time to perform some task

# Distribution Fitting

Fitting distributions to data consists in **choosing a probability distribution modeling the random variable**, as well as **finding parameter estimates** for that distribution.

---

<https://medium.com/the-researchers-guide/finding-the-best-distribution-that-fits-your-data-using-pythons-fitter-library-319a5a0972e9>

<http://www.cs.unitn.it/~taufer/Readings/2014-JSS-fitdistrplus-An%20R%20Package%20for%20Fitting%20Distributions.pdf>

# Service Demands

Each request (arrival) is handled by one or more system resources (stations)

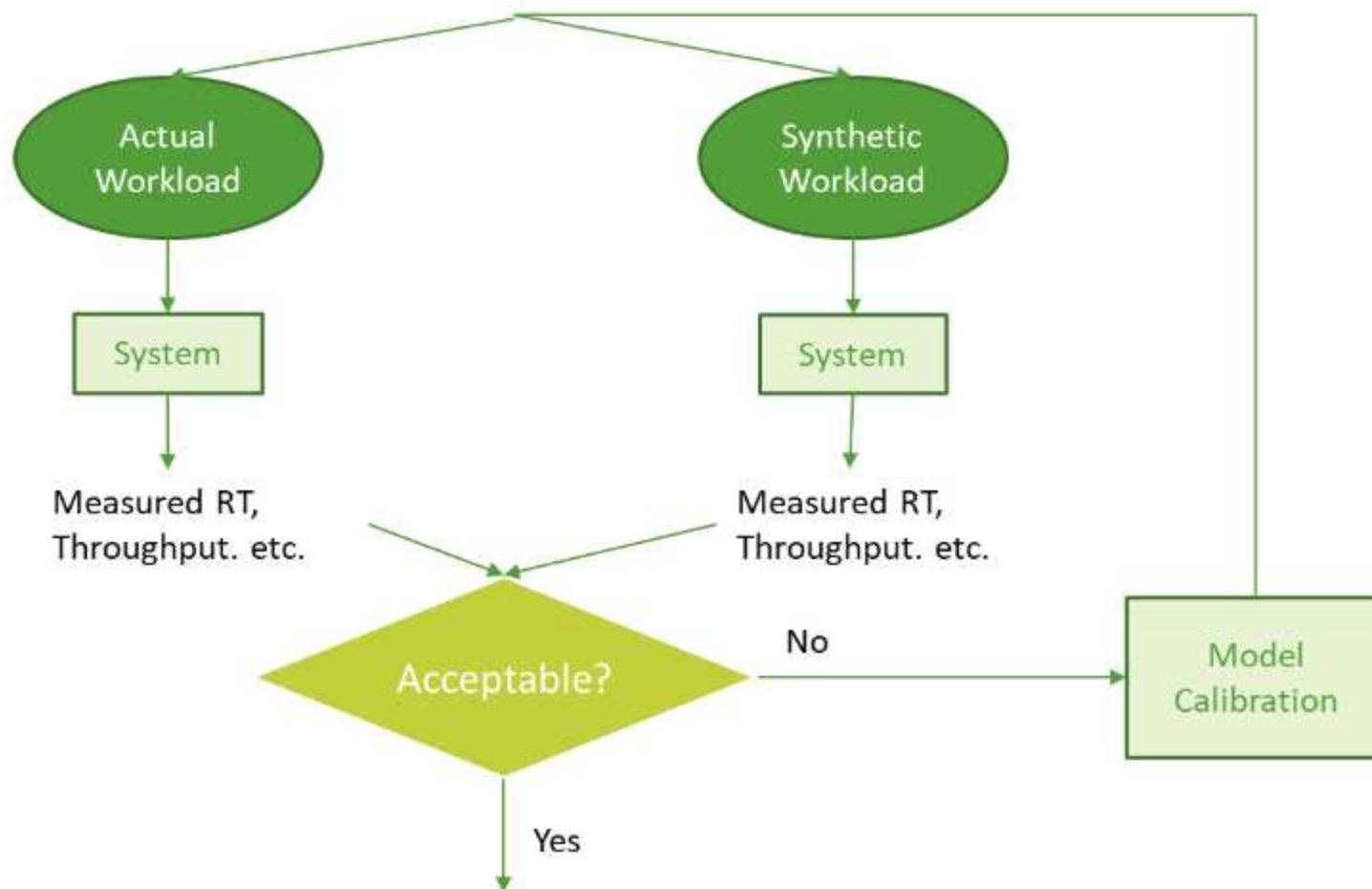
It has certain demands: access to a database, call a function, CPU time, memory usage etc.

Depending on the kind of system and on the purpose of the study different service demands can be considered

It must be understood which system components are interested by a request and how

e.g. a web service call may require an access to a database, a remote connection to another server etc.

# Workload model validation and calibration



# Workload forecasting

**Forecasting is the art and science of predicting future events.**

**Predicting how system workloads will vary in the future**

---

**It is a set of scenarios and assumptions**

- Evaluating the organization's workload trends;
- analyzing historical usage data;
- analyzing business or strategic plans;
- mapping plans into business processes

# References

- Chapter 3, 10 - D. A. Menascé, V. A. F. Almeida: *Capacity Planning for Web Services: metrics, models and methods*. Prentice Hall, PTR  
(Available in the library inside Dipartimento di Ingegneria informatica, automatica e gestionale Antonio Ruberti)
- Chapter 6 - A. M. Law - Simulation modeling and analysis  
<https://industri.fatek.unpatti.ac.id/wp-content/uploads/2019/03/108-Simulation-Modeling-and-Analysis-Averill-M.-Law-Edisi-5-2014.pdf>
- Calzarossa, Massari, Tessera. "Workload characterization: A survey revisited." ACM Computing Surveys (CSUR) 48.3 (2016): 1-43 <https://doi.org/10.1145/2856127>
- R. Jain, "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling,"  
[https://www.cin.ufpe.br/~rmfl/ADS\\_MaterialDidatico/PDFs/performanceAnalysis/Art%20of%20Computer%20Systems%20Performance%20Analysis%20Techniques.pdf](https://www.cin.ufpe.br/~rmfl/ADS_MaterialDidatico/PDFs/performanceAnalysis/Art%20of%20Computer%20Systems%20Performance%20Analysis%20Techniques.pdf)

For characterize a workload of a system mostly important to understand is the arrival pattern, how each request arrive in the system, how they are distribute in the time. Also important is the service demand, understand the involved component for each request. The usual way for do this is to use a random variable.

# Dependable Distributed Systems

## Master of Science in Engineering in Computer Science

AA 2022/2023

today characterize performance  
1 model that take in input the  
workload model defined

### LECTURE 18 : BUILDING A PERFORMANCE MODEL 1 – OPERATIONAL LAWS AND QUEUEING THEORY

# Introduction

Performance evaluation is required at every stage in the life cycle of a computer system:

- **Design:** compare alternatives designs and find the best design, possible organization for system
- **Operation:** determining how well it is performing and whether any improvements need to be made
- **Maintenance:** compare systems and decide which one is best for a given set of applications, make some update to the system

NOTE: the types of applications of computers are so numerous that it is not possible to have a standard measure of performance, a standard measurement environment (application), or a standard technique for all cases.

To measure the performance of a computer system, you need at least two tools:

- a tool to load the system (load generator) ↳ something that send inputs to the system to perform measurement.
- a tool to measure the results (monitor).

↗ before construct a model

↳ how arrival coming to the system, need some measure

# How to do performance evaluation?

- Measurement (Benchmarking)
- Analytical modeling
- Simulation modeling

just get the system, apply the output,  
check the measure utilization of a resource  
take measure of the actual system.

Measurements are possible only if the system already exists

→ during design phase, development

If it is a new concept, analytical modeling and simulation are the only techniques from which to choose

It would be more convincing to others if the analytical modeling or simulation is based on previous measurement

# How to do performance evaluation?

- Measurement (Benchmarking)
- Analytical modeling
- Simulation modeling

need to build a sort of program  
that try to mimic all the behaviour of  
an actual system.

Criterion	Analytical Modeling	Simulation	Measurement
1. Stage	Any <i>(also design)</i>	Any	Postprototype
2. Time required	Small	Medium	Varies
3. Tools	Analysts	Computer languages	Instrumentation
4. Accuracy <sup>a</sup>	Low <i>→ at least understand a lower bound</i>	Moderate	Varies
5. Trade-off evaluation	Easy	Moderate	Difficult
6. Cost	Small	Medium	High
7. Saleability	Low	Medium	High

# Performance Models

A model is **an abstraction** of a generalized overview of a real system

The **level of detail** of the model and the specific aspects of the real system that are considered in the model **depends on the purpose** of the model

→ should **not** be more complex than is necessary to achieve its goal

**Analytical** models: **set of formulas** and/or **computational algorithms**

- Lower level of detail -> Less accurate but more efficient

**Simulation** models: **computer programs**, **all resources and the dataflow are simulated**

- Higher level of detail -> **Expensive to run, develop and validate**

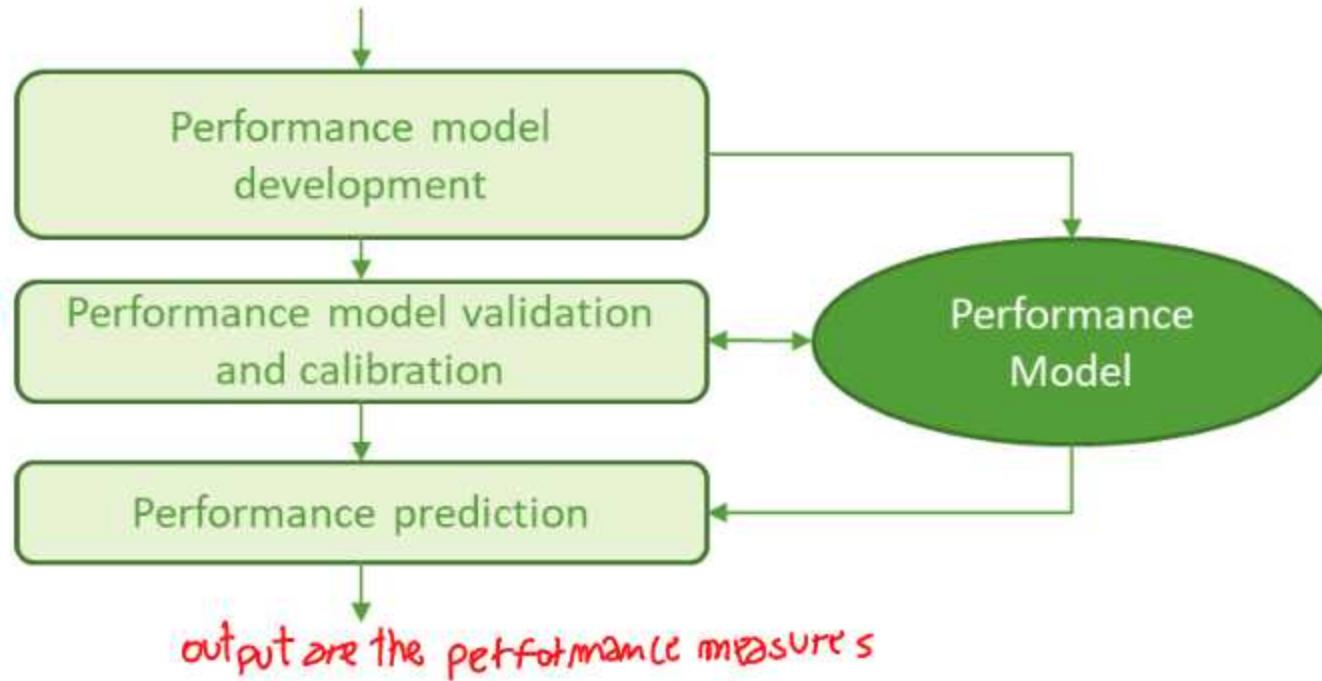
There are **some** system behaviours that analytic models cannot (or very poor) capture

# Performance model

Used to predict performance as function of system description and workload parameters

build model

check if the model  
is valid or not



output are the performance measures

Estimates performance measures of a computer system for a given set of parameters

Outputs: response times, throughputs, system resources utilizations, queue lengths, etc.

# Estimating Performance

general information of the system

## System Description

- System parameters
- Resources parameters *in terms of time*
- Workload parameters
  - service demands
  - workload intensity
  - burstiness

## Performance Measures

- Response time
- Throughput
- Utilization
- Queue length
- ...

## Performance Model

# Basics for performance evaluations



The performance of a system that gives services could be seen from two different viewpoints:

- **user**: the *time* to obtain a service or the waiting time before getting a service; (like the *response time*)
- **system**: the number of users served in the unit time or the resources utilization level.

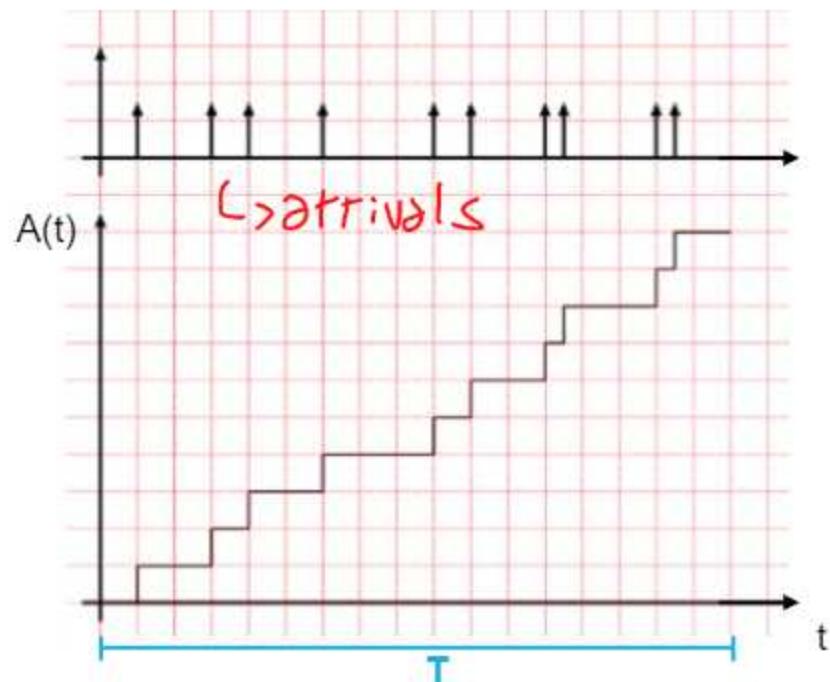
# Measurable Quantities



T: system *observation interval*;

↳ time interval which we repeat

A: *number of arrivals in the period T*;



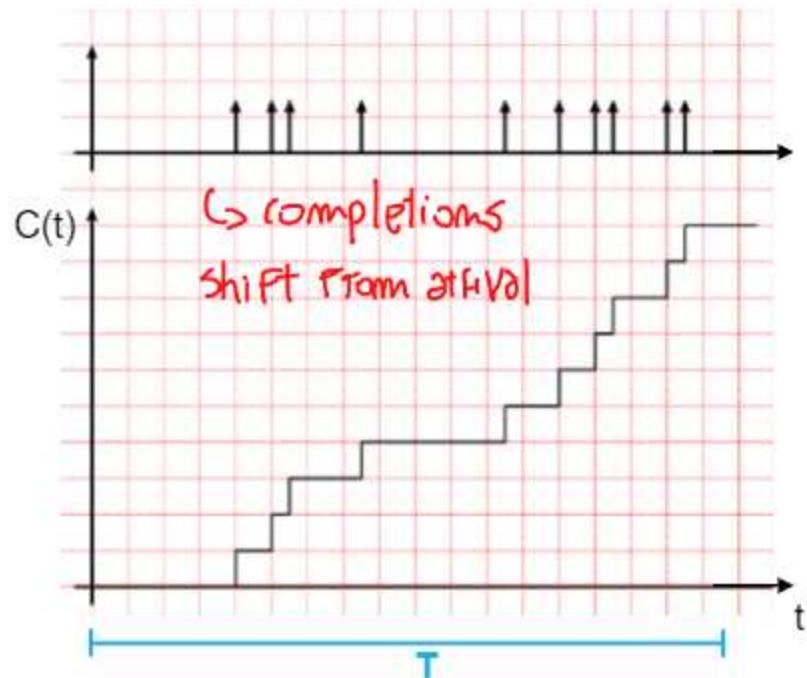
# Measurable Quantities



$T$ : system *observation interval*;

$C$ : *number of completions in the period  $T$* ;

*number of replies get out of system*



# Basic Relations (Valid for Every Performance Model)

T: system *observation interval*

A: *number of arrivals* in the period T

C: *number of completions* in the period T

$\lambda$ : *arrival rate*  
Average

$$\lambda = \underline{A/T}$$

how frequent the request arrive to out by stem.

X: *throughput*

$$\underline{X = C/T}$$

number of reply made in T, divide the time window

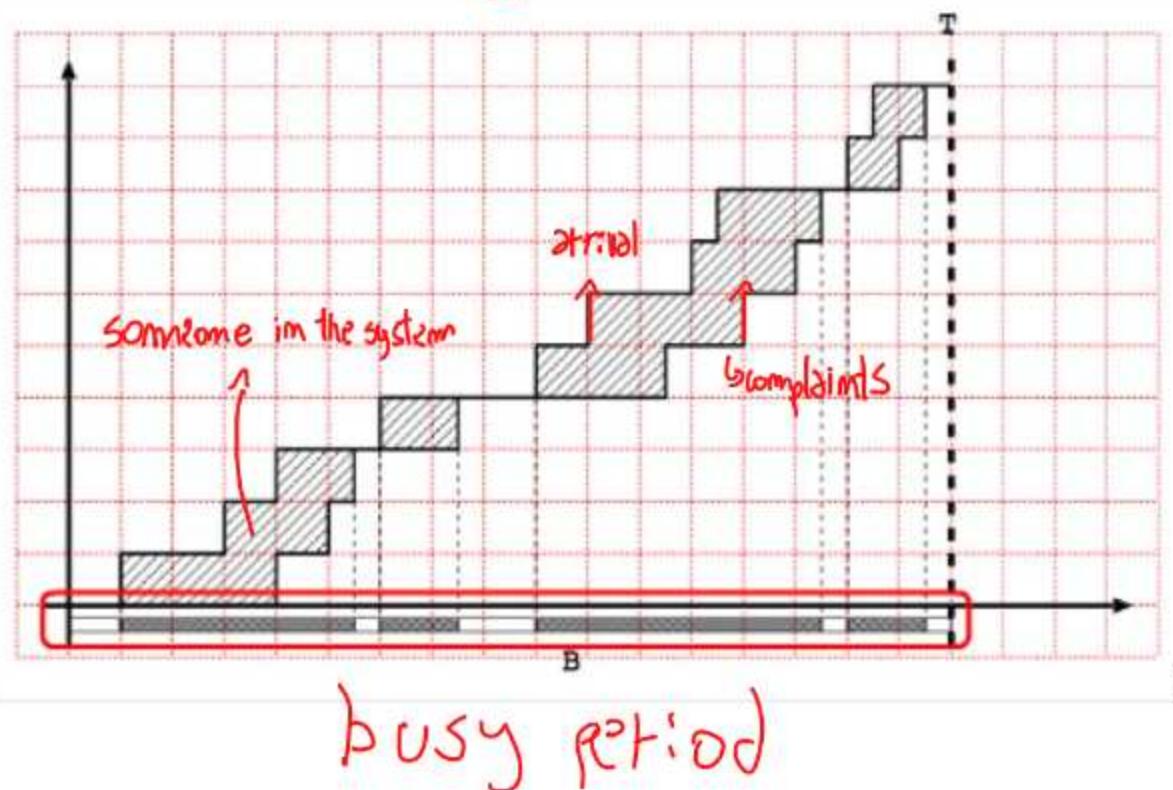
If  $A = C$ , i.e. the system is able to serve all its requests during the observation interval with no losses, the two quantities are equal.

$$\lambda = X$$

# Utilization Law

B: busy period time in the period T;

Union of the two graphs



# Utilization Law

**B:** busy period time in the period T;

**U:** system/component utilization

**S:** average service time per completion

**X:** throughput     $X = C/T$

Percentage of the time where  
doing something

$$U = B/T$$

$$S = B/C$$

$$U = \frac{B}{T} = \frac{B/C}{T/C} = \frac{S}{1/X} = S \cdot X$$

Utilization Law

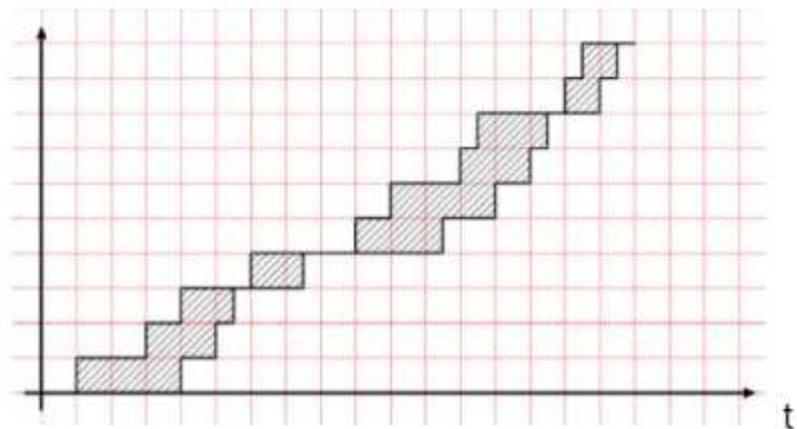
# Little Law



How many requests are in the system?

W: "the sum of the *amount of time* spent by all the request in the system in T"

Watered inside grey squares



If make horizontal line you understand how many requests are in the system in every period of time.

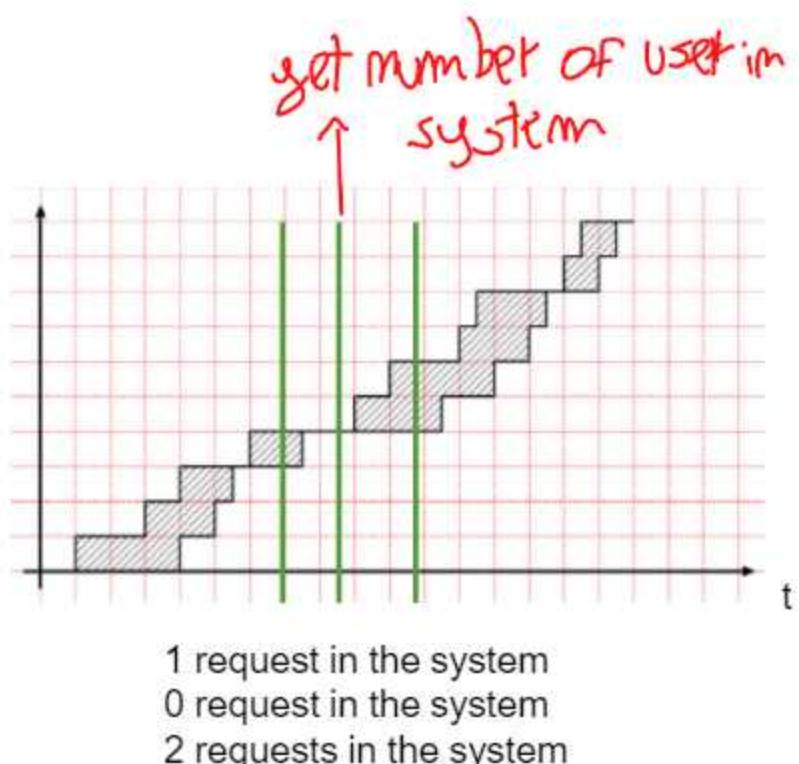
# Little Law

**W:** "the sum of the *amount of time* spent by *all the requests* in the system in  $T$ "

*How many requests are in the system?*

**N:** *average number of requests in the system*

$$\underline{N = W/T}$$



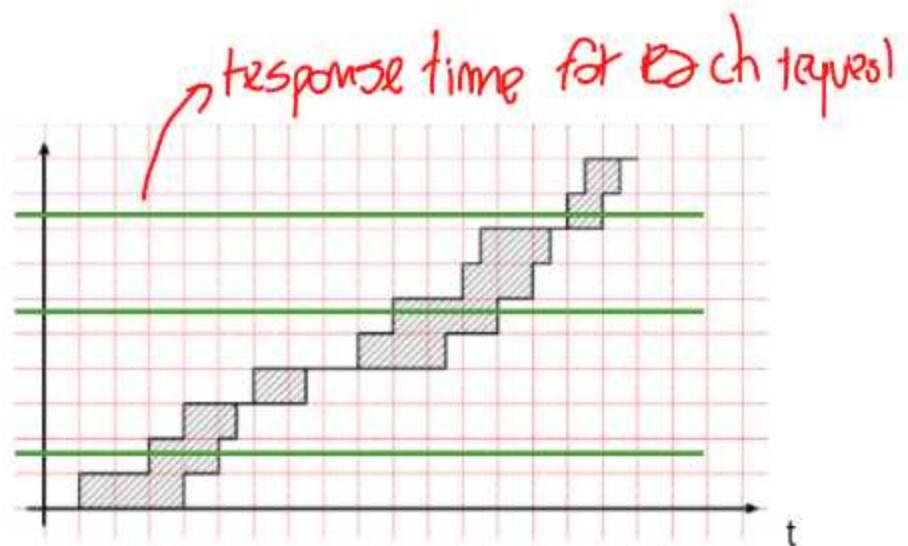
# Little Law

**W:** "the sum of the *amount of time* spent by all the requests in the system in  $T$ "

How many requests are in the system?

**R:** average *response time* (or *residence time*),  
*waiting time* + *service time*

$$R = W/C$$



When send a request, when receiving a reply

Waiting time: wait for get an answer

Service time: time for process the request

# Little Law (Recap.)

W: "the sum of the *amount of time spent by all the requests in the system in T*"

N: *average number of requests in the system*  $N = W/T$

R: *average response time*  $R = W/C$

X: *throughput*  $X = C/T$

How many requests are in the system?

$$N = \frac{W}{T} = \frac{W/C}{T/C} = \frac{R}{1/X} = R \cdot X$$

Little Law

## Common Performance Measures

If the system performs the service correctly, its performance is measured by the time taken to perform the service, the rate at which the service is performed, and the resources consumed while performing the service.

The utilization gives an indication of the percentage of time the resources of the gateway are busy for the given load level. The resource with the highest utilization is called the bottleneck.

# Common Performance Measures

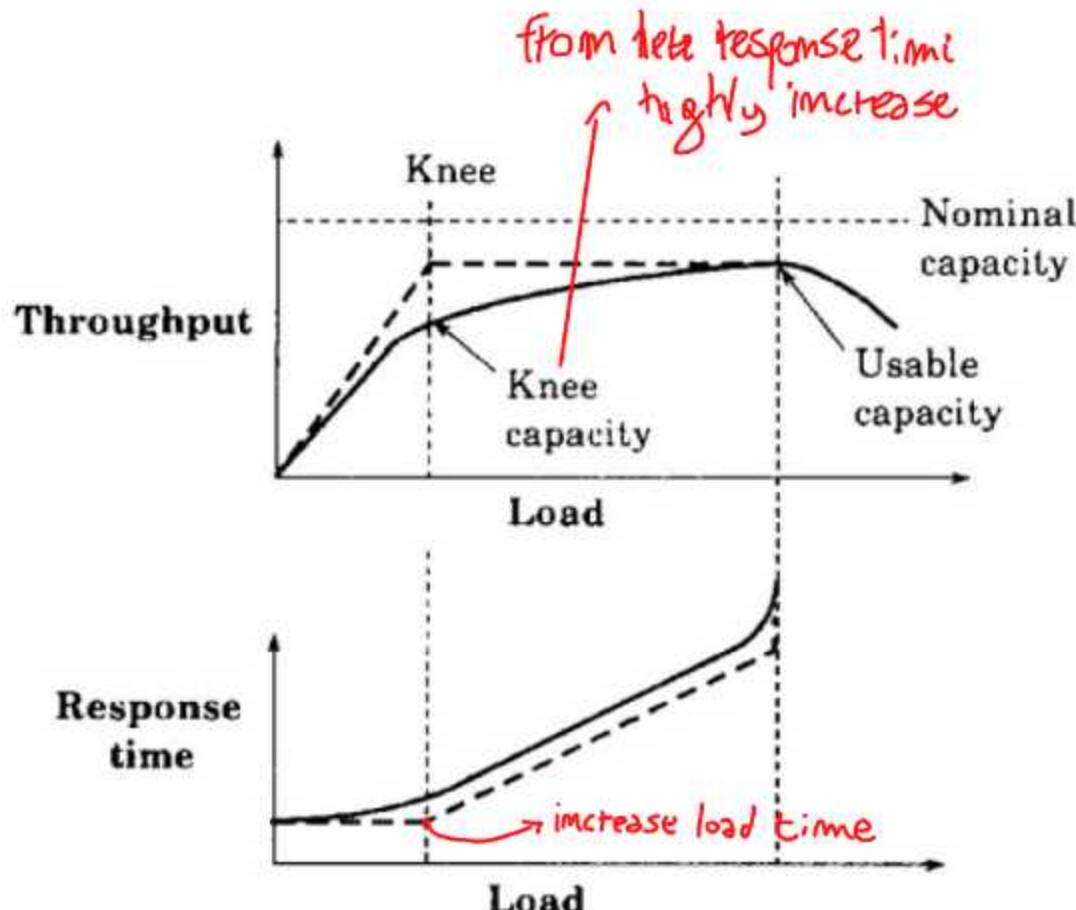
The maximum achievable throughput under ideal workload conditions is called **nominal capacity** of the system.

Often the response time at maximum throughput is too high to be acceptable. In such cases, it is more interesting to know the maximum throughput achievable without exceeding a prespecified response time limit.

This may be called **the usable capacity** of the system.

In many applications, **the knee of the throughput or the response-time curve** is considered the optimal operating point.

This is the point beyond which the response time increases rapidly as a function of the load but the gain in throughput is small.



## Stability condition

↳ system must be fast in processing requests to respect the state of requests arriving in the system.

B: busy time

U: utilization  $U = B/T$

$$B \leq T \Rightarrow U = B/T \leq 1$$

every resource can't work out each time interval

Serve all the requests,  $A = C \Rightarrow \lambda = X$

if  $\lambda > s$  the response time increases indefinitely

$$X \cdot S = \lambda \cdot S \leq 1$$

$$\lambda \leq \frac{1}{S} \text{ or } S \leq \frac{1}{\lambda}$$

## Recap Operational Laws (valid in all performance models)



$\lambda$ : arrival rate,  $X$ : throughput

$S$ : average service time per completion

$R$ : average response time (or residence time), waiting time + service time

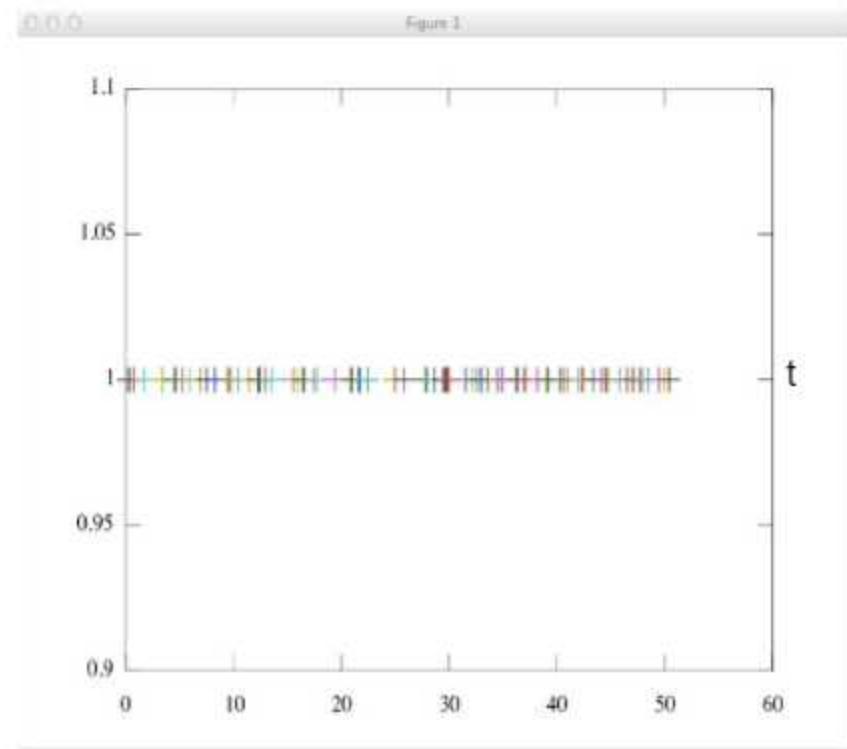
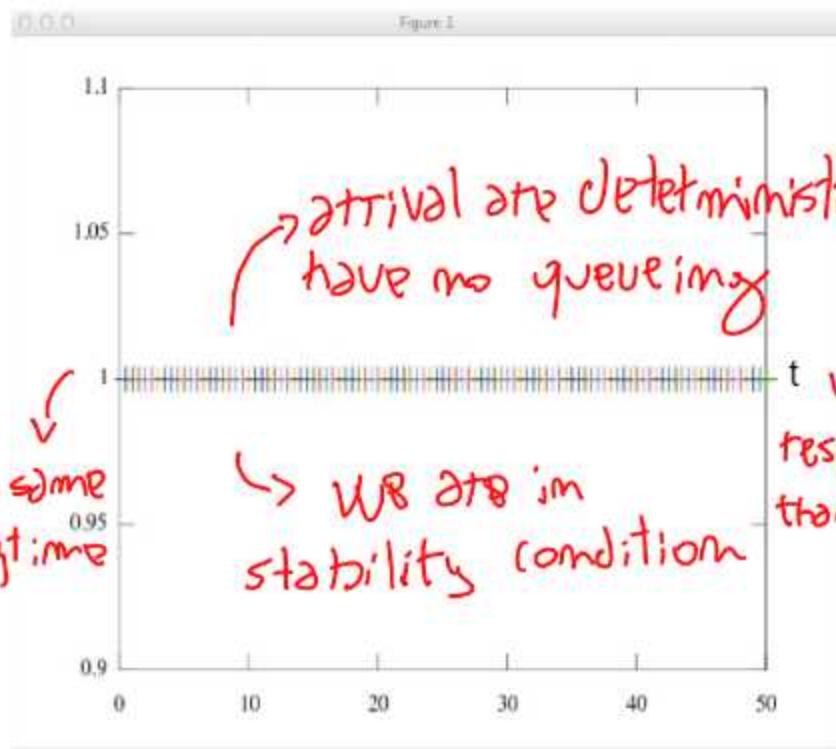
$U = S \cdot X$  (Utilization Law)

$N = R \cdot X$  (Little Law, number of requests IN the system)

**Stability condition:**  $\lambda \leq 1/S$

# Arrivals examples

$\lambda$  is not enough to characterize the workload of a system.

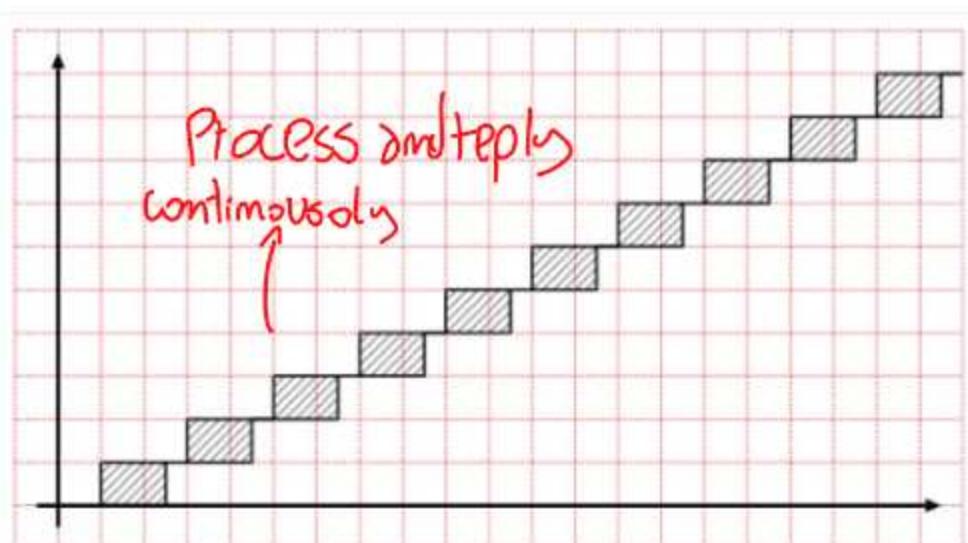


The arrival rate  $\lambda$  in the two workload is identical

but different pattern

# Deterministic Inter-arrival

only in deterministic  $\rightarrow$  no queues



Service time is assumed constant in the figure

Jobs never queues

$$U = \lambda \cdot S$$

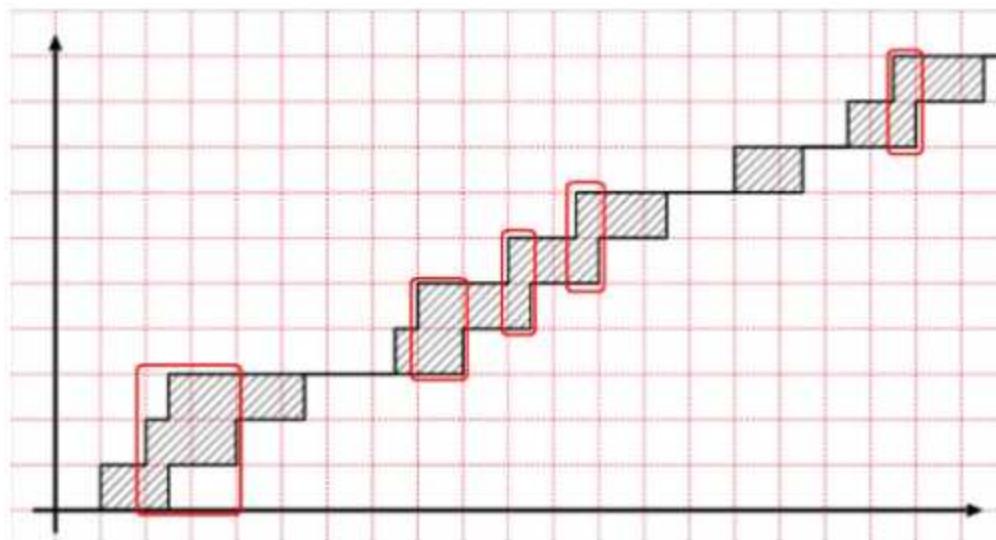
$$N = \lambda \cdot S$$

$$X = \lambda$$

$$R = S$$

only in synchronous system have best performances

# Random Inter-arrival



Service time is assumed  
constant in the figure

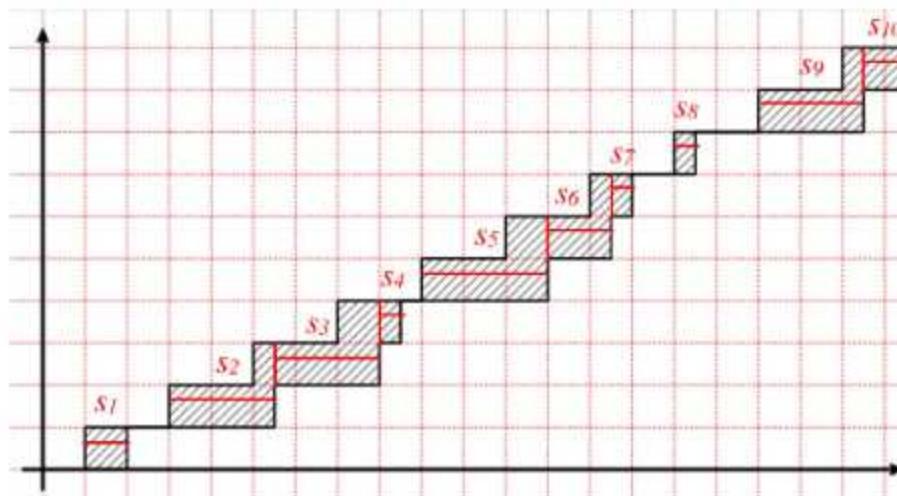
**Non-determinism  
in the arrivals  
creates queuing**

# Service Features

The same as arrivals:

- Their **rate** (i.e. how fast requests are served)
- Their **regularity** (i.e. the time that passes between two served requests)
- Their **correlation** (informally, subsequent service time are correlate?)

not always deterministic the time for reply



# Analytical Performance Model: Queuing Model

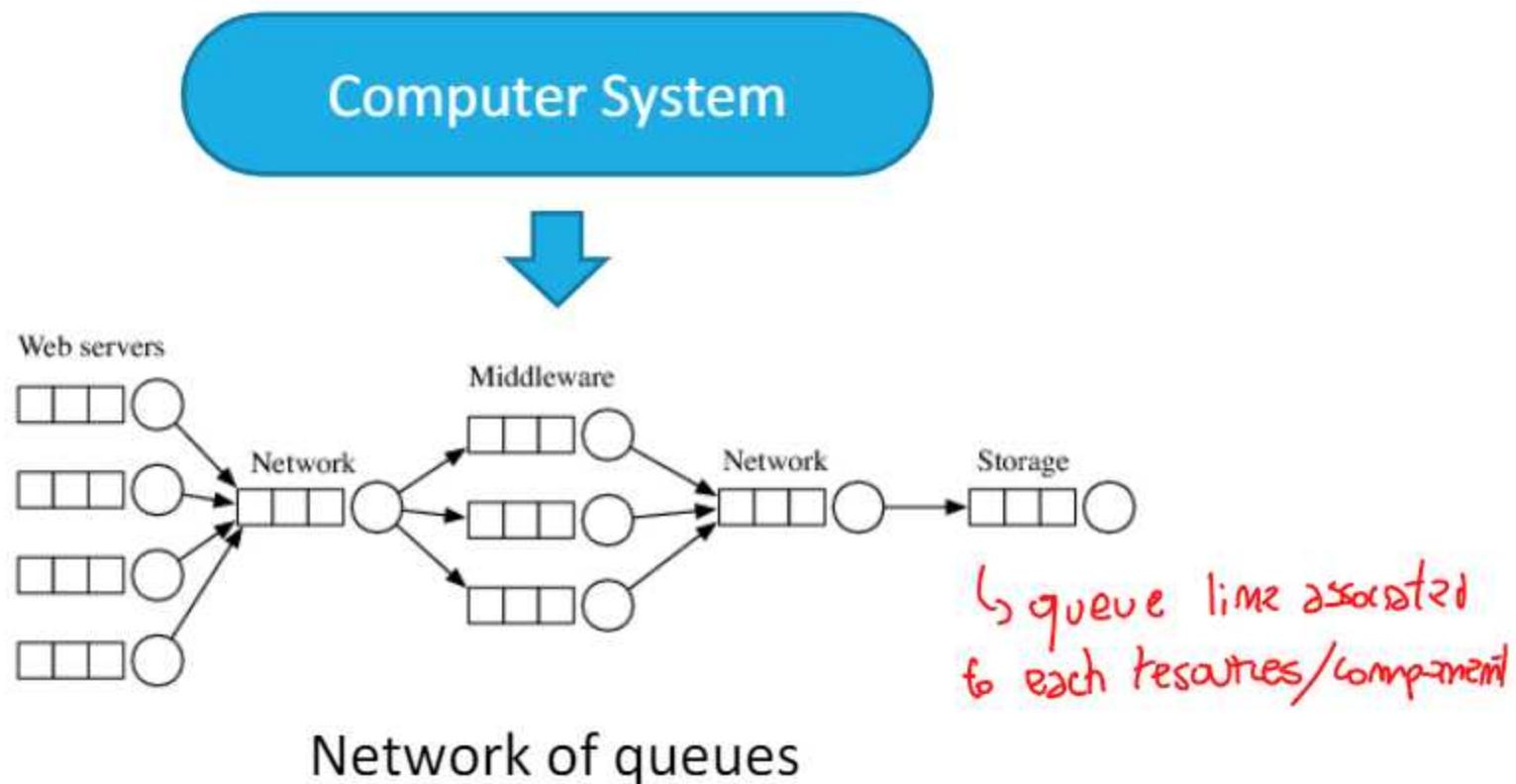
Computer systems, including software systems, are composed of a collection of resources (e.g., processors, disks, communication links, process threads, critical sections, database locks) that are shared by various requests (e.g., transactions, web requests, batch processes).

Usually, there are several requests running concurrently and many of them may want to access the same resource at the same time.

Since resources have a finite capacity of performing work (e.g., CPU finite number of instructions per seconds, disk transfers a certain amount of data per second, a communication link ...) waiting lines often build up in front of these resources.

→ how many user can wait inside your system Concurrently

# Queuing Model



# Exponential Distribution

assume all request are independent among the model

We say that a random variable  $X$  is distributed Exponentially with rate  $\lambda$ ,

$$X \sim \text{Exp}(\lambda)$$

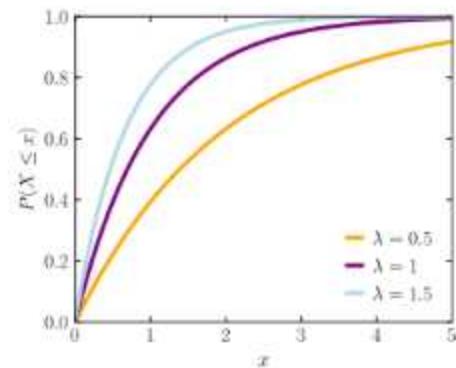
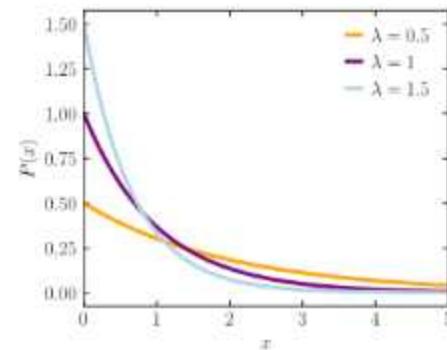
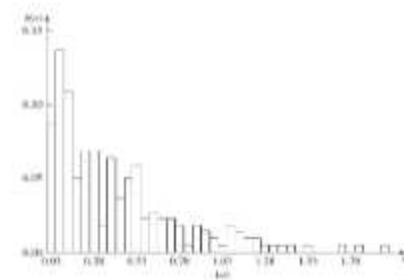
if  $X$  has the probability density function:

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0. \\ 0 & x < 0. \end{cases}$$

The cumulative distribution function,  $F(x) = \mathbf{P}\{X \leq x\}$ , is given by

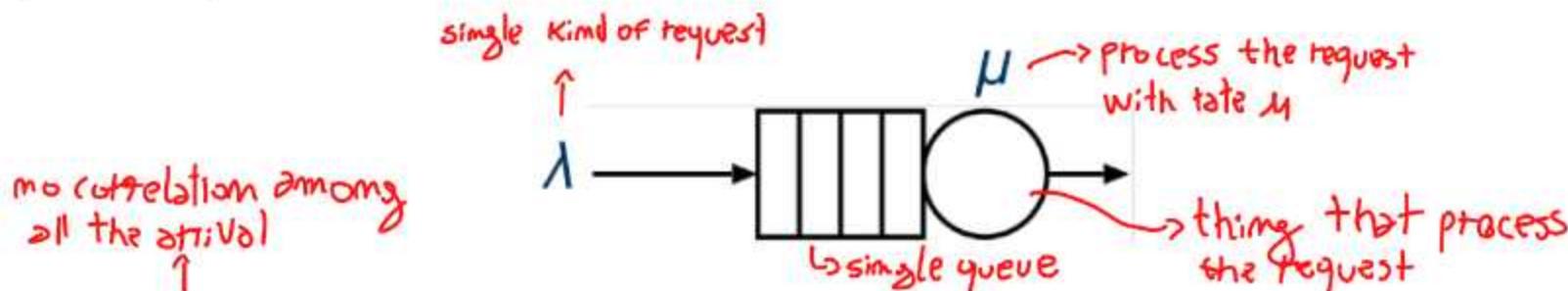
$$F(x) = \int_{-\infty}^x f(y) dy = \begin{cases} 1 - e^{-\lambda x} & x \geq 0. \\ 0 & x < 0. \end{cases}$$

$$\bar{F}(x) = e^{-\lambda x}, \quad x \geq 0.$$



# Single Open Queue – M/M/1

→ how many servers accommodate the request



- The inter-arrival time of the request is distributed in accordance to the exponential distribution, with arrival rate equal to  $\lambda$  (**M**) (**Poisson process**)  
(Poisson process, i.e., a process in which events occur **continuously and independently** at a constant average rate)
- The service time is distributed in accordance to the exponential distribution (load independent), with service time equals to  $1/\mu$  (**M**)
- **One single server (1)**
- The requests are statistically indistinguishable: single class, or homogeneous workload, or single workload component
- The server does not refuse any request: **infinite queue**

# Note

**$\lambda$  and  $\mu$  are rates** (respectively, arrival rate and service rate)

e.g.  $\lambda = 10$  requests/sec,  $\mu = 15$  requests/sec

The average inter-arrival time and the average service time are multiplicative inverse (reciprocal)

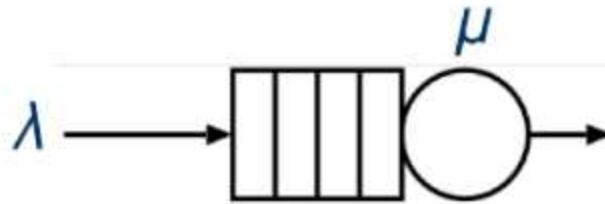
**$s = 1/\mu$**

**Average inter-arrival time =  $1/\lambda$**

$\lambda = 10$  requests/sec  $\rightarrow$  average inter-arrival time =  $1/\lambda = 0,1$  seconds

$\mu = 15$  requests/sec  $\rightarrow$  average service time =  $1/\mu = 0,067$  seconds

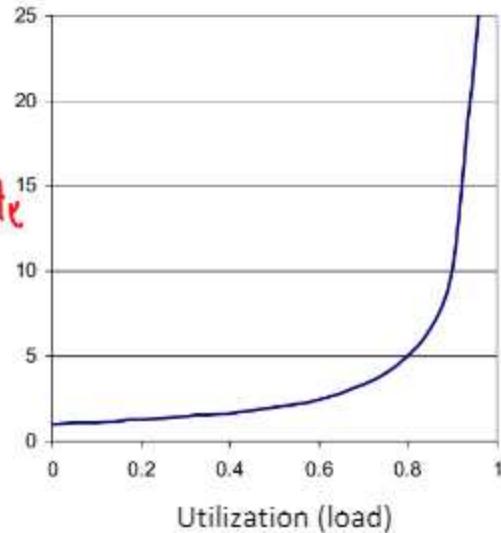
# Single Open Queue – M/M/1



$$R = \frac{1}{\mu \cdot \left(1 - \frac{\lambda}{\mu}\right)} = \frac{1}{\mu - \lambda}$$

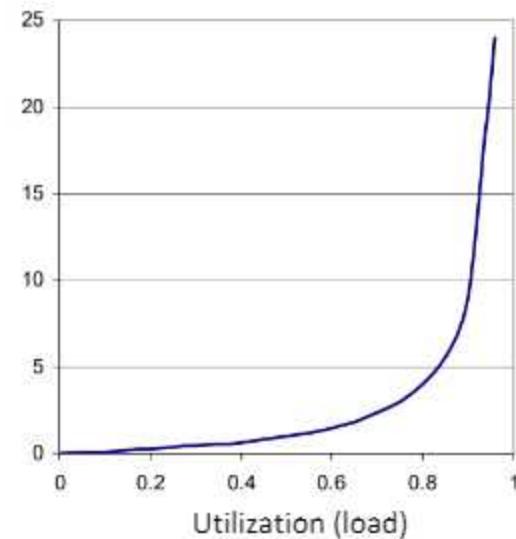
$$U = \rho = 1 - p_0 = \frac{\lambda}{\mu}$$

Expected utilization



Expected response time

$$N = \frac{\rho}{1 - \rho}$$

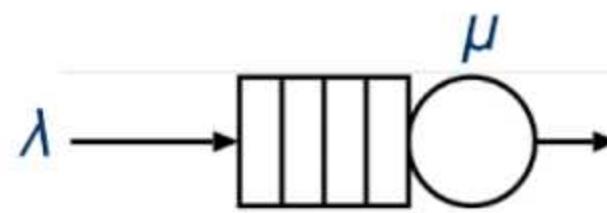


Expected number of requests in the system

## Kendall's notation

A special notation, composed by 3 to 6 terms, is used to identify  
the behavior of a single queue

$A / S / c [/ k [/ p]] [/ Q]$



# Kendall's notation

A / S / c [ / k [ / p ] ] [ / Q ]

The first term defines the arrival process

A

The second corresponds to the service time distribution.

S

c reports the number of servers → M/M/1

↑

Most common acronyms for  
Inter-arrival and service time  
distributions:

- M • exponential / Poisson (Markov)
- D • Deterministic
- E<sub>k</sub> • Erlang with  $k$  stages
- G • General

# Kendall's notation

A / S / **c** [ / **k** [ / **p** ] ] [ / **Q** ]

optionals, otherwise the default

↳ number of servers at the end of single line

**k** the maximum capacity of the station (queues + service centers)

**p** If the population from which the jobs that enter the system is finite

**Q** The Service Discipline or Priority, order that jobs in the queue:

**FIFO or FCFS**

- First-in first-out

**LIFO or LCFS**

- last-in last-out

**SIRO**

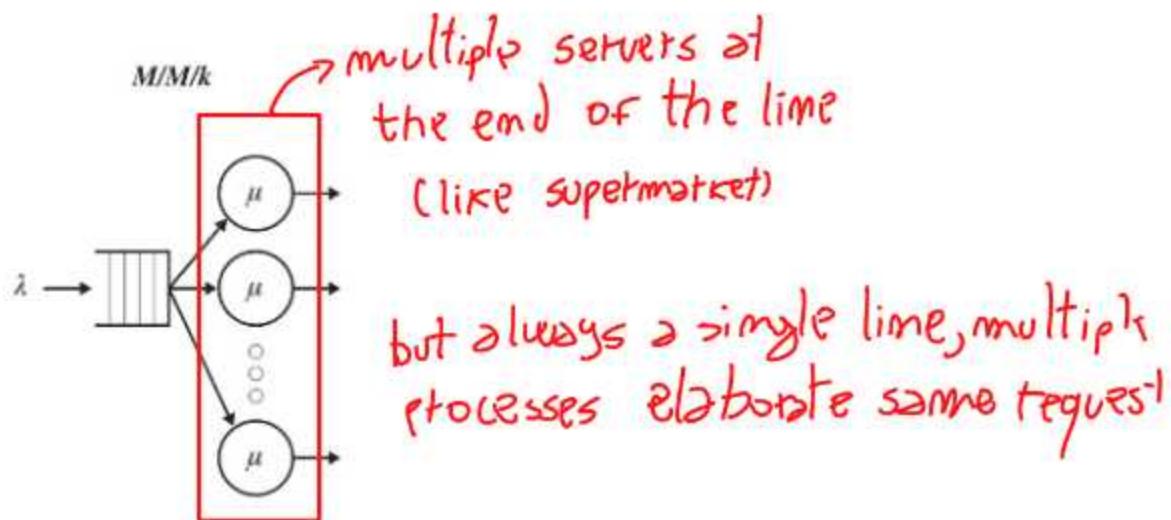
- service in random order

**PS**

- processor sharing

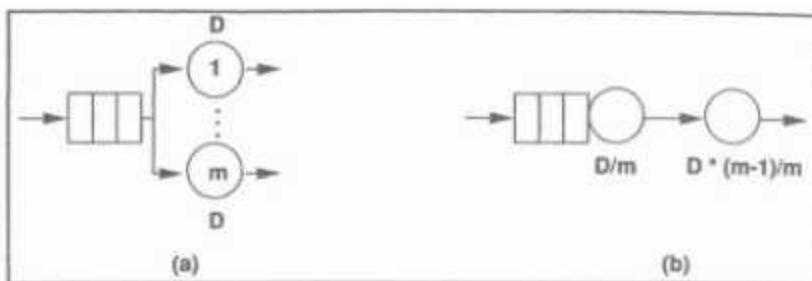
Default values are: **k =  $\infty$** , **p =  $\infty$** , **Q = FIFO**

# M/M/k



$$E[T]^{M/M/k} = \frac{1}{\lambda} \cdot P_Q \cdot \frac{\rho}{1 - \rho} + \frac{1}{\mu}$$

where  $\rho = \frac{\lambda}{k\mu}$ , and  $P_Q$  is the probability an arrival is forced to queue.

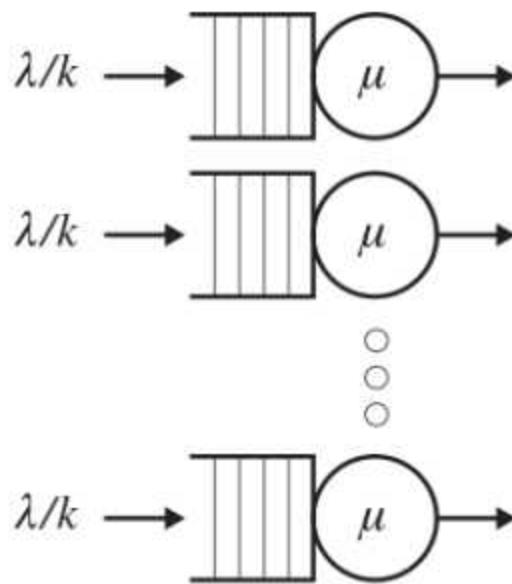


Approximation for multiple servers

# Comparing Server Organizations

less efficient

Frequency-division  
multiplexing



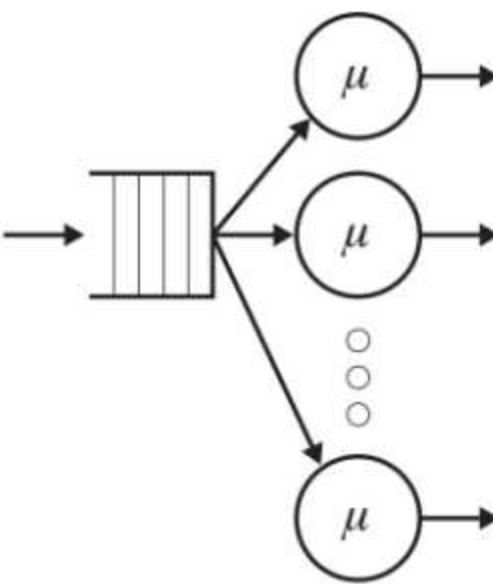
different services  
with split of requests

$$\rho = \frac{\lambda}{k\mu} \quad \text{same load}$$

$M/M/1$   
vertical scaling

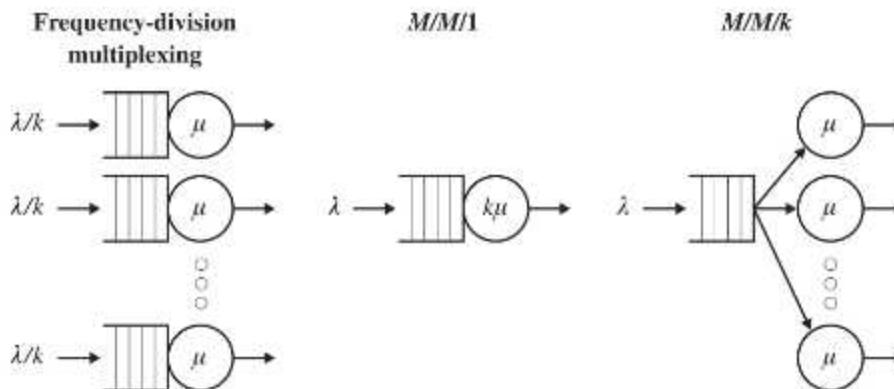


$M/M/k$   
single line - multiple servers



Which solution provides lower response time R ?

# Comparing Server Organizations



$$\mathbf{E}[T]_{\text{FDM}} = \frac{1}{\mu - \frac{\lambda}{k}} = \frac{k}{k\mu - \lambda}.$$

$$\mathbf{E}[T]_{\text{M/M/1}} = \frac{1}{k\mu - \lambda}.$$

$$\mathbf{E}[T]_{\text{M/M/k}} = \frac{1}{\lambda} \cdot P_Q \cdot \frac{\rho}{1 - \rho} + \frac{1}{\mu}$$

first one usually take more time

Response time:

- FDM > M/M/1

- Low load, M/M/1  $k$  times < M/M/k

- High load, M/M/1  $\approx$  M/M/k

equivalent

low load M/M/1 faster

# Comparing Server Organizations

Example

$$\mathbf{E}[T]_{\text{FDM}} = \frac{1}{\mu - \frac{\lambda}{k}} = \frac{k}{k\mu - \lambda}.$$

$$\mu = 10 \text{req/sec}$$

$$\mathbf{E}[T]_{\text{M/M/1}} = \frac{1}{k\mu - \lambda}.$$

$$\lambda = 30 \text{req/sec}$$

$$\mathbf{E}[T]_{\text{M/M/k}} : \xrightarrow{\quad \quad \quad} \boxed{\text{D/m}} \rightarrow \text{D} \rightarrow \text{D} \cdot (m-1)/m$$

$$K = 4$$

$$\rho = \frac{30}{4 \cdot 10} = 3/4$$

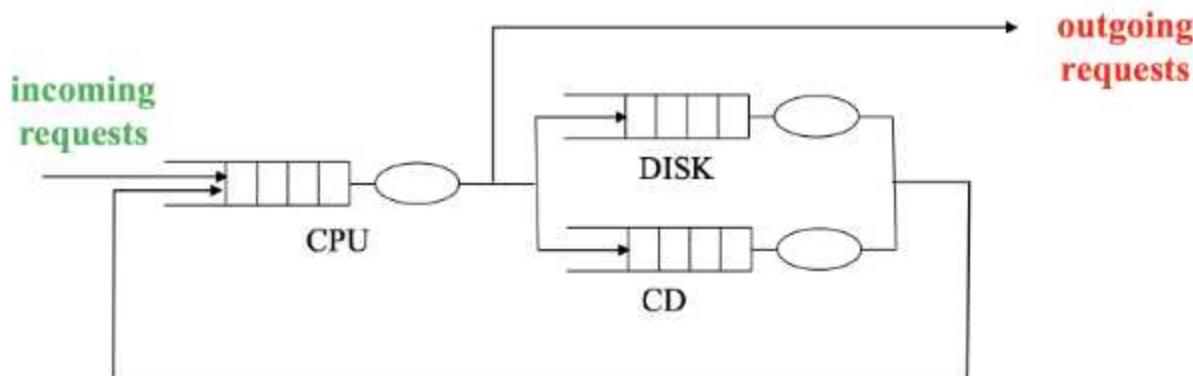
$$\text{FDM} = \frac{4}{4 \cdot 10 - 30} = 0,4 \text{ sec}$$

$$\text{M/M/1} = \frac{1}{40-30} = 0,1 \text{ sec}$$

$$\begin{aligned} \text{M/M/k} &= \frac{1}{40-30} + \frac{1}{10} \cdot \frac{3}{4} \\ &= 0,1 + 0,075 = 0,175 \end{aligned}$$

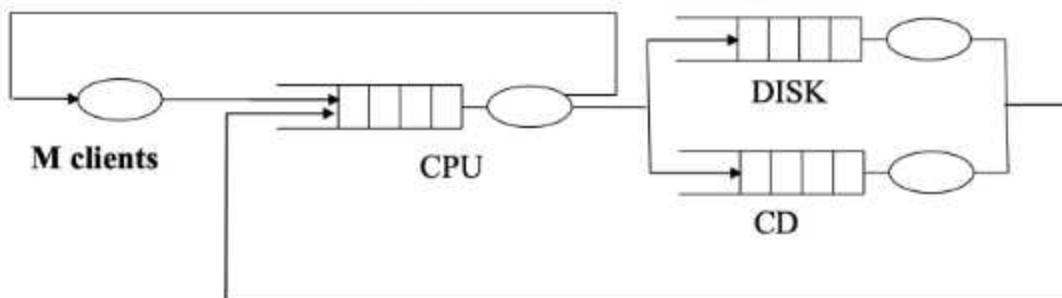
# Queuing Networks

## Open queuing networks



## Closed queuing networks

(finite number of users)



# Open Jackson Networks (simplified)

An open queueing network composed by M/M/k queues can be analyzed piecewise, namely the performance metrics of the single queues can be computed independently and the performance metrics for the entire network can be obtained by composition

All queues receives the same arrivals at rate  $\lambda$

↳ analyze all the networks independently

- \_> it is possible to assume that a request passes through a queue multiple times
- \_> it is possible to associate probabilities to the arrows interconnecting queues

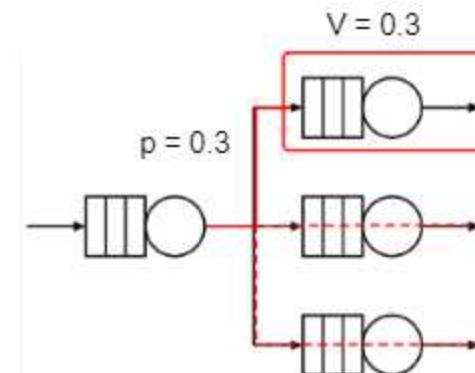
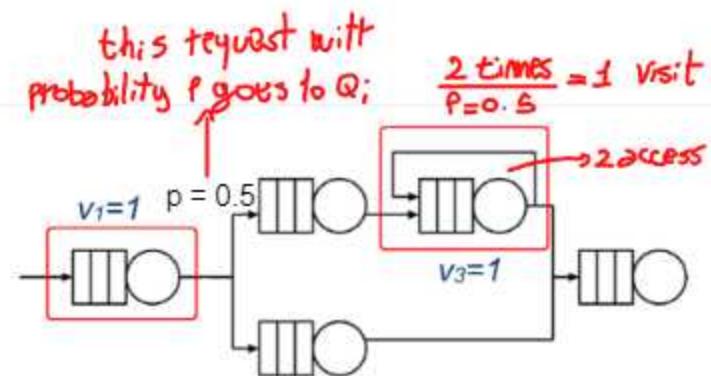
# Visits

The visits represent the average number of times a request pass through a station  $i$  for the moment it enters the system to the time it leaves

$$V_i = \frac{C_i}{C}$$

$V_i$  can be less than one if there could be paths that skips the considered node  $k$ .

Note that  $V_i = 1$  means that station  $i$  is visited on the average once during the service of one request: this does not exclude that the considered station can be skipped or that it can be re-entered by the job during its service.



# Service Demand

The **service demand**  $D_i$  of a queue  $i$  accounts for the **average time spent by a request** to the considered service center **during all its visits**, taking into account the fact that such resource might also be skipped.

$$D_i = \frac{B}{C} = \frac{B}{C} \cdot \frac{C_i}{C_i} = V_i \cdot S_i$$

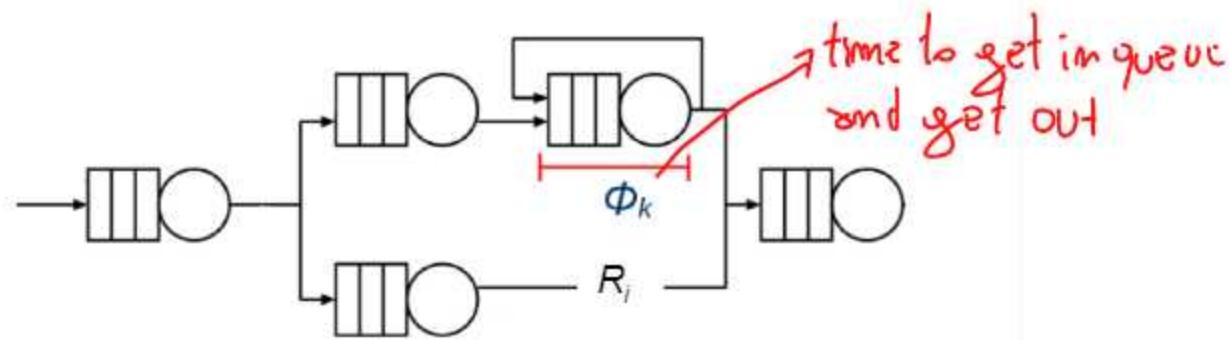
↳ demand of a single request to each specific subqueues

Note that:

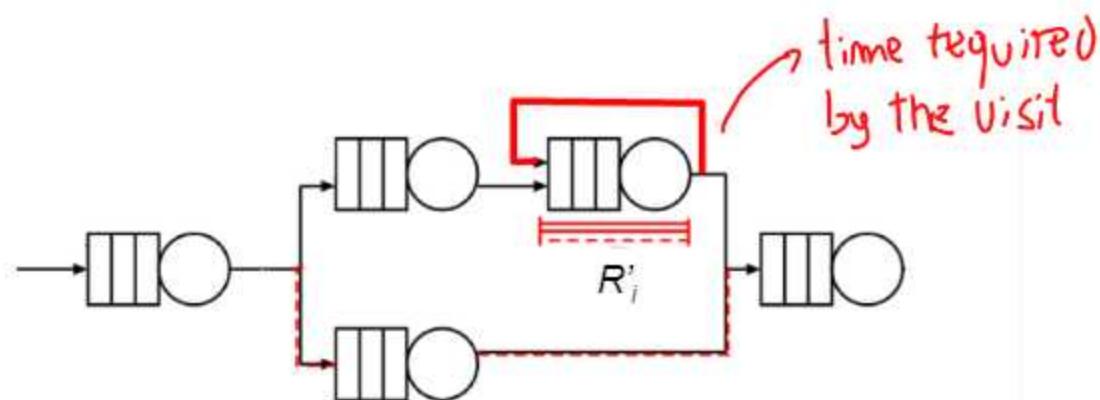
- Average **service time**  $S_i$  accounts for the **average time that a request spends in queue  $i$  when it is served**.
- Average **service demand**  $D_i$  accounts for the **average time a job spends in station  $i$  during its staying in the system**.

# Response Time and Residence Time

$R_i$  average **response time** is the average time spent in queue  $i$  when a request enters the station



$R'_i$  average **residence time** is the average time spent by a request at queue  $i$  during its staying in the system



It can be greater or smaller than the response time depending on the number of visits.

# Response, Residence, Service, and Demand

Note that the relation between Residence Time and Response Time is the same as the one between Demand and Service.

$$D_i = V_i \cdot S_i$$

$$R'_i = V_i \cdot R_i$$

Residence = visit<sub>i</sub> · response time

Also note that for single queue network  $V_i = 1$ . This implies that the average service time equals the service demand, and the response time and the residence time are identical.

For this reason when referring to models with a single station, demand and service times, or residence and response time can be used as synonymous.

## Little and Utilization Laws with Visits

$$U_i = X \cdot D_i = X \cdot V_i \cdot S_i = X_i \cdot S_i$$

$$N_i = X \cdot R'_i = X \cdot V_i \cdot R_i = X_i \cdot R_i$$

# Network Performance Indices

The workload of open networks is characterized by the arrival rate  $\lambda$

If the network is stable, the arrival rate corresponds to the system throughput

$$X = \lambda$$

The average total number of requests in the system can be computed as the sum of the average number of requests at each station:



$$N = \sum N_i$$

# Network Performance Indices

The average Response Time represents the average time spent by the requests in the system: it can be computed as the sum of the Residence Times of the jobs at all the stations

$$R = \sum R'_i$$

Note the discrepancy: the system **response time** is the sum of **residence times** and not of the response times at the stations.

Note that Little's law continues to be valid system-wise using the network performance indices:

$$N = \sum N_i = \sum X \cdot R'_i = X \cdot \sum R'_i = X \cdot R$$

# Single Queue Analysis in Open Jackson Network

$$R_i = \frac{S_i}{1-U_i}$$

Response time i

$$N_i = \frac{U_i}{1-U_i}$$

Number of requests i

$$U_i = \lambda \cdot V_i \cdot S_i$$

Utilization/load i

$$R'_i = V_i \cdot R_i = \frac{D_i}{1-U_i}$$

Residence time i

$$R = \sum R'_i$$

Response time network

# Bottleneck Identification (open network)

↳ For reduce response time, must identify bottleneck and do something there

In an open network the average frequency of users incoming into the network is fixed. For  $\lambda$  too much big the network will become unstable

$$U_i = X_i \cdot S_i = \lambda \cdot V_i \cdot S_i$$

$$D_i = V_i \cdot S_i$$

↳ Bottleneck is queue with highest utilization  
using max demand:

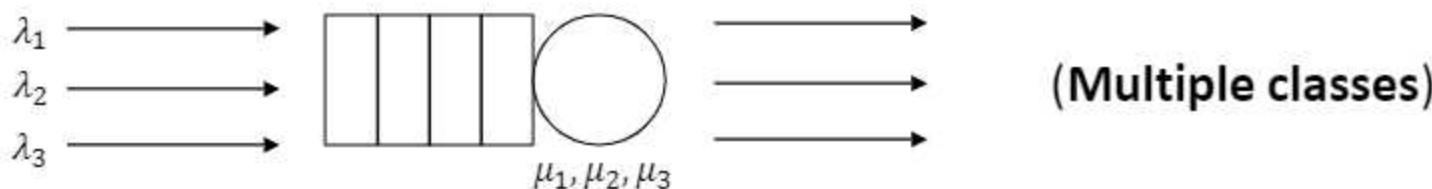
$U_i = 1$  is the greatest utilization factor of a queue  $U_i \leq 1$

$$\lambda \cdot D_i \leq 1$$

$$\lambda \leq \frac{1}{\max D_i}$$

For stability

# Open Jackson Networks - Multiple Classes



If the arrivals associated to each workload component (more than one) are all exponentially distributed, then the single workload components can be analyzed almost independently

It is necessary to know:

- All the arrival rates  $\lambda_r$
- All the demands associated to each class  $D_{i,r}$

Details available in Chapter 9 - D. A. Menascé, V. A. F. Almeida: Capacity Planning for Web Services: metrics, models and methods.

# Closed Networks

- Whenever get an answer directly get in queue, no thinking time.

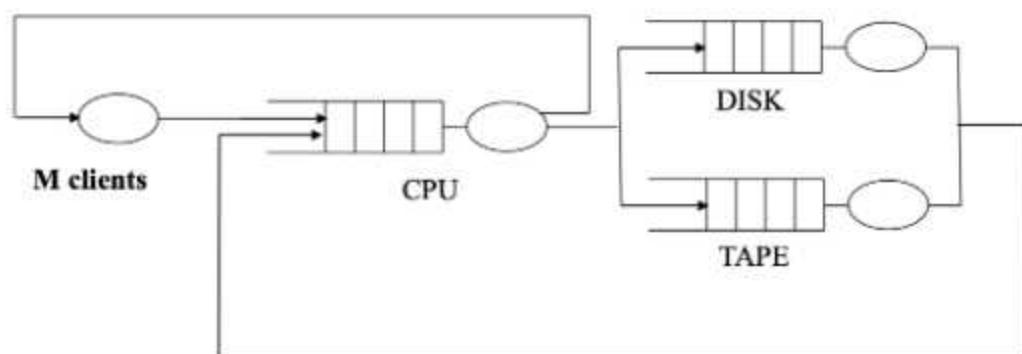
Finite population: **M** source of requests

Each source submits a request, waits for the response, composes a new request ("thinking") to be submitted to the system

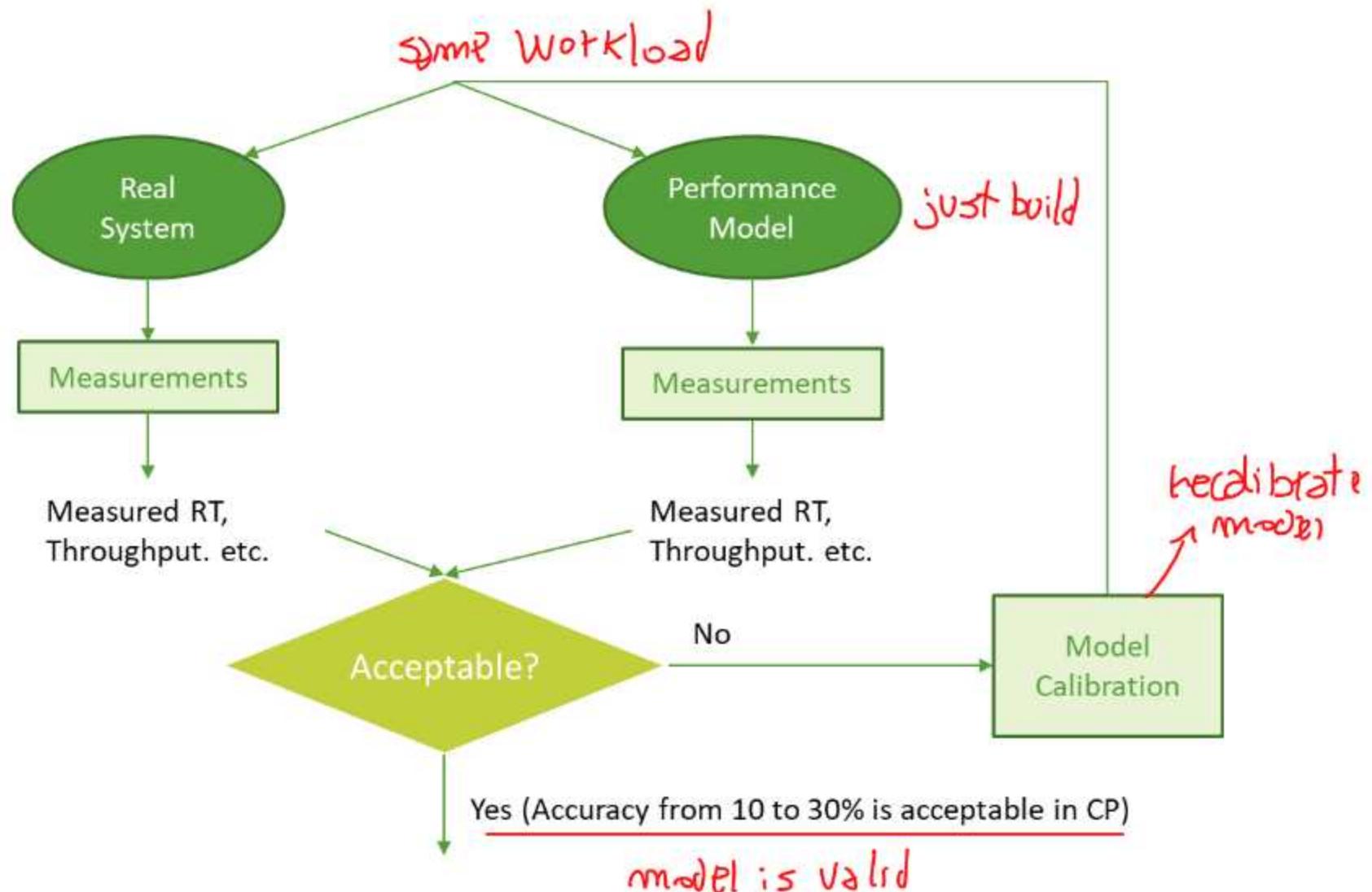
The time spent by the source from the time a response to a request is received and the next request is submitted is called **think time**

**Z = average think time**

Details available in Chapter 9 - D. A. Menascé,  
V. A. F. Almeida: Capacity Planning for Web  
Services: metrics, models and methods.



# Performance model validation and calibration



# References

- D. A. Menascé, V. A. F. Almeida: Capacity Planning for Web Services: metrics, models and methods. Prentice Hall, PTR
- Quantitative System Performance, Computer System Analysis Using Queueing Network Models. Edward D. Lazowska, John Zahorjan, G. Scott Graham, Kenneth C. Sevcik <https://homes.cs.washington.edu/~lazowska/qsp/>
- L. Kleinrock: Queueing Systems, Vol. 1: Theory, John Wiley & Sons
- Java Modelling Tools – JMT <https://jmt.sourceforge.net/>
- R. Jain, "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling," [https://www.cin.ufpe.br/~rmfl/ADS\\_MaterialDidatico/PDFs/performanceAnalysis/Art%20of%20Computer%20Systems%20Performance%20Analysis%20Techniques.pdf](https://www.cin.ufpe.br/~rmfl/ADS_MaterialDidatico/PDFs/performanceAnalysis/Art%20of%20Computer%20Systems%20Performance%20Analysis%20Techniques.pdf)

# Dependable Distributed Systems

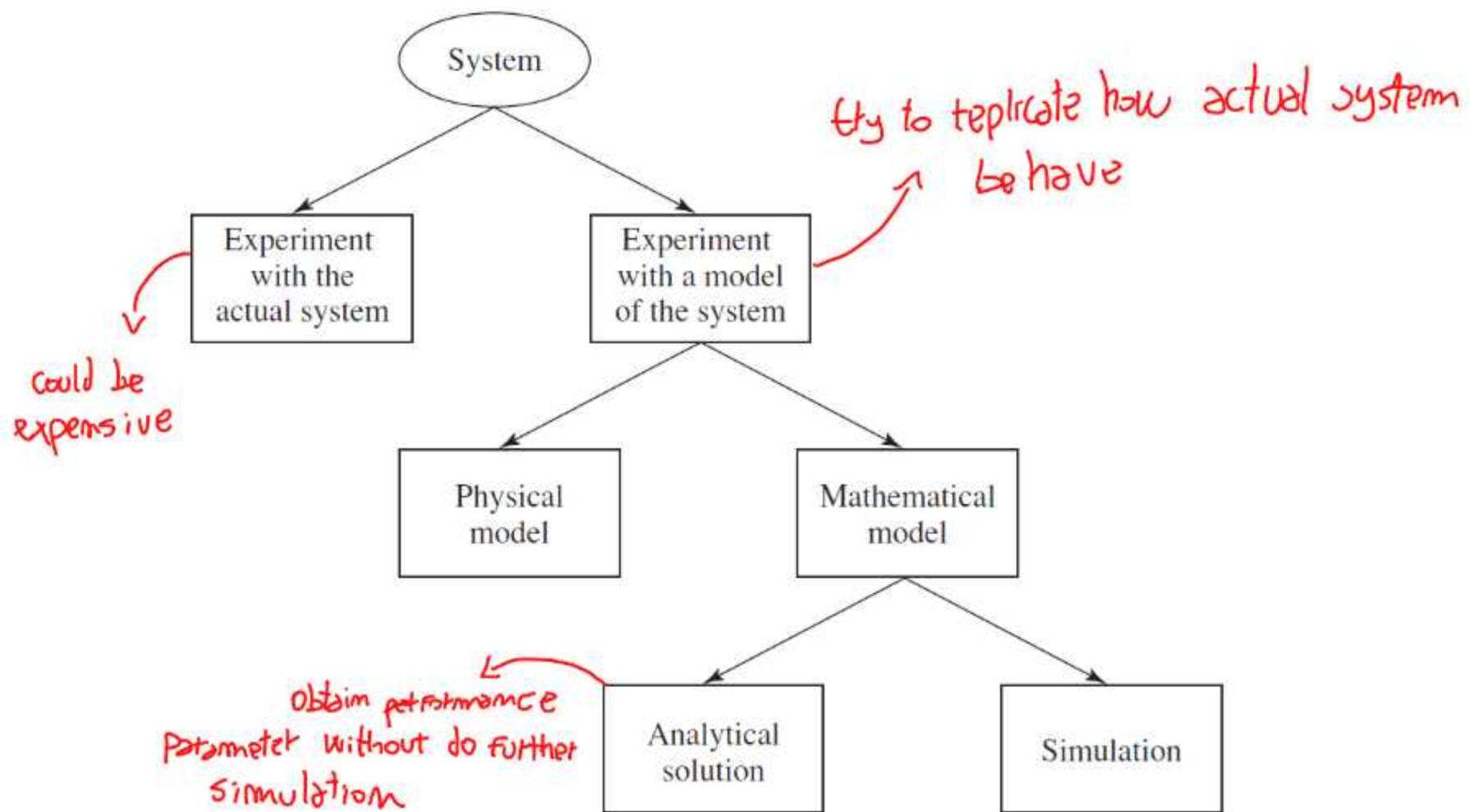
## Master of Science in Engineering in Computer Science

AA 2022/2023

---

LECTURE 19 : BUILDING A PERFORMANCE MODEL 2 – SIMULATION

# Ways to study a system



# When analytical models are not enough?

Many systems are highly complex

\_> Valid mathematical models of them are themselves complex, precluding any possibility of an analytical solution (like the ones we've seen for M/M/1 or Jackson's Networks)

In this case, the model must be studied by means of simulation, i.e., numerically exercising the model for the inputs in question to see how they affect the output measures of performance

# Systems Intro

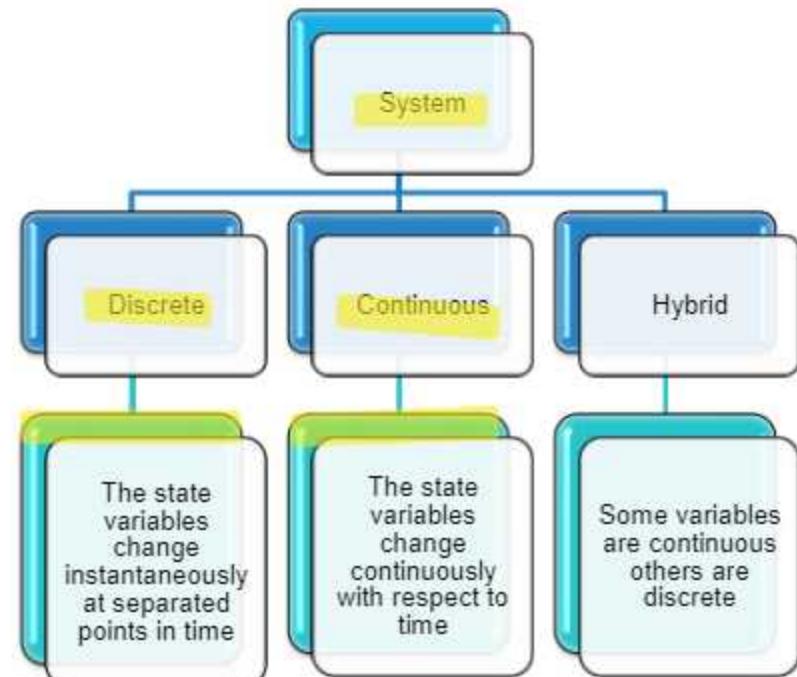
A **system** is defined to be a collection of entities that act and interact together toward the accomplishment of some logical end.

We define the **state of a system** to be that collection of variables necessary to describe a system at a particular time, relative to the objectives of a study.

We **categorize systems to be of two types, discrete and continuous.**

A **discrete system** is one for which the state variables change instantaneously at separated points in time.

A **continuous system** is one for which the state variables change continuously with respect to time.



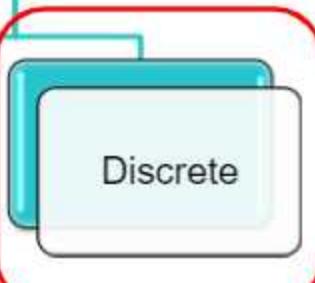
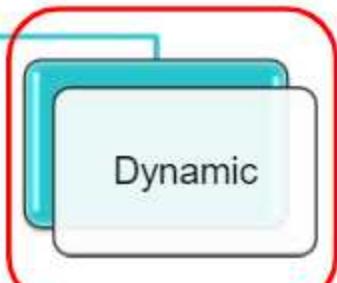
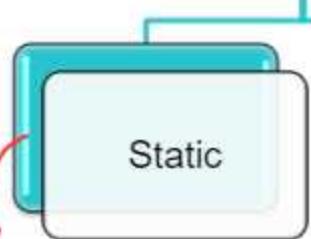
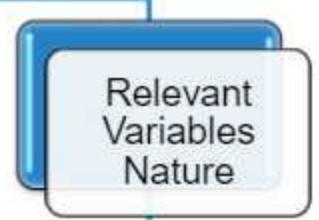
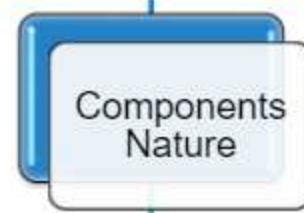
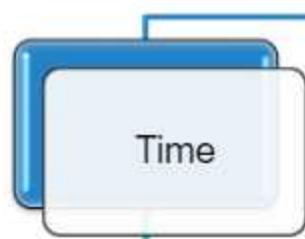
# Simulation Model

**Computer simulation** is the process of mathematical modelling, performed on a computer, which is designed to predict the behavior of, or the outcome of, a real-world or physical system

Simulation = Computer programs (in our context)

→ Dynamic time, stochastic component and discrete variable.

possible way to categorize s.m.:



not take care of evolution of the time, not influence simulation we are developing

↳ system evolve in time, in a simulated time.

every things occur in simulation is pre-define

↳ event follow certain probability distribution, introduce randomness

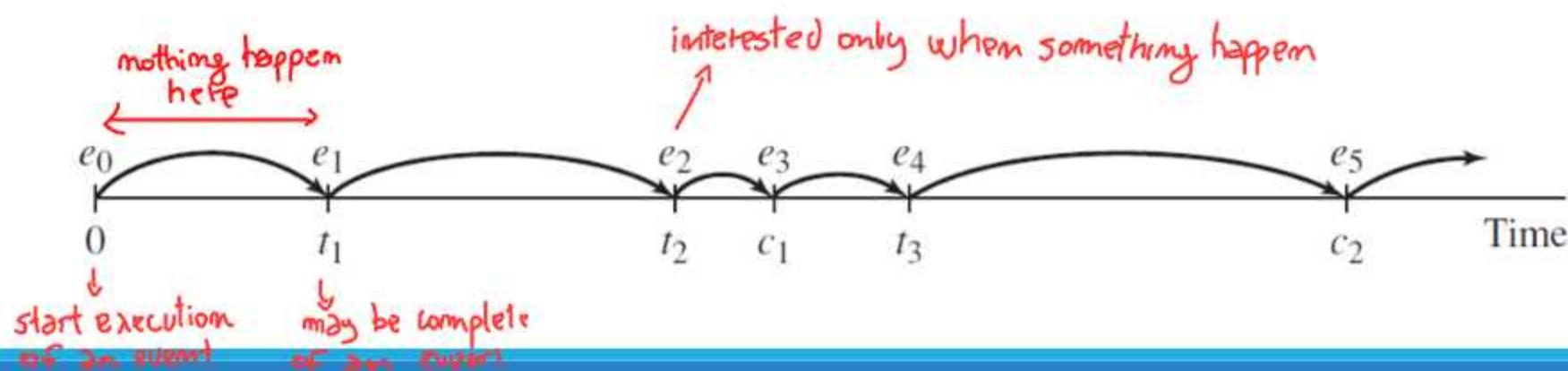
not evolve continuously in time, evolve in just fixed time

# Discrete-Event Simulation

Discrete-event simulation concerns the modeling of a system as it evolves over time by a representation in which **the state variables change** instantaneously **at separate points in time**. (In more mathematical terms, we might say that the system can change at only a countable number of points in time.)

These points in time are the ones at which an event occurs, where an event is defined as an instantaneous occurrence that may change the state of the system.

Between consecutive events, no change in the system is assumed to occur



# Discrete-Event Simulation

We must keep track of the current value of simulated time as the simulation proceeds, and we also need a mechanism to advance simulated time from one value to another

**simulation clock** is the variable in a simulation model that gives the current value of simulated time

Two principal approaches have been suggested for advancing the simulation clock

- **next-event time advance**: simulation time can directly jump to the occurrence time of the next event *most popular approach*
- **fixed-increment time advance**: where time is broken up into small time slices and the system state is updated according to the set of events/activities happening in the time slice

There is generally **no relationship between simulated time and the time needed to run a simulation on the computer**

# Components and Organization of a Discrete-Event Simulation Model

<b>System state</b>	The <u>collection of state variables necessary to describe the system at a particular time</u>
<b>Simulation clock</b>	A <u>variable giving the current value of simulated time</u>
<b>Event list</b>	A <u>list containing the next time when each type of event will occur</u>
<b>Statistical counters</b>	Variables used for storing statistical information about <u>system performance</u> , <i>result obtain from simulation</i>
<b>Initialization routine</b>	A <u>subprogram to initialize the simulation model at time 0</u>
<b>Timing routine</b>	A <u>subprogram that determines the next event from the event list and then advances the simulation clock to the time when that event is to occur</u>

# Components and Organization of a Discrete-Event Simulation Model

<b>Event routine</b>	A subprogram that updates the system state when a particular type of event occurs (there is one event routine for each event type)
<b>Library routines</b>	A set of subprograms used to generate random observations from probability distributions that were determined as part of the simulation model
<b>Report generator</b>	A subprogram that computes estimates (from the statistical counters) of the desired measures of performance and produces a report when the simulation ends
<b>Main program</b>	A subprogram that invokes the timing routine to determine the next event and then transfers control to the corresponding event routine to update the system state appropriately. The main program may also check for termination and invoke the report generator when the simulation is over.

## How Simulation Evolves: next-event time progression

With the **next-event time-advance approach**, the simulation clock is initialized to zero and the times of occurrence of future events are determined.

---

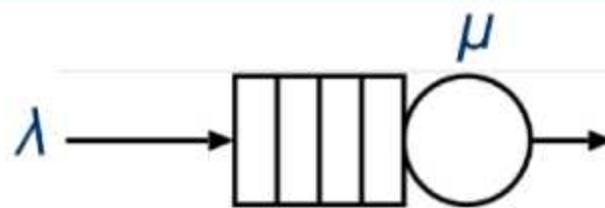
The simulation clock is then advanced to the time of occurrence of the most imminent (first) of these future events, at which point the state of the system is updated to account for the fact that an event has occurred, and our knowledge of the times of occurrence of future events is also updated. Then the simulation clock is advanced to the time of the (new) most imminent event, the state of the system is updated, and future event times are determined, etc.

---

**This process** of advancing the simulation clock from one event time to another **continues until some prespecified stopping condition is satisfied**. Since all state changes occur only at event times for a discrete event simulation model, periods of inactivity are skipped over by jumping the clock from event time to event time.

## Example: Simulation of a Single-server Queueing System

Consider a single-server queueing system for which the interarrival times  $A_1, A_2, \dots$  are independent and identically distributed (IID) random variables



event we consider  
is arrival and departure

The simulation will begin in the “empty-and-idle” state, i.e., no customers are present and the server is idle.

At time 0, we will begin waiting for the arrival of the first customer, which will occur after the first interarrival time  $A_1$

The simulation will stop when the  $n^{\text{th}}$  customer enters service

- Note that the time the simulation ends is thus a random variable, depending on the observed values for the interarrival and service-time random variables.

# Example: Simulation of a Single-server Queueing System

To measure the performance of this system, we will look at:

- $d(n)$ : the expected average delay in queue of the n customers
- $q(n)$ : average number of customers in the queue (but not being served)
  - Note that indicating n is necessary in the notation to underline that this average is taken over the time period needed to observe the n delays defining our stopping rule.
- $u(n)$ : expected utilization of the server i.e., the expected proportion of time during the simulation that the server is busy (i.e., not idle).

# Example: Simulation of a Single-server Queueing System

The **events** for this system are

- the **arrival** of a customer and
- the **departure** of a customer (after a service completion)

The **state variables** necessary to estimate  $d(n)$ ,  $q(n)$ , and  $u(n)$  are

- the **status of the server** (0 for idle and 1 for busy)
- the **number of customers** in the queue,
- the **time of arrival of each customer** currently in the queue (represented as a list), and

# Example: Simulation of a Single-server Queueing System

?

## EXAMPLE

We begin our explanation of how to simulate a single-server queueing system by showing how its simulation model would be represented inside the computer at time  $e_0 = 0$  and the times  $e_1, e_2, \dots, e_{13}$  at which the 13 successive events occur that are needed to observe the desired number ( $n = 6$ ) of delays in queue.

	1	2	3	4	5	6	7	8	9	...
$A_i$	0,4	1,2	0,5	1,7	0,2	1,6	0,2	1,4	1,9	...

Inter-arrival times

	1	2	3	4	5	6	7	8	9	...
$S_i$	2,0	0,7	0,2	1,1	3,7	0,6	...	...	...	...

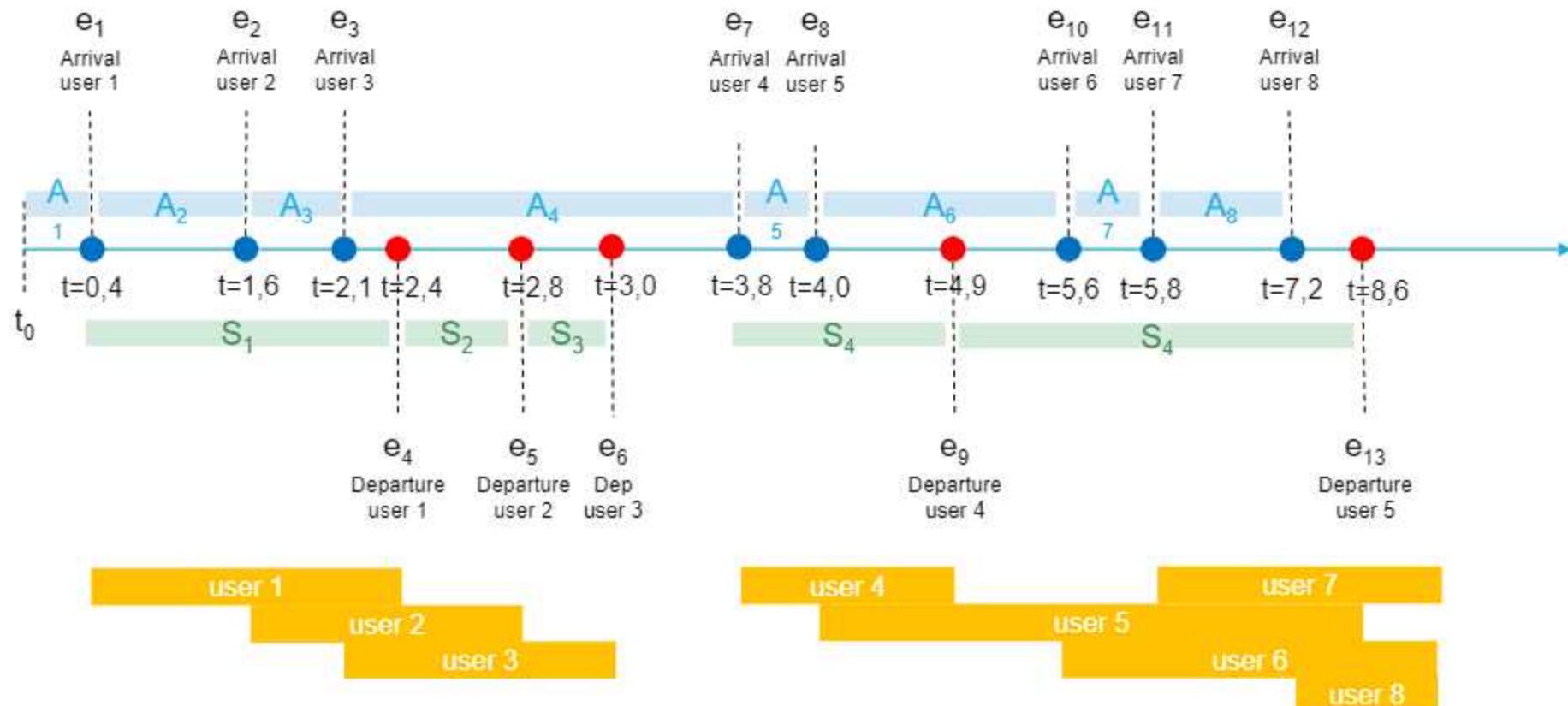
Service times

## WARNING!

It is not necessary to declare what the time units are (minutes, hours, etc.), but only to be sure that all time quantities are expressed in the same units

# Example: Simulation of a Single-server Queueing System

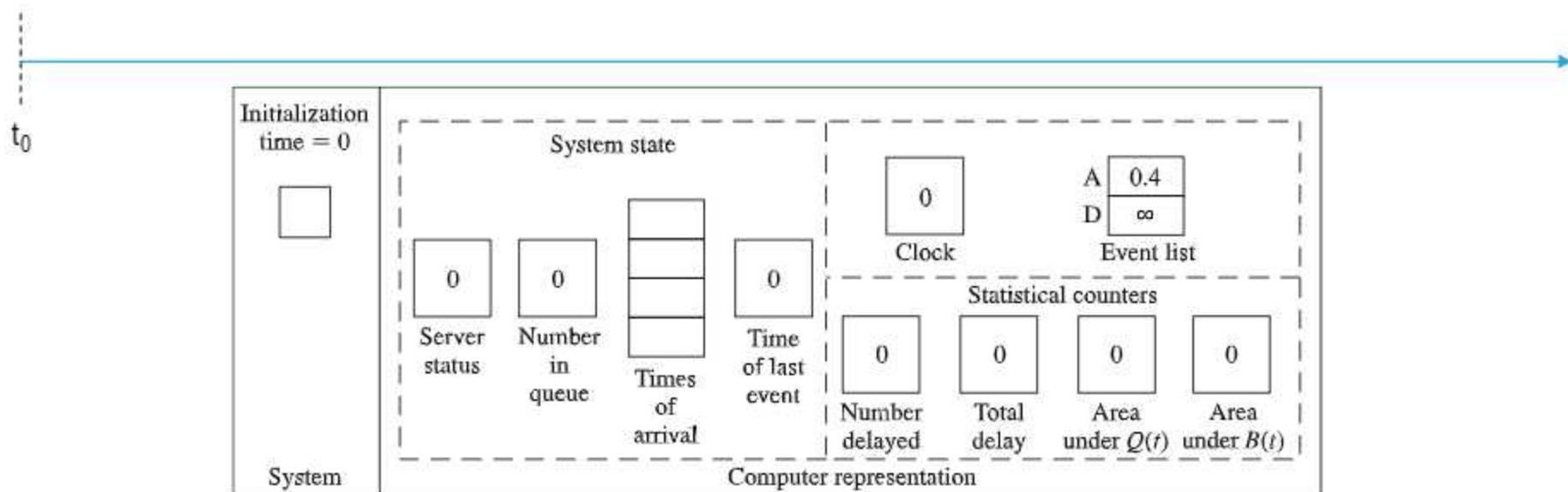
?



# Example: Simulation of a Single-server Queueing System

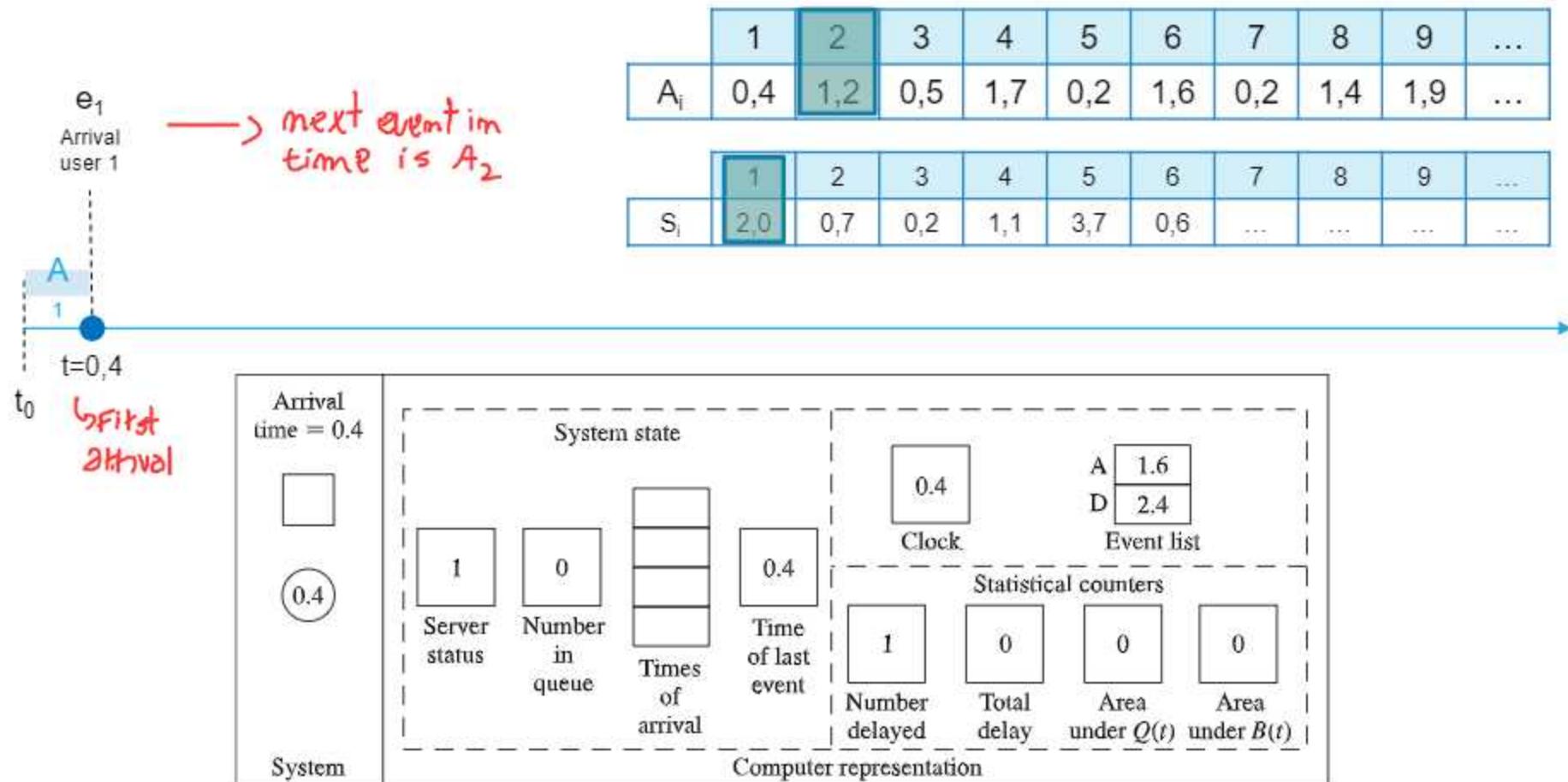
generate with random variable

	1	2	3	4	5	6	7	8	9	...
$A_i$	0.4	1.2	0.5	1.7	0.2	1.6	0.2	1.4	1.9	...



## Example: Simulation of a Single-server Queueing System

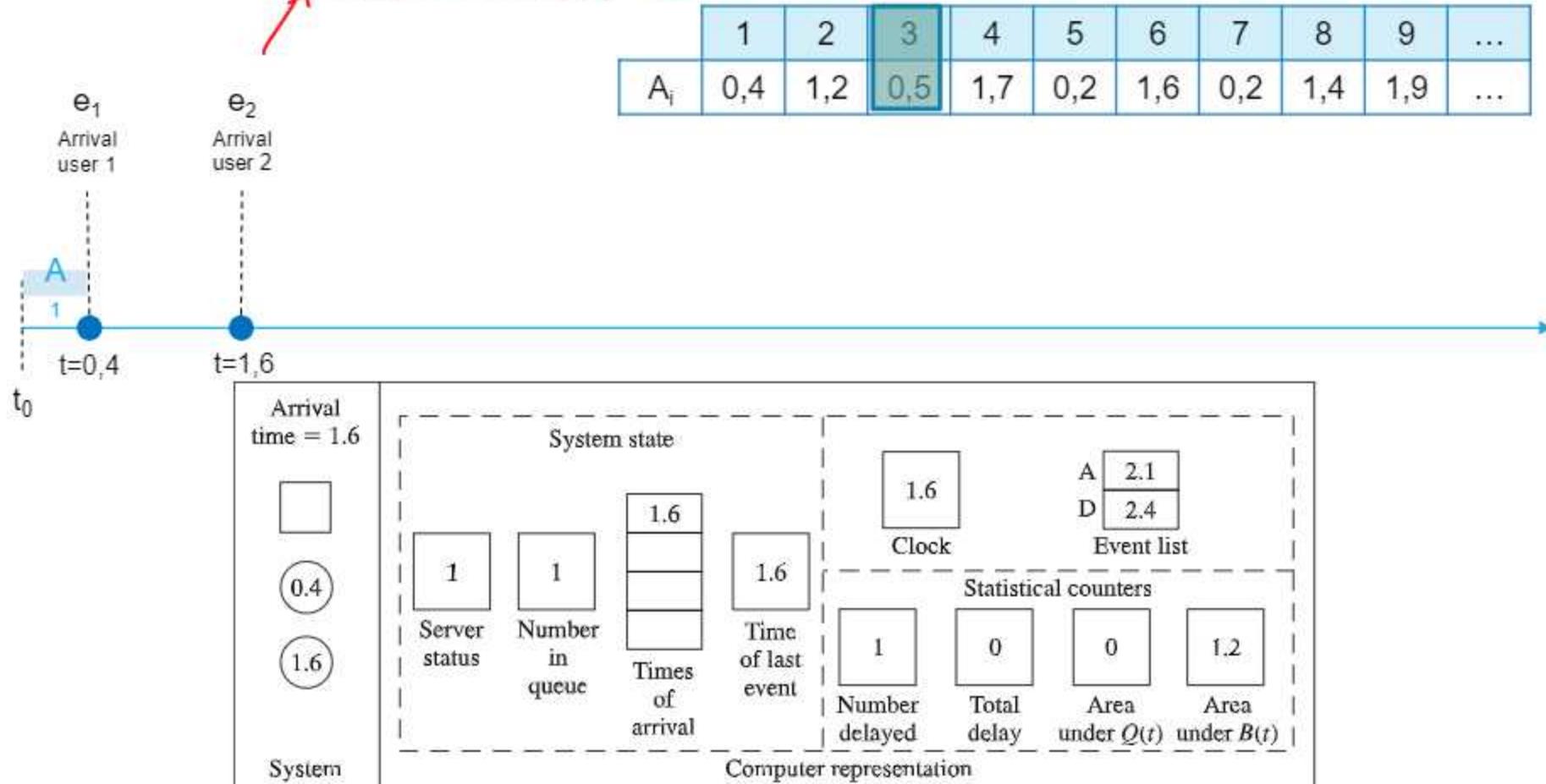
## EXAMPLE



# Example: Simulation of a Single-server Queueing System

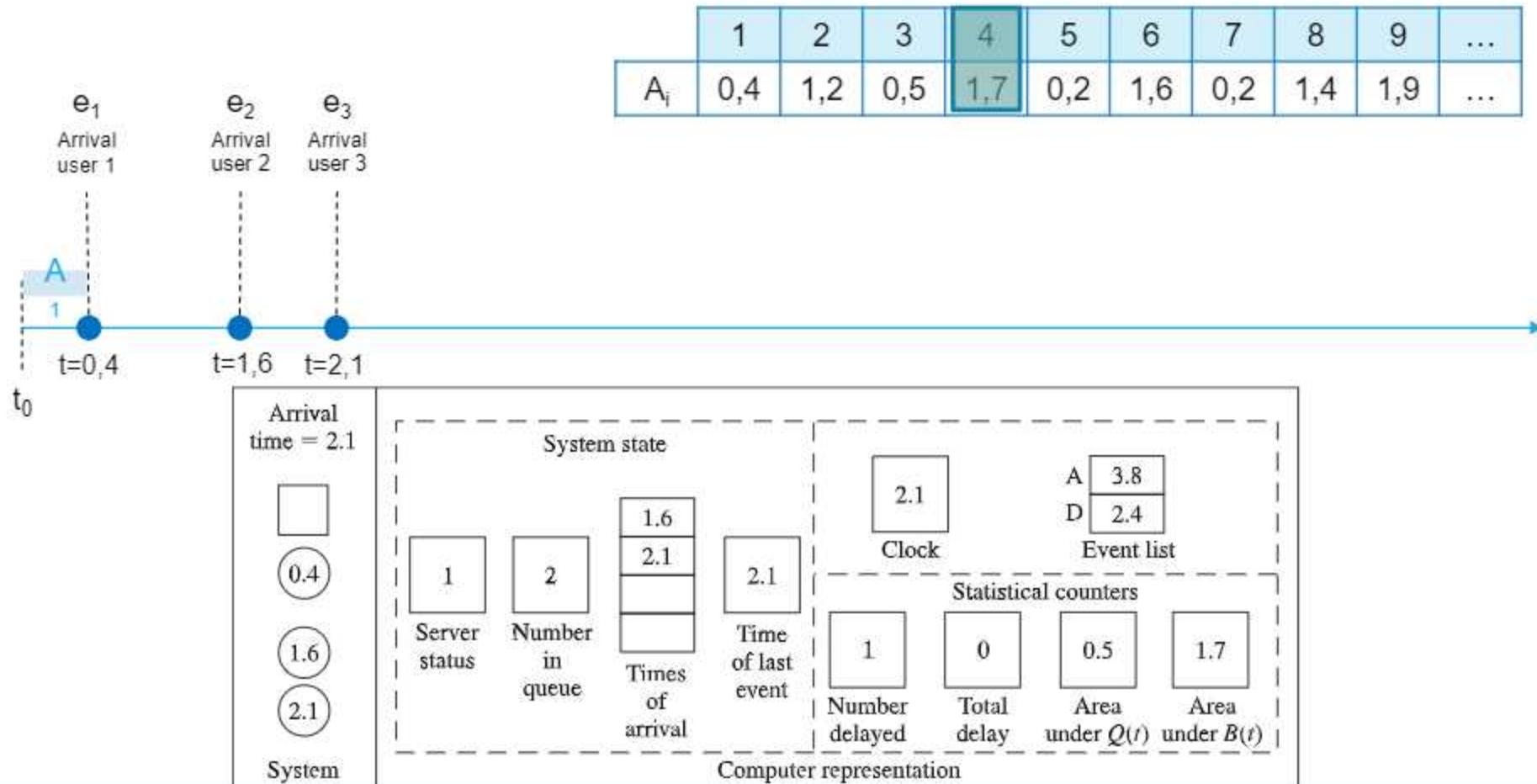
## EXAMPLE

*happens before  $S_1$*



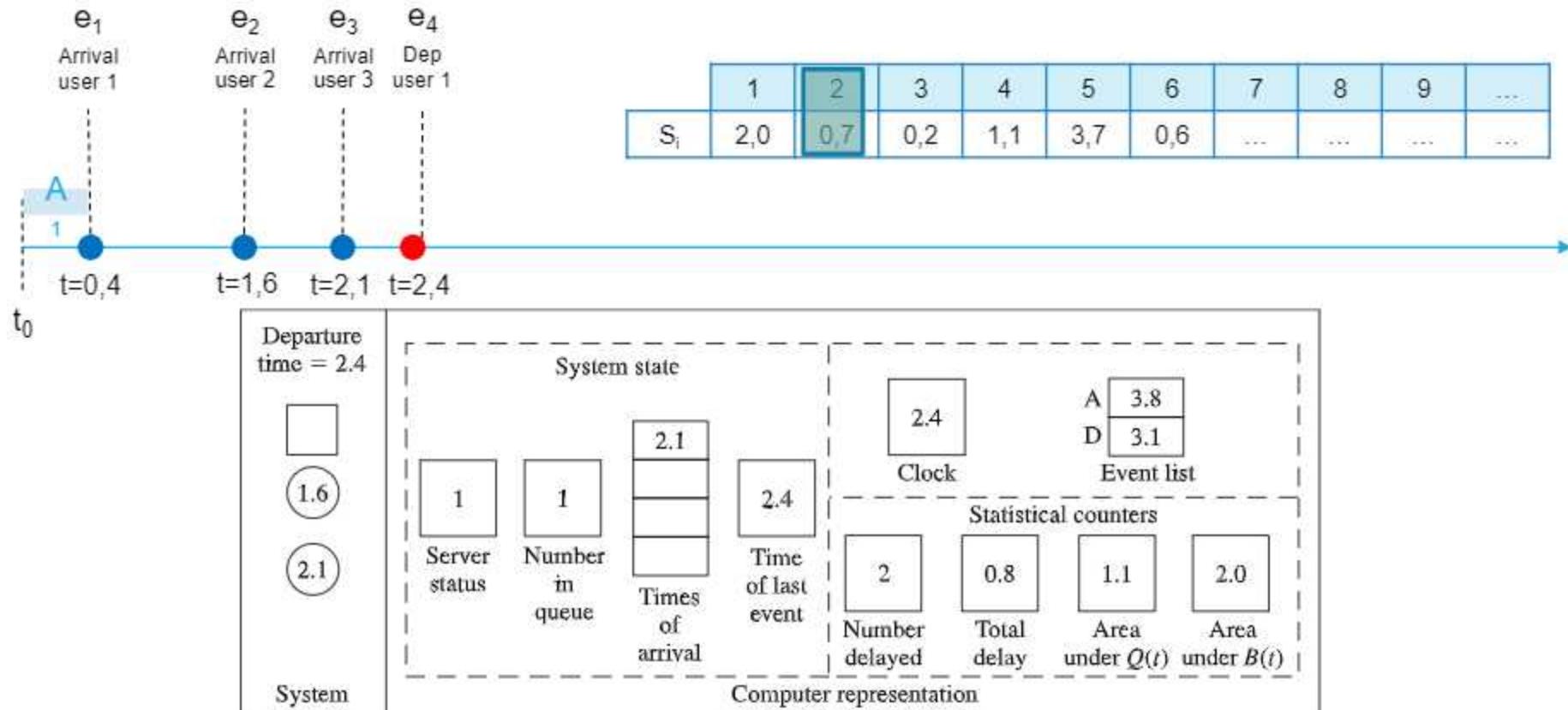
# Example: Simulation of a Single-server Queueing System

## EXAMPLE



# Example: Simulation of a Single-server Queueing System

## EXAMPLE



# SimPy

The easiest to use! (Python)

<https://simpy.readthedocs.io/en/latest/>



The behavior of active components (like vehicles, customers or messages) is modeled with processes.

All processes live in an *environment*.

They interact with the environment and with each other via *events*

# SimPy

**Processes** are described by simple **Python generators**.

Generator functions are a special kind of function that return a lazy iterator. These are objects that you can loop over like a list. However, unlike lists, lazy iterators do not store their contents in memory.

During their lifetime, **Processes create events** and **yield them in order to wait for them to be triggered**.

When a process yields an event, the process gets *suspended*. SimPy *resumes* the process, when the event occurs (we say that the event is *triggered*).

An important event type is the **timeout**.

Events of this type are triggered after a certain amount of (simulated) time has passed (e.g. to model service time).

# Simply Simple Example

```
>>> def car(env):
...     while True:
...         print('Start parking at %d' % env.now)
...         parking_duration = 5
...         yield env.timeout(parking_duration)
...
...         print('Start driving at %d' % env.now)
...         trip_duration = 2
...         yield env.timeout(trip_duration)
```

```
>>> import simpy
>>> env = simpy.Environment()
>>> env.process(car(env))
<Process(car) object at 0x...>
>>> env.run(until=15)
Start parking at 0
Start driving at 5
Start parking at 7
Start driving at 12
Start parking at 14
```

# Simply Process Interaction

```
>>> class Car(object):
...     def __init__(self, env):
...         self.env = env
...         # Start the run process everytime an instance is created.
...         self.action = env.process(self.run())
...
...     def run(self):
...         while True:
...             print('Start parking and charging at %d' % self.env.now)
...             charge_duration = 5
...             # We yield the process that process() returns
...             # to wait for it to finish
...             yield self.env.process(self.charge(charge_duration))
...
...             # The charge process has finished and
...             # we can start driving again.
...             print('Start driving at %d' % self.env.now)
...             trip_duration = 2
...             yield self.env.timeout(trip_duration)
...
...     def charge(self, duration):
...         yield self.env.timeout(duration)
```

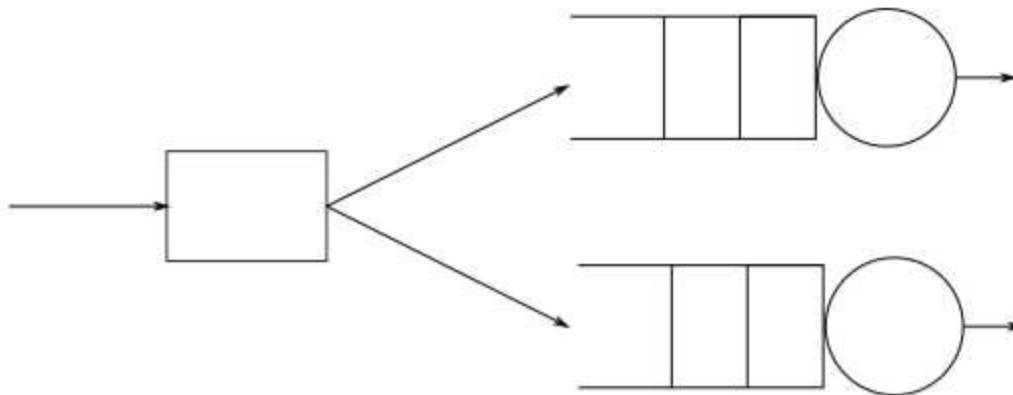
# SimPy Resources

SimPy offers three types of **resources** that help you modeling problems, where multiple processes want to use a resource of limited capacity

```
>>> import simpy
>>> env = simpy.Environment()
>>> bcs = simpy.Resource(env, capacity=2)

>>> def car(env, name, bcs, driving_time, charge_duration):
...     # Simulate driving to the BCS
...     yield env.timeout(driving_time)
...
...     # Request one of its charging spots
...     print('%s arriving at %d' % (name, env.now))
...     with bcs.request() as req:
...         yield req
...
...         # Charge the battery
...         print('%s starting to charge at %s' % (name, env.now))
...         yield env.timeout(charge_duration)
...         print('%s leaving the bcs at %s' % (name, env.now))
```

# SimPy Queues Example



Join the Shortest Queue (JSQ)

<https://bit.ly/2R1yWou>

# OMNeT++ *quite hard to use*

Issue: how to model a specific network? It is always possible to measure the real-world?

OMNeT++ is an object-oriented modular **discrete event network simulation framework**. It has a generic architecture, so it can be (and has been) used in various problem domains:

- modeling of wired and wireless communication networks
- protocol modeling
- modeling of queueing networks
- modeling of multiprocessors and other distributed hardware systems
- **evaluating performance aspects of complex software systems**
- in general, modeling and simulation of **any system** where the discrete event approach is suitable and **can be conveniently mapped into entities communicating by exchanging messages.**

# OMNeT++

OMNeT++ provides infrastructure and tools for writing simulations.

---

An OMNeT++ model consists of **modules that communicate with message passing**.

**Simple modules can be grouped into compound modules** and so forth; the number of hierarchy levels is unlimited

**The whole model, called network** in OMNeT++, is itself a compound module.

# OMNeT++

**Modules communicate with messages that may contain arbitrary data**, in addition to usual attributes such as a timestamp.

Simple modules typically send messages via **gates**, but it is also possible to send them directly to their destination modules.

**Gates are the input and output interfaces of modules**: messages are sent through output gates and arrive through input gates.

**An input gate and output gate can be linked by a connection.**

# OMNeT++ : Building a Simulation

An OMNeT++ model consists of the following parts:

- NED language topology description(s) (**.ned files**) that **describe the module structure with parameters, gates, etc.** (it provides a general description of the model). It always includes a *network* description, characterizing the whole model.
- **Simple module sources.** They are C++ files, with **.h/.cc** suffix. (they provide the logic of the modules defined through NED files)
- **Configuration .ini file** (**provides the simulation parameter**, used to describe several simulation scenarios)
- (optional) Message definitions (**.msg files**) that let one define message types and add data fields to them. OMNeT++ will translate message definitions into full-fledged C++ classes.

# OMNeT++: Parameters

**Modules can have parameters.** Parameters are used mainly to pass configuration data to simple modules, and to help define model topology.

Parameters such as propagation delay, data rate and bit error rate, can be assigned to connections.

Parameters can be of type double, int, bool, string and xml

Parameters can be assigned in either the NED files or the **configuration file** `omnetpp.ini`., usually employed to describe several simulation scenarios

# Simple Modules

In OMNeT++, events occur inside simple modules. **Simple modules** encapsulate C++ code that **generates events and reacts to events**, implementing the behaviour of the module.

Every simple module is derived from a basic class (cSimpleModule).

The **handleMessage()** function will be called for every message that arrives at the module.

The function should process the message and return immediately after that. **The simulation time is potentially different in each call. No simulation time elapses within a call to handleMessage().**

**Modules with handleMessage() are NOT started automatically.** This means that you have to schedule **self-messages** (i.e. an event) from the **initialize()** function if you want a handleMessage() simple module to start working “by itself”, without first receiving a message from other modules.

Simple modules may also define the **finish()** function which is called at the end of the simulations. It is usually employed to store collected statistics.

# Starting an OMNeT++ Simulations

Work on Linux or WSL: you will cry less!

<https://omnetpp.org/>

– before launching the simulation, we have to configure it by creating an initialization file (with suffix .ini) through New/Initialization File (ini).

This file has to indicate at least one ned file containing the description of a network;

– the launching of a simulation under IDE is done by clicking on a file with suffix ini (generally omnetpp.ini).

Some OMNeT examples <https://drive.google.com/drive/u/2/folders/1DMu8A8nao5PKP-fjQGgExANjI3qTQA0>

# Plugins

There are a lots of plugins defined for Omnet++

- INET model suite for wired, wireless and mobile networks <https://inet.omnetpp.org/>
  - INET provides modules that act as bridges between the simulated and real domains: you can simulate both the network and the application/system or only one of them
  - You can write your service and connect it to a simulated network in Omnet++ <https://inet.omnetpp.org/docs/users-guide/ch-emulation.html>
- Oversim: overlay and peer-to-peer network simulation framework <http://www.oversim.org/>
- Many others:



# Many other simulators

There exist so many simulators, each specialized in characterizing a specific domain and providing certain pre-defined models.

QUANTAS: Quantitative User-friendly Adaptable Networked Things Abstract Simulator

<https://dl.acm.org/doi/10.1145/3524053.3542744>

We present QUANTAS: a simulator that enables quantitative performance analysis of distributed algorithms.

SimGrid <https://simgrid.org/>

A framework for developing simulators of distributed applications targeting distributed platforms, which can in turn be used to prototype, evaluate and compare relevant platform configurations, system designs, and algorithmic approaches.

# References

- A. M. Law - Simulation modeling and analysis  
<https://industri.fatek.unpatti.ac.id/wp-content/uploads/2019/03/108-Simulation-Modeling-and-Analysis-Averill-M.-Law-Edisi-5-2014.pdf>
- [https://en.wikipedia.org/wiki/Computer\\_simulation](https://en.wikipedia.org/wiki/Computer_simulation)
- [https://en.wikipedia.org/wiki/Discrete-event\\_simulation](https://en.wikipedia.org/wiki/Discrete-event_simulation)
- Ken Chen. Performance Evaluation by Simulation and Analysis with Applications to Computer Networks  
<https://onlinelibrary.wiley.com/doi/book/10.1002/9781119006190>

# Dependable Distributed Systems

## Master of Science in Engineering in Computer Science

AA 2022/2023

---

LECTURE 21 : DEPENDABILITY EVALUATION

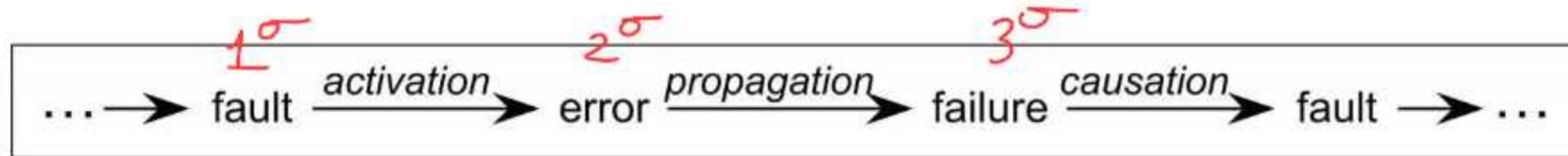
# Recalls: **Dependability**

**Dependability** is the ability of a system to deliver a service that can justifiably be trusted, it is the ability to avoid service failures that are more frequent and more severe than is acceptable.

- **Availability**: readiness for correct service.
- **Reliability**: continuity of correct service.
- ...

**Robustness**: Dependability with respect to external faults which characterizes a system reaction to a specific class of faults

# Recalls: Threats



Having a **service failure** means that there exists at least a deviation of the system behavior from the correct service state

The **deviation from the correct state is called an error**

The adjudged or hypothesized cause of an error is called a fault



# Recalls: Means

## Fault Prevention

Prevent the occurrence or introduction of faults

## Fault Tolerance

Avoid service failures in the presence of faults

## Fault Removal

Reduce the number and severity of faults

## Fault Forecasting

Estimate the present number, the future incidence and the likely consequences of faults

Aim to provide  
the ability to  
deliver a service  
that can be  
trusted

Aim to justify that the  
system is likely to meet  
its functional,  
dependability and  
security requirements

# Example: Registers

Which means are applied (Prevention, Tollerance, Removal, Forecasting)?

## (1,N) regular register

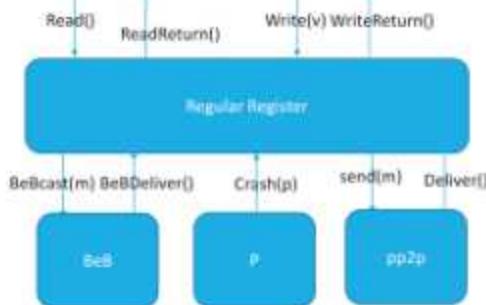
### Algorithm 4.1: Read-One Write-All

Implements:  
(1, N)-RegularRegister, instance onrr.

Uses:

BestEffortBroadcast, instance beb;  
PerfectPointToPointLinks, instance pt;  
PerfectFailureDetector, instance P.

```
upon event { onrr, Init } do
  val := ⊥;
  correct := ∅;
  writeset := ∅;
  upon event { P, Crash | p } do
    correct := correct \ {p};
```



## (1,N) regular register

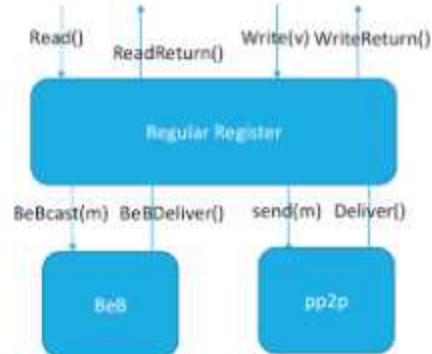
### Algorithm 4.2: Majority Voting Regular Register

Implements:  
(1, N)-RegularRegister, instance onrr.

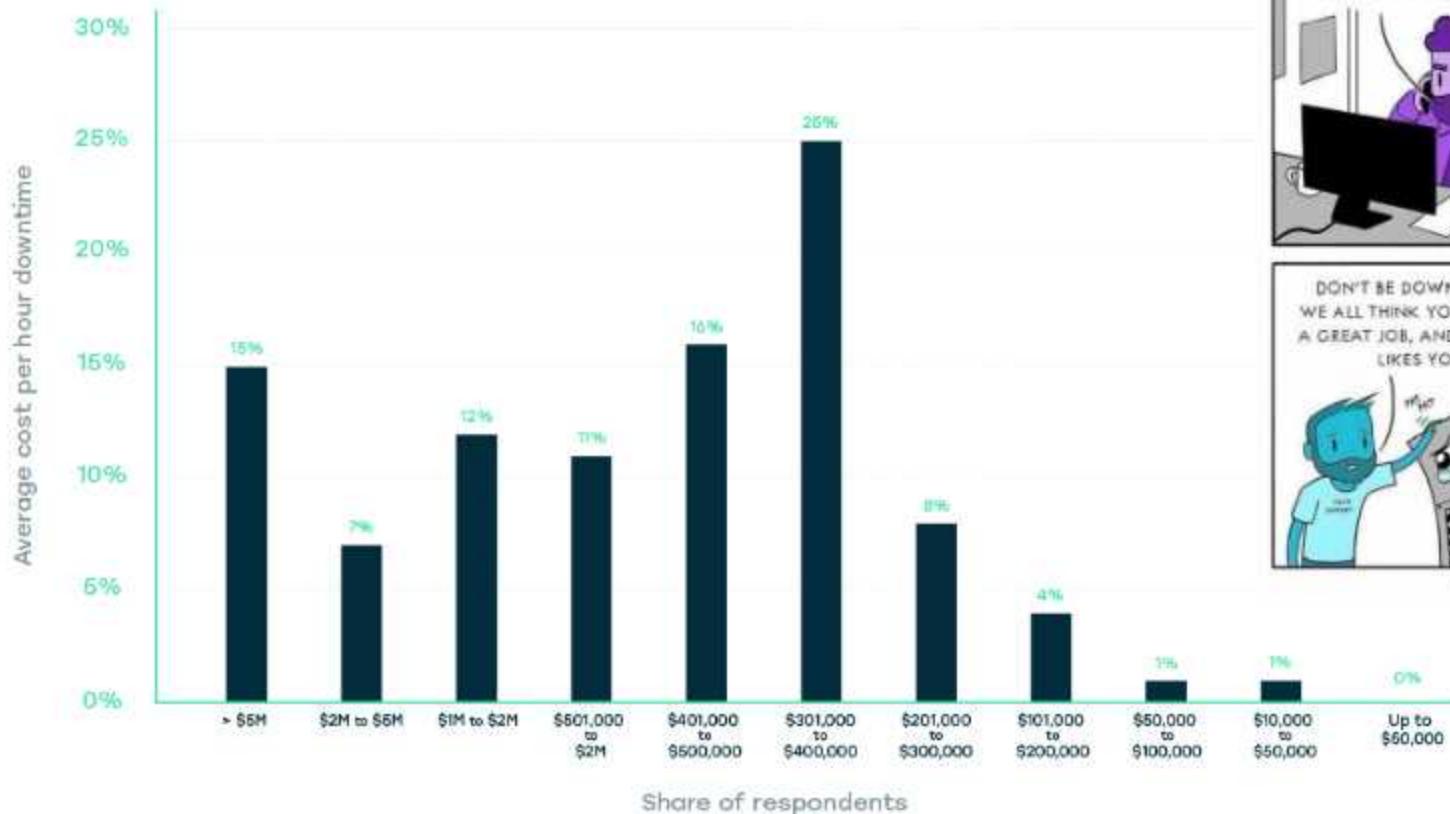
Uses:

BestEffortBroadcast, instance beb;  
PerfectPointToPointLinks, instance pt.

```
upon event { onrr, Init } do
  (ts, val) := (0, ⊥);
  wts := 0;
  acks := 0;
  rid := 0;
  readlist := [⊥]N;
```



# Downtime Cost



“Instead of minimizing the downtime cost, minimize the chance of it happening!”

<https://medium.com/@FedakV/how-much-does-your-server-downtime-cost-it-outsourcing-company-it-svit-9cf2a206f28>

# Why (again) fault tolerance?

Should we necessarily care about making a system fault tolerant?

Think about the number of involved component and their scale

The probability of a single fault occurrence

**X** number of “components”

**X** dependencies ( \_> cascading effects)

**“at Google’s scale failures with one in a million odds are occurring several times a second”**

# Redundancy

**Fault tolerance** in computer system is achieved through **redundancy** in hardware, software, information, and/or time.

**Redundancy is simply the addition of information, resources, or time beyond what is needed for normal system operation.**

**Redundancy can provide additional capabilities** within a system. But, **redundancy can have very important impact** on a system's performance, size, weight and power consumption.

# Redundancy

- **Hardware** redundancy is the addition of extra hardware, usually for the purpose either detecting or tolerating faults.
- **Software** redundancy is the addition of extra software, beyond what is needed to perform a given function, to detect and possibly tolerate faults.
- **Information** redundancy is the addition of extra information beyond that required to implement a given function; for example, error detection codes.
- **Time** redundancy uses additional time to perform the functions of a system such that fault detection and often fault tolerance can be achieved. Transient faults are tolerated by this approach.

## Redundancy vs Diversity

Redundancy contributes to improve the system dependability...

... but it does not guarantee alone a higher level of security (robustness)

- If you replicate using exact clones, the attacker will not have a harder time.

To improve security you need to apply diversity in

- Hardware
- Software
- Network configuration

# Techniques for Dependability Evaluation

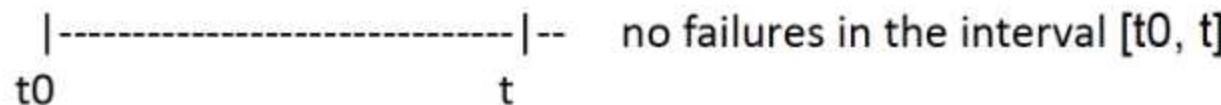
The dependability evaluation of a system can be carried out either:

- **experimentally (heuristic)**: a system prototype is built, and empirical statistical data are used to evaluate the system's metrics:
  - by far more expensive and complex than the analytic approach
  - building a system prototype may be impossible
  - experimental evaluation of dependability requires long observation periods
- **analytical**: dependability metrics are obtained by a mathematical model of the system:
  - mathematical models may not adequately represent the real system's structure or the behavior of its components
  - *simulation models* may be a complementary helpful tool

# Definition of Dependability Attributes

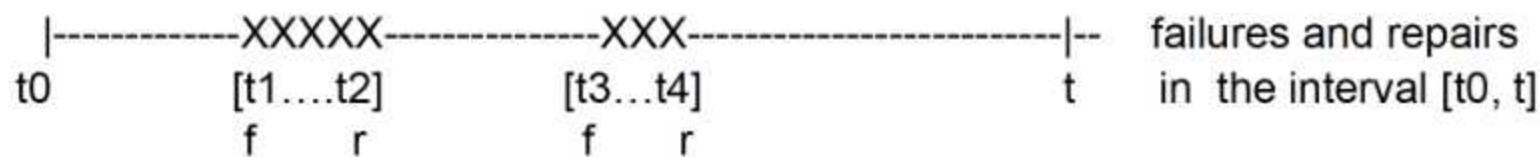
## Reliability - $R(t)$

conditional probability that the system performs correctly throughout the interval of time  $[t_0, t]$ , given that the system was performing correctly at the instant of time  $t_0$



## Availability - $A(t)$

the probability that the system is operating correctly and is available to perform its functions at the instant of time  $t$



# Definition of Dependability Attributes

**MTTF (Mean Time To Failure)**. Expected time before a failure, or expected operational time of a system before the occurrence of the first failure.

**$\lambda(t)$  failure rate**: the rate at which a component fails

$$\text{Failure Rate } (\lambda) = \frac{\text{Number of Failures}}{\text{Operating Time}}$$

$$\text{MTTF} = \theta = \frac{\text{Operating Time (Cycle)}}{\text{Number of Failures}}$$

$$\lambda = \frac{1}{\theta}$$

# Example

Let us assume we have **20 independent replicas of a component**

Let them **operate for a period of time**, let us say **1000 hours**

During the **period of 1000 hours**, **6 of them fail**.

Specifically, they **fail at the following hours** (assuming  $t_0 = 0h$ ):

550, 480, 680, 790, 860, 620

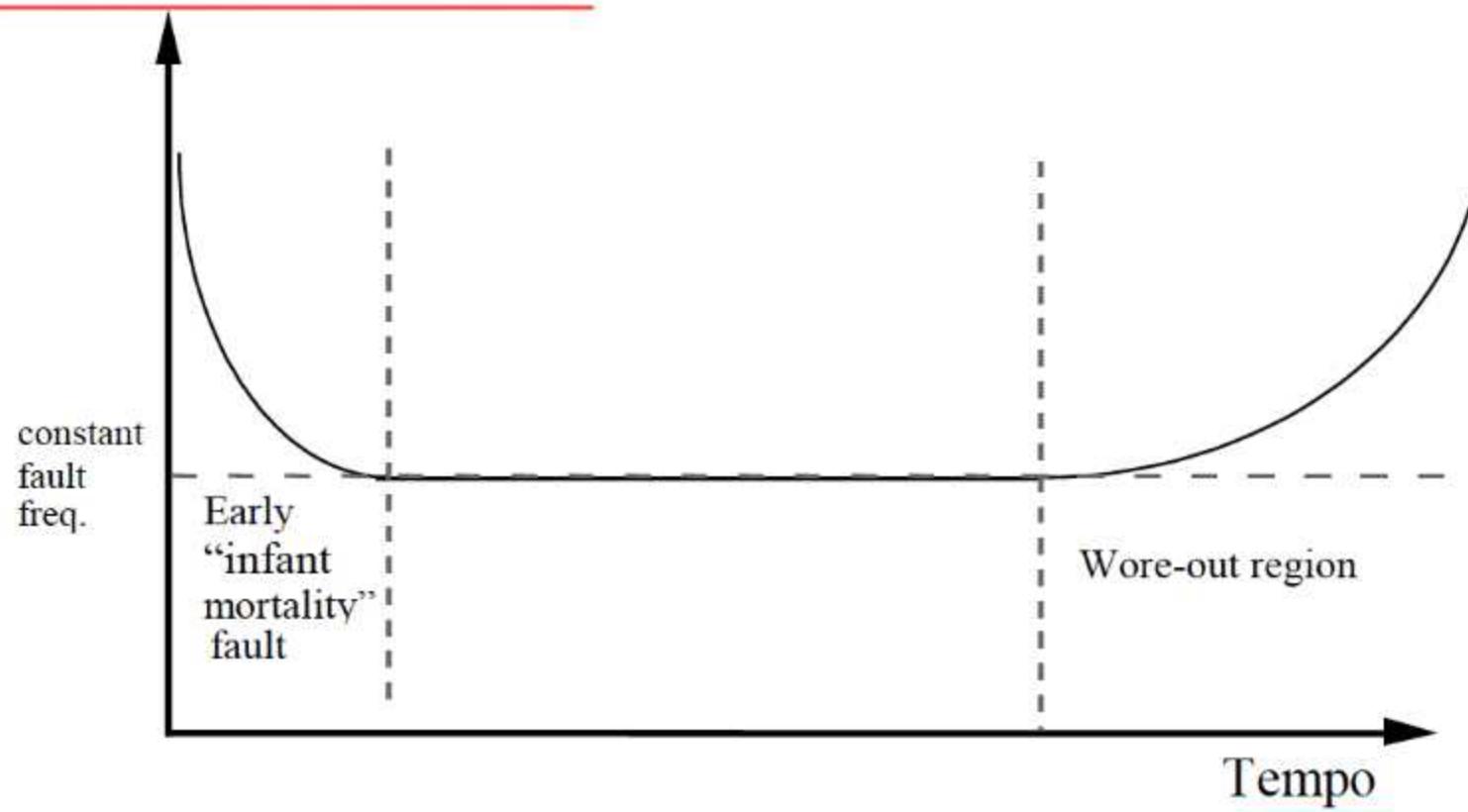
$$\text{Failure Rate } (\lambda) = \frac{6}{550 + 480 + 680 + 790 + 860 + 620 + (14 * 1,000)} = \frac{6}{17,980}$$

*↳ 14 correct replicas*

**Failure Rate  $(\lambda)$  = 0.000334 Failures Per Hour**

# Bathtub Curve – Hardware Failure Frequency

Failure frequency function

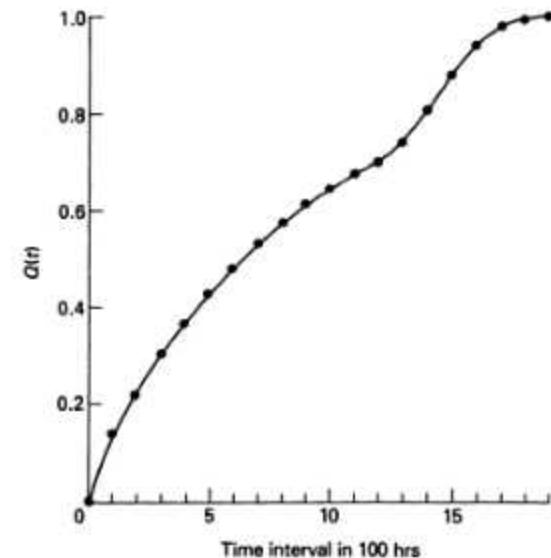


# Definitions

## Failure Function $Q(t)$ :

- probability that a component fails for the first time in the time interval  $(0, t)$
- it's a cumulative distribution function:

$$\begin{aligned} Q(t) &= 0 && \text{for } t = 0 \\ 0 \leq Q(t) &\leq Q(t + \Delta t) && \text{for } \Delta t \geq 0 \\ Q(t) &= 1 && \text{for } t \rightarrow +\infty \end{aligned}$$



# Definitions: Reliability function

## Reliability Function $R(t)$ :

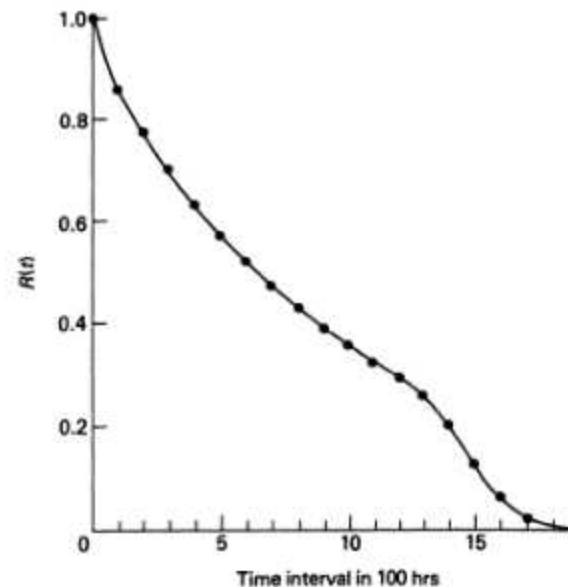
- probability that a component functions correctly in the time interval  $(0, t)$

$$R(t) = 1 - Q(t)$$

$$R(t) = 1 \quad \text{for } t = 0$$

$$1 \geq R(t) \geq R(t + \Delta t) \quad \text{for } \Delta t \geq 0$$

$$R(t) = 0 \quad \text{for } t \rightarrow +\infty$$

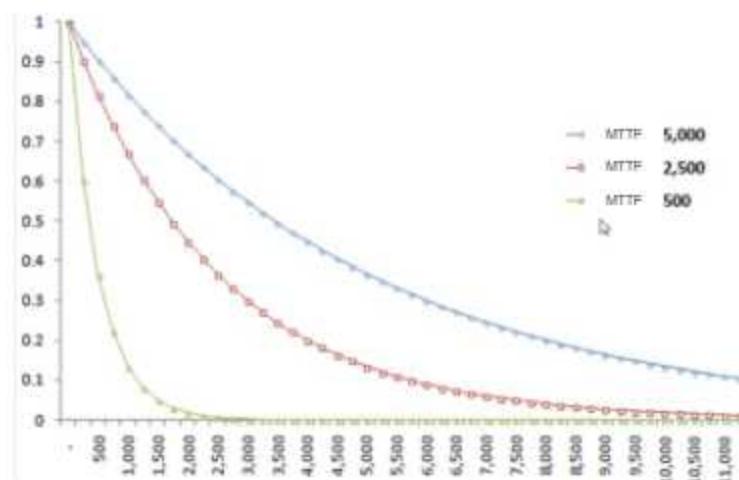


# Definitions: Reliability function

$$R(t) = \exp \left[ - \int_0^t \lambda(t) dt \right]$$

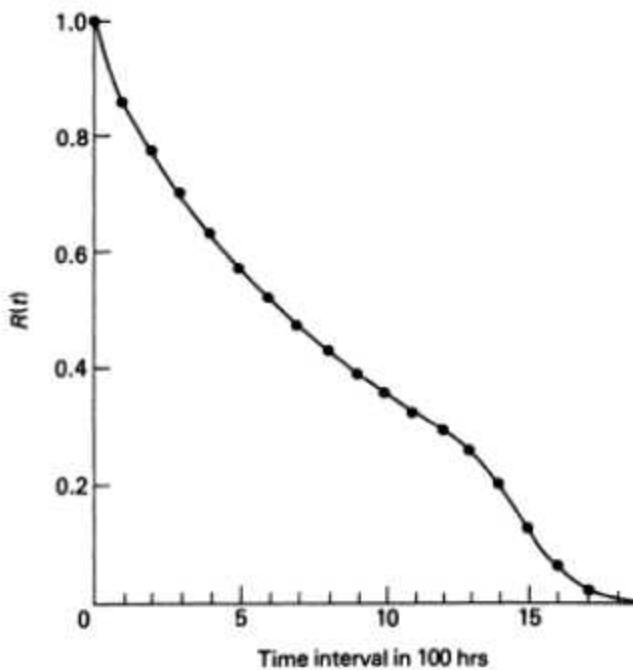
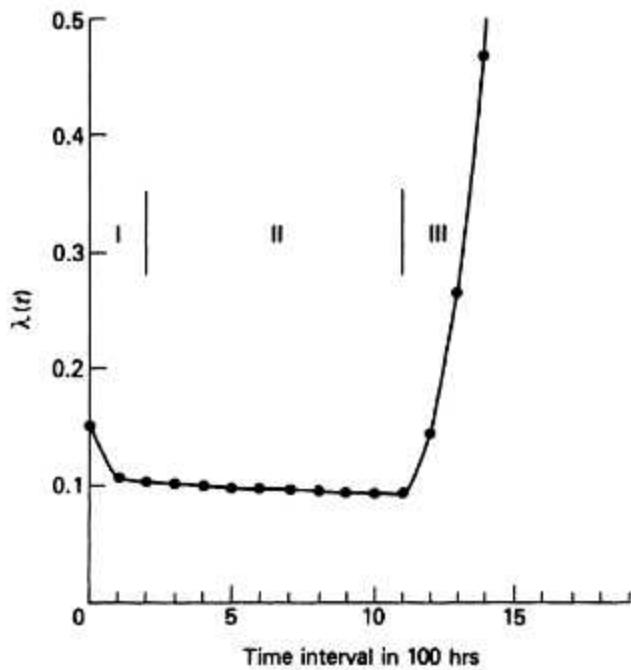
For the special case in which  $\lambda(t)$  is a constant and independent of time

$$R(t) = e^{-\lambda t}$$

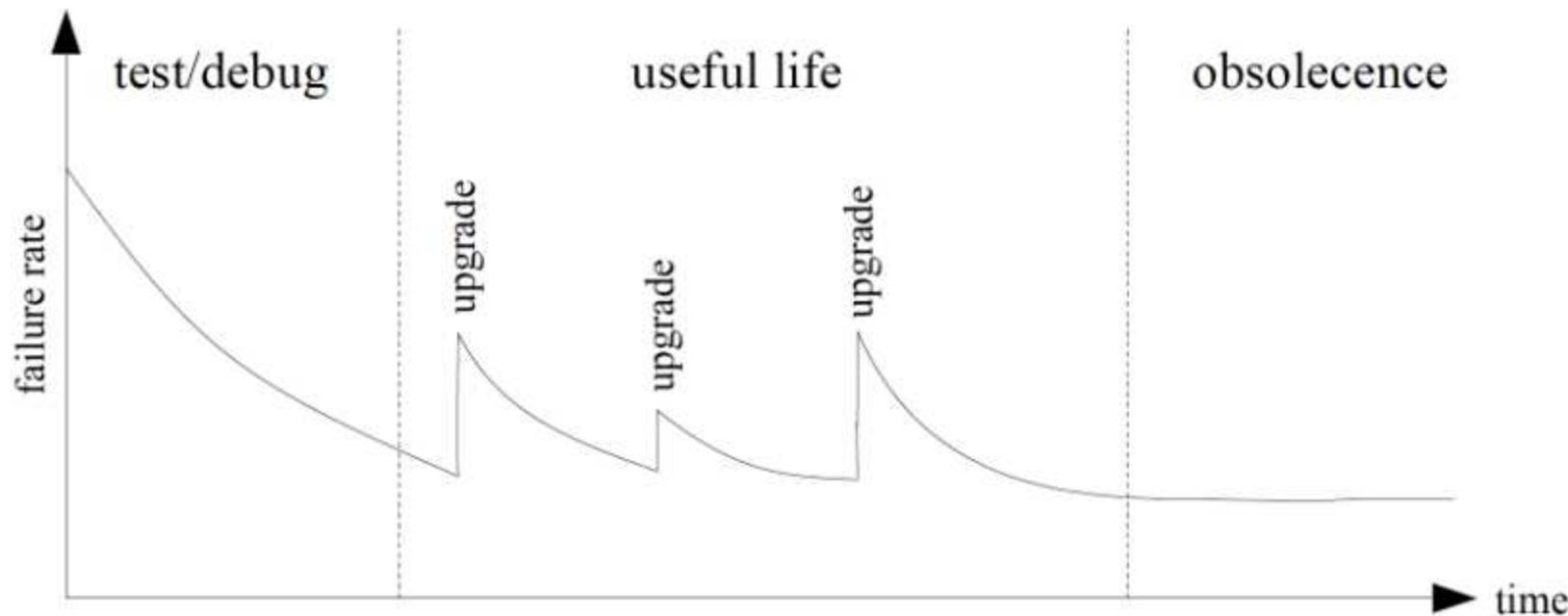


It is possible to evaluate the reliability of a component also cases where  $\lambda(t)$  is not constant through the Weibull distribution

# Example ([1] Section 6.4)



# Software Failure Rate over Lifetime



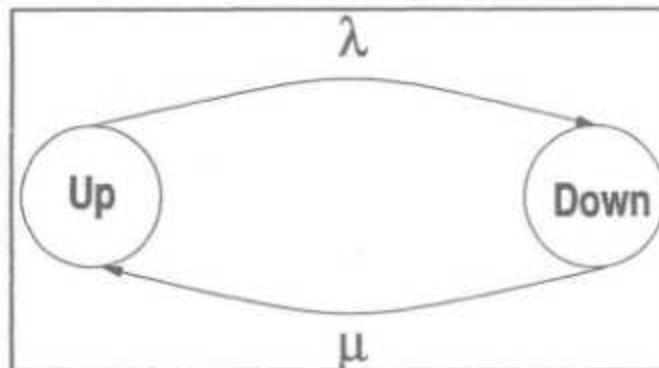
Software Failure Rate over Lifetime

# Repairable Systems

In the case of repairable systems, besides the “fault occurrence” event, the event “repairing” or “replacement” of the faulty components has to be considered:

**MTBF** Mean Time Between Fault: Expected time before a new failure

**MTTR (Mean Time To Repair)**: The average time to repair or replace a faulty entity



# Availability

## Point Availability, $A(t)$

Instantaneous (or point) availability is the probability that a system (or component) will be operational (up and running) at a specific time,  $t$ .

## Steady State Availability, $A(\infty)$

The steady state availability of the system is the limit of the availability function as time tends to infinity.

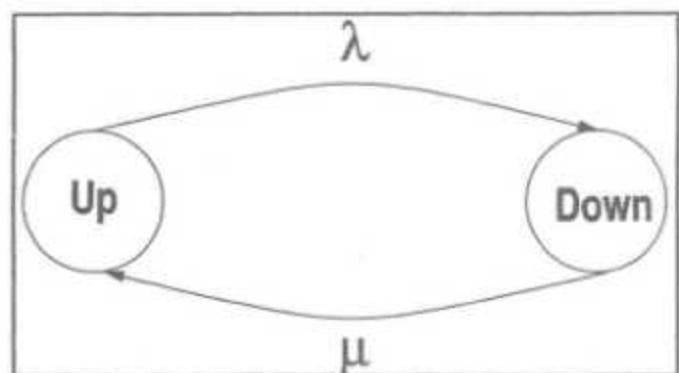
$$A(\infty) = \lim_{t \rightarrow \infty} A(t)$$

However, it must be noted that the steady state also applies to mean availability.

For ease of simplicity, we assume that all faults occurrences and repairs are independent among them other

## (Steady-State) Availability Combinatorial Evaluation

Availability is the the fraction of time that a component/system is operational



$$\lambda = \frac{1}{\text{MTBF}} \quad \mu = \frac{1}{\text{MTTR}}$$

*(expect time to new failure)      (expect time to repair)*

$$\lambda \cdot p_{up} = \mu \cdot p_{down}$$

$$p_{up} + p_{down} = 1$$

$$A = p_{up}$$

$$A = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

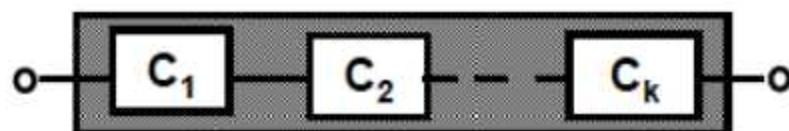
## Reliability/Availability Combinatorial Evaluation - Interconnections

Most common component interconnections are:

- Serial
- Parallel
- Hybrid M out of N

# Serial Interconnection

$K$  entities are serially interconnected when the functioning of the system depends on the correct functioning of all the  $K$  entities.



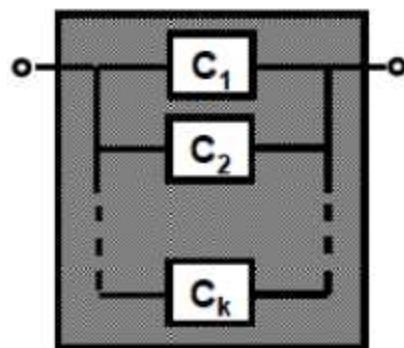
$$A_{\text{serial}} = \prod A_i$$

The probability of an event expressed as the intersection of independent event is the product of the probabilities of the independent events

$$R(t) = \prod_{i=1}^K R_i(t)$$

# Parallel Interconnection

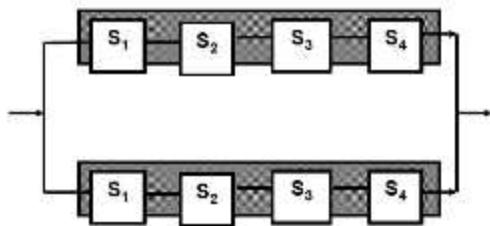
$k$  entities are interconnected in parallel when the functioning of the system is guaranteed even if just a single entity works.



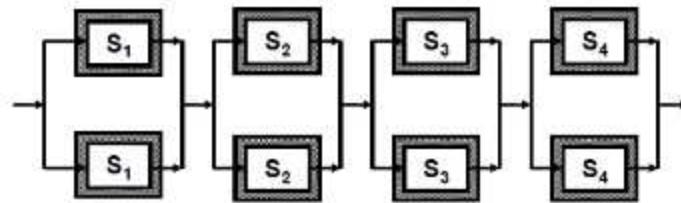
$$A_{parallel} = 1 - \prod(1 - A_i)$$

$$R(t) = 1 - (1 - R_1(t))(1 - R_2(t)) \dots (1 - R_K(t))$$

# Evaluation Example



$$A_{d1} = 1 - (1 - A_s)^2$$



$$A_{td} = 1 - (1 - A_i)^2$$

$$A_{d2} = A_{1d} A_{2d} A_{3d} A_{4d}$$

Which solution is more available?

$$A_i = 0,9$$

$$A_s = 0,6561$$

$$A_{d1} = 0,8817$$

$$A_{d2} = 0,9606$$

# Hybrid M out of N

The system works as long as there are at least  $M$  correct entities, namely at most  $K = N - M$  entities fail.

$$A = \sum_{i=0}^K \binom{N}{i} A_C^{N-i} (1 - A_C)^i$$

$$R(t) = \sum_{i=0}^K \binom{N}{i} R_C^{N-i}(t) (1 - R_C(t))^i$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

# SLA – Classes of Availability

Availability Class	Availability	Unavailable (min/year)	System Type
1	90.0%	52,560	Unmanaged
2	99.0%	5,256	Managed
3	99.9%	526	Well-Managed
4	99.99%	52.6	Fault-Tolerant
5	99.999%	5.3	Highly Available
6	99.9999%	0.53	Very Highly Available
7	99.99999%	0.0053	Ultra Available

# Intro to Chaos Engineering

---

Solutions to distributed system problems are designed over a set of assumptions (the system model)

\_> their correctness rely on these assumptions.

What events can invalidate an assumption?

What is the probability of occurrence of such events?

What happens if part of them are not verified?

# Intro to Chaos Engineering

---



Practically, **how dependability is evaluated and proven today** by the bigger tech companies:

Netflix, Google, Amazon, Microsoft, Dropbox, Yahoo!, Uber, cars.com, Gremlin Inc., University of California, Santa Cruz, SendGrid, North Carolina State University, Sendence, Visa, New Relic, Jet.com, Pivotal, ScyllaDB, GitHub, DevJam, HERE, Cake Solutions, Sandia National Labs, Cognitect, Thoughtworks, and O'Reilly Media ...

It follows dependability evaluation and testing

Prerequisites for Chaos Engineering:

- **Be almost sure that the system is resilient to real-world events** such as service failures and network latency spikes
- **Have a monitoring system** that can be used to determine the current state of the system

# Intro to Chaos Engineering

---

It is still a relatively new discipline within software development

*“Over the past three years, the question has changed from “Should we do Chaos Engineering?” to “What’s the best way to get started doing Chaos Engineering?””*

**The story begins at Netflix in 2008,**

moving from their datacenter to the cloud: necessitate horizontally scaled components + the single points of failure.

It took eight years to fully extract themselves from the datacenter.

AWS: instances occasionally blinked out of existence with no warning.

The streaming service was facing availability deficits due to the high frequency of instance instability events.

# Intro to Chaos Engineering

---

Chaos Monkey, a very simple app: go through a list of clusters, pick one instance at random from each cluster, and at some point, during business hours, turn it off without warning (vanishing instances affected service availability).

Chaos Monkey gave them a way to proactively test everyone's resilience to the failure and do it during business hours so that people could respond to any potential fallout when they had the resources to do so.

Chaos Monkey forced everyone to be highly aligned toward the goal of being robust enough to handle vanishing instances, it doesn't suggest the solution.

# Intro to Chaos Engineering

---

**December 24, 2012, Christmas Eve.** AWS suffered a rolling outage of elastic load balancers (ELBs). These components connect requests and route traffic to the compute instances where services are deployed.

---

Could something similar be built to solve the problem of vanishing regions?

**AWS wouldn't allow Netflix to take a region offline** (something about having other customers in the region) so instead this was simulated.

**Chaos Kong** were routinely conducted to verify that Netflix had a plan of action in case a single region went down.

**2015:** Netflix had **Chaos Monkey and Chaos Kong**, working on the small scale of vanishing instances and the large scale of vanishing regions, respectively.

# Dependability of Complex Systems

---

Chaos Engineering was born of necessity in a **complex** distributed software system.

Simple systems	Complex systems
Linear	Nonlinear
Predictable output	Unpredictable behavior
Comprehensible	Impossible to build a complete mental model

A **simple system** is one in which a person can comprehend all of the parts, how they work, and how they contribute to the output.

A **complex system**, by contrast, has so many moving parts, or the parts change so quickly that no person is capable of holding a mental model of it in their head.

In **simple systems**, one person, usually an experienced engineer, can orchestrate the work of several engineers.

# Birth of Chaos Engineering

---

Principles of Chaos Engineering <https://principlesofchaos.org/>

The discipline was officially formalized.

**Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production.**

The facilitation of experiments to uncover systemic weaknesses

# Principles lists

---

The Principles lists five advanced practices that set the gold standard for a Chaos Engineering practice:

- **Build a hypothesis around steady-state behavior**
- **Vary real-world events**
- **Run experiments in production**
- **Automate experiments to run continuously**
- **Minimize blast radius**

# 1) Hypothesize about steady state

---

Model with a steady state = Expected values of the business metrics

Balance for the choice of the metrics  $\rightarrow$  Relationship between the chosen metric and the impact on the clients

Engineering effort required to collect the data : measurability and accuracy

$\rightarrow$  Latency between the metric and the ongoing behavior of the system (almost real time)

1. Start by defining “steady state” as some measurable output of a system that indicates normal behavior.

2. Hypothesize that this steady state will continue in both the control group and the experimental group.

$\rightarrow$  focusing on the way the system is expected to behave and capturing that in a measurement.

## 2) Vary Real-World Events

---

**Introduce variables that reflect real-world events** like servers that crash, hard drives that malfunction, network connections that are severed, etc.

From the distributed system perspective, almost all interesting availability experiments can be driven by **affecting latency or response type**.

Terminating an instance is a special case of infinite latency.

**It follows that most availability experiments can be constructed with a mechanism to vary latency and change status codes.**

\_> Try to disprove the hypothesis by looking for a difference in steady state

# 3) Run Experiments in Production

---

Experimentation teaches you about the system you are studying.

**If you are experimenting on a Staging environment, then you are building confidence in that environment.**

This principle is not without controversy. Certainly, in some fields **there are regulatory requirements that preclude the possibility of affecting the Production systems**. In some situations, there are insurmountable technical barriers to running experiments in Production.

**In most situations, it makes sense to start experimenting on a Staging system, and gradually move over to Production once the kinks of the tooling are worked out.**

# Experimentation, not testing

---

Testing, strictly speaking, does not create new knowledge.

Testing requires that the engineer writing the test knows specific properties about the system that they are looking for.

Complex systems are opaque to that type of analysis.

Tests make an assertion, based on existing knowledge, and then running the test collapses the valence of that assertion, usually into either true or false.

Experimentation, on the other hand, creates new knowledge.

Experiments propose a hypothesis, and as long as the hypothesis is not disproven, confidence grows in that hypothesis. If it is disproven, then we learn something new.

## 4) Automate Experiments to Run Continuously

---

To cover a larger set of experiments than humans can cover manually.

Automation provides a means to scale out the search for vulnerabilities that could contribute to undesirable systemic outcomes.

To empirically verify our assumptions over time, as unknown parts of the system are changed.

## 5) Minimize Blast Radius

---

By using a **tightly orchestrated control** experiments can be constructed in such a way that the **impact of a disproved hypothesis on customer traffic in Production is minimal.**

# About Minimize Blasting Radius

---

The **Chernobyl disaster** of 1986 is one of the most infamous examples of a catastrophic industrial failure. **Workers** at the nuclear power plant **were carrying out an experiment** to see whether the core could still be sufficiently cooled in the event of a power loss. Despite the potential for grave consequences, safety personnel were not present during the experiment, nor did they coordinate with the operators to ensure that their actions would minimize risk.

**During the experiment, what was supposed to just be a shutdown of power led to the opposite;** the power surged and led to several explosions and fires, leading to radioactive fallout that had catastrophic consequences for the surrounding area. Within a few weeks of the failed experiment, thirty-one people died, two of whom were workers at the plant and the rest of whom were emergency workers who suffered from radiation poisoning.

## 5) Minimize Blasting Radius

---

Try to **architect your experimentation** in a way that lets you have a **high level of granularity**, and **target the smallest possible unit**, whatever that is in your system.

You **should always have an eye on your metrics** and be able to **quickly terminate an experiment** if anything seems to be going **wrong in unexpected and impactful ways**.

# “Hope is not a strategy”

---

Disaster testing helps prove a system's resilience when failures are handled gracefully and **exposes reliability risks in a controlled fashion** when things are less than graceful.

**Exposing reliability risks during a controlled incident** allows for thorough analysis and preemptive mitigation, **as opposed to waiting for problems to expose themselves** by means of circumstance alone, when issue severity and time pressure amplify missteps and force risky decisions based on incomplete information

Becoming familiar with and **preparing for uncommon combinations of failures**.

# “Hope is not a strategy”

---

As engineers we are frequently unaware of the implicit assumptions that we build into systems until these assumptions are radically challenged by unforeseen circumstances.

There are events purposefully imposed: software updates, OS patches, regular hardware and facility maintenance, and other threats that commonly occur.

Need to be open to unexpected consequences, and practice recovery as best we can.

One of the most difficult aspects of running a successful Chaos Engineering practice is proving that the results have business value.

# Getting to Basic Fault Tolerance

---

First and foremost, **keep spare capacity online**.

Once you have spare capacity available, **consider how to remove malfunctioning computers from service automatically**. Don't stop at automatic removal,

**Carry on to automatic replacement**.

**Replacement instances must enter service in less than the mean time between failures.**

# DiRT (Google)

---

DiRT tests must have **no service-level** objective breaking **impact on external systems or users**

**Service levels should not be degraded below the standard objectives** already set by the owning teams.

A **key point of Google's approach to Chaos Engineering is that we prefer "controlled chaos" when possible.**

When **designing a test, you should weigh the costs** of internal user disruptions and loss of goodwill carefully **against what you stand to learn.**

# Example of Software Experiments

---

- Run at service levels
- Run without dependencies
- People outages
- Release and rollback
- Incident management procedures
- Datacenter operations
- Capacity management
- Business continuity plans
- Data integrity
- Networks
- Monitoring and alerting
- Reboot everything

# Prioritize Experiments

---

There is no way to identify all possible incidents in advance, much less to cover them all with chaos experiments.

To maximize the efficiency of your effort to reduce system failures you should prioritize different classes of incidents.

**How often does this happen?**

**How likely are you to deal with the event gracefully?**

There will be events that you don't need to investigate, since you accept total loss. On the other hand, there will be events you certainly won't allow to take you down. Prioritize experiments against those to make sure your assumptions on reliability and robustness are being met conscientiously and continuously.

**How likely is this to happen?** once the event is imminent, you must prioritize testing against such events ahead of anything else.

Your answers to these questions should be based on the business needs of your product or service. The satisfaction of reliability goals with respect to business needs should match expectations set with your customers. It's always best to explicitly check with stakeholders whether your plans for reliability land within acceptable margins.

# The Case for Security Chaos Engineering

---

SCE allows teams to proactively, safely discover system weakness before they disrupt business outcomes.

There is no single root cause for failure

The reality is that **most of the malicious** code such as viruses, malware, ransomware, and the like habitually take advantage of low-hanging fruit.

↳ easy things

This can take the form of weak passwords, default passwords, outdated software, unencrypted data, weak security measures in systems, and most of all they take advantage of unsuspecting humans' lack of understanding of how the complex system actually functions.

# The Case for Security Chaos Engineering

---

## Remove the Low-Hanging Fruit!

If we always operated in a culture where we expect humans and systems to behave in unintended ways, then perhaps we would act differently and have more useful views regarding system behavior.

Instead of simply reacting to failures, the security industry has been overlooking valuable chances to further understand and nurture incidents as opportunities to proactively strengthen system resilience.

# The Case for Security Chaos Engineering

---

The most common way we discover security failures is when a security incident is triggered.

Security incidents are not effective signals of detection, because at that point it's already too late.

SCE introduces observability plus rigorous experimentation to illustrate the security of the system. Observability is crucial to generating a feedback loop.

Injecting security events into our systems helps teams understand how their systems function as well as increase the opportunity to improve resilience.

By running security experiments continuously, we can evaluate and improve our understanding of unknown vulnerabilities before they become crisis situations.

# The Case for Security Chaos Engineering

---

**SCE addresses a number of gaps in contemporary security methodologies such as Red and Purple Team exercises.**

The goal of these exercises is the collaboration of offensive and defensive tactics to improve the effectiveness of both groups in the event of an attempted compromise.

These techniques remain valuable but differ in terms of goals and techniques. Combined with SCE, they provide a more objective and proactive feedback mechanism to prepare a system for an adverse event than when implemented alone.

**SCE utilizes simple isolated and controlled experiments instead of complex attack chains involving hundreds or even thousands of changes.**

# Security Chaos Engineering Tool: ChaoSlingr

---

**ChaoSlingr** was originally designed to operate in Amazon Web Services (AWS).

**It proactively introduces known security failure conditions through a series of experiments to determine how effectively security is implemented.**

ChaoSlingr was developed to uncover, communicate, and address significant weaknesses proactively, before they impacted customers in production.

ChaoSlingr demonstrates how Chaos Engineering experiments can be constructed and executed to provide security value in distributed systems.

**The majority of organizations utilizing ChaoSlingr have since forked the project and constructed their own series of security chaos experiments using the framework provided by the project as a guide.**

# Final Comments

---

All of these advanced **principles** are presented to **guide and inspire, not to dictate.**

It is currently under introduction of several no tech realities

Chaos Engineering serves as an instrument to discover real systems behavior and real dependency graphs.

Chaos Engineering is also applied extensively in companies and industries that aren't considered digital native, like large financial institutions, manufacturing, and healthcare.

**Remember you're not doing anything that would not happen in real life.**

# Chaos Engineering References

---

- \_> Ali Basiri, Niosha Behnam, Rudd de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, Casey Rosenthal, “**Chaos Engineering**,” in IEEE Software, vol. 33, no. 3, pp. 35-41, May-June 2016,  
<https://doi.org/10.1109/MS.2016.60>
- \_> Casey Rosenthal, Lorin Hochstein, Aaron Blohowiak, Nora Jones, Ali Basiri, **Chaos Engineering**, O'Reilly Media, Inc., August 2017
- \_> Casey Rosenthal and Nora Jones, **Chaos Engineering – System Resiliency in practice**, O'Reilly Media, Inc., April 2020,
- \_> <https://www.gremlin.com/state-of-chaos-engineering/2021/>

# References

- Availability, MTTR, and MTBF for SaaS Defined  
<https://medium.com/@daveowczarek/availability-mttr-and-mtbf-for-saas-defined-66b618ac1533>
- A. Avizienis, J.-C. Laprie, B. Randell, C. E. Landwehr: *Basic Concepts and Taxonomy of Dependable and Secure Computing*. <https://ieeexplore.ieee.org/document/1335465/>
- Billinton, Allan - Reliability Evaluation of Engineering Systems
- D. A. Menascé, V. A. F. Almeida: Capacity Planning for Web Services: metrics, models and methods. Prentice Hall, PTR

# Dependable Distributed Systems

## Master of Science in Engineering in Computer Science

AA 2022/2023

---

LECTURE 21 : INTRO TO EXPERIMENTAL DESIGN

# Note on output analysis

---

Each simulation run is just a particular realization of the random variables

If not properly analysed, results of the executed runs may hide an high variance and lead to wrong inference about the system under analysis

# Independence Across Runs Property

Let  $Y_1, Y_2, \dots$  be an output stochastic process from a single simulation run

- e.g.,  $Y_i$  might be the throughput (production) in the  $i$ -th hour for a manufacturing system

Let  $y_{1,1}, y_{1,2}, \dots, y_{1,m}$  be a realization of the random variables  $Y_1, Y_2, \dots, Y_m$  resulting from making the simulation run 1 of length  $m$  by using the random numbers  $u_{1,1}, u_{1,2}, \dots$

## OBSERVATION

If we run a second simulation (run 2) with a different set of random numbers  $u_{2,1}, u_{2,2}, \dots$ , then we will obtain a different realization  $y_{2,1}, y_{2,2}, \dots, y_{2,m}$  of the random variables  $Y_1, Y_2, \dots, Y_m$ .

$\dots, Y_m$

# Independence Across Runs Property

Let  $Y_1, Y_2, \dots$  be an output stochastic process from a single simulation run

- e.g.,  $Y_i$  might be the throughput (production) in the  $i$ -th hour for a manufacturing system

Let  $y_{1,1}, y_{1,2}, \dots, y_{1,m}$  be a realization of the random variables  $Y_1, Y_2, \dots, Y_m$  resulting from making the simulation run 1 of length  $m$  by using the random numbers  $u_{1,1}, u_{1,2}, \dots$

Performing  $n$  independent runs we get

simulation run		not depend by the model	
RUN	1	$y_{1,1}$	$\dots$
RUN	2	$y_{2,1}$	$\dots$
	$\vdots$	$\vdots$	$\vdots$
RUN	$n$	$y_{n,1}$	$\dots$
		$y_{n,i}$	$\dots$
			$y_{n,m}$

in line all output

The observations from a particular replication (row) are clearly not IID

The observation of a particular realization  $i$  across multiple runs is IID

# Experimental Design and Analysis

---

- Design a proper set of experiments for measurement or simulation
- Develop a model that best describes the data obtained
- Estimate the contribution of each alternative to the performance
  - Isolate the measurement errors
  - Estimate confidence intervals for model parameters
  - Check if the alternatives are significantly different
  - Check if the model is adequate

↳ component, algorithm, ...

# An old example

---

Personal workstation design

1. Processor: 68000, Z80, or 8086.
2. Memory size: 512K, 2M, or 8M bytes
3. Number of Disks: One, two, three, or four

→ all these choice have  
an impact

want to find best alternative to achieve a goal

# Terminology

things that want to maximize or minimize

**Response Variable:** Outcome. E.g., throughput, response time

**Factors:** Variables that affect the response variable. E.g., **CPU type**, memory size, number of disk drives.

**Levels:** The values that a factor can assume, E.g., the **CPU type** has three levels: 68000, 8080, or Z80

**Replication:** Repetition of all or some experiments

**Design:** The number of experiments, the factor level and number of replications for each experiment

**Experimental Unit:** Any entity that is used for experiments

# Terminology

---

**Interaction:** Two factors A and B are said to interact if the effect of one depends upon the level of the other

Table 1: Noninteracting Factors

	$A_1$	$A_2$
$B_1$	3	5
$B_2$	6	8

Table 2: Interacting Factors

	$A_1$	$A_2$
$B_1$	3	5
$B_2$	6	9

# Types of Experimental Designs

Given  $k$  factors, with the  $i$ -th factor having  $n_i$  levels

□ **Simple Designs:** Vary one factor at a time

$$\# \text{ of Experiments} = 1 + \sum_{i=1}^k (n_i - 1)$$

change level of single factor for experiment

The response of a base configuration is measured first

- Not statistically efficient.
- Wrong conclusions if the factors have interaction.
- Not recommended.

□ **Full Factorial Design:** All combinations.

$$\# \text{ of Experiments} = \prod_{i=1}^k n_i$$

- Can find the effect of all factors.
- Too much time and money.
- May try  $2^k$  design first.

3 bad

# Types of Experimental Designs

---

## □ Fractional Factorial Designs: Less than Full Factorial

- Save time and expense.
- Less information.
- May not get all interactions.
- Not a problem if negligible interactions

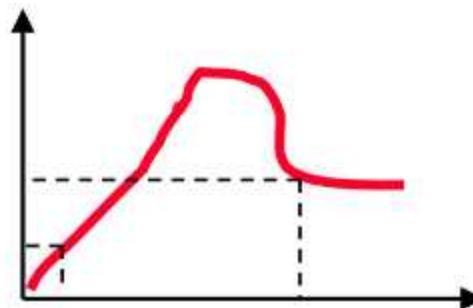
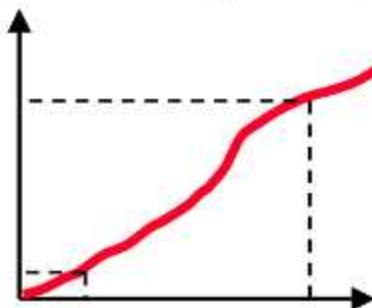
{ good

↓  
just part of all  
possible combination

# $2^k$ Factorial Designs

in precedent example  $k=3$  CPU, RAM, DISC

- $k$  factors, each at two levels. At your choice
- Easy to analyze.
- Helps in sorting out impact of factors.
- Good at the beginning of a study.
- Valid only if the effect is unidirectional.  
E.g., memory size, the number of disk drives



# $2^2$ Factorial Factorial Designs

---

- Two factors, each at two levels.

Performance in MIPS		
Cache Size	Memory Size	
	4M Bytes	16M Bytes
1K	15	45
2K	25	75

*$\pm 1$  at each possible configuration*

$$x_A = \begin{cases} -1 & \text{if 4M bytes memory} \\ 1 & \text{if 16M bytes memory} \end{cases}$$
$$x_B = \begin{cases} -1 & \text{if 1K bytes cache} \\ 1 & \text{if 2K bytes cache} \end{cases}$$

# 2<sup>2</sup> Factorial Factorial Designs, Computation of effects

<u>Experiment</u>	A	B	<u>y</u>
1	-1	-1	$y_1$
2	1	-1	$y_2$
3	-1	1	$y_3$
4	1	1	$y_4$

1MB MEMORY, 1 KB CACHE

y = response variable

base output independent by A and B

$$y = q_0 + q_A x_A + q_B x_B + q_{AB} x_A x_B$$

effect given by combination of the two factors

y = response

qi = effects

$$y_1 = q_0 - q_A - q_B + q_{AB}$$

$$y_2 = q_0 + q_A - q_B - q_{AB}$$

$$y_3 = q_0 - q_A + q_B - q_{AB}$$

$$y_4 = q_0 + q_A + q_B + q_{AB}$$

single effect of the single choice

# 2<sup>2</sup> Factorial Factorial Designs, Computation of effects

---

$$q_0 = \frac{1}{4}(y_1 + y_2 + y_3 + y_4)$$

$$q_A = \frac{1}{4}(-y_1 + y_2 - y_3 + y_4)$$

$$q_B = \frac{1}{4}(-y_1 - y_2 + y_3 + y_4)$$

$$q_{AB} = \frac{1}{4}(y_1 - y_2 - y_3 + y_4)$$

# 2<sup>2</sup> Factorial Factorial Designs, Computation of effects

---

Experiment	A	B	y
1	-1	-1	$y_1$
2	1	-1	$y_2$
3	-1	1	$y_3$
4	1	1	$y_4$

$$q_A = \frac{1}{4}(-y_1 + y_2 - y_3 + y_4)$$

$$q_B = \frac{1}{4}(-y_1 - y_2 + y_3 + y_4)$$

Notice:

$$q_A = \text{Column A} \times \text{Column y}$$

$$q_B = \text{Column B} \times \text{Column y}$$

*vectorial*

# Allocation of variation

→ For evaluation of the importance of a factor

The importance of a factor is measured by the proportion of the total variation in the response that is explained by the factor

$$\text{Total Variation of } y = \text{SST} = \sum_{i=1}^{2^2} (y_i - \bar{y})^2$$

(EXAMPLE OF BEFORE

the mean of  
responses  
from all four  
experiments

- For a  $2^2$  design:

$$\text{SST} = 2^2 q_A^2 + 2^2 q_B^2 + 2^2 q_{AB}^2 = \text{SSA} + \text{SSB} + \text{SSAB}$$

- Variation due to A = SSA =  $2^2 q_A^2$
- Variation due to B = SSB =  $2^2 q_B^2$
- Variation due to interaction = SSAB =  $2^2 q_{AB}^2$
- Fraction explained by A =  $\frac{\text{SSA}}{\text{SST}}$  Variation ≠ Variance

# Allocation of variation example

## □ Memory-cache study:

understand where to act first

$$\bar{y} = \frac{1}{4}(15 + 55 + 25 + 75) = 40$$

$$\begin{aligned}\text{Total Variation} &= \sum_{i=1}^4 (y_i - \bar{y})^2 \\ &= (25^2 + 15^2 + 15^2 + 35^2) \\ &= 2100 \\ &= 4 \times 20^2 + 4 \times 10^2 + 4 \times 5^2\end{aligned}$$

## □ Total variation = 2100

Variation due to Memory = 1600 (76%)  $\rightsquigarrow$  improving A is better

Variation due to cache = 400 (19%)

Variation due to interaction = 100 (5%)  $\rightsquigarrow$  not very dependent A and B

# $2^k$ Design Example

---

- ❑ Three factors in designing a machine:

- Cache size
- Memory size
- Number of processors

Factor	Level -1	Level 1
A   Memory Size	4MB	16MB
B   Cache Size	1kB	2kB
C   Number of Processors	1	2

# $2^k$ Design Example

---

Cache Size	4M Bytes		16M Bytes	
	1 Proc	2 Proc	1 Proc	2 Proc
1K Byte	14	46	22	58
2K Byte	10	50	34	86

I	A	B	C	AB	AC	BC	ABC	y
1	-1	-1	-1	1	1	1	-1	14
1	1	-1	-1	-1	-1	1	1	22
1	-1	1	-1	-1	1	-1	1	10
1	1	1	-1	1	-1	-1	-1	34
1	-1	-1	1	1	-1	-1	1	46
1	1	-1	1	-1	1	-1	-1	58
1	-1	1	1	-1	-1	1	-1	50
1	1	1	1	1	1	1	1	86
320	80	40	160	40	16	24	9	Total
40	10	5	20	5	2	3	1	Total/8

# $2^k$ Design Example

---

$$\begin{aligned}\text{SST} &= 2^3(q_A^2 + q_B^2 + q_C^2 + q_{AB}^2 + q_{AC}^2 + q_{BC}^2 + q_{ABC}^2) \\ &= 8(10^2 + 5^2 + 20^2 + 5^2 + 2^2 + 3^2 + 1^2) \\ &= 800 + 200 + 3200 + 200 + 32 + 72 + 8 = 4512 \\ &= 18\% + 4\% + 71\% + 4\% + 1\% + 2\% + 0\% \\ &= 100\%\end{aligned}$$

- ❑ Number of Processors (C) is the most important factor.
-

# $2^k r$ Factorial Designs

---

- $r$  replications of  $2^k$  Experiments  
⇒  $2^k r$  observations.  
⇒ Allows estimation of experimental errors.
- Model:  
$$\underline{y = q_0 + q_A x_A + q_B x_B + q_{AB} x_A x_B + e}$$
- $e$  = Experimental error

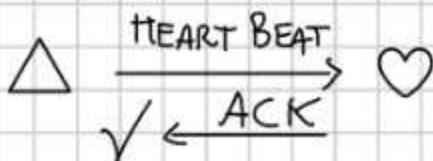
# References

- R. Jain, "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling,"  
[https://www.cin.ufpe.br/~rmfl/ADS\\_MaterialDidatico/PDFs/performan](https://www.cin.ufpe.br/~rmfl/ADS_MaterialDidatico/PDFs/performanceAnalysis/Art%20of%20Computer%20Systems%20Performance%20Analysis%20Techniques.pdf)  
<ceAnalysis/Art%20of%20Computer%20Systems%20Performance%20Analysis%20Techniques.pdf>
- A. M. Law - Simulation modeling and analysis  
<https://industri.fatek.unpatti.ac.id/wp-content/uploads/2019/03/108-Simulation-Modeling-and-Analysis-Averill-M.-Law-Edisi-5-2014.pdf>

## Example 1 Perfect Failure detector

Protocol with timeout =  $\Delta$ , after  $\Delta$  elapse send a message to all processes in the system to see if they are alive and wait for an ACK. Assumption system is synchronized.

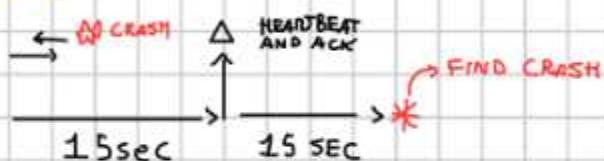
Set of processes  $\Delta P$



$d$  = delay is time for deliver a message, to a side to another.  $\Delta$  is at least equal to round trip time ( $2d$ ).

Hp: assume we can identify any possible crash occurred in the system at most in 30 second.

What is maximum value for  $\Delta$ ?



For identifying crashes in 30 sec,  $\Delta \leq 15 \text{ sec}$

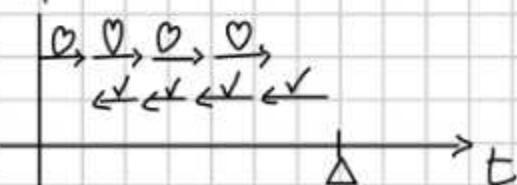
### Actual system:

the channel of our system have a capacity of  $M = 3 \text{ messages/sec}$

$\frac{1}{M}$  the two channel are independent  $\hookrightarrow$  FIXED BANDWIDTH

- Which maximum number of processes that we can deploy in this system, with this  $M$  for individual crashes in 30 sec?  $M = 44$

P:

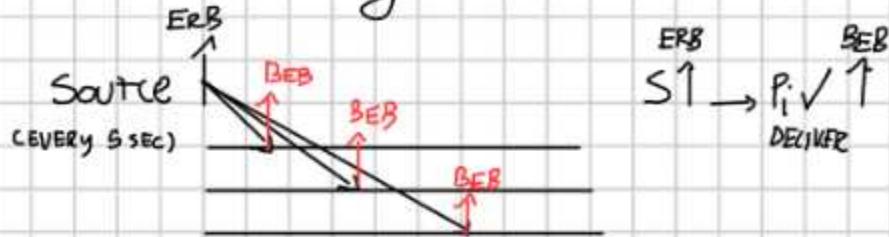
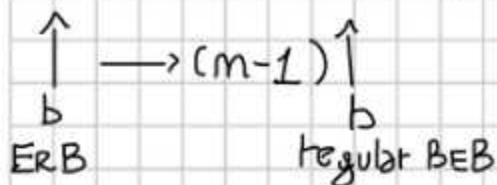


$$\frac{M+1}{15} \leq 3 \rightarrow M = 44$$

$\hookrightarrow$  length of  $\Delta$

$M+1$  because he must receive last heartbeat.

**Example 2** Eager reliable broadcast (messages can be lost) the first time a process receive a broadcast message, he broadcast again the same message.



We want to use this primitive (ERB) with this rate  $\lambda = 0.2 \text{ REQUEST/Sec}$ . The broadcast primitive is triggered with an exponential distribution and the rate of this exp distribution is  $\lambda$ . Every  $5 \text{ sec}$  a process start an ERB, we assume only one process (source) start it.

Assume to have a channel bandwidth  $M^{(n)} = 10 \text{ messages/Sec}$ , No - Crosses, what is **EXPECTED TIME TO DELIVER** ?

we have 2 channels for send and receiving.

1)  $M = 30 \text{ processes}$

2)  $M_{\max} = ?$

1. we assume  $M/M/1$  queue

$$R = \text{EXPECT TIME TO DELIVER} = \frac{1}{M_i - 30 \cdot \lambda_i} = \frac{1}{10 - 30 \cdot 0.2} = 0.25 \text{ sec}$$

### Example 3 SOFTWARE REPLICATION

FAULT = CRASH

MTBF = 50 μs

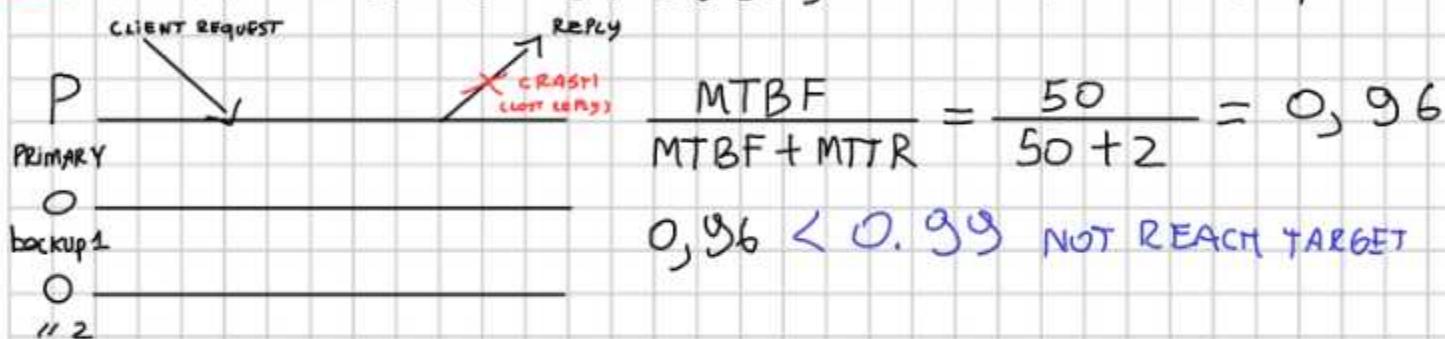
1) PASSIVE

MTTR = 2 μs

2) ACTIVE

We want a TARGET 99% for STEADY STATE AVAILABILITY.

1. Passive is less reliable, we have a PRIMARY



Increase number of replica is not a solution, what we can do is:

increase MTBF and decrease MTTR

better

• Here need a primary that must be alive.

• Less expensive to implement, interaction only between P and C.

2.  $1 - (1 - 0,96)^n = 1 - (0,04)^n$



We have a better availability respect to PASSIVE model.

• Here at least one process must be alive, is sufficient to receive one answer.

• We need a broadcast.

# Dependable Distributed Systems

## Master of Science in Engineering in Computer Science

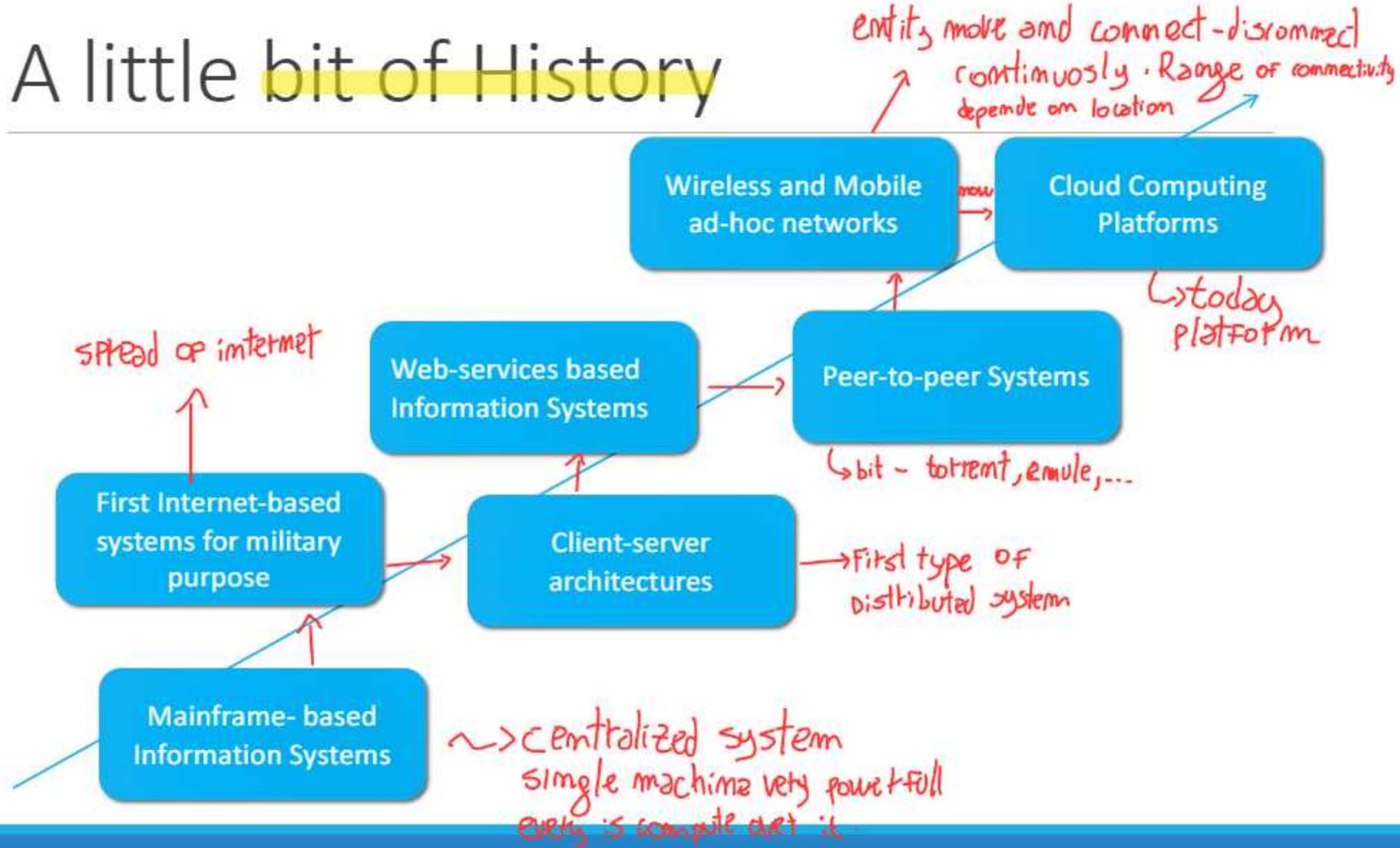
AA 2022/2023

---

LECTURE 22: CAP THEOREM

↳ Consistency - Availability - Partition tolerant

# A little bit of History



# Relational Databases History

---

Relational Databases – mainstay of business

Web-based applications caused spikes

- Especially true for public-facing e-Commerce sites

Developers begin to front RDBMS with memcache or integrate other caching mechanisms within the application

↳ problem with manage the replicas

# Scaling Up

how we can scale and increase number of query?

Issues with scaling up when the dataset is just too big

RDBMS were not designed to be distributed

Began to look at multi-node database solutions

Known as 'scaling out' or 'horizontal scaling'

backup are expensive

Different approaches include: for consistency

- Master-slave (primary backup style) → master keep in charge of all update and share with all replica
- Sharding

↳ split the load horizontal between replica  
replicate between replica only a part of database

# Scaling RDBMS – Master/Slave

---

## Master-Slave

- All writes are written to the master. All reads performed against the replicated slave databases
- Critical reads may be incorrect as writes may not have been propagated down
- Large data sets can pose problems as master needs to duplicate data to slaves

# Scaling RDBMS - Sharding

---

## Partition or sharding

- Scales well for both reads and writes
- Not transparent, application needs to be partition-aware
- Can no longer have relationships/joins across partitions
- Loss of referential integrity across shards

# Other ways to scale RDBMS

---

## Multi-Master replication

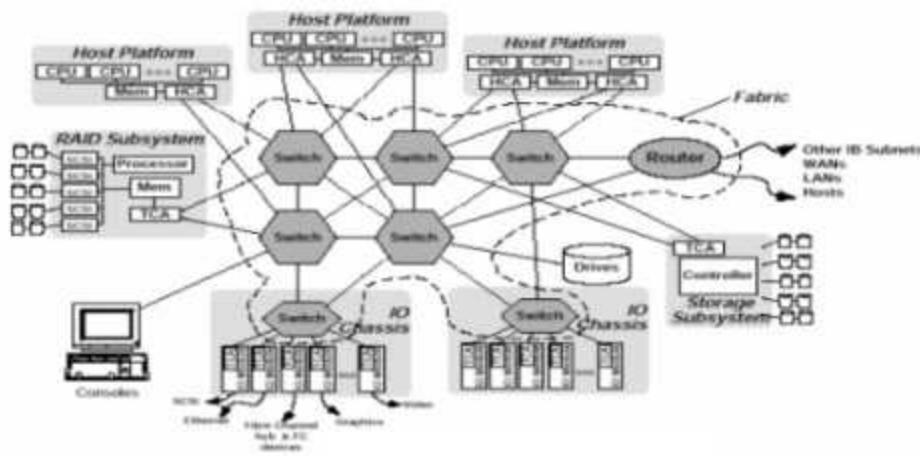
INSERT only, not UPDATES/DELETES *or just few*

No JOINs, thereby reducing query time

- This involves de-normalizing data *→ most costly operations*

## In-memory databases

Today... now we have milion of users and much more replica



# Replication

---

PRELIMINARY NOTIONS

# Replication as a way to scale up

---

N clients  $c_1, c_2, \dots, c_n$  access a set  $X$  of objects (tables, tuples, etc.)

*↳ could be the database (relational or not)*

Each object  $x \in X$  has its own internal state

Clients access the object  $x$  through the invocation of operations

The set of operations that allow clients to interact with the object define its semantics

# Replication Model

for every  
table, key  
value, ...

→ Each object x is developed by a set  $\{x^1, x^2 \dots x^m\}$  of physical copies called "replicas"

Each replica is located in a different physical location

# Requirements

---

## Transparency:

- Clients must have the illusion to interact with a single object
- Object interfaces do not change

## Consistency:

- Operations must produce results as if they are executed on a single object

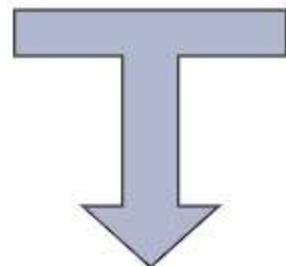
# Consistency: Linearizability

## Sequential System:

- At each time unit only one process interacts with the object
- Operations are specified by pre conditions and post conditions

## Concurrent System:

- Multiple clients concur to access the object
- Management of concurrent updates



## LINEARIZABILITY

Specify how a concurrent object must behave according with its sequential specification

# Linearizability: Idea

---

Clients should have the illusion of interacting with a unique physical object even in case of concurrency

---

Each operation must produce the same effect it would produce if executed in isolation

---

The order between sequential operations must be preserved

---

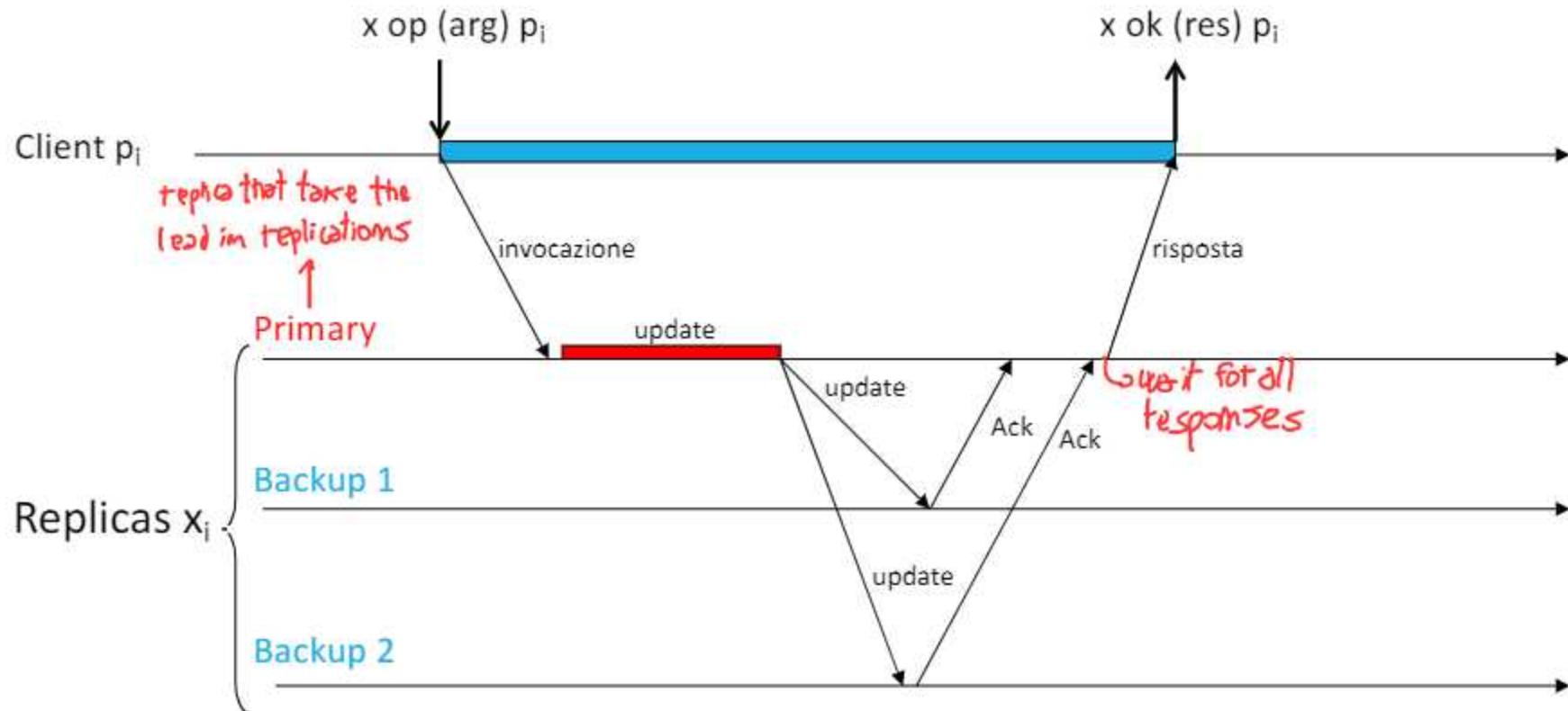
## Basic Replication Techniques

---

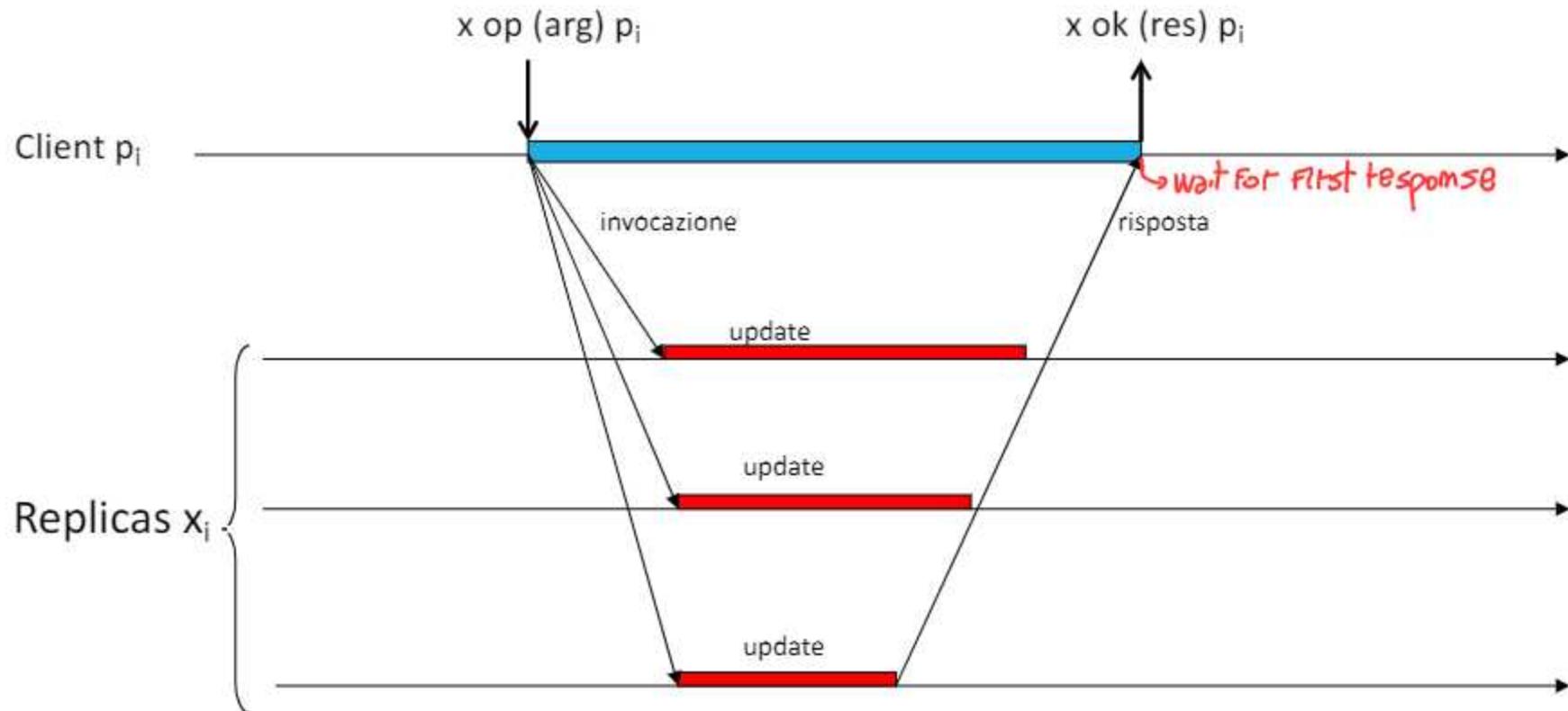
Two basic approaches to replication:

- Primary Backup (passive replication)
- Active Replication

# Passive Replication

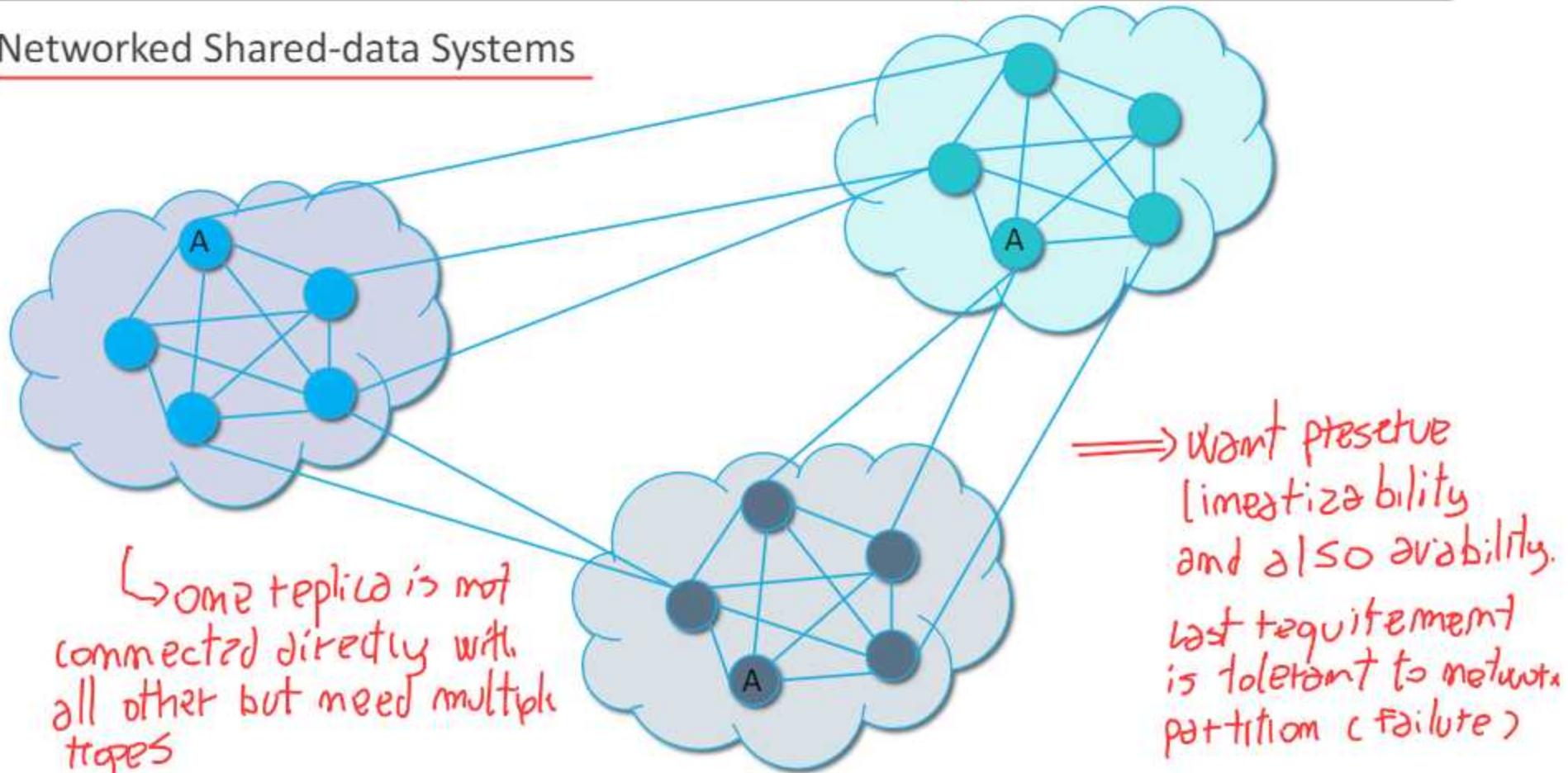


# Active Replication



# Context

## Networked Shared-data Systems



# Fundamental Properties

---

## Consistency

- (informally) “every request receives the right response”
- E.g. If I get my shopping list on Amazon I expect it contains all the previously selected items

## Availability

- (informally) “each request eventually receives a response”
- E.g. eventually I access my shopping list

## tolerance to network Partitions

- (informally) “servers can be partitioned in to multiple groups that cannot communicate with one other”

# CAP Theorem

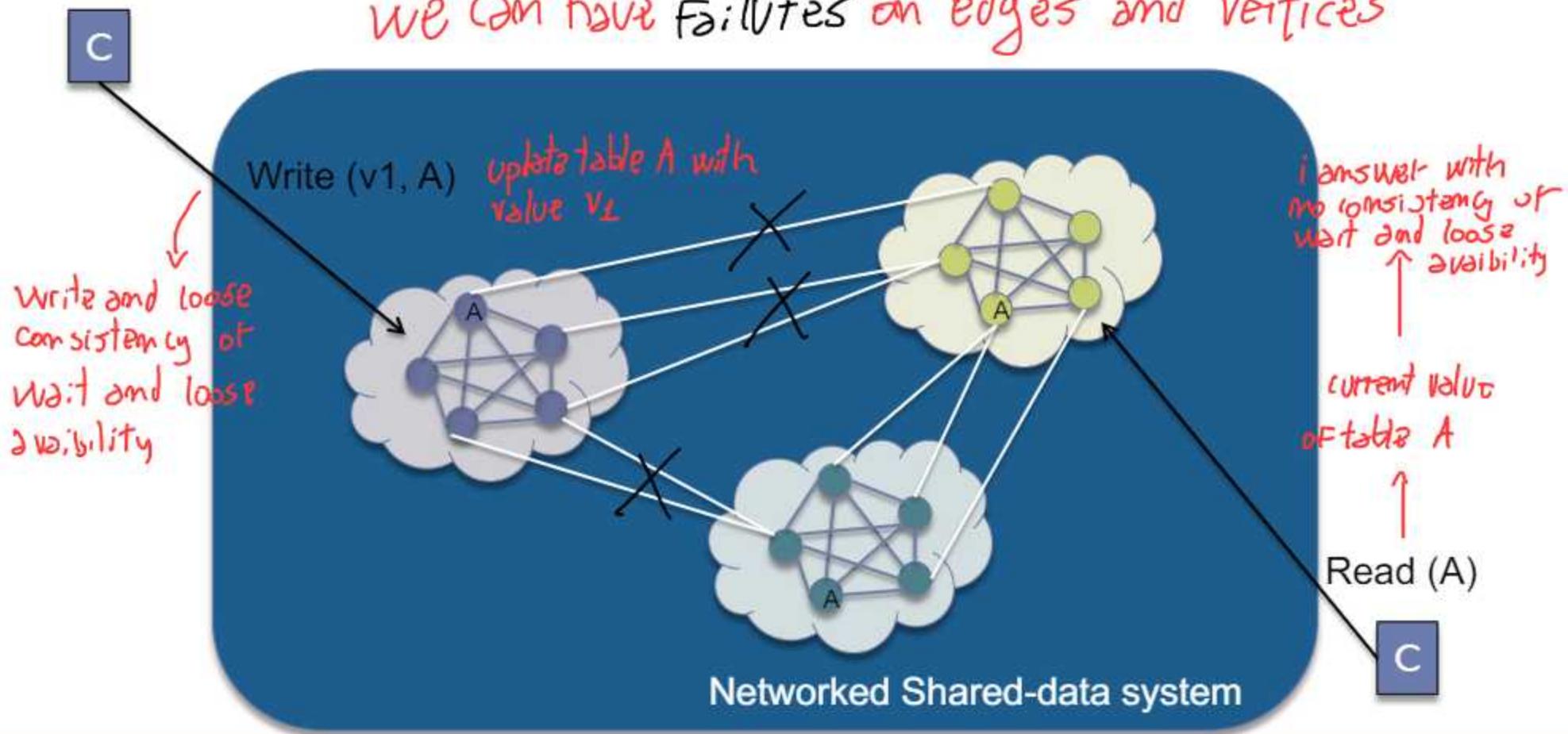
---

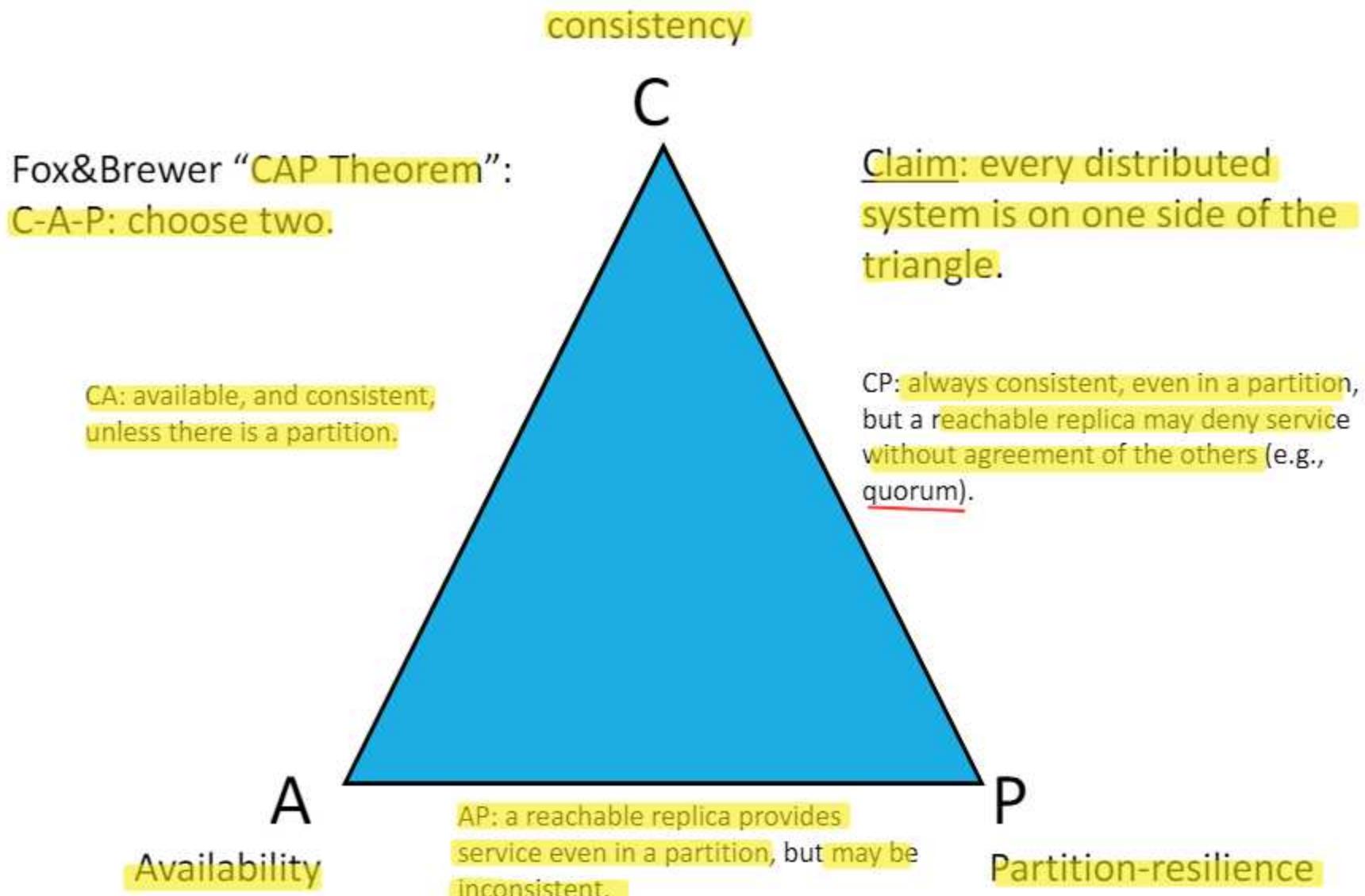
- 2000: Eric Brewer, PODC conference keynote
- 2002: Seth Gilbert and Nancy Lynch, ACM SIGACT News 33(2)

*“Of three properties of shared-data systems (Consistency, Availability and tolerance to network Partitions) only two can be achieved at any given moment in time.”*

# Proof Intuition

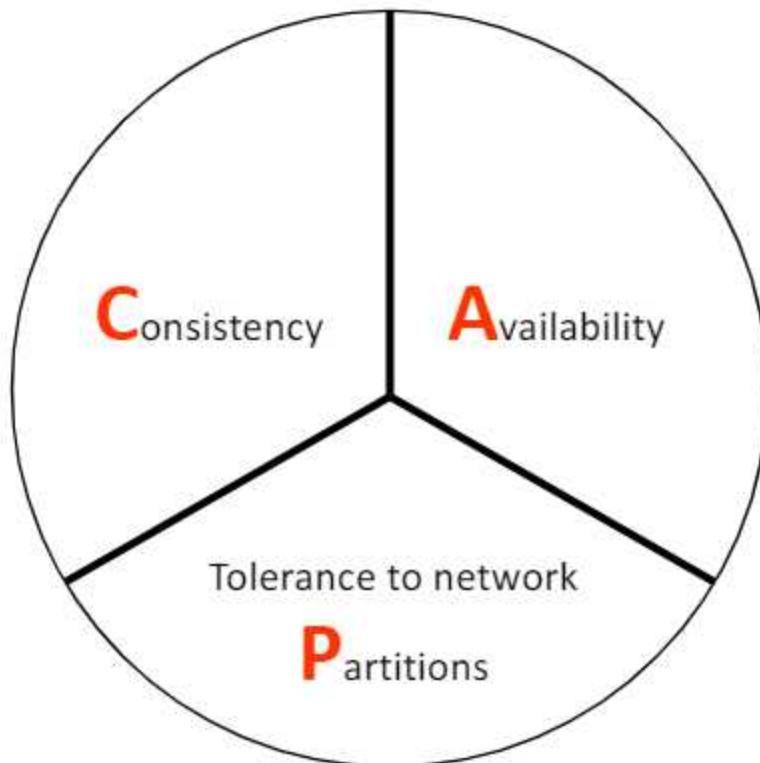
we can have failures on edges and vertices





# The CAP Theorem

---



**Theorem:** You can have at most two of these invariants for any shared-data system

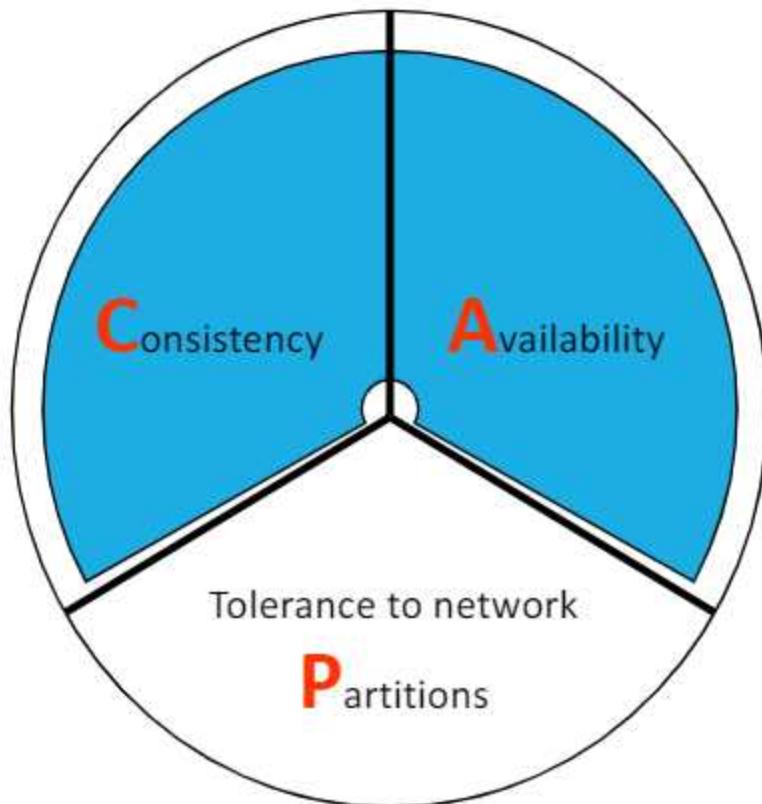
---

**Corollary:** consistency boundary must choose A or P

---

# Forfeit Partitions

CA system



## Examples

- Single-site databases can not be partitioned
- Cluster databases
  - LDAP
  - Fiefdoms

## Traits

- 2-phase commit ~ could generate dead-lock
- cache validation protocols
- The “inside”

# Observations

---

CAP states that in case of failures you can have at most two of these three properties for any shared-data system

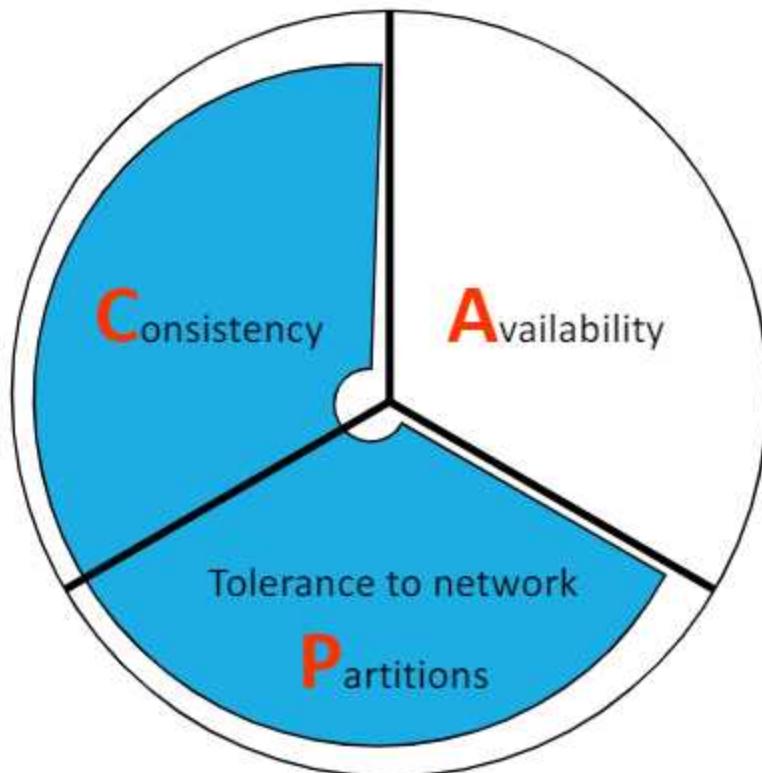
*For large scale system you don't really have an option, must preserve P for scalability purpose*

To scale out, you have to distribute resources.

- P is not really an option but rather a need
- The real selection is among consistency or availability
- In almost all cases, you would choose availability over consistency

# Forfeit Availability

CP system



## Examples

Distributed databases → relational

Distributed locking

Majority protocols

wait until everybody replies

## Traits

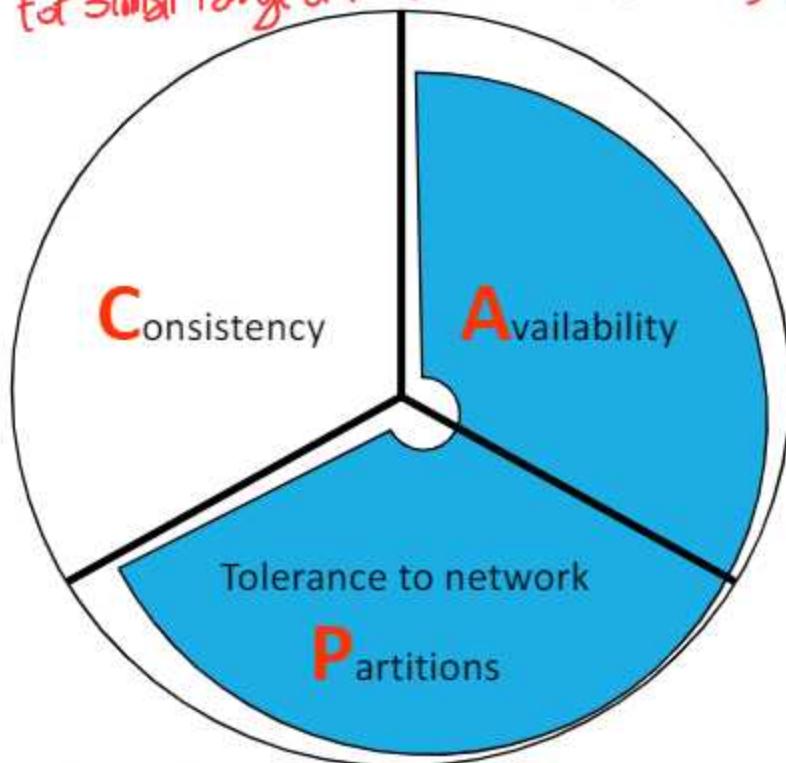
Pessimistic locking algorithm

Make minority partitions unavailable

# Forfeit Consistency

AP system

for small range of time not have linearity property



can tolerate some degree of inconsistency

## Examples

Coda

Web caching

DNS

→ want quickly an answer

Emissaries

## Traits

expirations/leases

conflict resolution

Optimistic

The “outside”

# Consistency Boundary Summary

---

We can have consistency & availability within a cluster.

- No partitions within boundary!

OS/Networking better at A than C

---

Databases better at C than A

---

Wide-area databases can't have both

---

Disconnected clients can't have both

---

↳ in client-server style

# Visual Guide to NoSQL Systems



→ atomicity - consistency - integrity - durability

## CAP, ACID and BASE

BASE stands for Basically Available Soft State Eventually Consistent system.

Basically Available: the system available most of the time and there could exists a subsystems temporarily unavailable

Soft State: data are “volatile” in the sense that their persistence is in the hand of the user that must take care of refresh them

Eventually Consistent: the system eventually converge to a consistent state

# CAP, ACID and BASE

---

Relation among ACID and CAP is more complex

Atomicity: every operation is executed in “all-or-nothing” fashion

Consistency: every transaction preserves the consistency constraints on data

Integrity: transaction does not interfere. Every transaction is executed as it is the only one in the system

Durability: after a commit, the updates made are permanent regardless possible failures

# CAP, ACID and BASE

---

CAP

C here looks to single-copy consistency

A here look to the service/data availability

C = (C U I of ACID)

ACID

C here looks to constraints on data and data model

A looks to atomicity of operation and it is always ensured

I is deeply related to CAP. It can be ensured in at most one partition

D is independent from CAP

# Warning!

What CAP says:

- When you have a partition in the network you cannot have both C and A

You have not to choose FOREVER, we have  
also period with no FAILURE, at some point you  
need to choose.

- During Normal Periods (i.e. period with no partitions) both C and A can be achieved

# 2 out of 3 is misleading

Partitions are rare events *but may happen*

- there are little reasons to forfeit by design C or A

*99% of the time  
our system is Apac*

Systems evolve along time

- Depending on the specific partition, service or data, the decision about the property to be sacrificed can change

C, A and P are measured according to continuum

- Several level of Consistency (e.g. ACID vs BASE)
- Several level of Availability
- Several degree of partition severity

# 2 of 3 is misleading

---

In principle every system should be designed to ensure both C and A in normal situation

When a partition occurs the decision among C and A can be taken

When the partition is resolved the system takes corrective action coming back to work in normal situation

# Consistency/Latency Trade Off

---

CAP does not force designers to give up A or C but why there exists a lot of systems trading C?

**LATENCY !**

System can be not partitioned, but network can be congested

CAP does not explicitly talk about latency...

... however latency is crucial to get the essence of CAP

Great problem is to trade off consistency with latency

# Consistency/Latency Trade Off

1.  High Availability
  - High Availability is a strong requirement of modern shared-data systems

2.  Replication
  - To achieve High Availability, data and services must be replicated , increase load

3.  Consistency
  - Replication impose consistency maintenance

4.  Latency
  - Every form of consistency requires communication and a stronger consistency requires higher latency

how many replica put in the system, depend on constraint impose from customer

# PACELC

Abadi proposes to revise CAP as follows:

in case of partition we need  
↑ to choose A or P

"**PACELC** (pronounced pass-elk): if there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?"

A quick answer can be no consistent constraint between latency and consistency when there are no partition

# Consistency Spectrum

---

# Consistency Criteria

Different perspective between Data base community and Distributed System community

- Data-centric vs client-centric ↗ want to optimize experience of client
  - ↳ optimize interaction with data

ACID → Strong Consistency

↳ Data-centric

Several Degrees of Weak Consistency

- Eventual Consistency
- Read-your-writes
- Monotonic read
- Monotonic write

in normal state we must trade off latency and consistency.  
Quick response  $\Rightarrow$  less consistency

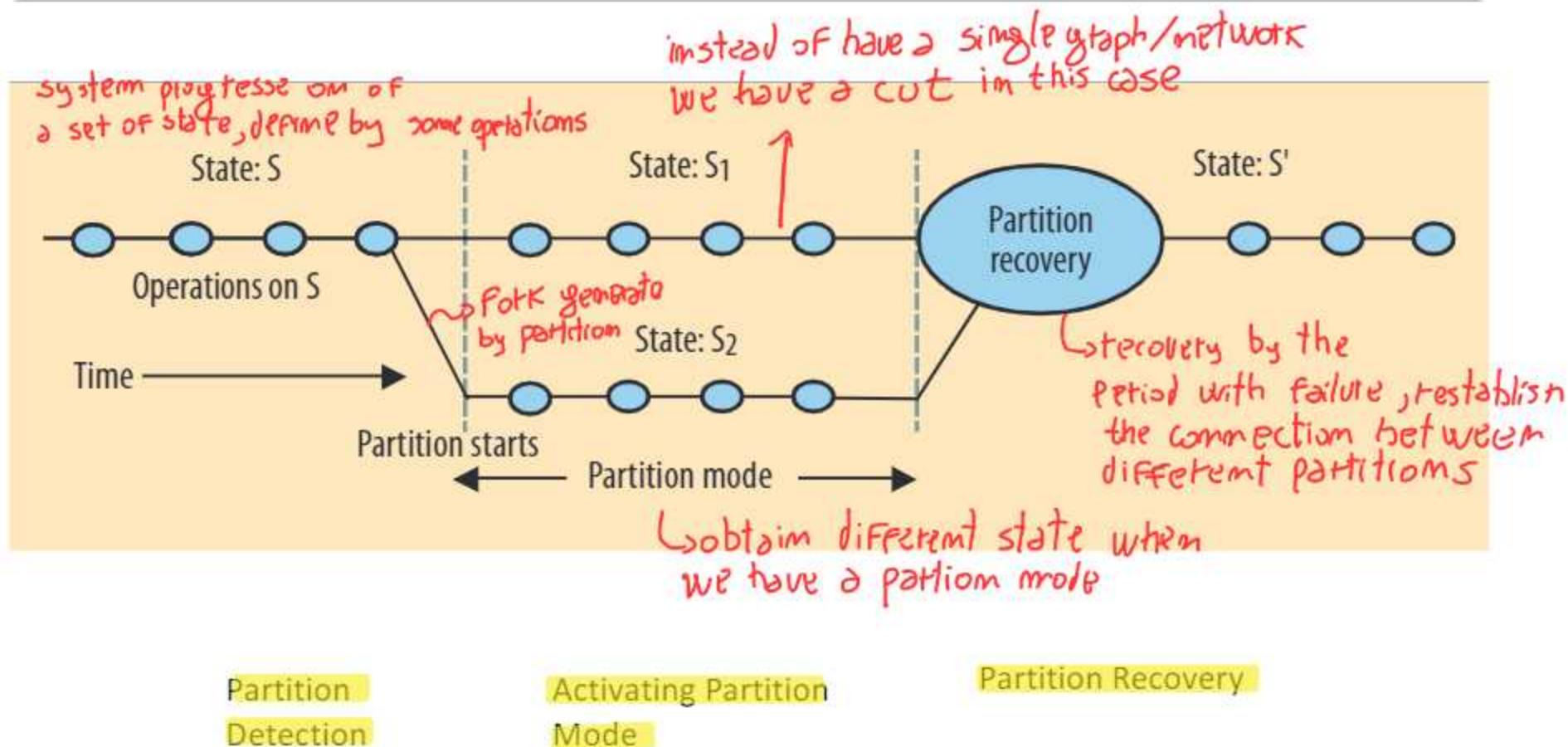
# Partition Management

---

A-C-P  $\rightarrow$  could guarantee only two when we encounter failures, in normal state we can achieve A-C-P

→ event that happen in system consequence of failures

## Partitions Management



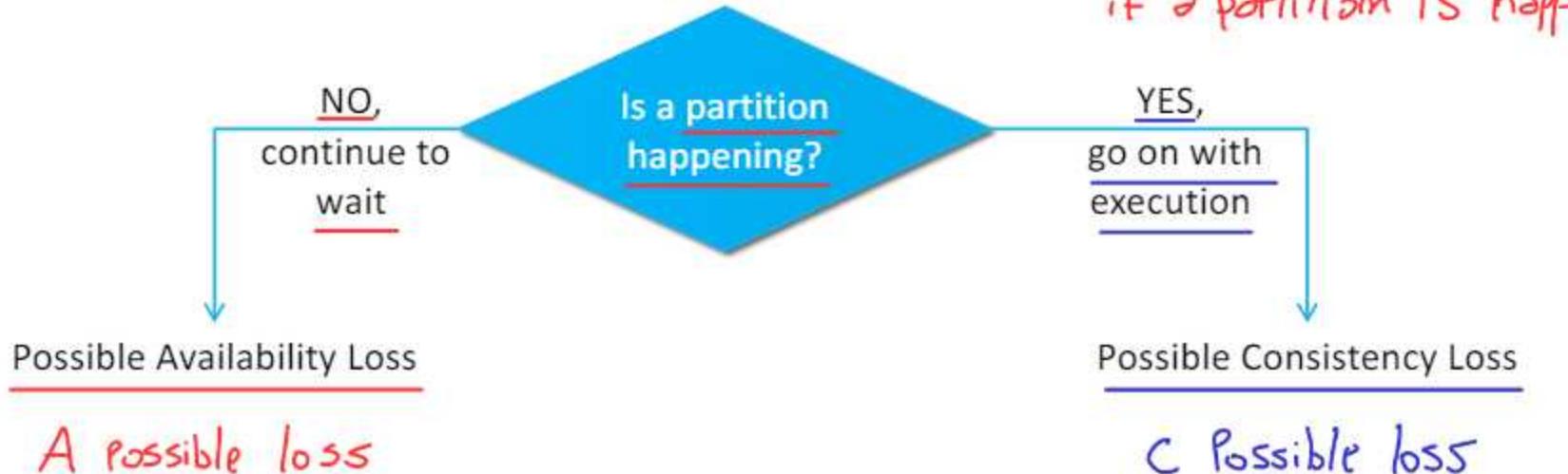
# Partition Detection

CAP does not explicitly talk about latencies

However...

- To keep the system live time-outs must be set
- When a time-out expires the system must take a decision

*use a timer for Fin J  
if a partition is happening*



we have a single graph, you can communicate only with neighbours, some nodes could not see any anomaly!

## Partition Detection

Partition Detection is not global

- An interacting part may detect the partition, the other not.
- Different processes may be in different states (partition mode vs normal mode)

we only implement an eventually failure detector

When entering Partition Mode the system may

- Decide to block risk operations to avoid consistency violations
- Go on limiting a subset of operations

# Which Operations Should Proceed?

Live operation selection is an hard task

- Knowledge of the severity of invariant violation
- Examples
  - every key in a DB must be unique
    - Managing violation of unique keys is simple
    - Merging element with the same key or keys update
  - every passenger of an airplane must have assigned a seat
    - Managing seat reservations violation is harder
    - Compensation done with human intervention
  - Log every operation for a possible future re-processing

choose only operation  
that not change  
the state  
↳ difficult

we can assign a  
different key  
while we have conflict

after the period  
where have a portion

typically we prefer consistency, roll back the operation

# Partition Recovery

When a partition is repaired, partitions' logs may be used to recover consistency

**Strategy 1:** roll-back and execute again operations in the proper order (using version vectors)

**Strategy 2:** disable a subset of operations (Commutative Replicated Data Type - CRDT)

↓  
particular data structure where  
all operation is commutative

like vector clocks  
↑

# Basic Techniques: Version Vector

In the version vector we have an entry for any node updating the state

Each node has an identifier

Each operation is stored in the log with attached a pair  $\langle \text{nodeId}, \text{timeStamp} \rangle$

Given two version vector A and B, A is newer than B if

- For any node in both A and B,  $\text{ta}(B) \leq \text{ts}(A)$  and
- There exists at least one entry where  $\text{ta}(B) < \text{ts}(A)$

# Version Vectors: example

<table border="1"><tr><td>1</td></tr><tr><td>0</td></tr><tr><td>0</td></tr></table>	1	0	0	<table border="1"><tr><td>1</td></tr><tr><td>1</td></tr><tr><td>0</td></tr></table>	1	1	0
1							
0							
0							
1							
1							
0							
ts(B)	Ts(A)						

$ts(A) < ts(B)$  then  $A \rightarrow B$

*A is newer than B*

<table border="1"><tr><td>1</td></tr><tr><td>0</td></tr><tr><td>0</td></tr></table>	1	0	0	<table border="1"><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>1</td></tr></table>	0	0	1
1							
0							
0							
0							
0							
1							
ts(B)	Ts(A)						

$ts(A) \neq ts(B)$  then  $A \parallel B$

**POTENTIALLY INCONSISTENT!**

*not comparable*



*(> human intervention)*

# Basic Techniques: Version Vector

Using version vectors it is always possible to determine if two operations are causally related or they are concurrent (and then dangerous)

Using vector versions stored on both the partitions it is possible to re-order operations and raising conflicts that may be resolved by hand

Recent works proved that this consistency is the best that can be obtained in systems focussed on latency

for latency version vector is  
best choice

second strategy without toll back

# Basic Techniques: CRDT

Commutative Replicated Data Type (CRDT) are data structures that provably converges after a partition (e.g. set).

## Characteristics:

- All the operations during a partition are commutative (e.g. add(a) and add(b) are commutative) or *→ topology structure start by a value and keep track of updates*
- Values are represented on a lattice and all operations during a partitions are monotonically increasing wrt the lattice (giving an order among them)
  - Approach taken by Amazon with the shopping cart.
- Allows designers to choose A still ensuring the convergence after a partition recovery

# Basic Techniques: Mistake Compensation

---

Selecting A and forgoing C, mistakes may be taken

- Invariants violation

To fix mistakes the system can

- Apply deterministic rule (e.g. “last write win”)
- Operations merge
- Human escalation

General Idea:

- Define specific operation managing the error
  - E.g. re-found credit card

# References

---

1. Brewer "CAP twelve years later: How the "rules" have changed"  
<http://ieeexplore.ieee.org/document/6133253/> (see NOTE above)
2. Abadi "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story"  
<http://ieeexplore.ieee.org/document/6127847/> (see NOTE above)

**NOTE:** Use the Sapienza proxy to access this paper. Instruction on how to do it can be found here  
<https://web.uniroma1.it/sbs/easybixy/easybixy>

Dependable Distributed Systems  
Master of Science in Engineering in  
Computer Science

AA 2022/2023

---

LECTURES 23: CONSISTENCY CRITERIA FOR DISTRIBUTED  
SHARED MEMORIES

# Motivation

---

Distributed Shared Memories (DSMs) are an alternative to message passing for allowing inter-process communication

---

## DSM Advantages

- support developers with the shared variables programming paradigm
    - abstract the underlying system
  - increasing transparency with respect to portability, load balancing and process migration
- 

The semantic of a shared memory is expressed by a consistency criterion

---

# Consistency criteria

A consistency criterion defines the result returned by an operation

- It can be seen as a contract between the programmer and the system implementing replication



# Notation

---

A shared memory system is composed by a set of sequential processes  $p_1, p_2, \dots, p_n$  interacting with a finite set of shared objects

---

Each object can be accessed by invoking read and write operations

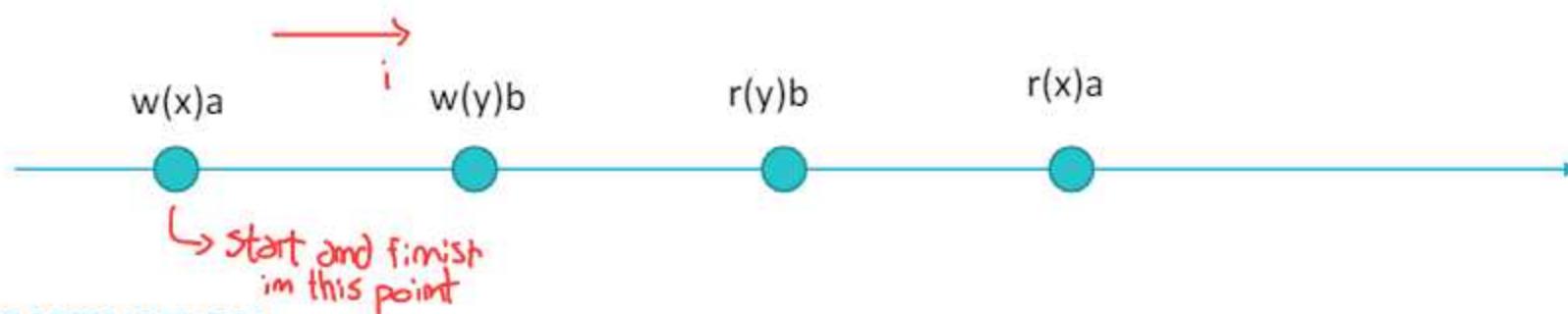
- $w_i(x)v$  denote a write operation issued by  $p_i$  on the object  $x$  and writing the value  $v$
- $r_j(x)v$  denote a read operation issued by  $p_j$  on the object  $x$  and returning the value  $v$
- $op_i(x)v$  denote a generic operation issued by  $p_i$  on the object  $x$  and writing/returning the value  $v$

# Histories

Informally, an history represents the partial order of all the operations executed on the shared objects

The Local history  $\hat{h}_i$  is the sequence of operations issued by  $p_i$

- if  $op_1$  and  $op_2$  are issued by  $p_i$  and  $op_1$  is issued first, then we say that  $op_1$  precedes  $op_2$  in  $p_i$ 's process order and we will denote it as  $op_1 \rightarrow_i op_2$



## OBSERVATION

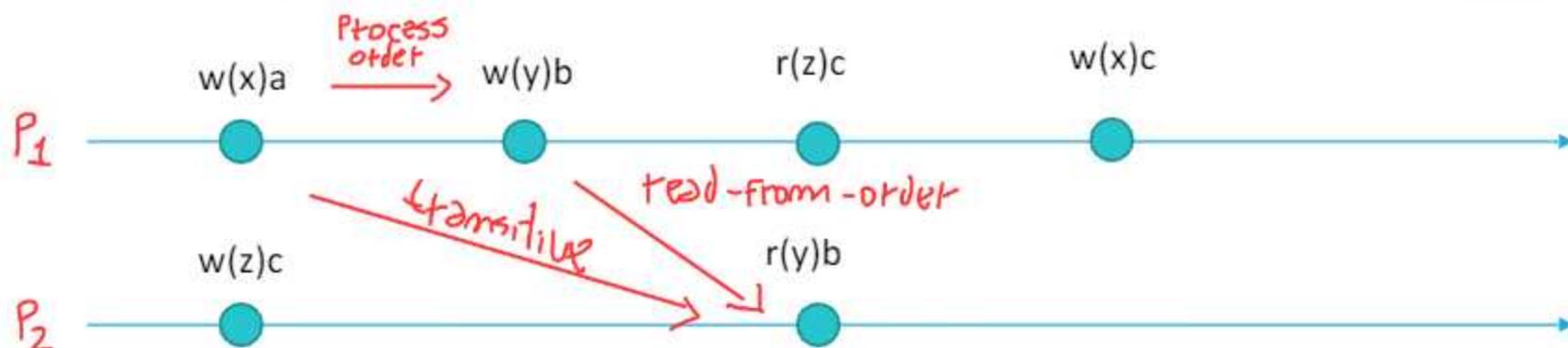
- Given the set of operations  $h_i$  issued by  $p_i$ , the local history  $\hat{h}_i$  is the total order  $(h_i, \rightarrow_i)$   
*↳ given that the processes are sequential*

# Execution history

An execution history  $\hat{H}$  of a shared memory system is a partial order  $(H, \rightarrow_H)$  such that

- $H = \bigcup_i h_i$  ↳ all operations executed by any process
  - $op_1 \rightarrow_H op_2$  if:
    - $op_1$  and  $op_2$  are in **process-order** relation (i.e., there exists a  $p_i$  such that  $op_1 \rightarrow_i op_2$ ) OR
    - $op_1$  and  $op_2$  are in **read-from-order** relation (i.e.,  $op_1 = w_i(x)v$  and  $op_2 = r_j(x)v$ ) OR
    - there exists  $op_3$  such that  $op_1 \rightarrow_H op_3$  and  $op_3 \rightarrow_H op_2$
- ↳ happened before relationship
- ↳ transitive relationship
- ↳ executed by same process  
↑ satisfy local order

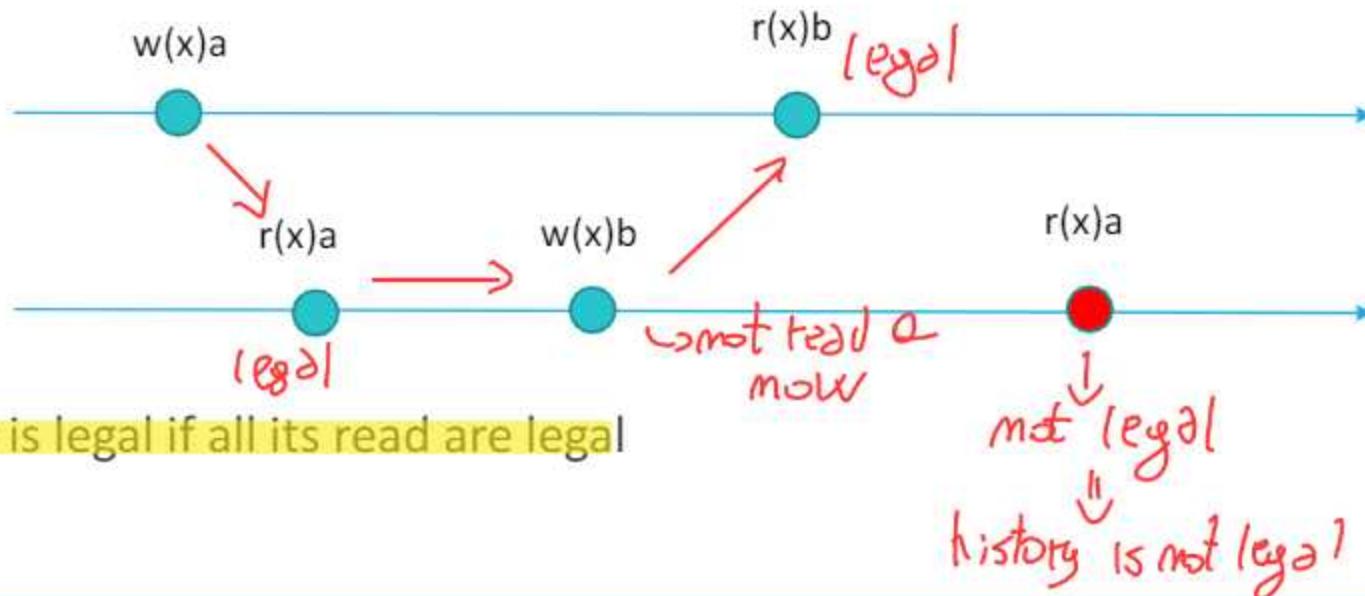
If neither  $op_1 \rightarrow_H op_2$  nor  $op_2 \rightarrow_H op_1$  then  $op_1$  and  $op_2$  are said to be **concurrent**



# Legality

A read operation  $r(x)v$  is **legal** if

1. there exists a write  $w(x)v$  preceding it in the history (i.e.,  $\exists w(x)v : w(x)x \rightarrow_H r(x)v$ ) AND
2. there not exists any other operation in between that write/read a different value  $u$  (i.e.,  $\nexists op(x)u : (u \neq v) \wedge w(x)v \rightarrow_H op(x)u \rightarrow_H r(x)v$ )



A **history** is legal if all its read are legal

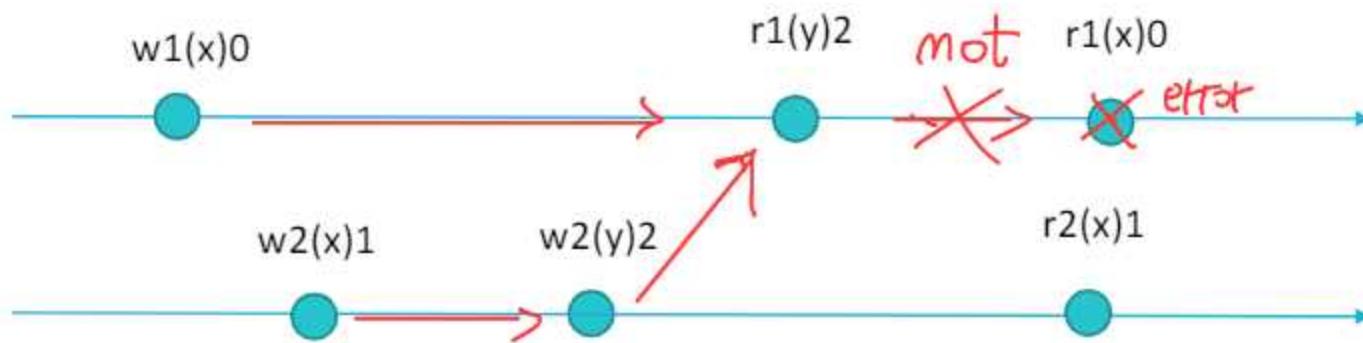
# Sequential Consistency

A history  $\hat{H} = (H, \rightarrow_H)$  is sequentially consistent if it admits a linear extension in which all the reads are legal.

A linear extension  $\hat{S} = (S, \rightarrow_S)$  of a partial order  $\hat{H} = (H, \rightarrow_H)$  is a topological sort of this partial order (i.e.,  $\hat{S}$  creates a total order by maintaining the order of all ordered pairs in  $\hat{H}$ )

↳ don't need preserve real time of operation, like in linearizability

# Example

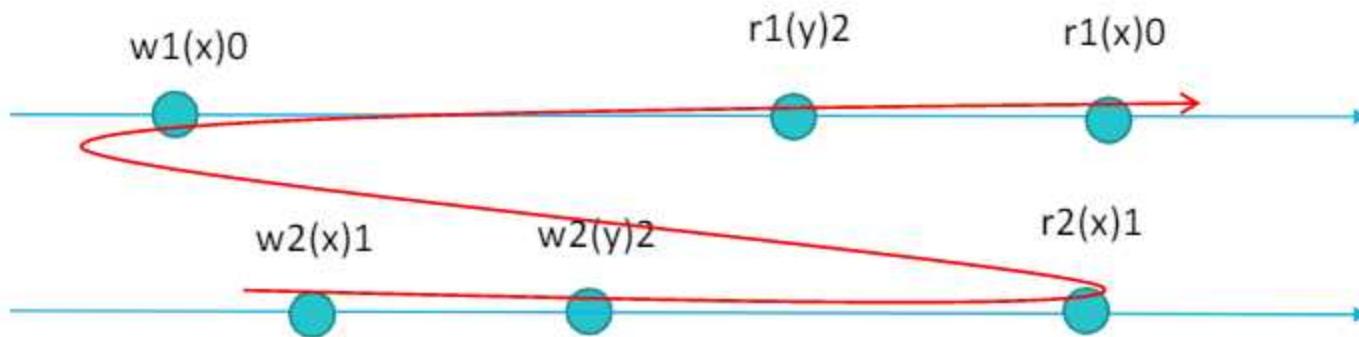


**The execution is not linearizable!**

$$\hat{S} = w1(x)0, w2(x)1, w2(y)2, r1(y)2, r2(x)1, \cancel{r1(x)0}$$

# Example

---



**However, it is sequential consistent**

$$\hat{S} = w_2(x)1, w_2(y)2, r_2(x)1, w_1(x)0, r_1(y)2, r_1(x)0$$

# Linearizability vs Sequential Consistency

---

## RECALL

- Linearizability requires that the linear extension  $\hat{S}$  also respects the real time of invocation
  - Sequential consistency removes the real time aspect and just focus on logical time
- 

Linearizability  $\Rightarrow$  Sequential Consistency

Sequential Consistency  $\not\Rightarrow$  Linearizability

# Causal Consistency

---

Let  $\widehat{H} = (H, \rightarrow_H)$  be an execution history and let  $\widehat{H}_i$  be the sub-history of  $\widehat{H}$  from which all the read operations not issued by  $p_i$  are removed.

*skip all write and my read operation*

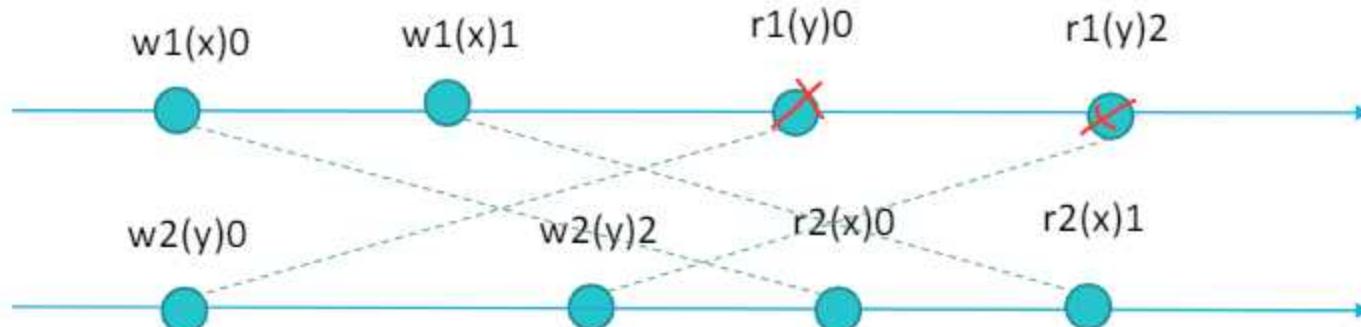
A history  $\widehat{H} = (H, \rightarrow_H)$  is *causally consistent* if, for each process  $p_i$ , all the read operations of  $\widehat{H}_i$  are legal.

## OBSERVATION

- in a causally consistent history, all processes see the same partial order of operations but each process may see a different linear extension.

# Example

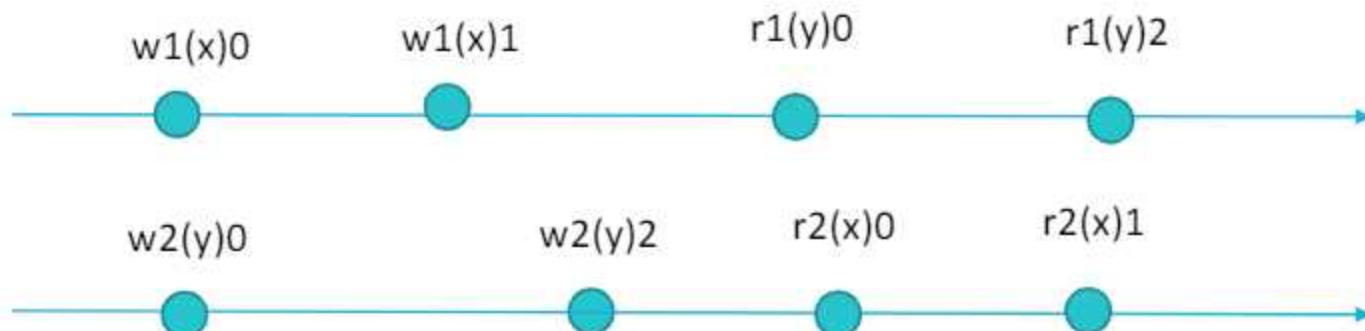
*depending on crossing each process*



**The execution is not sequential consistent!**

$\hat{S} = w1(x)0, w2(y)0, w2(y)2, r2(x)0, w1(x)1, r2(x)1, r1(y)0, r1(y)2$

# Example



**However, it is causal  
consistent**

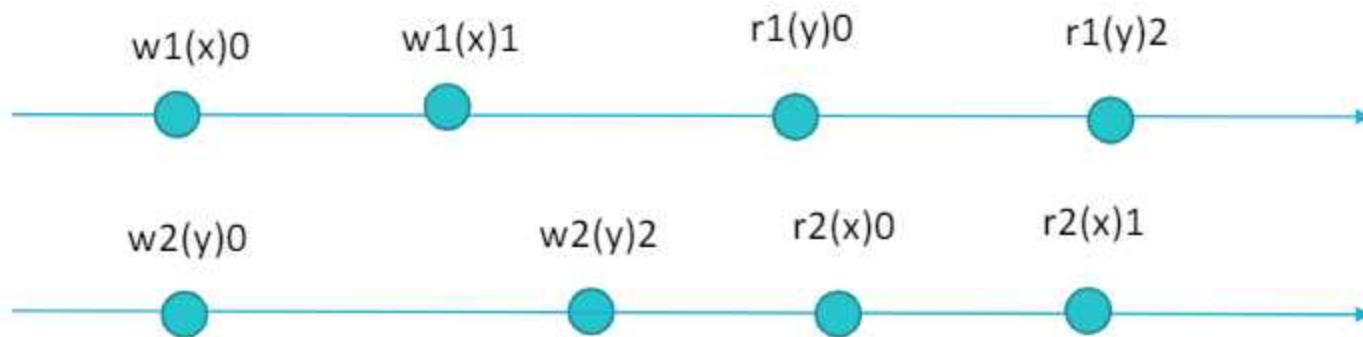
$$\hat{S}_1 = w1(x)0, w1(x)1, w2(y)0, r1(y)0, w2(y)2, r1(y)2$$

*remove head of  $P_2$*

$$\hat{S}_2 = w2(y)0, r1(y)0, w1(x)0, r2(x)0, w1(x)1, r2(x)1$$

*$\sqsubset$  of  $P_1$*

# Example



Linear ~~⇒~~ Sequential ~~⇒~~ causal

Sequential Consistency  $\Rightarrow$  Causal Consistency  
Causal Consistency  $\not\Rightarrow$  Sequential Consistency

# References

---

Michel Raynal and André Schiper: “A suite of formal definitions for consistency criteria in distributed shared memories”

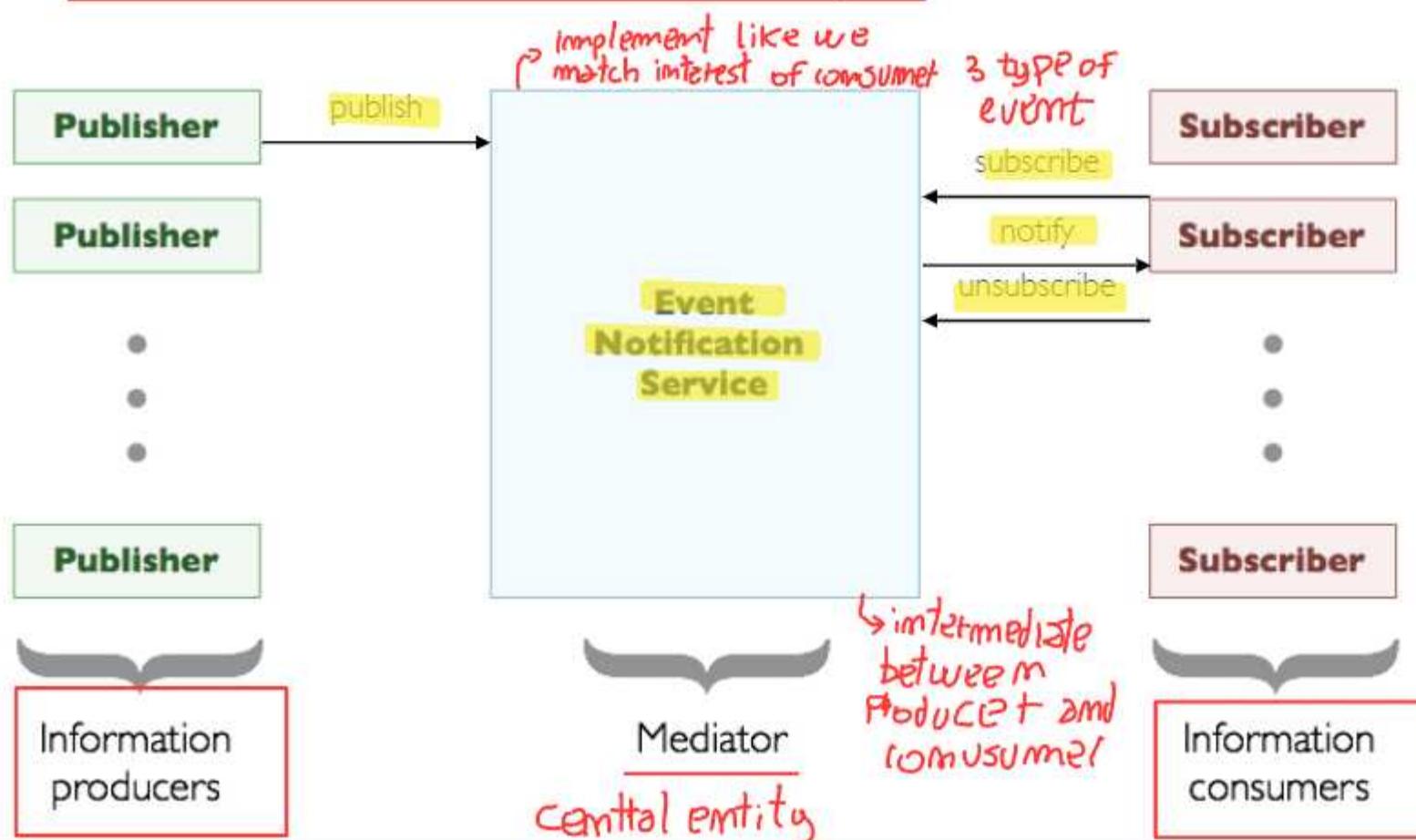
available at:

<https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.6880&rep=rep1&type=pdf>

# Information Dissemination in large scale systems: Publish/Subscribe

- The publish/subscribe communication paradigm:

- Publishers:** produce data in the form of **events**. *(or operations)*
- Subscribers:** declare interests on published data with **subscriptions**.
- Each subscription** is a filter on the set of published events.
- An **Event Notification Service (ENS)** notifies to each subscriber every published event that matches at least one of its subscriptions.



- Publish/subscribe was thought as a comprehensive solution for those problems:
  - **Many-to-many communication model** - Interactions take place in an environment where various information producers and consumers can communicate, all at the same time. Each piece of information can be delivered at the same time to various consumers. Each consumer receives information from various producers. *~ every entity or system can be producer and consumer*
  - **Space decoupling** - Interacting parties do not need to know each other. Message addressing is based on their content. *↳ producer P2P communication with ECS, consumer also P2P with ECS*
  - **Time decoupling** - Interacting parties do not need to be actively participating in the interaction at the same time. Information delivery is mediated through a third party.
  - **Synchronization decoupling** - Information flow from producers to consumers is also mediated, thus synchronization among interacting parties is not needed.
  - **Push/Pull interactions** - both methods are allowed. *~ publisher push to ECS and it push to consumer of ECS or ECS pull from publisher and consumer pull from ECS*
- These characteristics make **pub/sub** perfectly suited for distributed applications relying on **document-centric communication**.

Producet produce an event when information is ready

- Events represent information structured following an event schema.
- The event schema is fixed, defined a-priori, and known to all the participants.
- It defines a set of fields or attributes, each constituted by a name and a type. The types allowed depend on the specific implementation, but basic types (like integers, floats, booleans, strings) are usually available.
- Given an event schema, an event is a collection of values, one for each attribute defined in the schema.

■ Example: suppose we are dealing with an application whose purpose is to distribute updates about computer-related blogs.

name	type	allowed values
blog_name	string	ANY
address	URL	ANY
genre	enumeration	[hardware, software, peripherals, development]
author	string	ANY
abstract	string	ANY
rating	integer	[1-5]
update_date	date	>1-1-1970 00:00

Event



name	value
blog_name	Prad.de
address	<a href="http://www.prad.de/en/index.html">http://www.prad.de/en/index.html</a>
genre	peripherals
author	Mark Hansen
abstract	"The review of the new TFT panel..."
rating	4
update_date	26-4-2006 17:58

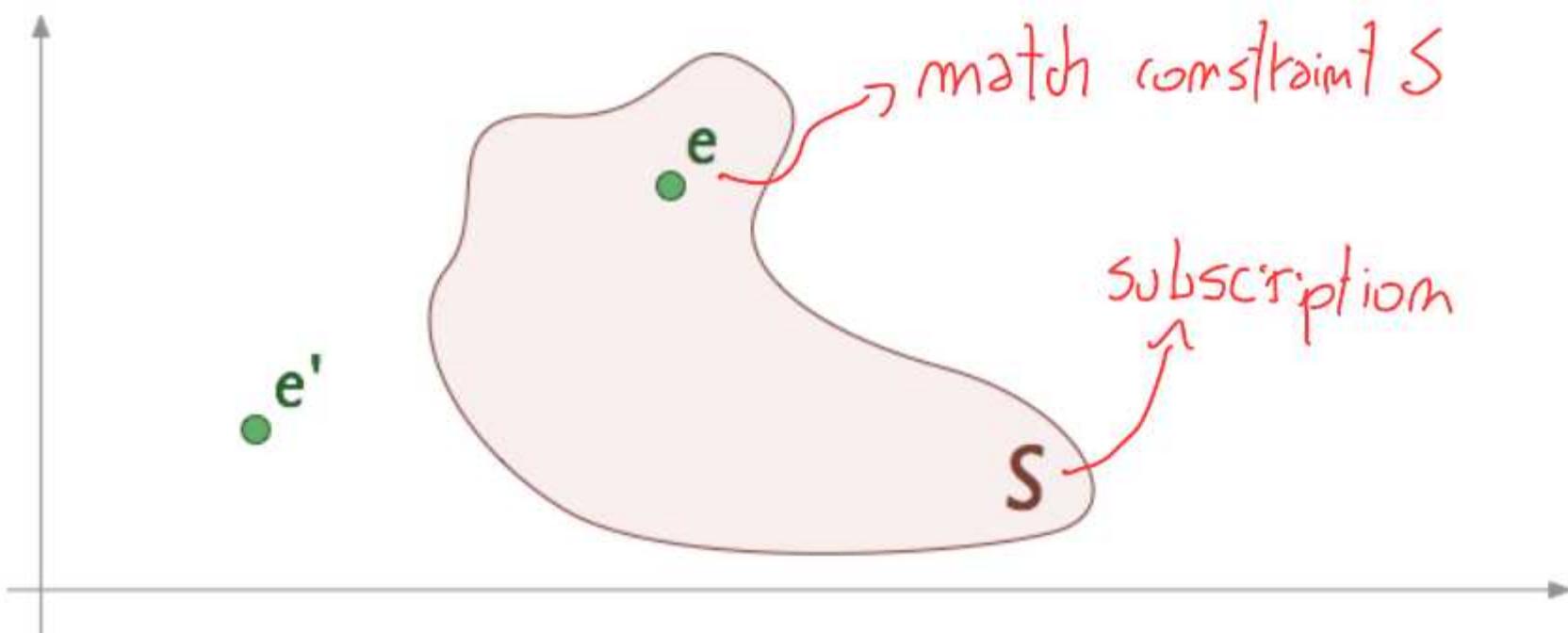
→ define structure of element values

Event Schema

the message with a value coherent with schema

- Subscribers express their interests in specific events issuing subscriptions.
- A subscription is a constraint expressed on the event schema.
- The Event Notification Service will notify an event  $e$  to a subscriber  $x$  only if the values that define the event satisfy the constraint defined by one of the subscriptions  $s$  issued by  $x$ . In this case we say that  **$e$  matches  $s$** .
- Subscriptions can take various forms, depending on the subscription language and model employed by each specific implementation.
- Example: a subscription can be a conjunction of constraints each expressed on a single attribute. Each constraint in this case can be as simple as a  $>=$  operator applied on an integer attribute, or complex as a regular expression applied to a string.

- From an abstract point of view the **event schema** defines an n-dimensional event space (where n is the number of attributes).
- In this space each event e represents a point.
- Each subscription s identifies a subspace.
- An event e matches the subscription s if, and only if, the corresponding point is included in the portion of the event space delimited by s.

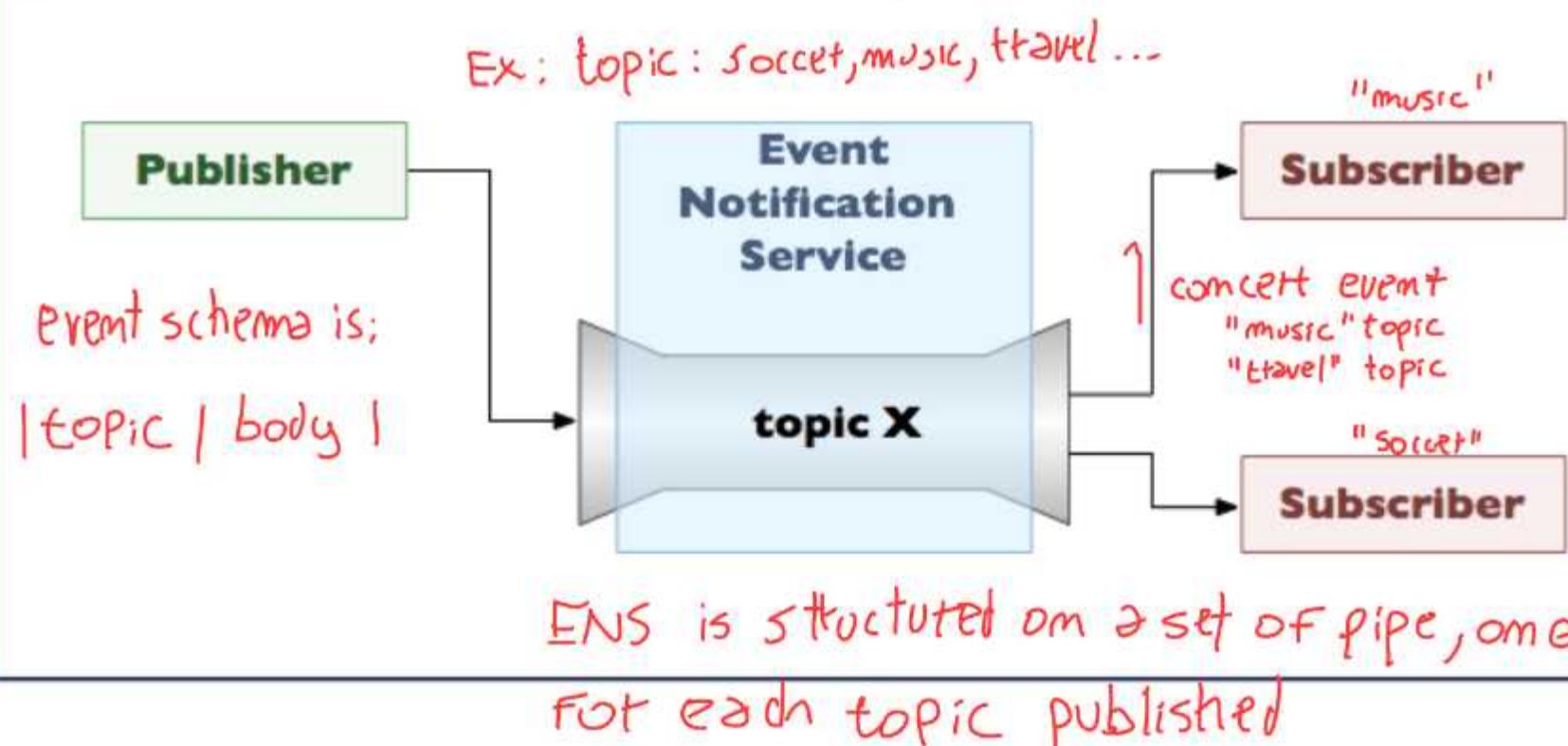


- Depending on the subscription model used we distinguish various flavors of publish/subscribe:

- Topic-based
- Hierarchy-based
- Content-based
- Type-based
- Concept-based
- XML-based
- .....

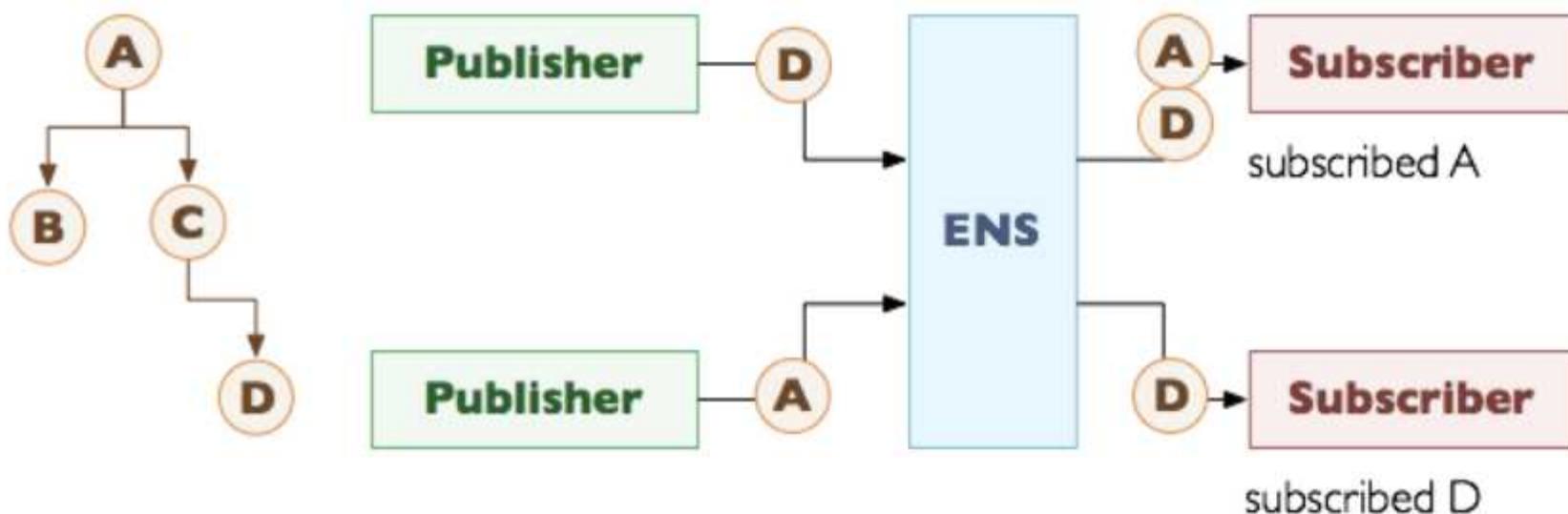
→ most spread paradigm

- **Topic-based selection:** data published in the system is mostly unstructured, but each event is "tagged" with the identifier of a **topic** it is published in. Subscribers issue subscriptions containing the topics they are interested in.
- A **topic** can be thus represented as a "virtual channel" connecting producers to consumers. For this reason, the **problem of data distribution in topic-based publish/subscribe systems** is considered quite close to group communications.

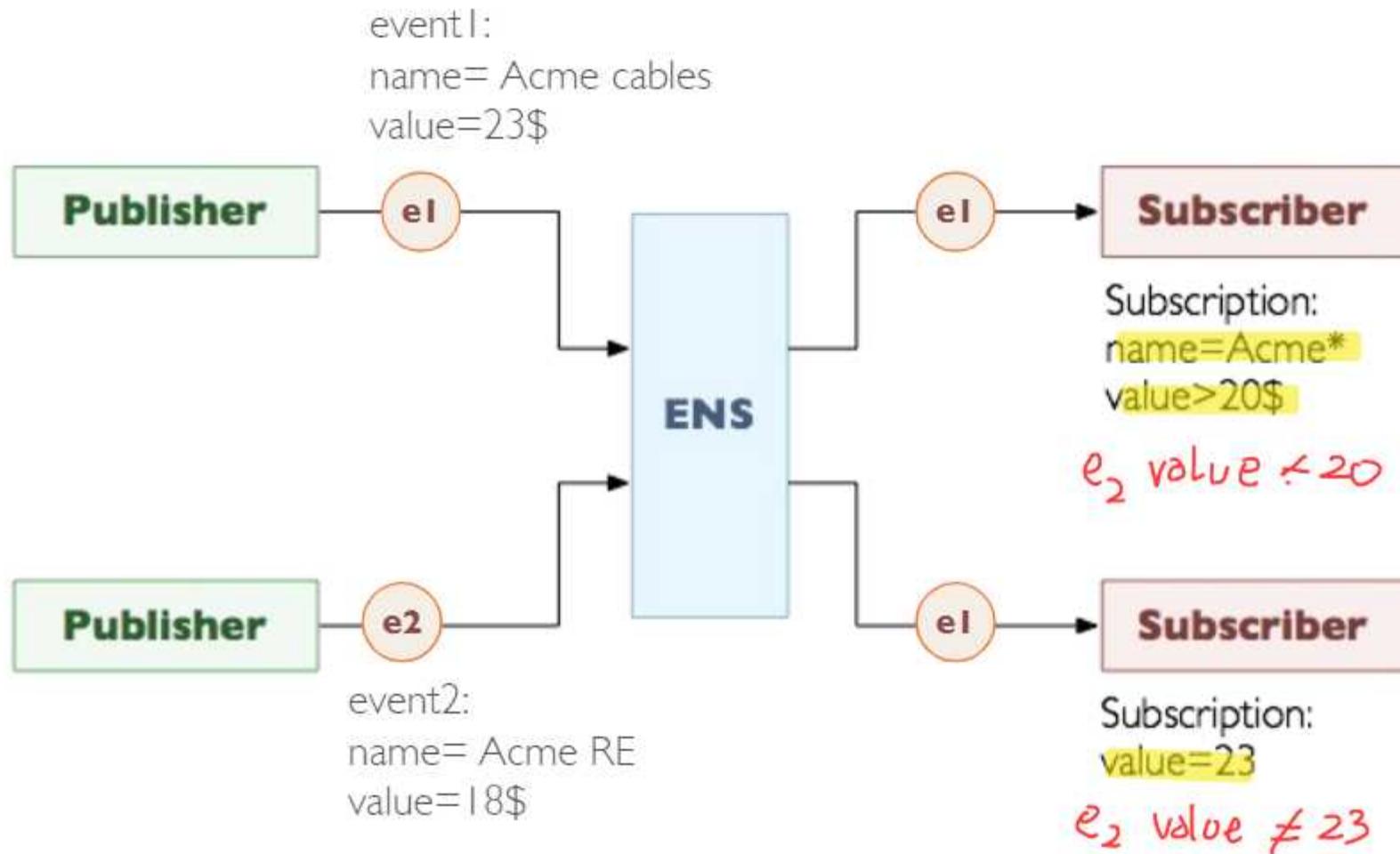


■ **Hierarchy-based selection:** even in this case each event is “tagged” with the topic it is published in, and Subscribers issue subscriptions containing the topics they are interested in.

■ Contrarily to the previous model, here topics are organized in a hierarchical structure which express a notion of containment between topics. When a subscriber subscribe a topic, it will receive all the events published in that topic and in all the topics present in the corresponding sub-tree.



■ **Content-based selection:** all the data published in the system is mostly structured. Each subscription can be expressed as a conjunction of constraints expressed on attributes. The Event Notification Service filters out useless events before notifying a subscriber.



## ■ Where Publish subscribe is used

- Transport component of EDA architectures
- Data Distribution Services
- Transport component of context aware distributed applications

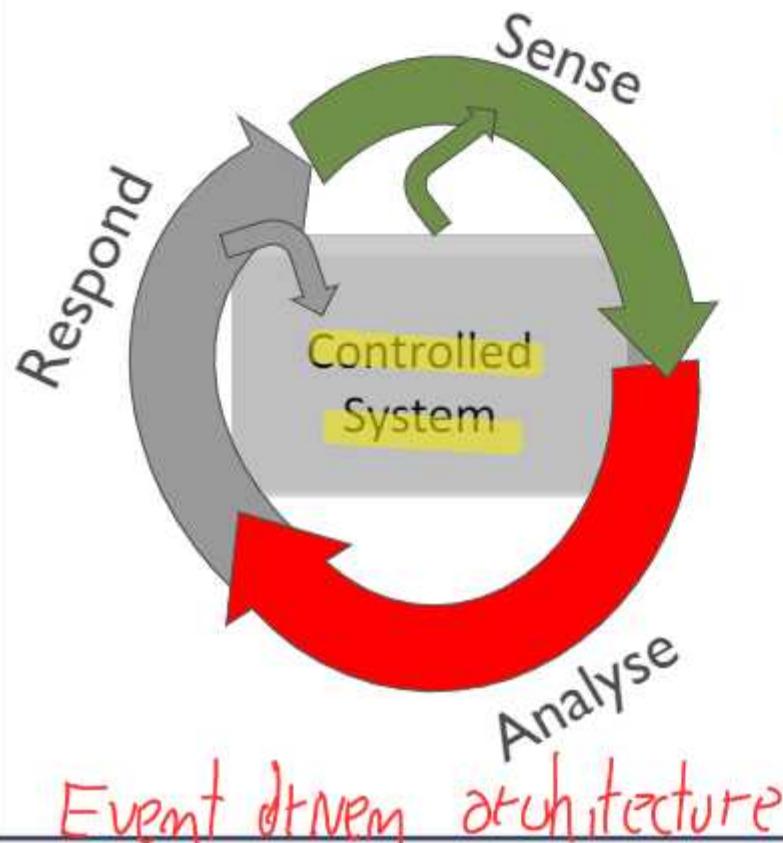
Event driven architecture  
do something when something  
event, all algorithm seem.

advertisement services, blogs,....

## ■ **EDA characteristics:** three principal building blocks

- **Sense:** The sensing block gathers data from within and outside the controlled system
- **Analyse:** Data are analyzed
- **Respond:** Analysis used to determine whether appropriate actions are to be timely undertaken in response to what has been sensed (respond block)

SMART HOME : Get value with sensor from system, with actuator respond to measurement



### ■ A **control loop** is enabled

- The sensing part obtains data that defines the "real" state
- The analysis part correlates data in order to determine the current state
- When reality deviates from that expected then the respond part acts on the system (e.g, sends alerts)

## ■ EDA Architecture

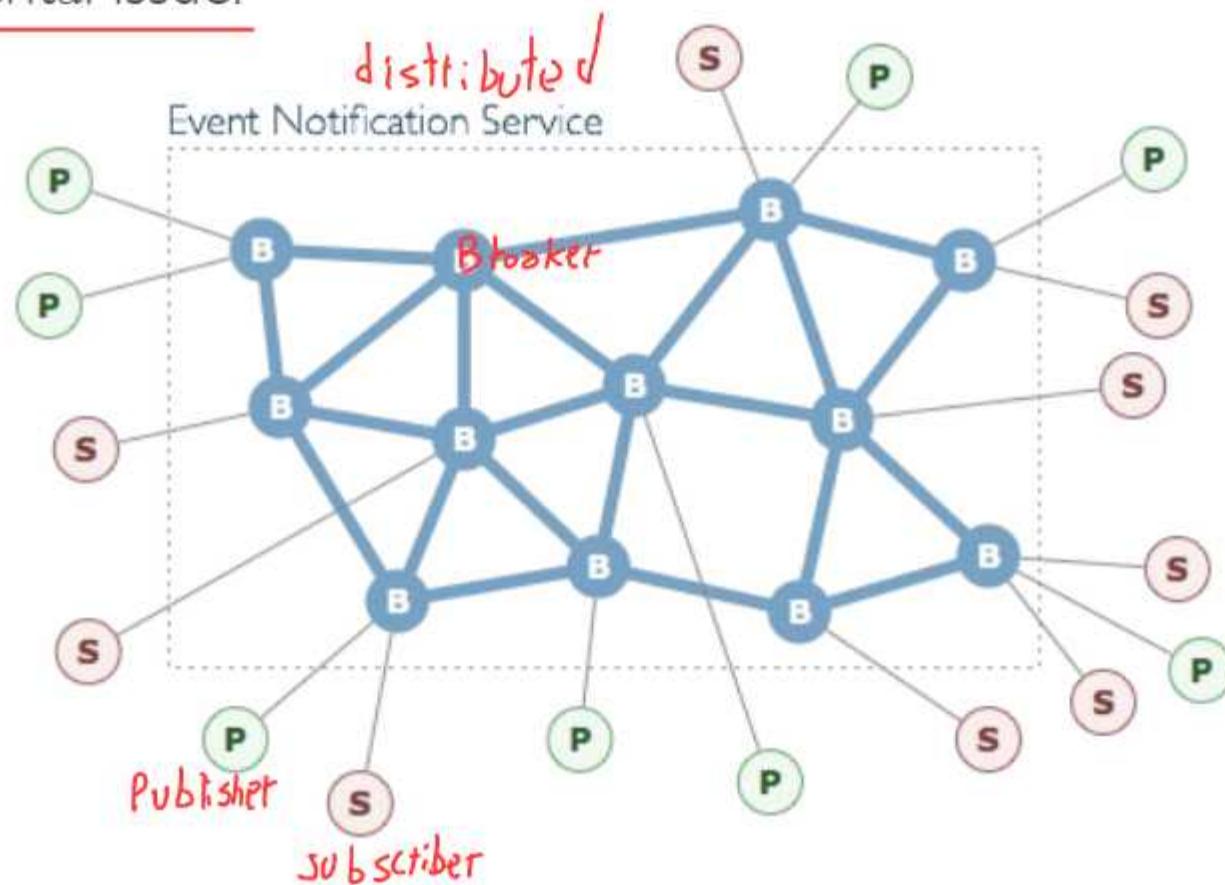
- Event Correlation and Processing
- Event Distribution *→ tight measurement to tight component*
- Publish subscribe help in doing some pre-processing of the events as well as reducing the network load

Complex Event Processing

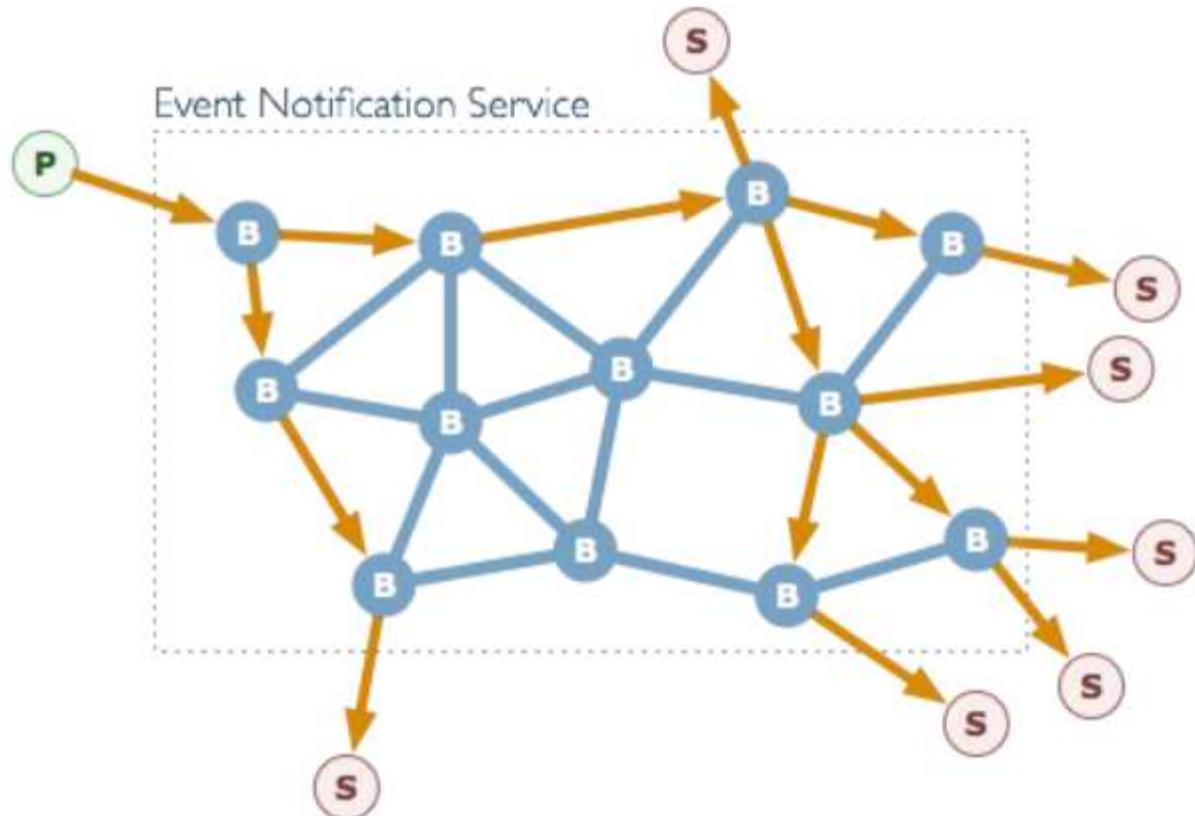
→ acquire data from multiple sources and combine them for respond correctly

Event Dissemination  
(Publish-Subscribe)

- The Event Notification Service is usually implemented as a:
  - **Centralized service**: the ENS is implemented on a single server. *system small and low load*
  - **Distributed service**: the ENS is constituted by a set of nodes, event brokers, which cooperate to implement the service.
- The latter is usually preferred for large settings where scalability is a fundamental issue.



- Modern ENSs are implemented through a set of processes, called *event brokers*, forming an overlay network.
- Each *client* (publisher or subscriber) accesses the service through a *broker* that masks the system complexity.



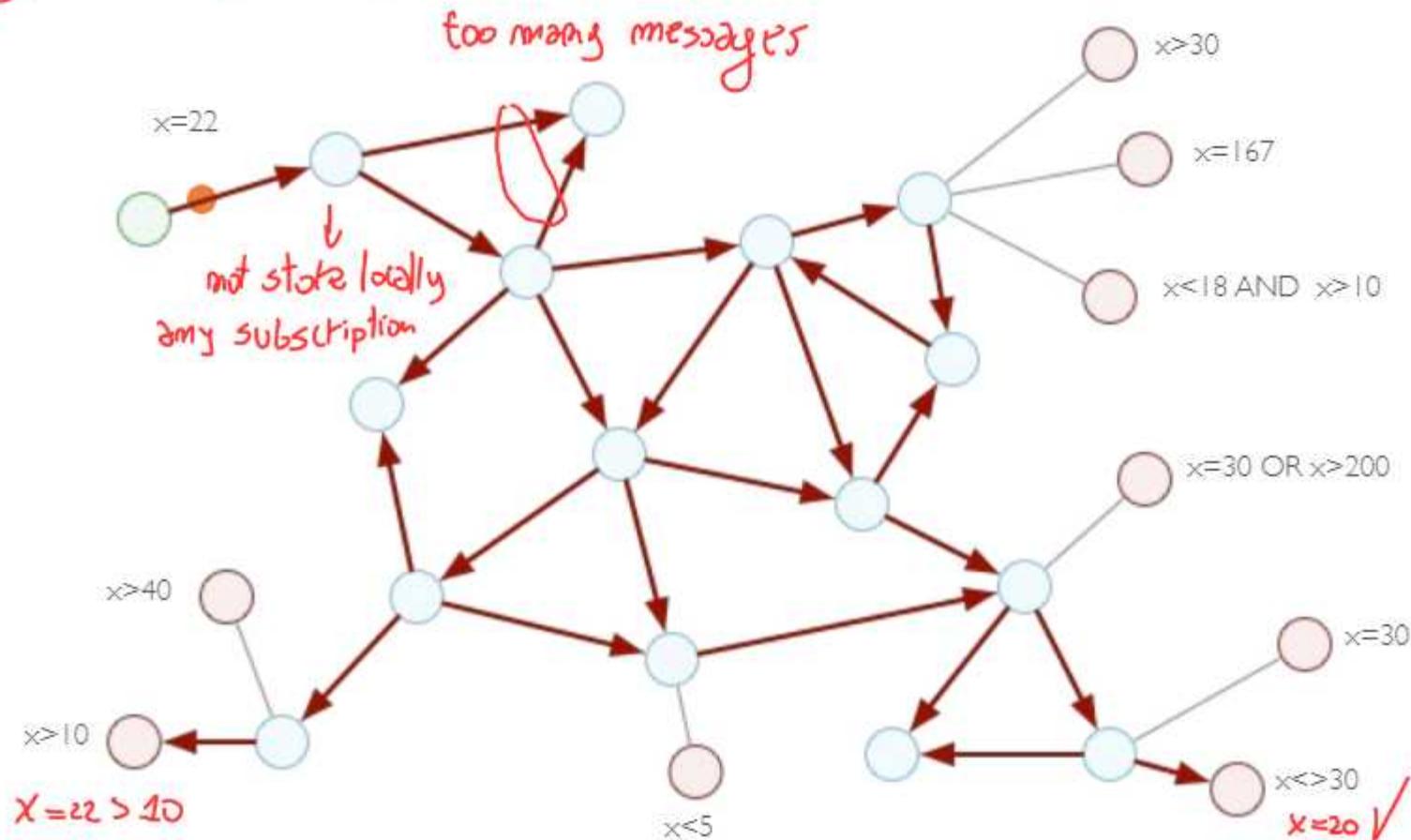
- An *event routing mechanism* routes each event inside the ENS from the broker where it is published to the broker(s) where it must be notified.

■ **Event flooding:** each event is broadcast from the publisher in the whole system.

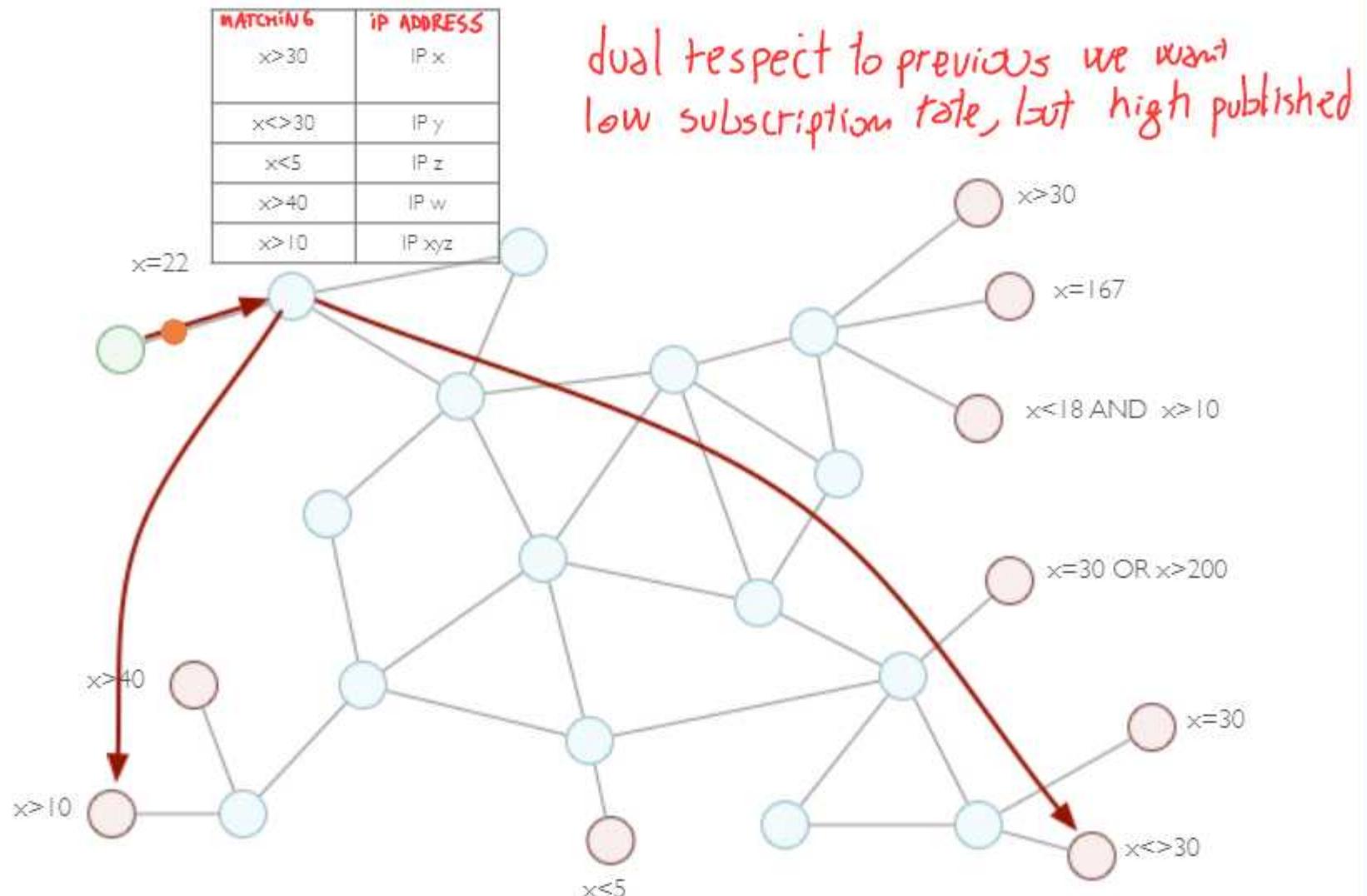
■ The implementation is straightforward but very expensive.

■ This solution has the highest message overhead with no memory overhead.

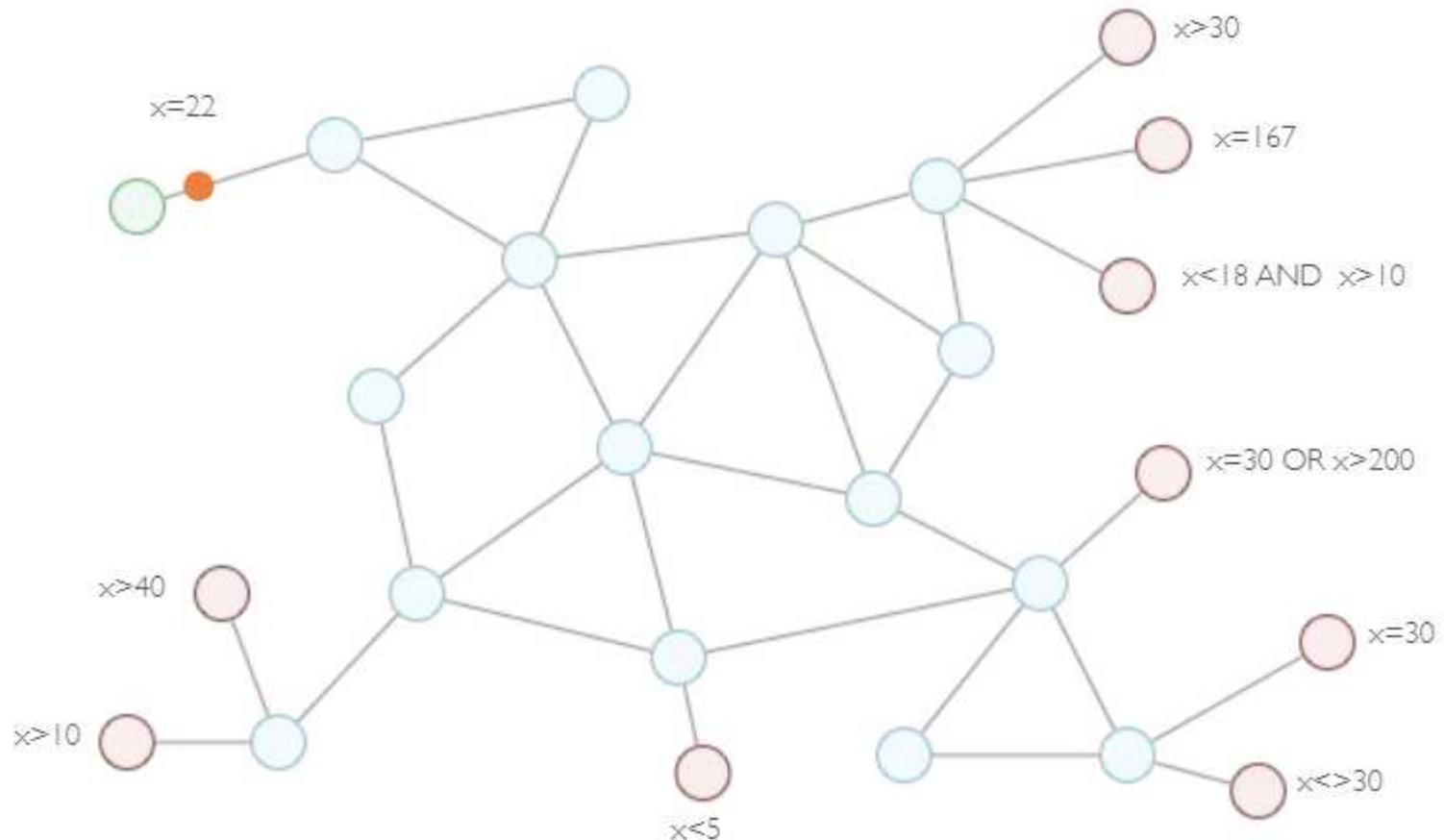
is good in system quite static from genetatic contam, but frequent in interest



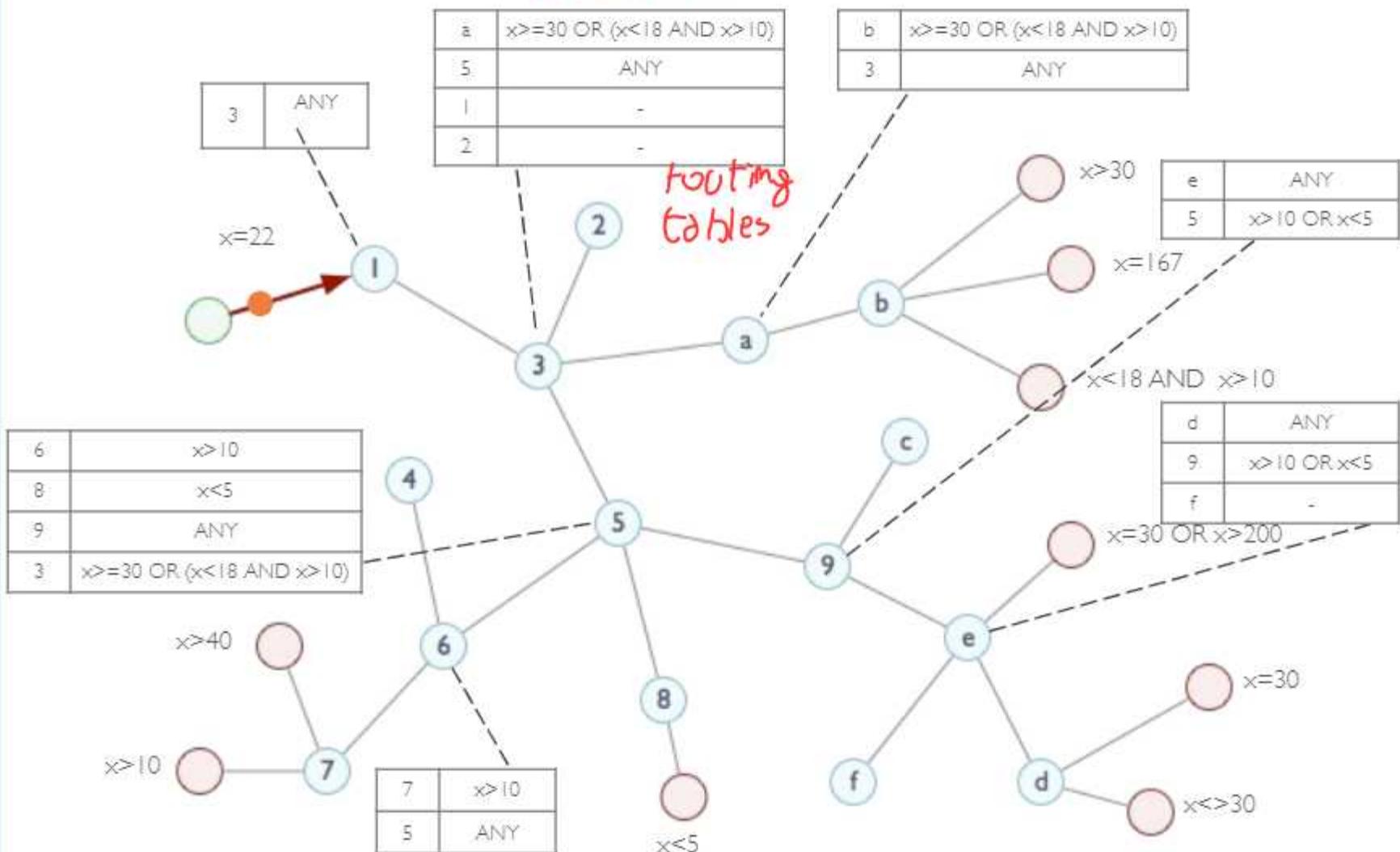
■ **Subscription flooding:** each subscription is copied on every broker, in order to build locally complete subscription tables. These tables are then used to locally match events and directly notify interested subscribers. This approach suffers from a large memory overhead, but event diffusion is optimal. It is impractical in applications where subscriptions change frequently.



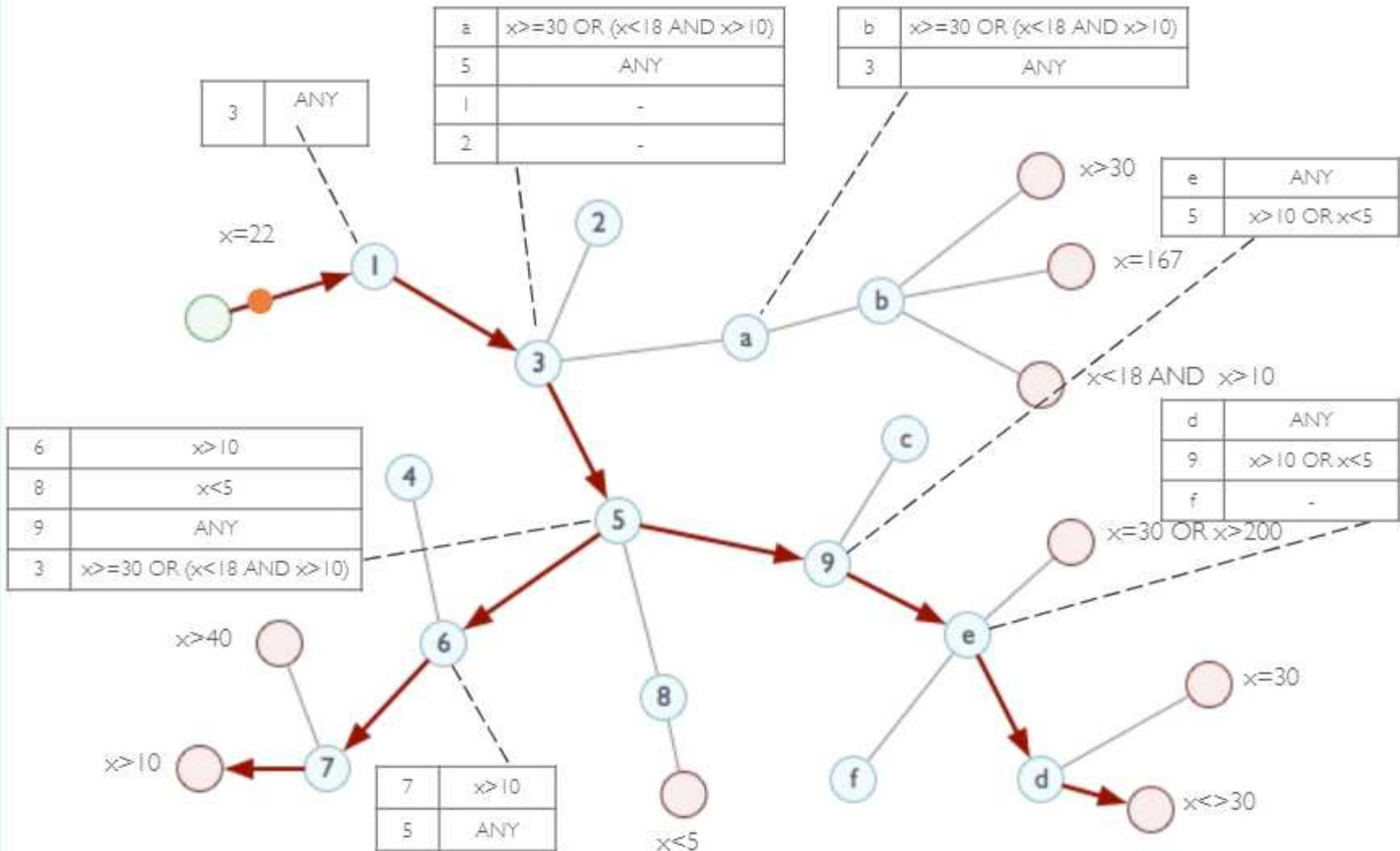
**Filter-based routing:** subscriptions are partially diffused in the system and used to build *routing tables*. These *tables*, are then exploited during event diffusion to dynamically build a multicast tree that (hopefully) connects the publisher to all, and only, the interested subscribers.



**Filter-based routing:** subscriptions are partially diffused in the system and used to build *routing tables*. These tables, are then exploited during event diffusion to dynamically build a multicast tree that (hopefully) connects the publisher to all, and only, the interested subscribers. *where to route based on condition*



**Filter-based routing:** subscriptions are partially diffused in the system and used to build *routing tables*. These tables, are then exploited during event diffusion to dynamically build a multicast tree that (hopefully) connects the publisher to all, and only, the interested subscribers.

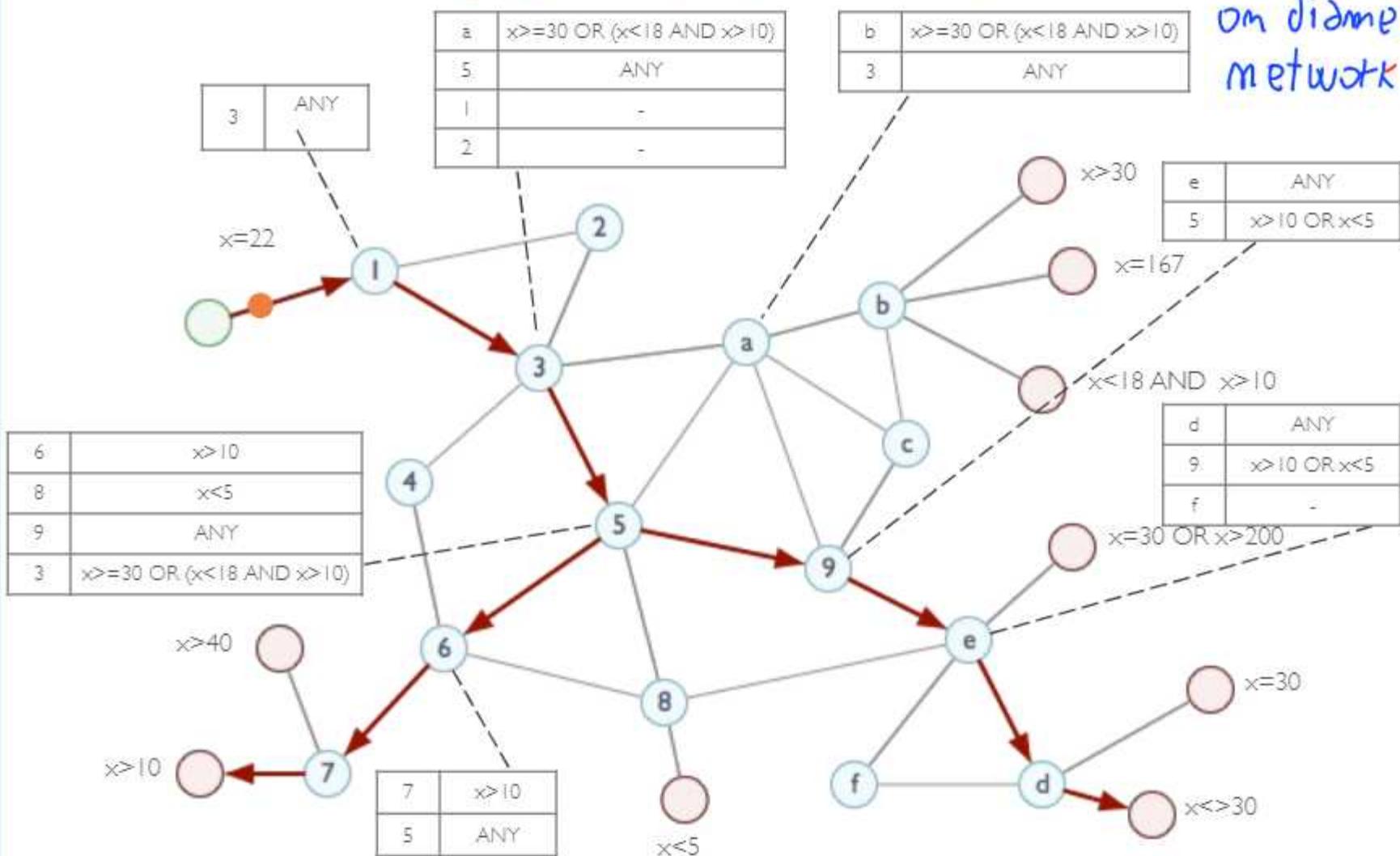


used to reduce number of hops, p and s must be near

good for case in middle of precedent

a good tradeoff with rate of publisher  
and subscription → problem with latency that depend

on diameter of the network



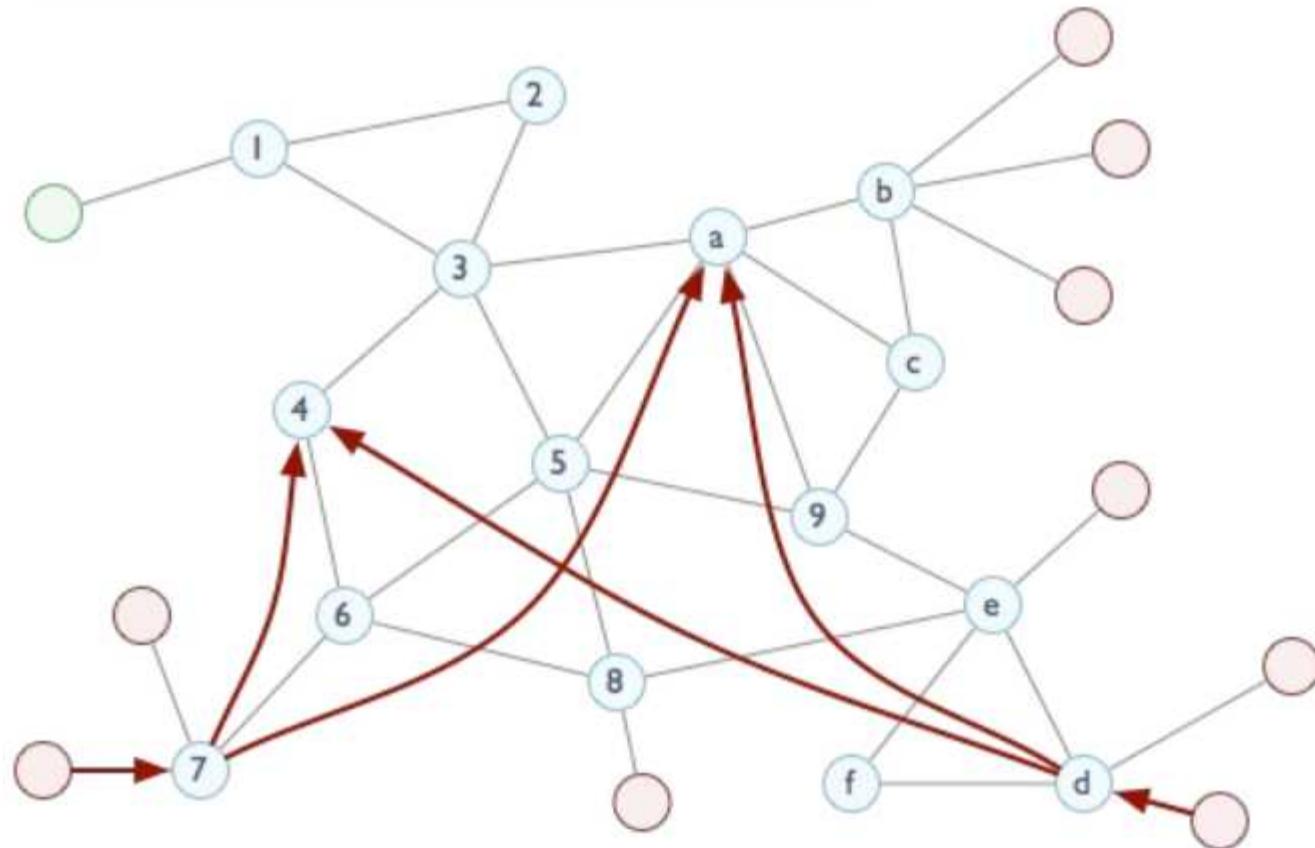
■ **Rendez-Vous routing:** it is based on two functions, namely  $SN$  and  $EN$ , used to associate respectively subscriptions and events to brokers in the system.

- Given a subscription  $s$ ,  $SN(s)$  returns a set of nodes which are responsible for storing  $s$  and forwarding received events matching  $s$  to all those subscribers that subscribed it.
- Given an event  $e$ ,  $EN(e)$  returns a set of nodes which must receive  $e$  to match it against the subscriptions they store.
- Event routing is a two-phases process: first an event  $e$  is sent to all brokers returned by  $EN(e)$ , then those brokers match it against the subscriptions they store and notify the corresponding subscribers.
- This approach works only if for each subscription  $s$  and event  $e$ , such that  $e$  matches  $s$ , the intersection between  $EN(e)$  and  $SN(s)$  is not empty (**mapping intersection rule**).

Using distributed hash table , a particular  
time

## ■ **Rendez-Vous routing**: example.

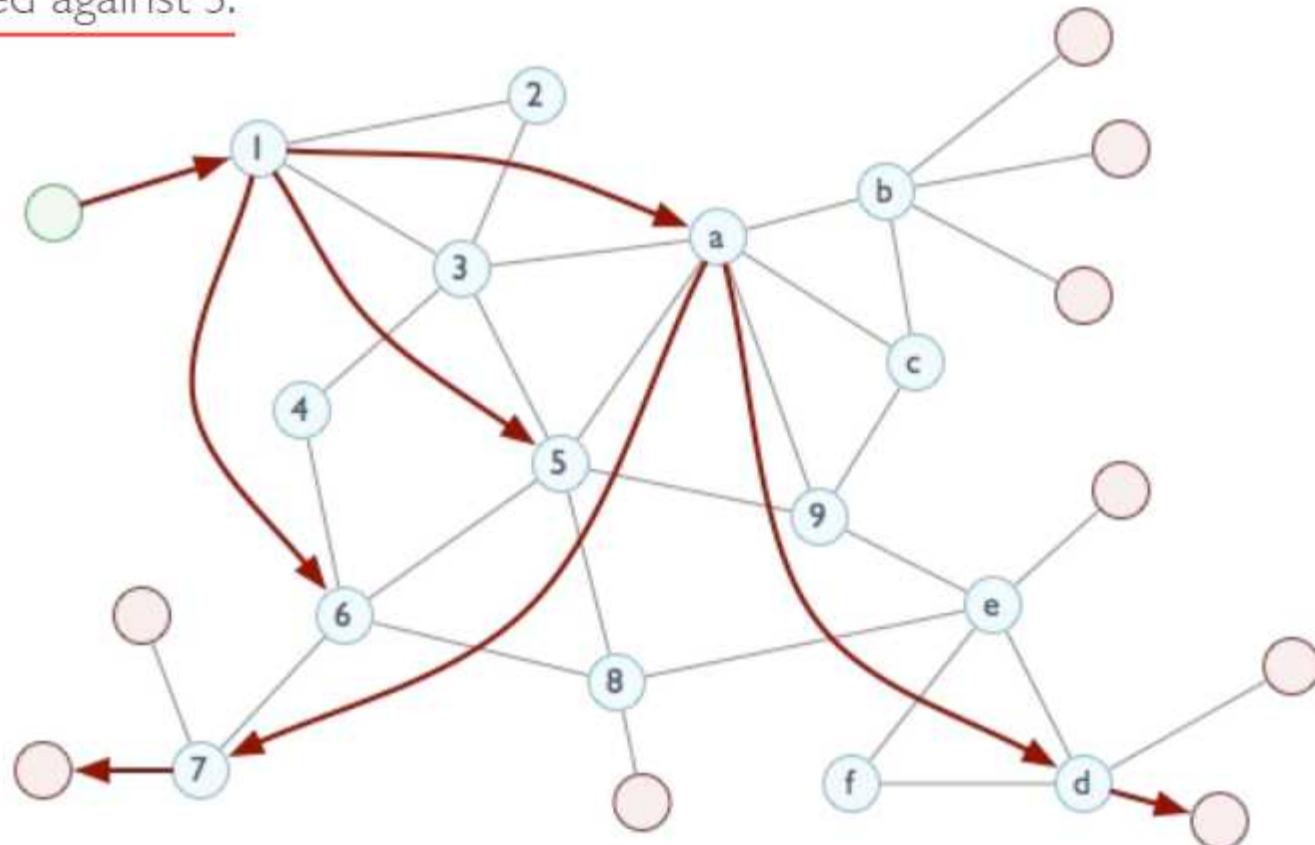
- **Phase I:** two nodes issue the same subscription  $S$ .



- $SN(S) = \{4, a\}$

## ■ **Rendez-Vous routing**: example.

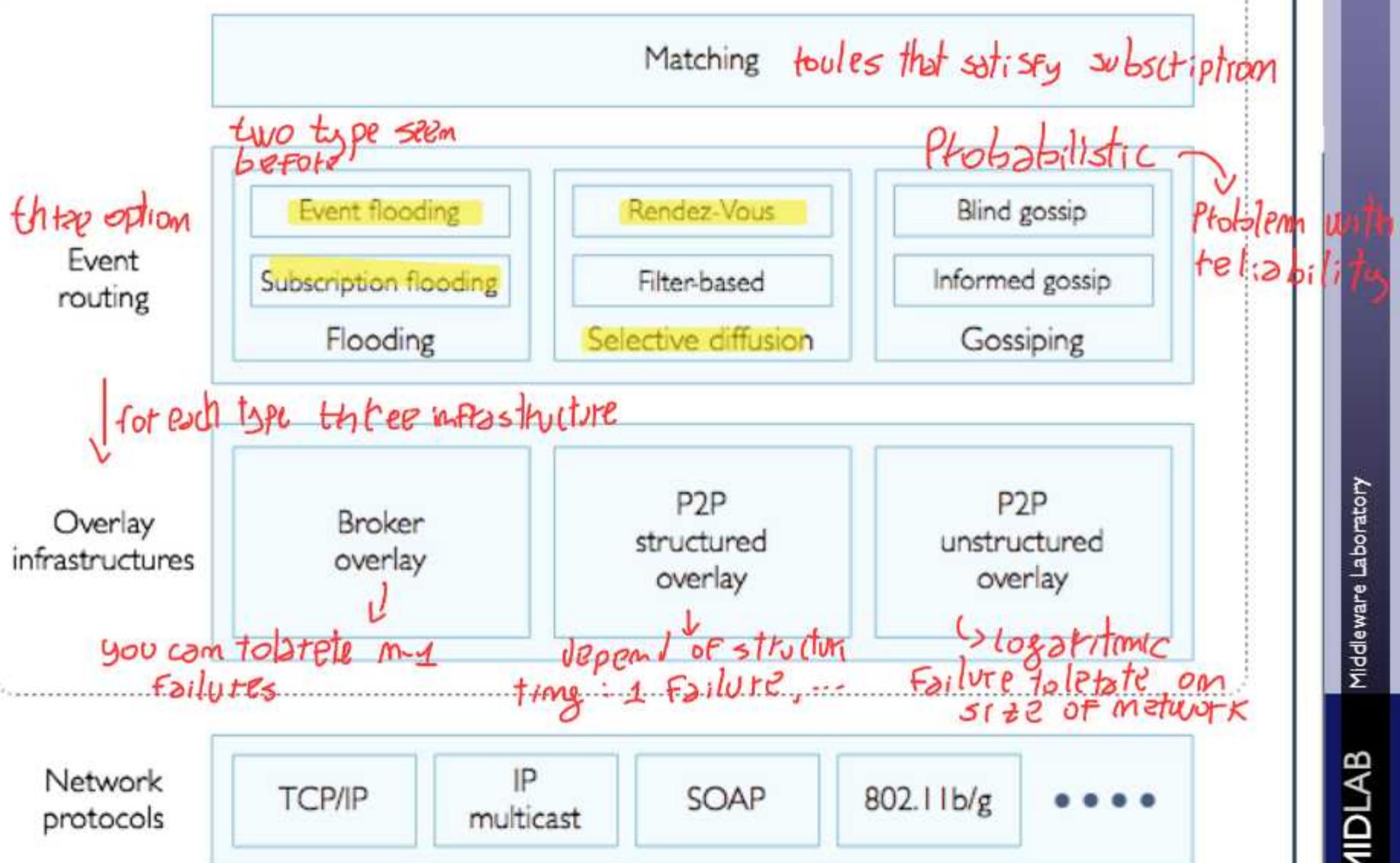
- **Phase II**: an event  $e$  matching  $S$  is routed toward the rendez-vous node where it is matched against  $S$ .



- $EN(e) = \{5, 6, a\}$
- Broker **a** is the rendez-vous point between event  $e$  and subscription  $S$ .

## ■ A generic architecture of a publish/subscribe system:

### Pub/Sub Architecture



From "Distributed Event Routing in Publish/Subscribe Communication Systems: a survey" R.Baldoni, L. Querzoni, S.Takoma, A.Virgillito

# Dependable Distributed Systems

## Master of Science in Engineering in Computer Science

AA 2022/2023

---

LECTURE 25: ~~OVERLAY NETWORKS~~

# Peer to Peer (P2P)

organize in a hierarchy way

P2P systems comprise **self-organized** equal and autonomous entities  
**(peers)** aiming to **share distributed resources** in a given network

Peers are **both client and servers**, each node carries out the same tasks

**Local knowledge:** nodes only know a small set of other nodes

**Decentralization:** nodes must self-organize in a decentralized way

**Robustness:** several nodes may fail with little or no impact

**High scalability:** high aggregate capacity, load distribution

**Low-cost:** storage and bandwidth are contributed by users

# Peer to Peer (P2P)

---

P2P systems attempt to provide a long (not exhaustive) **list of features**:

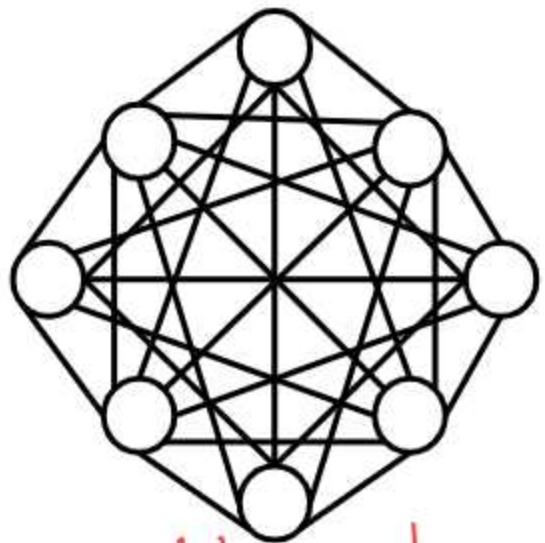
---

- Selection of nearby peers
- redundant storage
- efficient search/location of data items
- data permanence or guarantees
- hierarchical naming
- trust and authentication
- anonymity
- ...

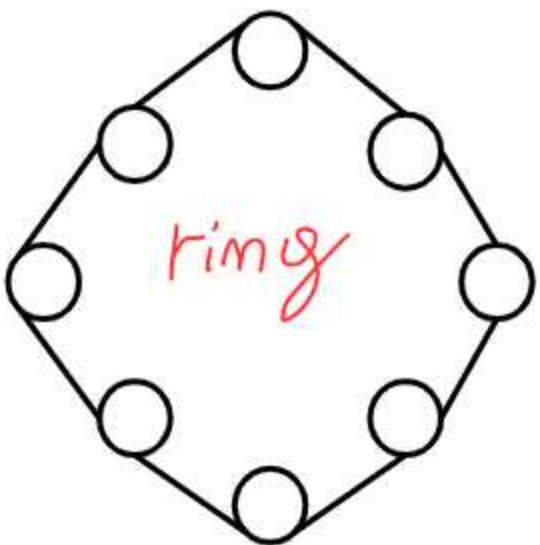
P2P networks potentially offer an **efficient routing architecture** that is **self-organizing, massively scalable, and robust** in the wide-area, **combining fault tolerance, load balancing, and explicit notion of locality**

# Topology

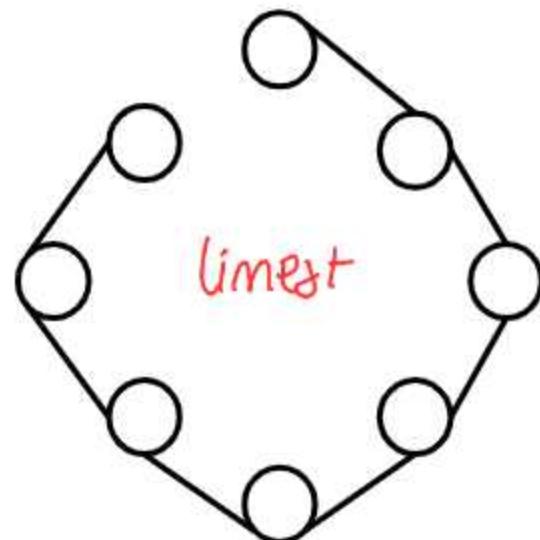
Way to abstract interconnection in distributed system.



complete graph



ring



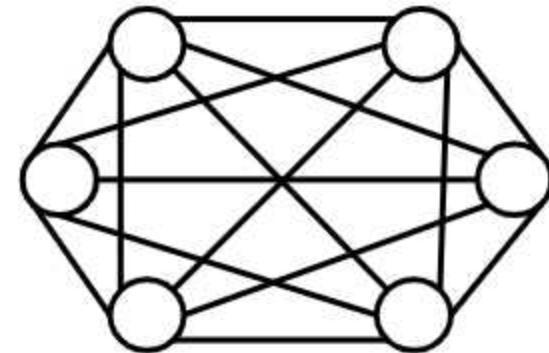
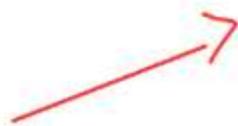
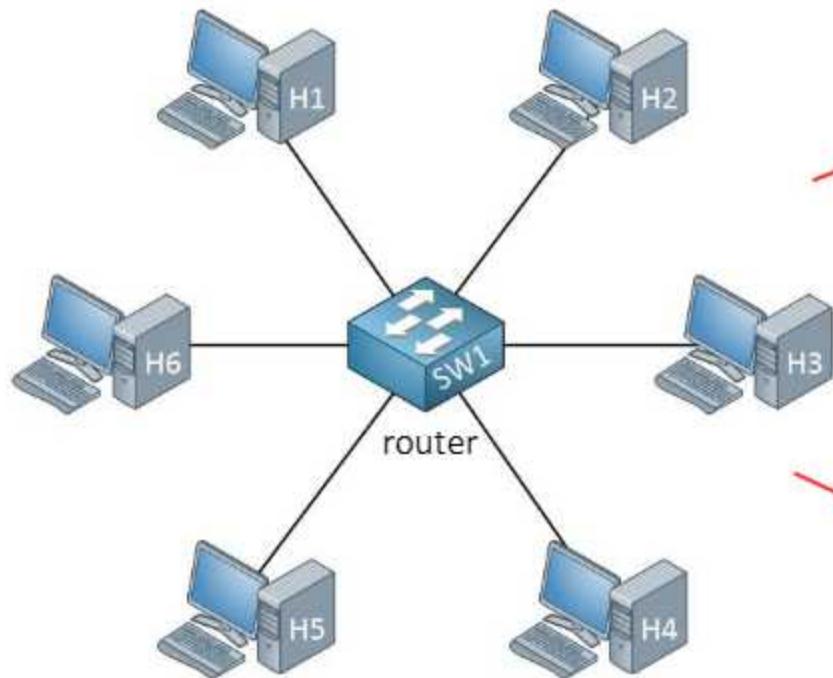
linear

Links: model the capability of a pair of processes in exchanging messages.

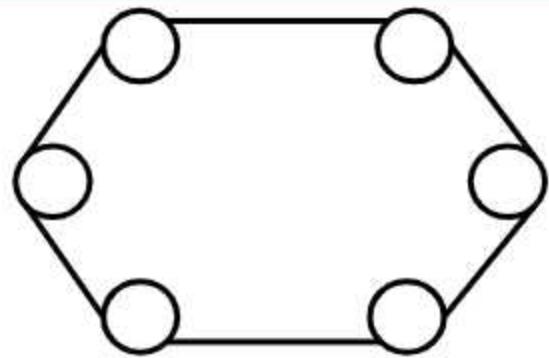
same set of processes but different topology of links

# Topology (behind)

is just an abstraction



If all machines can exchange messages among them

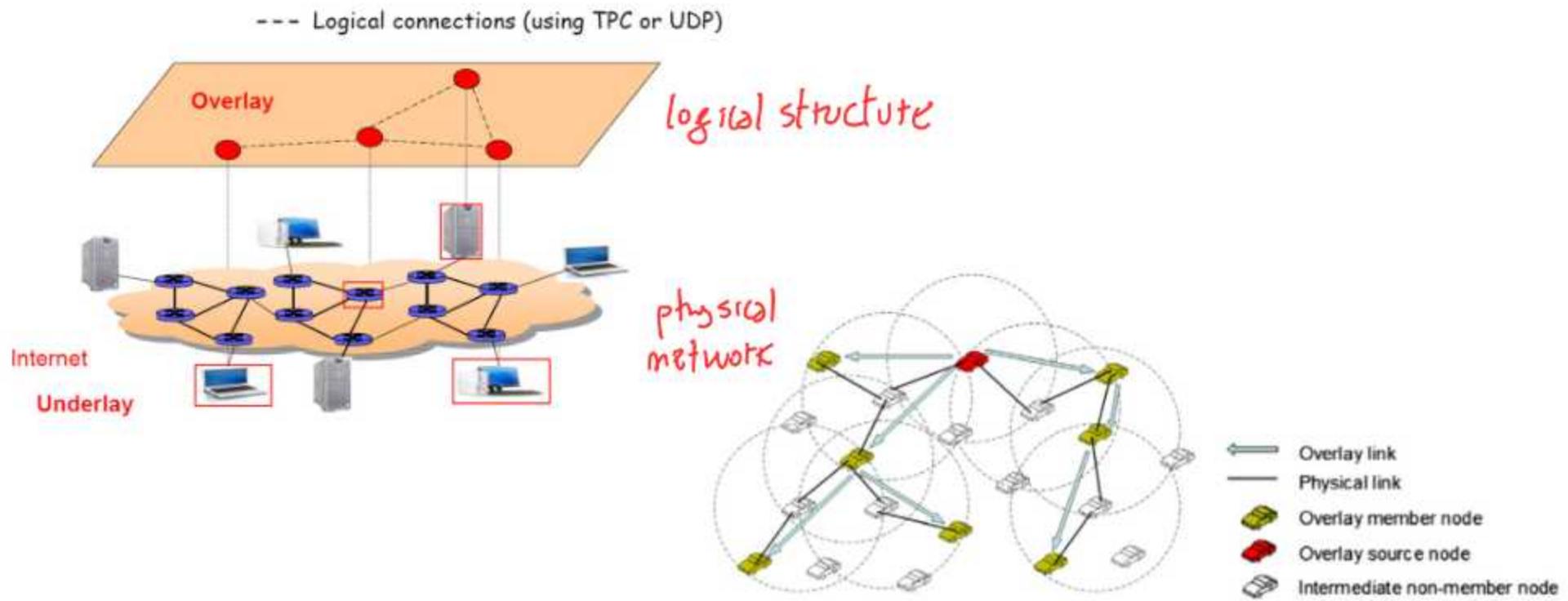


two possibility, depend on rules

If every machine  $i$  can exchange messages only with  $i \% 6 \pm 1$

# Overlay Network

A logical structure over a physical network



# Overlay Network

---

An overlay network is a network that is **built on top of an existing network**.

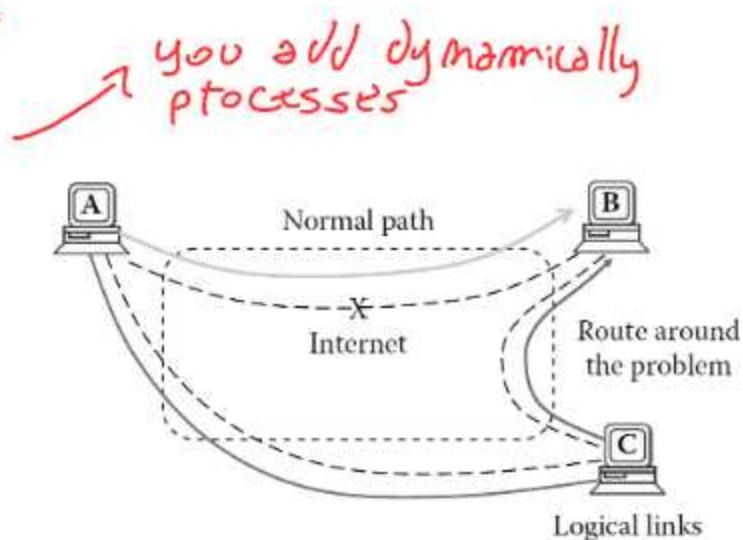
The overlay therefore **relies on the so-called underlay network for basic networking functions, namely routing and forwarding**

Today, most overlay networks are built in the application layer on top of the TCP/IP networking suite.

The nodes in an overlay network are connected via logical links that can span many physical links. A link between two overlay nodes may take several hops in the underlying network.

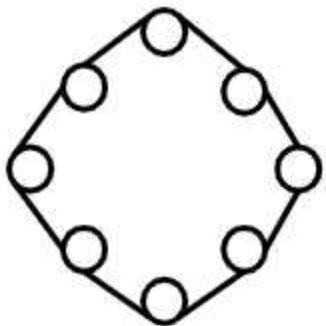
# Why set up an overlay?

- **Performance and/or scalability**
- **Induced:** processes may not know all the peers of the system (**membership**), or **reachability issues**
- **Incremental deployment:** do not require changes to the existing routers, can grow node by node
- **Adaptable:** The overlay algorithm can utilize a number of metrics when making routing and forwarding decisions. Thus the overlay can take application-specific concerns into account
- **Robust:** to node and network failures due to its adaptable nature; with a sufficient number of nodes in the overlay, the network may be able to offer multiple independent (router-disjoint) paths to the same destination. At best, overlay networks are able to route around faults.

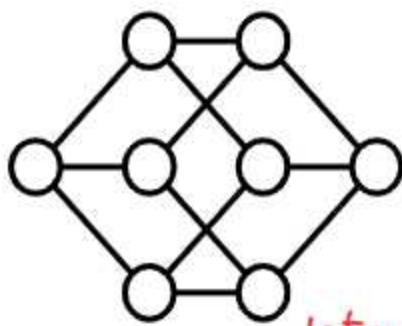


# Basic Graph Metrics

---



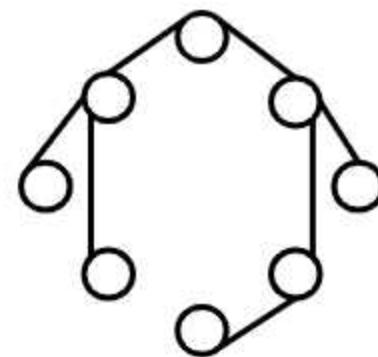
**Distance(a,b):**  
length of the  
shortest path  
between nodes  
a and b



**Diameter:** max  
among all  
distances

*latency*  
↑

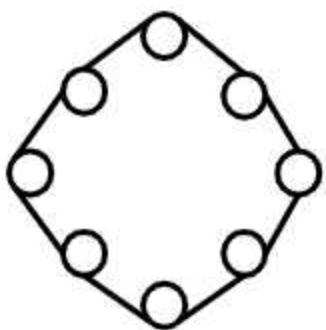
**Closeness**  
**centrality:** the  
average length  
of the shortest  
path between  
the node and all  
other nodes in  
the graph



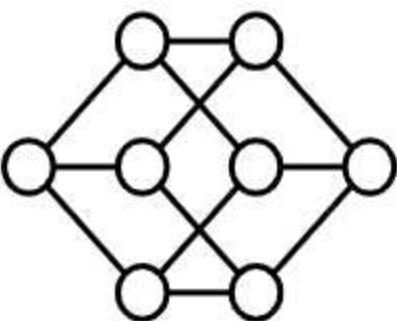
**Betweenness**  
**centrality:** how  
important a node is  
to the shortest  
paths  
through the  
network

# Basic Graph Metrics

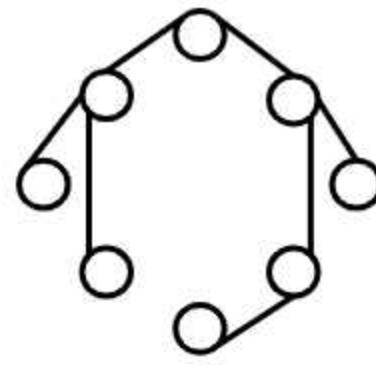
---



**Connected:** it  
exists a path  
between every  
two nodes



**Edge-Connectivity:**  
minimum number of  
edges that has to be  
removed to  
disconnect the  
network

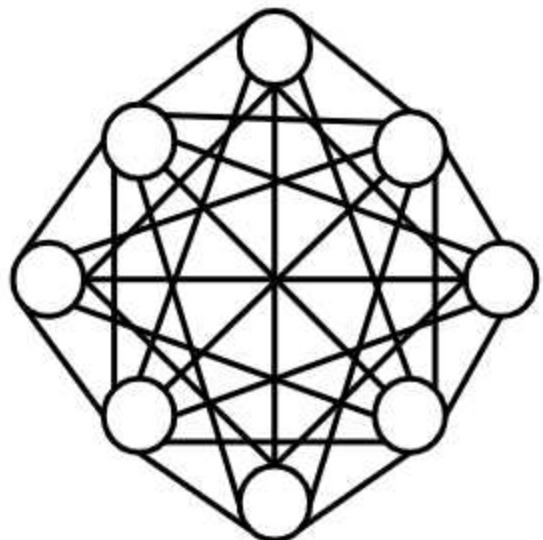


**Node-Connectivity:**  
minimum number of  
nodes that has to be  
removed to  
disconnect the  
network

↳ maximum  
number of processes  
that can crash

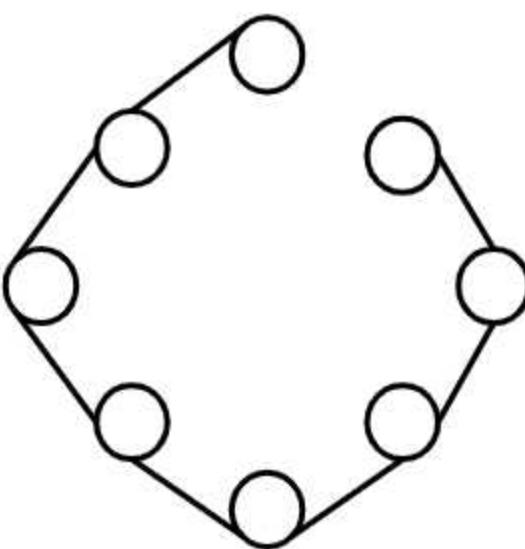
node-based fault tolerance

# Example (BEB - Performance)

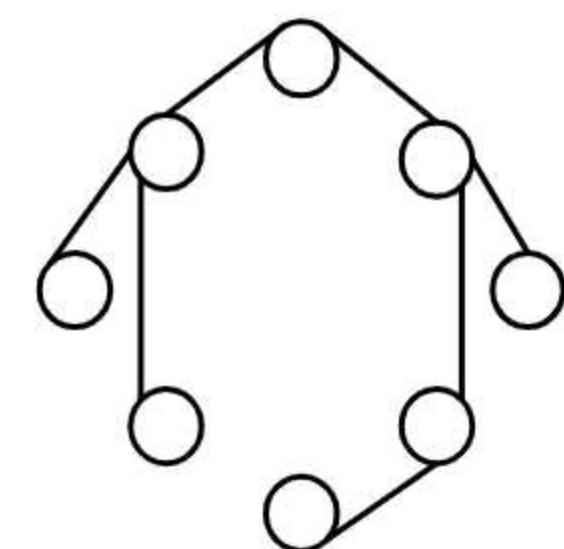


**Load:** the source has to  
send n messages  
**Latency:**  $O(1)$  hop

↳ communicate directly  
with all



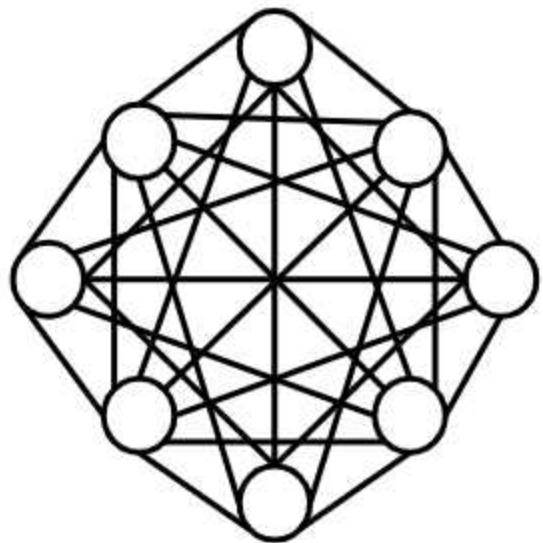
**Load:** the source has to  
send 1 message  
**Latency:**  $O(n)$  hops



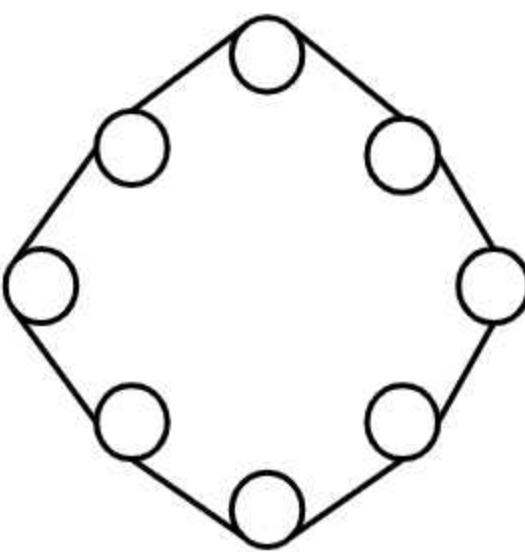
**Load:** the source has  
to send 3 messages  
**Latency:**  $\approx O(\log_2(n))$   
hops

# Example (BEB - Fault Tolerance)

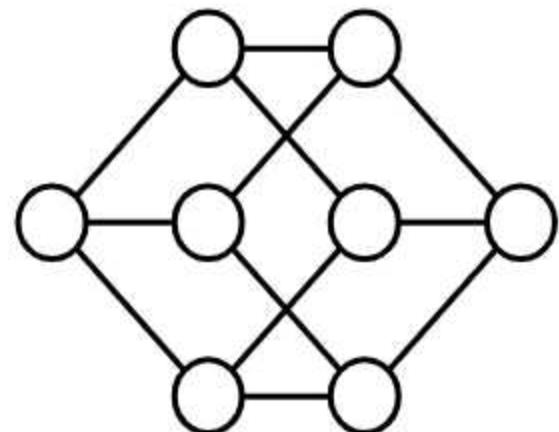
---



Correctness:  $n-1$  crashes



Correctness: 1 crash  
(worst case)



Correctness: 2 crashes  
(worst case)

# Dynamic Distributed Systems

---

Informally, it is a distributed systems that evolves over time

There are several ways in which a distributed system may evolve,  
we consider two types of evolutions:

- **Set of links** (i.e. the available links between the processes change over time)
- **Set of processes** (i.e. the actual processes in the system change over time)

In an actual distributed system **they may both change**

**Overlay network protocols** allows to preserve correctness of distributed protocols despite the evolution of the system

Note: the evolution of a distributed system can either be intentional or due to faults

system evolves in natural behaviour

or enter in a network of losses (intentional)

# Dynamic Distributed Systems

## Main Classes

---

Dynamicity of the processes:

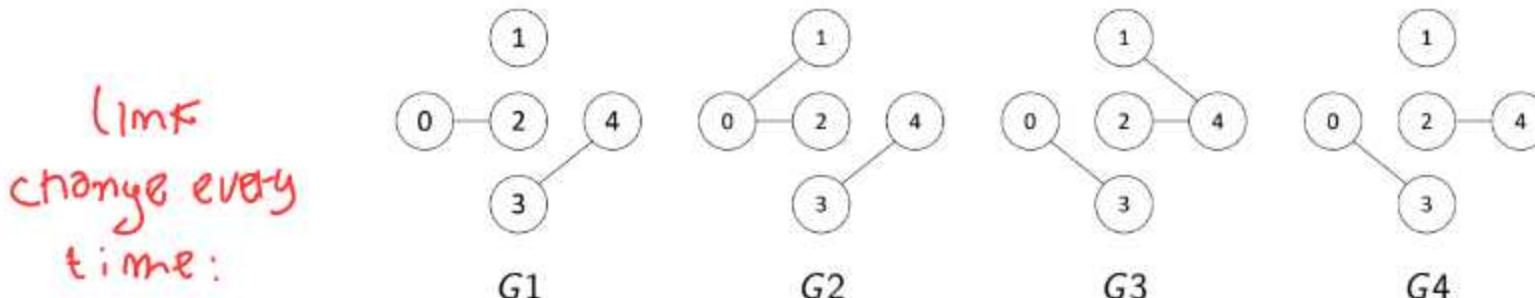
**Reconfiguration:** a protocol manages join and leaves (request to join, request to leave, reconfiguration of the system)

**Churn:** joins and leaves are unmanaged, the churn is characterized by a pattern (e.g. churn rate)

↳ Don't manage by system, know only pattern of arrivals and leaves

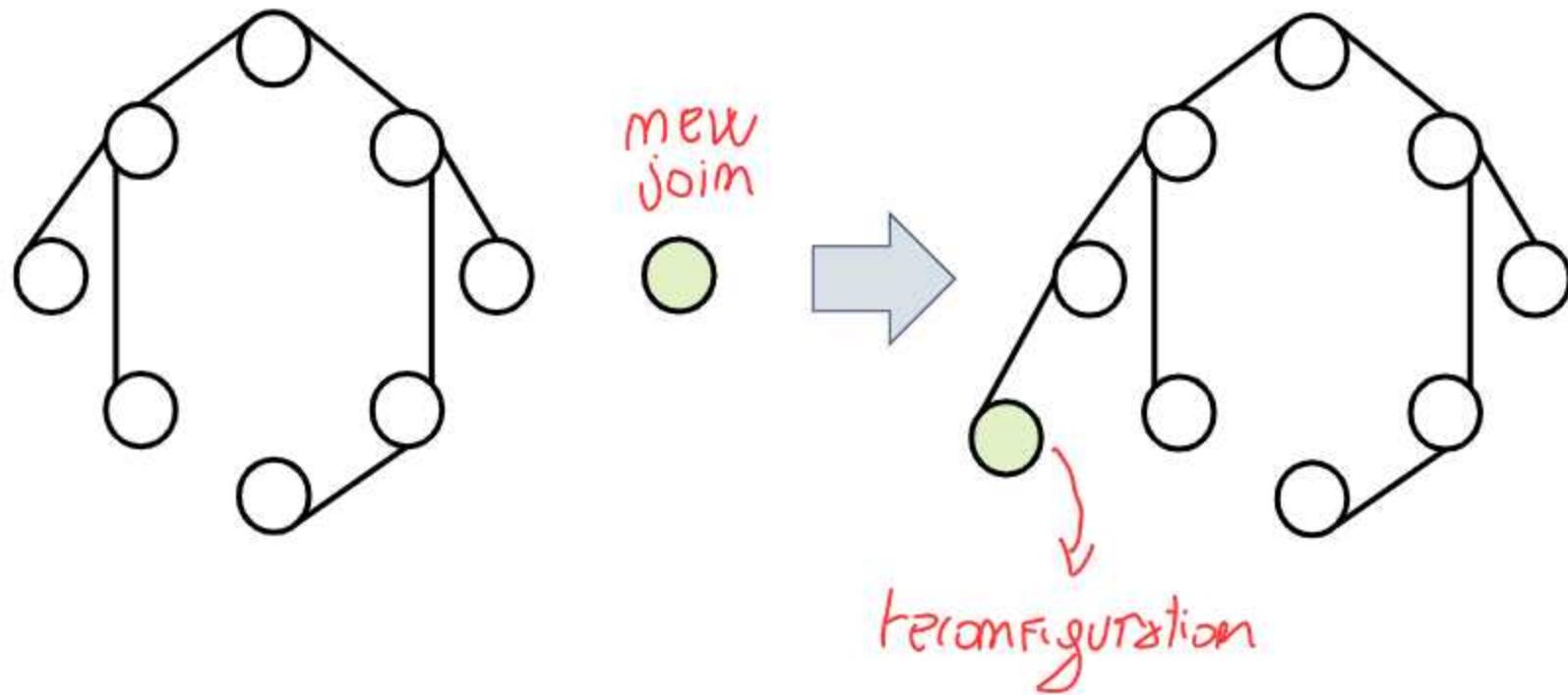
Dynamicity of the links:

Time-varying graphs, general network features, ...



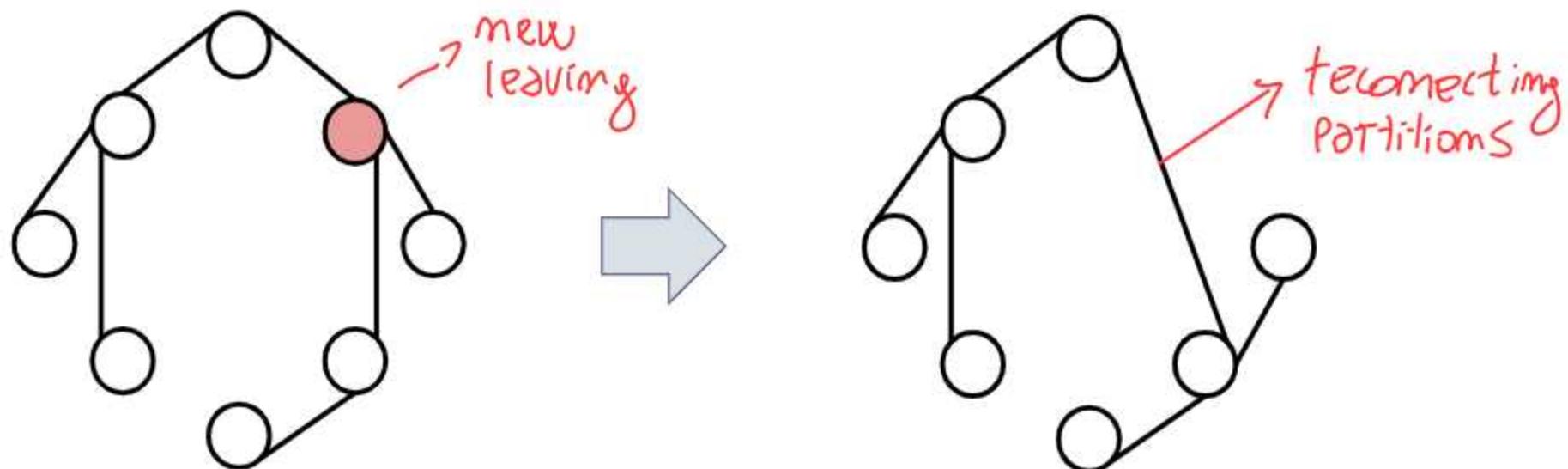
# What an overlay network protocol does (reconfiguration)

---



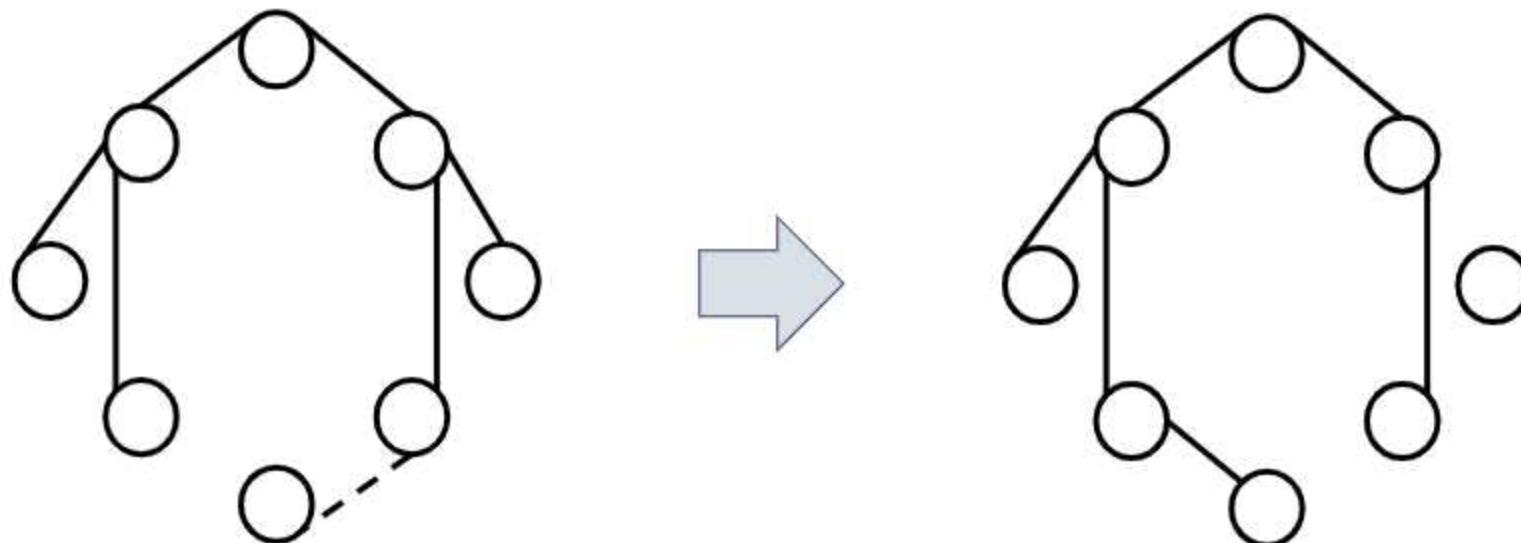
# What an overlay network protocol does (reconfiguration)

---



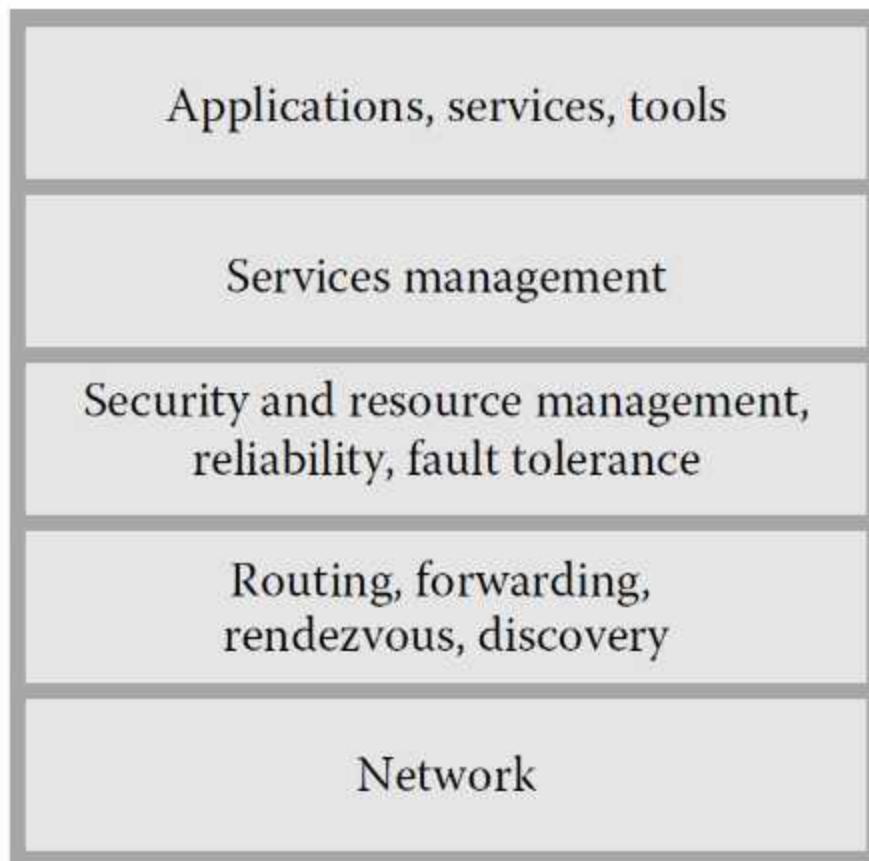
# What an overlay network protocol does (reconfiguration)

---



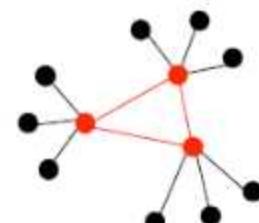
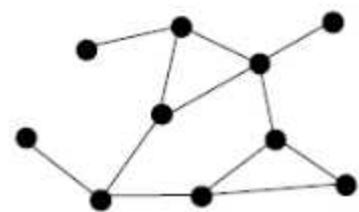
# What an overlay network protocol does

---



# Classes of Overlay Networks

- **Structured** *must remain in that topology*
  - nodes are arranged in a restricted structure (ring, tree, etc.)
- **Unstructured** *(must preserves, despite evolution)*
  - the topology results from some loose rules, without any prior knowledge of the topology
- **Hybrid and/or multi-level:**
  - supernodes form a small overlay



# Classes of Overlay Networks: Features

---

- **Structured**

- efficient data location
- long join and leave procedures
- less robust in high-churn environments

you can easy find data  
→ guarantee on latency

- **Unstructured**

- fast join procedure
- Usually very tolerant to churn
- good for data dissemination, bad for location
- support more complex queries

→ loose connectivity and high  
number of nodes in one time

→ high interconnected

→ not have an organization

- **Hybrid and/or multi-level:**

- Custom features
- Usually large state and high load on supernodes

# Overlay Networks: Applications

---

- **Data/file sharing:** BitTorrent, Bitcoin
  - **CDN:** Content caching to reduce delay and cost
  - **Routing and forwarding:** Reduce routing delays and cost, resiliency, flexibility
  - **Security:** To enhance end-user security and offer privacy protection. For example, virtual private networks (VPNs), onion routing, anonymous content storage, censorship resistant overlays.
  - Other: publish/subscribe, etc.
- all streaming infrastructure*  
*↳ distribution or replica*

amount of data change in time, also DHT

**DHT** map data structure distribute and dynamic

---

**Distributed hash tables** are a class of decentralized distributed algorithms that **offer a lookup service**

---

DHTs **store (key, value) pairs**, and they support the **lookup** of the value associated with a given key

---

The **keys and values are distributed** in the system, and the DHT system must ensure that the nodes have sufficient information of the global state to be able to forward and process lookup requests properly

---

Data object (or value) location information is placed deterministically, at the peers with identifiers corresponding to the data object's unique key.

---

**Linear Hashing** and **LH\***: support table expansion

# DHT

---

The key-based structured algorithms have a **desirable property**: namely, that **they can find data locations within a bounded number of overlay hops**

---

The **DHT algorithm** is responsible for distributing the **keys and values** in such a way that efficient lookup of the value corresponding to a key becomes possible.

---

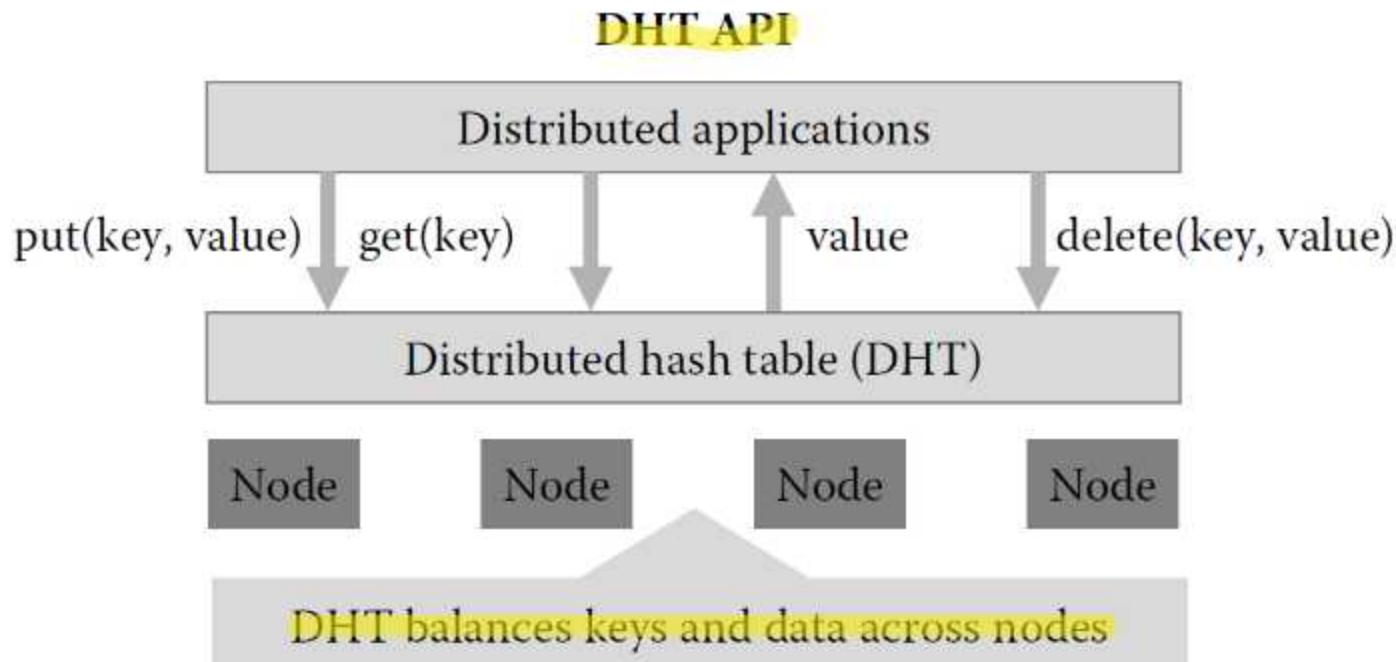
Since peer nodes may come and go, this requires that **the algorithm be able to cope with changes in the distributed system**. In addition, the locality of data plays an important part in all overlays, since they are executed on top of an existing network, typically the Internet

---

The overlay should **take the network locations of the peers into account** when deciding where data is stored, and where messages are sent, in order to **minimize networking overhead**

# DHT

---



# Retrieve data in a P2P system

- Central server (e.g. Napster) scalability issue, and robustness issue
- Flooding Search (e.g. Gnutella) just send a message everywhere and wait for a reply
- Distributed Indexing (e.g. Chord)

three main approach

↳ is DHT, mapping that associates data to a node

# Locating Data

---

= Routing the request to the final destination

## Structured Networks:

- low hop count for large network sizes
- each object must be tracked by a different node
- the overlay must be structured according to a given topology in order to achieve a low hop count
- routing tables must be updated every time a node joins or leaves the overlay

# Locating Data

## Unstructured Networks:

- Flooding or Gossip: send a message to all nodes
  - simple, no topology constraint
  - high network overhead (huge traffic generated by each search request)
  - flooding stopped by TTL
  - only applicable to small number of nodes
  - it may fail
- Random walk, Expanding-ring time-to-live, etc.
- Gossip characterized mainly by maximum number of hop and the fan-out

ask multiple times  
same thing



# Performance in P2P Networks

---

In general, what impact most the performance metrics of P2P applications are:

- The topology of the overlay network → how many hop a message must do
- The routing protocol

# Bootstrapping

---

Bootstrapping basically describes the process of **integrating a new node into a P2P system**

- Dedicated bootstrap servers for join contact a specific server
  - Local host cache, remember precedent configuration the join
  - Random Address Probing
-

# Blockchain Networks

---

**Blockchains in general and cryptocurrencies such as Bitcoin typically run on top of a P2P network**

**In permissionless protocols, such as Bitcoin and Ethereum, any machine can join the network and become a node of the P2P network.**

**A node bootstraps its operation with a discovery protocol to establish connections with other nodes in the system**

**Transactions and blocks are typically propagated in the network using a flooding or gossip protocol**

**Many cryptocurrency networks are characterized by frequent broadcast operations**

**Knowledge of the network topology can give parties an advantage in the dispersal of information (blocks, transactions), which can lead to security risks**

*(not have a total knowledge of network)*

# Bitcoin Network

---

Bitcoin and Ethereum rely on flat **random graph topologies**

The **Bitcoin P2P network topology** is formed by *each peer connecting to 8 nodes (outbound connections) and accepting up to 125 incoming connections*

**Outbound destinations are randomly selected among known identities**

**Before being able to send and receive protocol messages a node has to find other nodes to connect to join the network**

With the **first messages** exchanged between new peers, **they inform each other of a random subset of locally known addresses** with a timestamp of at most 3h ago

# Bitcoin Network Bootstrap

---

In Bitcoin,

- a node first tries to connect to nodes it knows from participating previously.
- If no connections can be established this way, or if the node connects for the very first time, then it queries a list of well-known DNS seeds. The DNS seeds are maintained by Bitcoin community members
- As a last resource, it will try to connect to hardcoded seed nodes

# Bitcoin Network

---

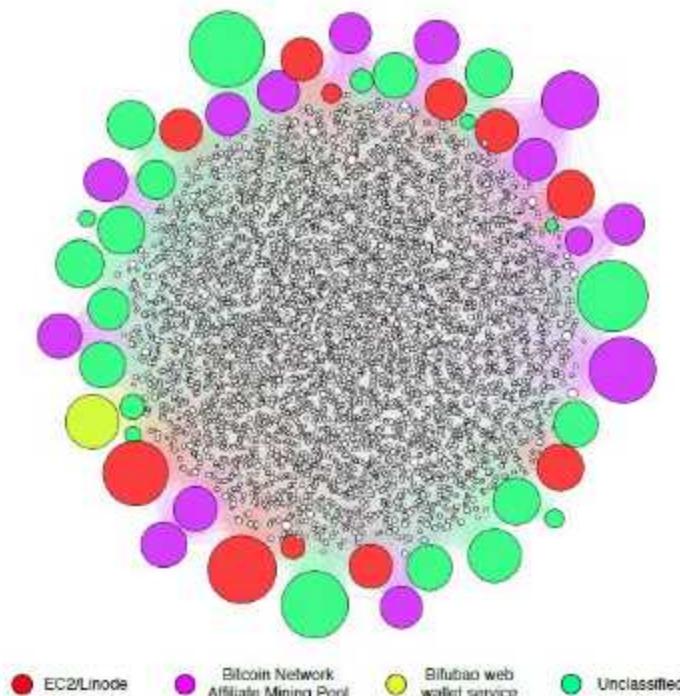


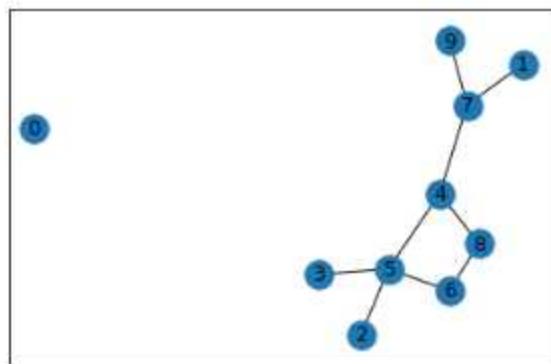
Figure 7: A snapshot of the (reachable) Bitcoin network discovered by AddressProbe on Nov. 5. The highest degree nodes (with degrees ranging 90–708) are colored.

From Miller, Andrew, et al. "Discovering bitcoin's public topology and influential nodes." *et al* (2015).

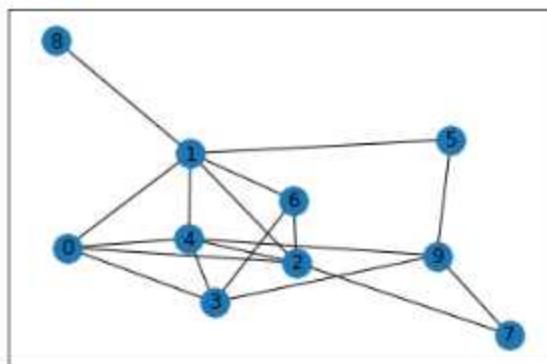
# Erdős-Rényi Random Graphs Model

---

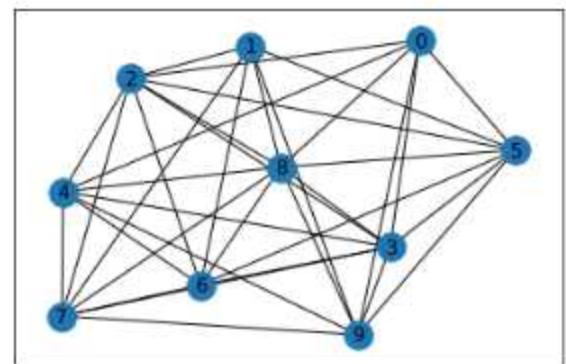
In the  $G(n,p)$  model, a graph is constructed by connecting labeled nodes randomly. Each edge is included in the graph with probability  $p$ , independently from every other edge.



$G(10, 0.2)$



$G(10, 0.5)$



$G(10, 0.8)$

# Dependable Distributed Systems

## Master of Science in Engineering in Computer Science

AA 2022/2023

---

LECTURE 26: BROADCAST IN PRESENCE OF BYZANTINE  
PROCESSES

# Recap on Byzantine processes

---

Byzantine processes may

1. deviate arbitrarily from the instructions that an algorithm assigns to them
  - creating fake messages
  - dropping messages
  - delay the deliveries
  - altering the content of messages
  - ...
2. act as if they were deliberately preventing the algorithm from reaching its goals

↳ For example you can't implement failure detector, you can't detect correct behaviour to faulty one

# Basic step to fight Byzantine processes

→ not guarantee that content is coherent with the logic



Using cryptographic mechanisms to implement the authenticated perfect links abstraction

↳ simple encode everything, guarantee only that content is send by original sender

... But, cryptography alone does not allow to tolerate Byzantine processes

- Considering an arbitrary-faulty sender, asking him/her to digitally sign every broadcast message does not help at all (it may simply sign the two different messages)

↳ only authentication of the two endpoint

# Correct and faulty state

As in the crash failure model, we distinguish between *faulty* and *correct* processes

NOTE: a Byzantine process may act arbitrarily, and no mechanism can guarantee anything that relates to its actions.

byzantine are  
autonomous



We do not define any “uniform” variants of primitives in the Byzantine failure model.

cannot force a correct process to do what a faulty process ~~not~~ disclose to other

# Authenticated Perfect Link

P2P abstraction

**Module 2.5:** Interface and properties of authenticated perfect point-to-point links

**Module:**

**Name:** AuthPerfectPointToPointLinks, **instance** *al*.

**Events:**

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

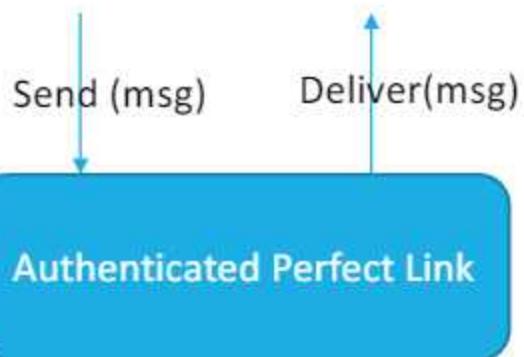
**Indication:**  $\langle al, Deliver \mid p, m \rangle$ : Delivers message *m* sent by process *p*.

**Properties:**

**AL1: Reliable delivery:** If a correct process sends a message *m* to a correct process *q*, then *q* eventually delivers *m*.

**AL2: No duplication:** No message is delivered by a correct process more than once.

**AL3: Authenticity:** If some correct process *q* delivers a message *m* with sender *p* and process *p* is correct, then *m* was previously sent to *q* by *p*.



Same as Perfect  
point-to-point links

↳ use private/public key on top of messages, we apply cryptographic primitive to our algorithm

# Byzantine consistent broadcast specification $\approx BEB$

Module 3.11: Interface and properties of Byzantine consistent broadcast

Module:

Name: ByzantineConsistentBroadcast, instance  $bcb$ , with sender  $s$ .

Events:

**Request:**  $\langle bcb, \text{Broadcast} \mid m \rangle$ : Broadcasts a message  $m$  to all processes. Executed only by process  $s$ .

**Indication:**  $\langle bcb, \text{Deliver} \mid p, m \rangle$ : Delivers a message  $m$  broadcast by process  $p$ .

Properties:

**BCB1: Validity:** If a correct process  $p$  broadcasts a message  $m$ , then every correct process eventually delivers  $m$ .

**BCB2: No duplication:** Every correct process delivers at most one message.

**BCB3: Integrity:** If some correct process delivers a message  $m$  with sender  $p$  and process  $p$  is correct, then  $m$  was previously broadcast by  $p$ .

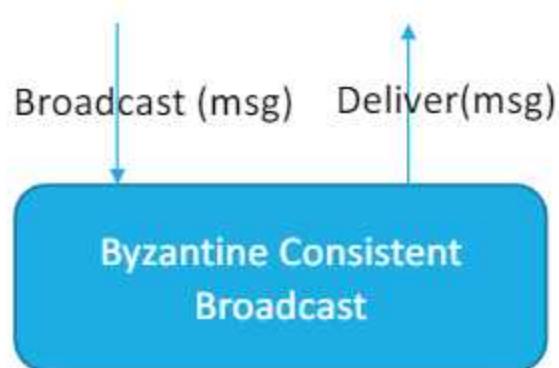
**BCB4: Consistency:** If some correct process delivers a message  $m$  and another correct process delivers a message  $m'$ , then  $m = m'$ .

↳ some form of agreement

Byzantine can send messages at random, can't trust received message



The specification refers to a single broadcast event!



# Byzantine consistent broadcast implementation

asynchronous system  
without failure detector is  
traversable synchrony

## Algorithm 3.16: Authenticated Echo Broadcast

Implements:

ByzantineConsistentBroadcast, **instance** *bcb*, with sender *s*.

Uses:

AuthPerfectPointToPointLinks, **instance** *al*.

```
upon event ( bcb, Init ) do
    sentecho := FALSE;
    delivered := FALSE;
    echos := [⊥]N;
    → implicit consensus
    → for no duplications

upon event ( bcb, Broadcast | m ) do
    forall q ∈ Π do
        trigger ( al, Send | q, [SEND, m] );
    // only process s

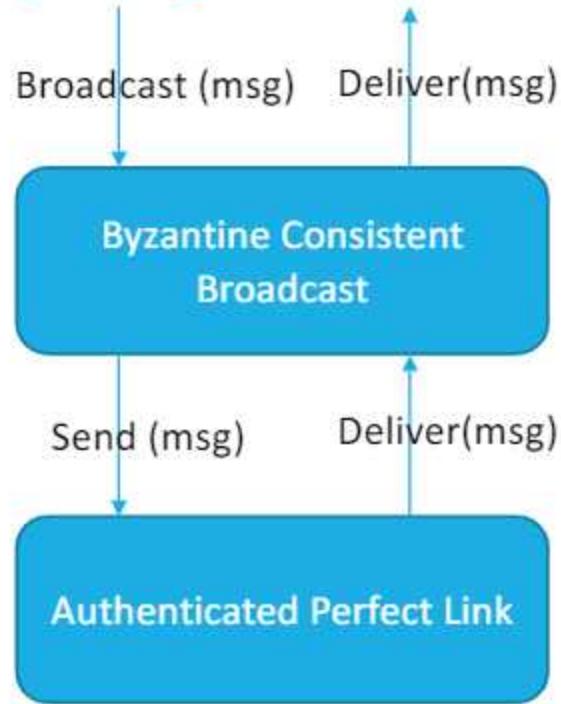
upon event ( al, Deliver | p, [SEND, m] ) such that p = s and sentecho = FALSE do
    sentecho := TRUE;
    forall q ∈ Π do
        trigger ( al, Send | q, [ECHO, m] );

upon event ( al, Deliver | p, [ECHO, m] ) do
    if echos[p] = ⊥ then
        echos[p] := m;

upon exists m ≠ ⊥ such that #({p ∈ Π | echos[p] = m}) >  $\frac{N+f}{2}$ 
    and delivered = FALSE do
        delivered := TRUE;
        trigger ( bcb, Deliver | s, m );
```

at least  $\frac{N+1}{2}$  processes voted *m*

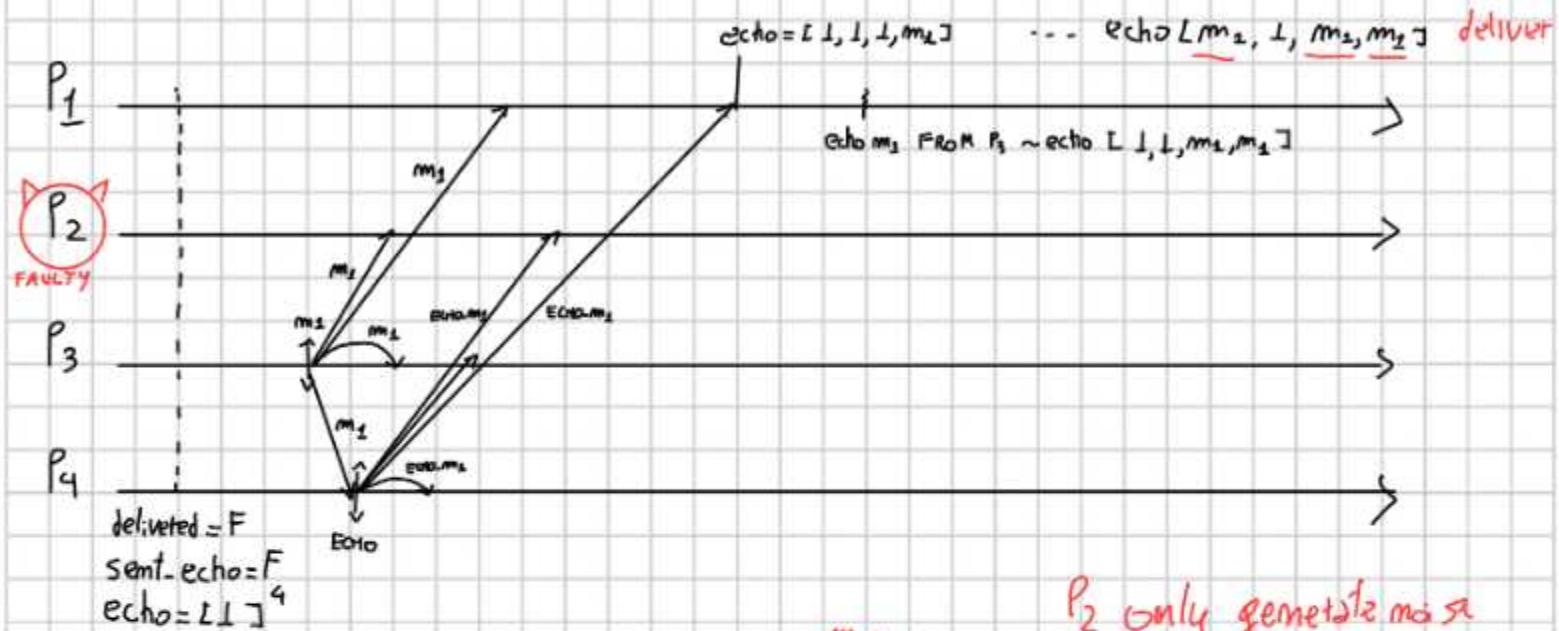
Correctness is ensured if  
 $N > 3f$



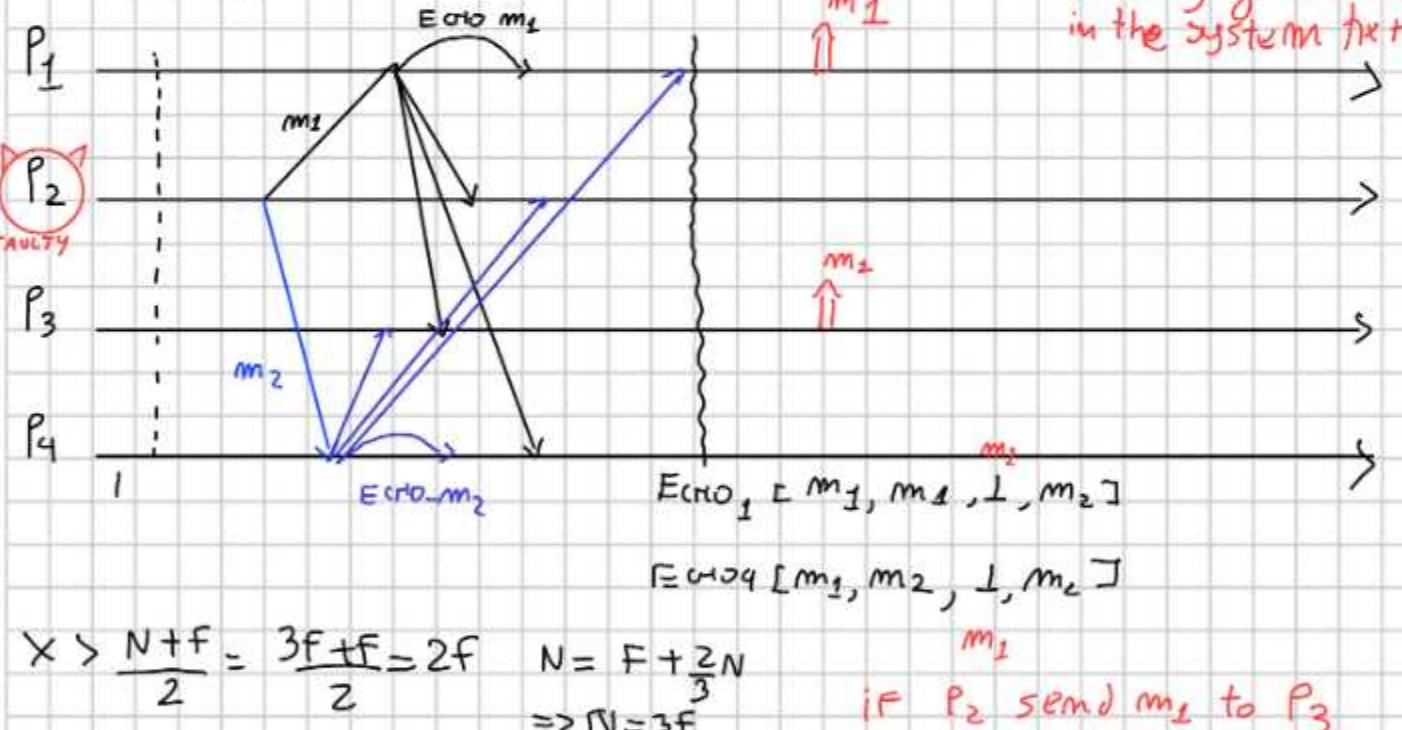
## Byzantine consistent broadcast

$N > 3f$  number of processes must be 3 times the number of failure we want to tolerate, for  $f = 1 \Rightarrow N \geq 4$

$$N > \frac{N+f}{2} = \frac{4+1}{2} = 2.5 \times 3$$



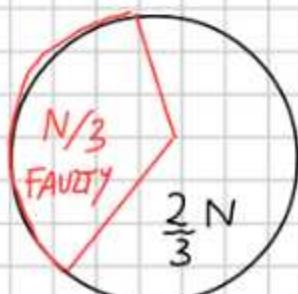
$P_2$  only generate noise in the system here



$$N > \frac{N+f}{2} = \frac{3f+f}{2} = 2f \quad N = f + \frac{2N}{3} \quad \Rightarrow 3N = 3f$$

$m_1$   
if  $P_2$  send  $m_1$  to  $P_3$

# copies  $> 2f$



$N$

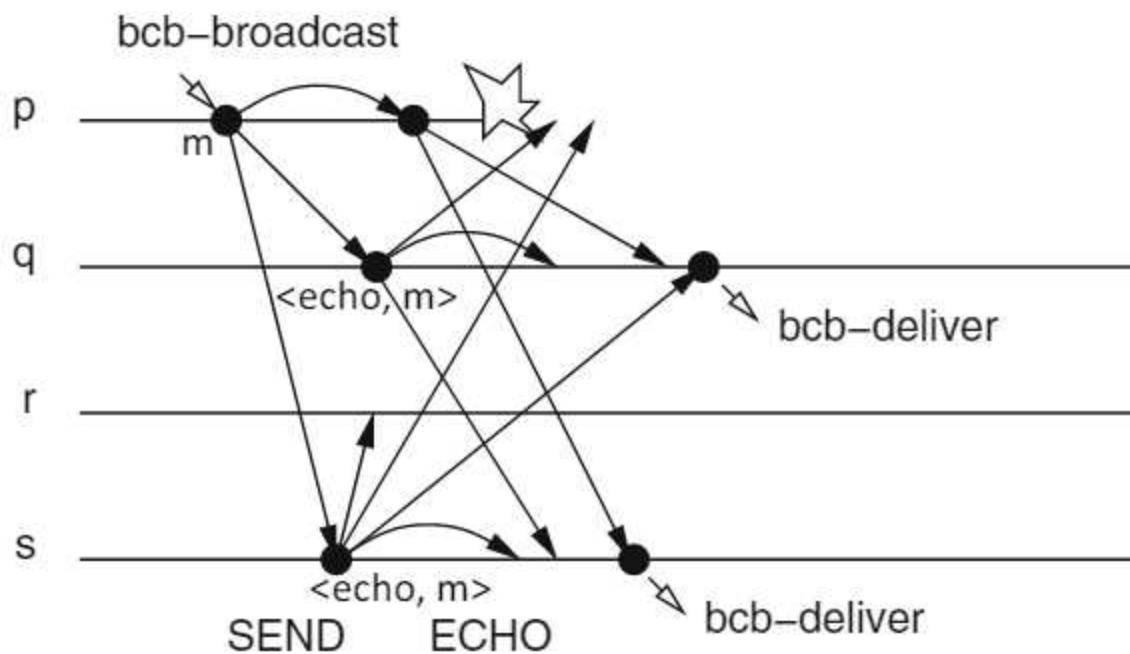
# distinct  $\leq N-f$

# copies  $> 2f$

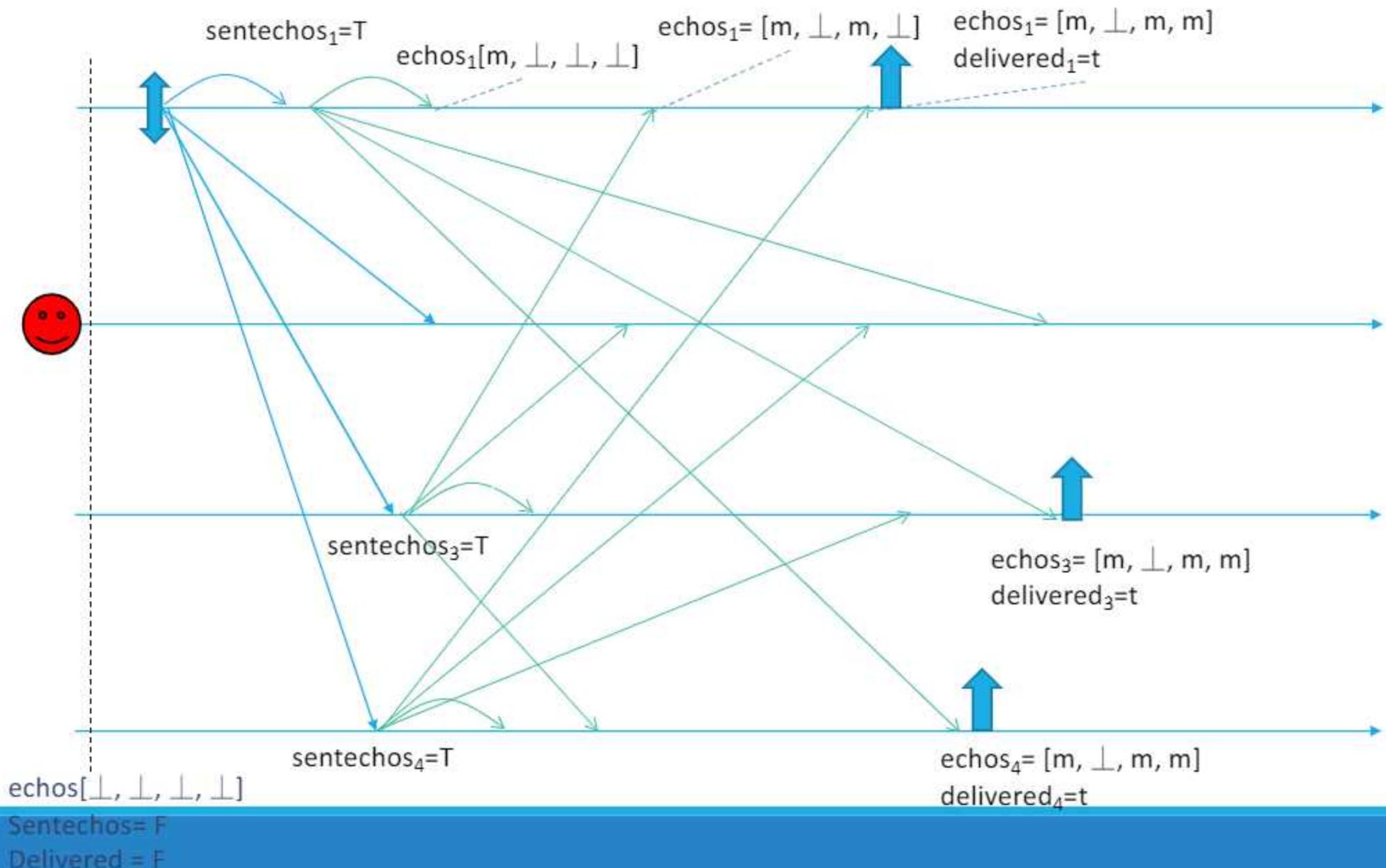
$m_1, \dots, m_1, m_2, \dots, m_2$   
 $\underbrace{1}_{f \text{ copies of } m_1} \quad \underbrace{1}_{f \text{ copies of } m_2}$

# Byzantine consistent broadcast example

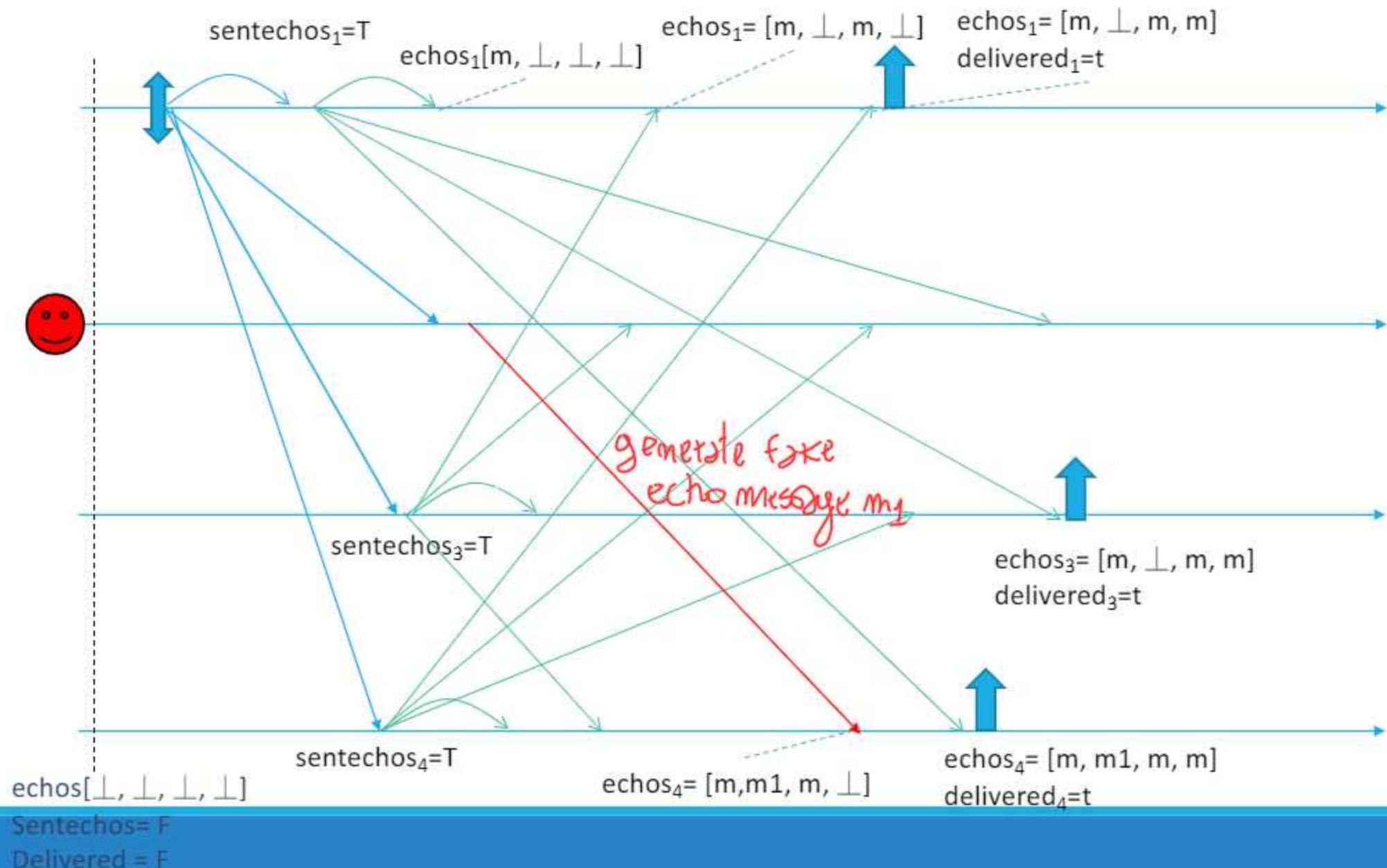
---



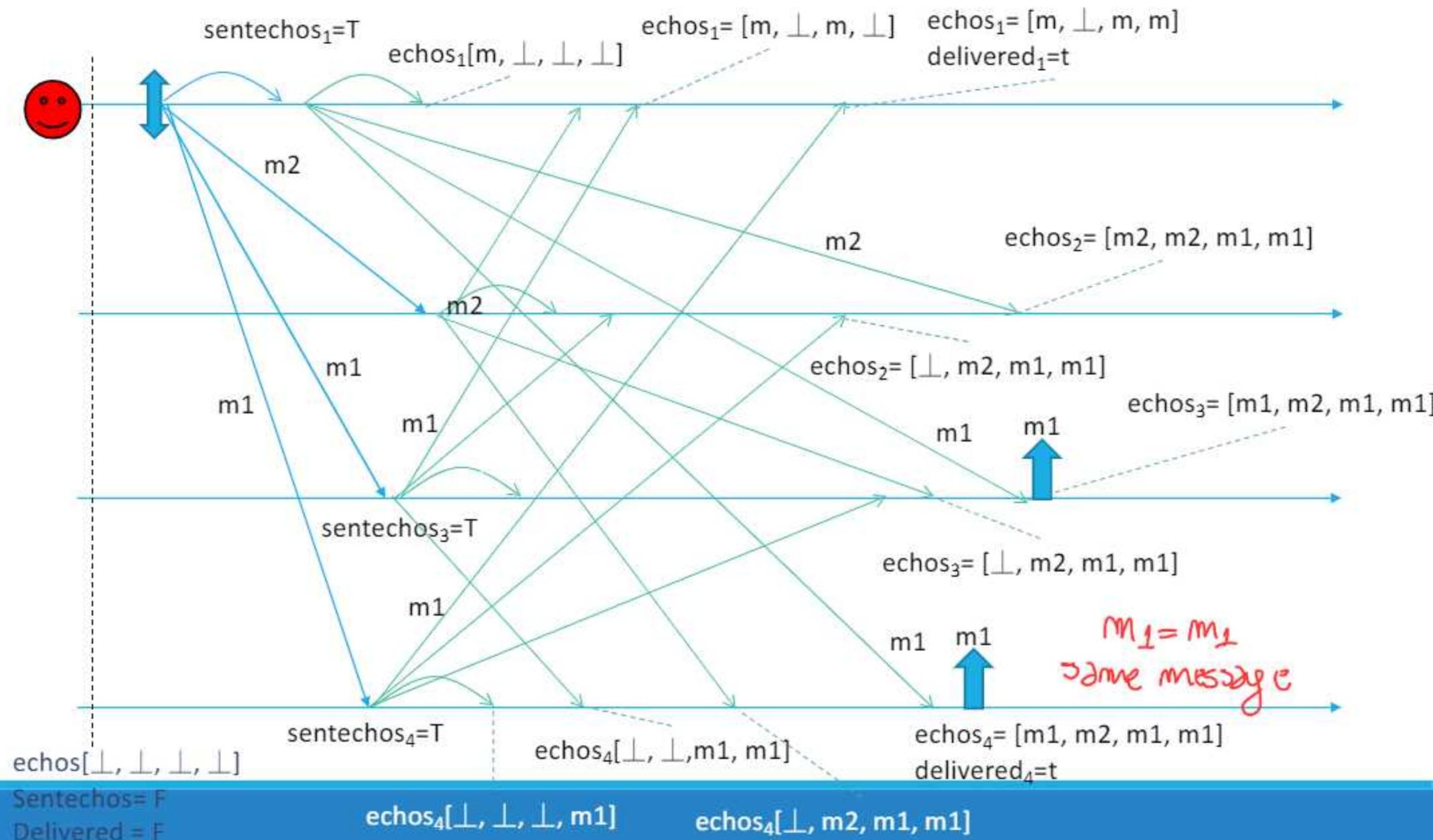
Assumption for Correctness:  $N > 3f \rightarrow f=1$ ,  $N=4$  with p1 acting as sender (correct sender)



Assumption for Correctness:  $N > 3f \rightarrow f=1$ ,  $N=4$  with  $p_1$  acting as sender (correct sender)



Assumption for Correctness:  $N > 3f \rightarrow f=1$ ,  $N=4$  with **p1 acting as sender (faulty sender)**



# Byzantine Reliable Broadcast specification

Module 3.12: Interface and properties of Byzantine reliable broadcast

Module:



The specification refers to a single broadcast event!

Name: ByzantineReliableBroadcast, instance  $brb$ , with sender  $s$ .

Events:

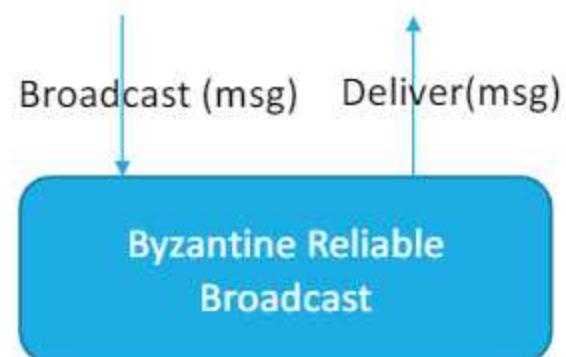
**Request:**  $\langle brb, Broadcast \mid m \rangle$ : Broadcasts a message  $m$  to all processes. Executed only by process  $s$ .

**Indication:**  $\langle brb, Deliver \mid p, m \rangle$ : Delivers a message  $m$  broadcast by process  $p$ .

Properties:

**BRB1–BRB4:** Same as properties BCB1–BCB4 in Byzantine consistent broadcast (Module 3.11).

**BRB5: Totality:** If some message is delivered by any correct process, every correct process eventually delivers a message.



# Byzantine Reliable Broadcast implementation

*we have double echo stage*

**Algorithm 3.18:** Authenticated Double-Echo Broadcast

**Implements:**

ByzantineReliableBroadcast, **instance** *brb*, with sender *s*.

**Uses:**

AuthPerfectPointToPointLinks, **instance** *al*.

```
upon event ( brb, Init ) do
  sentecho := FALSE;
  sentready := FALSE;
  delivered := FALSE;
  echos := [⊥]N;
  readyss := [⊥]N;

upon event ( brb, Broadcast | m ) do
  forall q ∈  $\Pi$  do
    trigger ( al, Send | q, [SEND, m] );
  // only process s

upon event ( al, Deliver | p, [SEND, m] ) such that p = s and sentecho = FALSE do
  sentecho := TRUE;
  forall q ∈  $\Pi$  do
    trigger ( al, Send | q, [ECHO, m] );

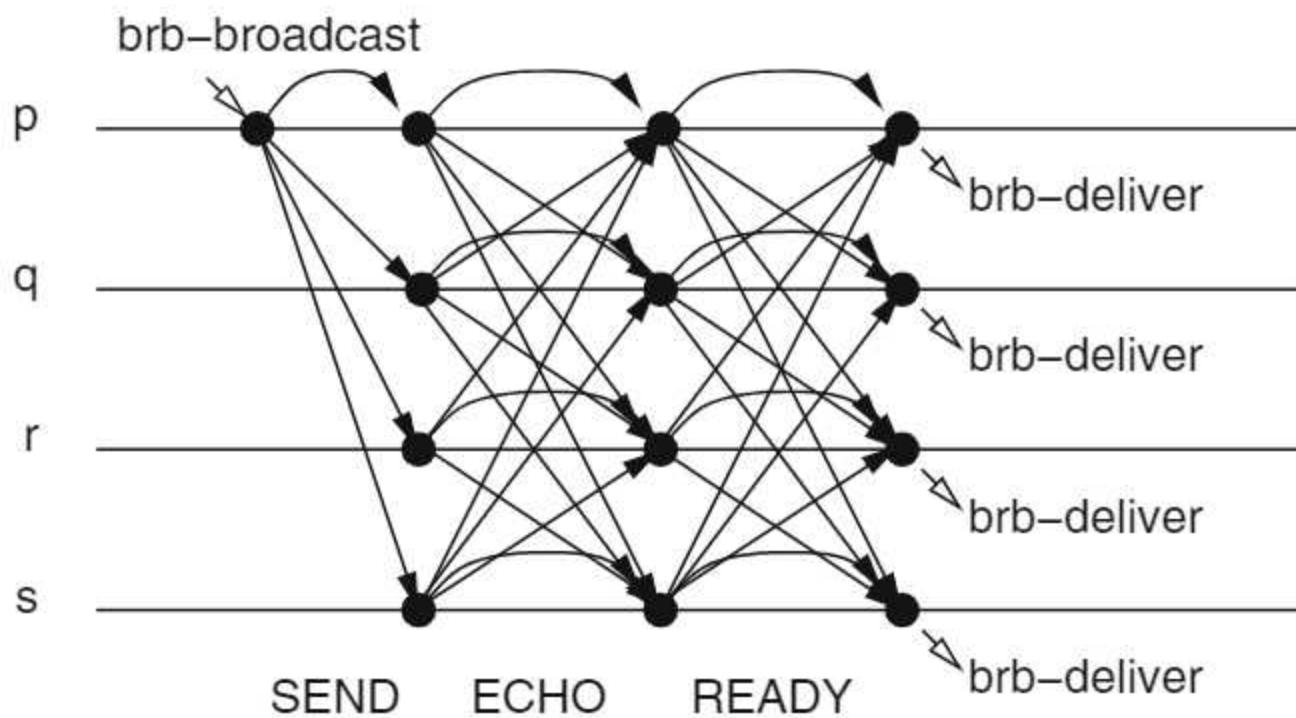
upon event ( al, Deliver | p, [ECHO, m] ) do
  if echos[p] = ⊥ then
    echos[p] := m;
```

```
upon exists m ≠ ⊥ such that #({p ∈  $\Pi$  | echos[p] = m}) >  $\frac{N+f}{2}$ 
  and sentready = FALSE do
    sentready := TRUE;
    forall q ∈  $\Pi$  do
      trigger ( al, Send | q, [READY, m] );
    → instead of
    → deliver, send
    → to other that i'm
    → ready to deliver
  upon event ( al, Deliver | p, [READY, m] ) do
    if readyss[p] = ⊥ then
      readyss[p] := m;

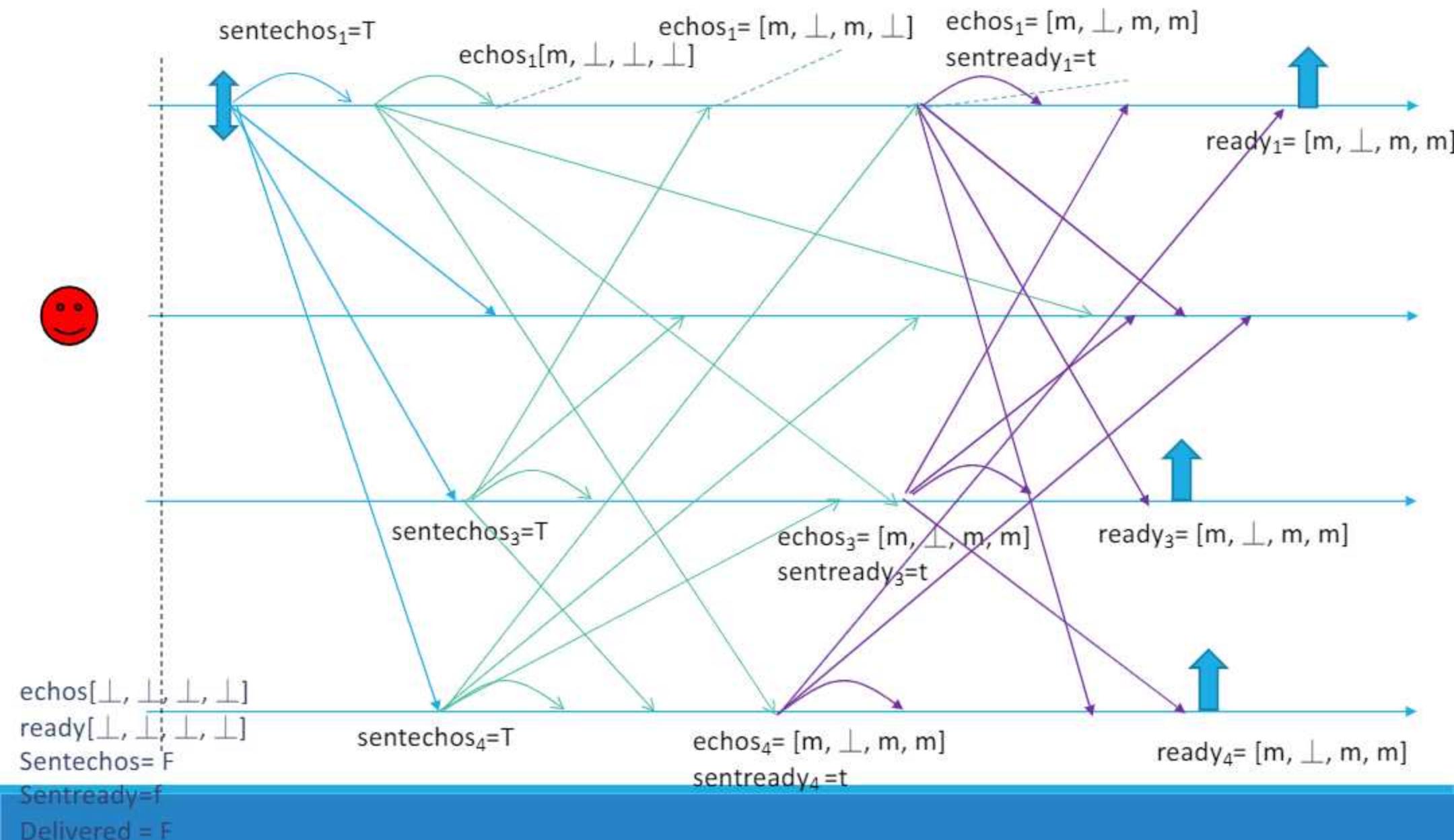
upon exists m ≠ ⊥ such that #({p ∈  $\Pi$  | readyss[p] = m}) > f
  and sentready = FALSE do
    sentready := TRUE;
    forall q ∈  $\Pi$  do
      trigger ( al, Send | q, [READY, m] );
  upon exists m ≠ ⊥ such that #({p ∈  $\Pi$  | readyss[p] = m}) > 2f
    and delivered = FALSE do
      delivered := TRUE;
      trigger ( brb, Deliver | s, m );
```

# Byzantine Reliable Broadcast example

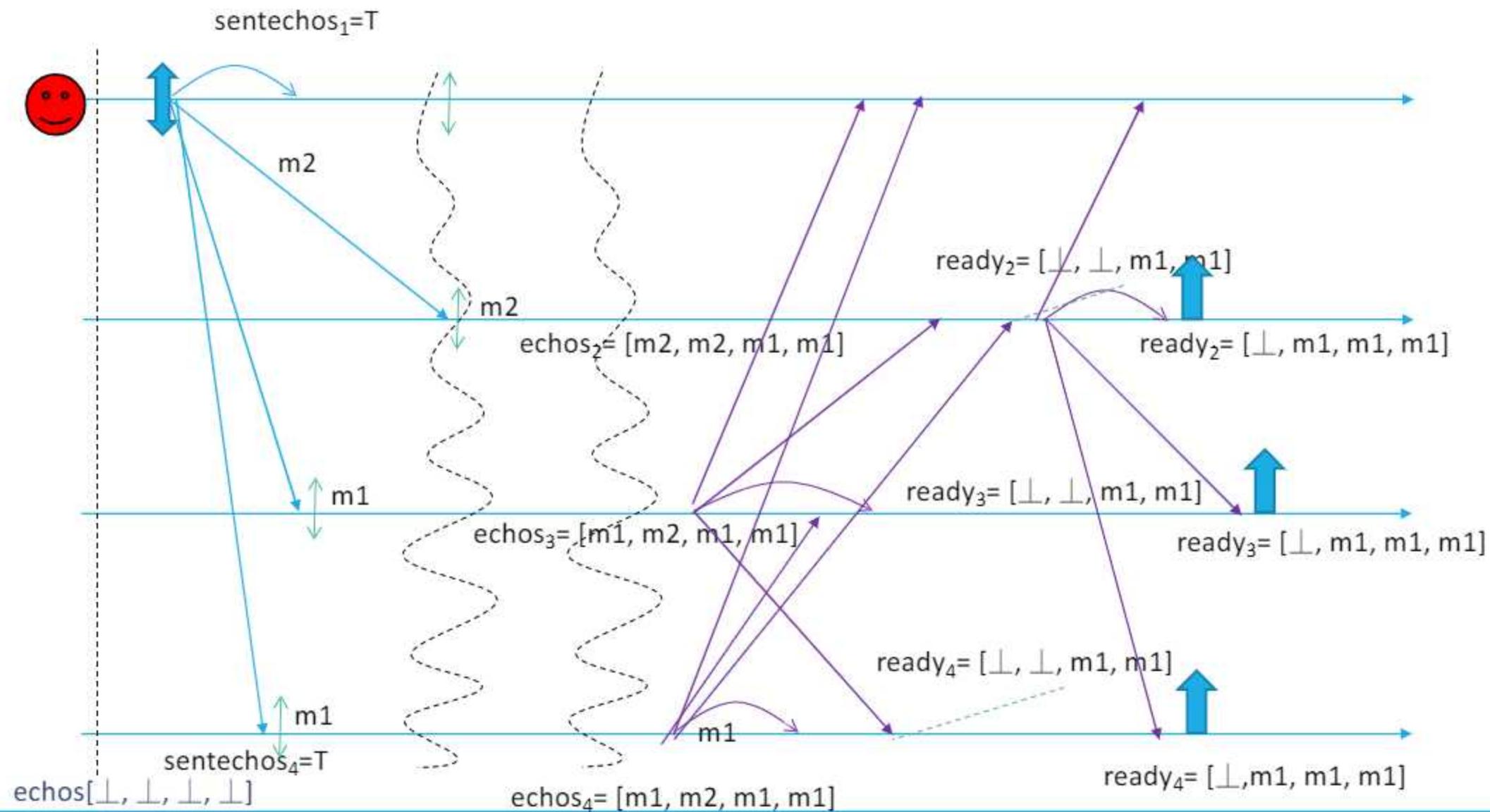
---



Assumption for Correctness:  $N > 3f \rightarrow f=1$ ,  $N=4$  with  $p_1$  acting as sender (correct sender)

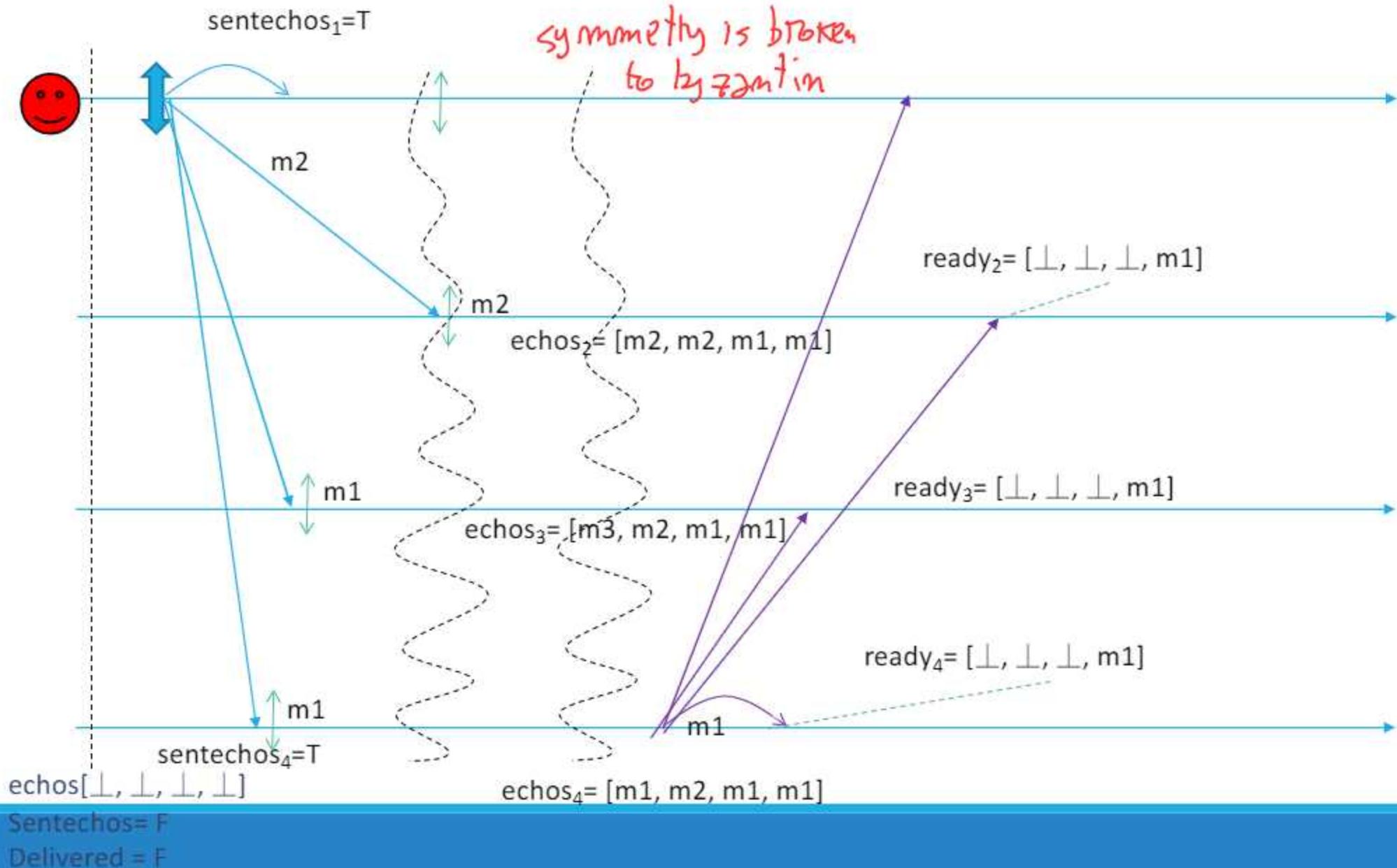


Assumption for Correctness:  $N > 3f \rightarrow f=1$ ,  $N=4$  with  $p_1$  acting as sender (faulty sender)

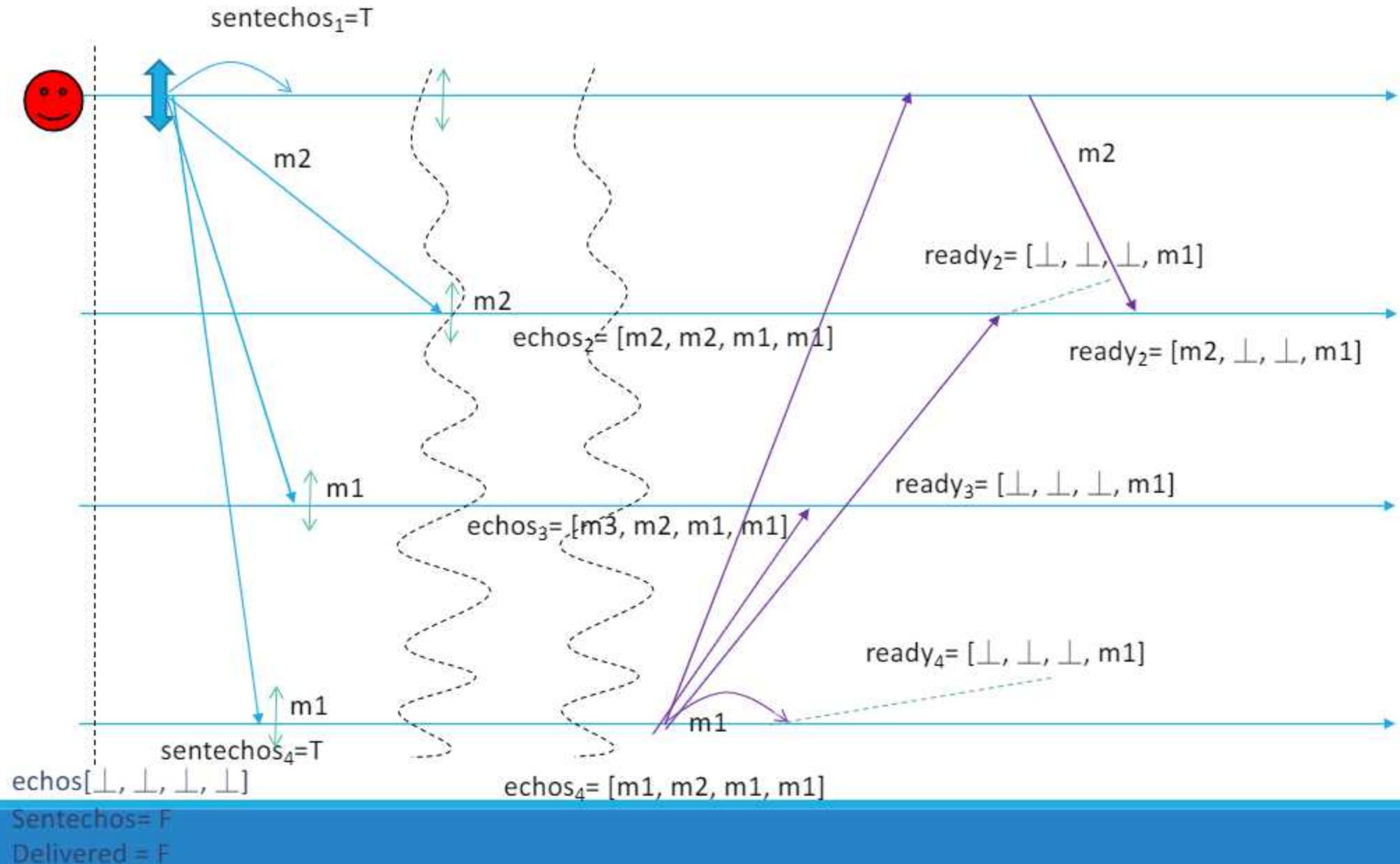


Sentechos= F  
Delivered = F

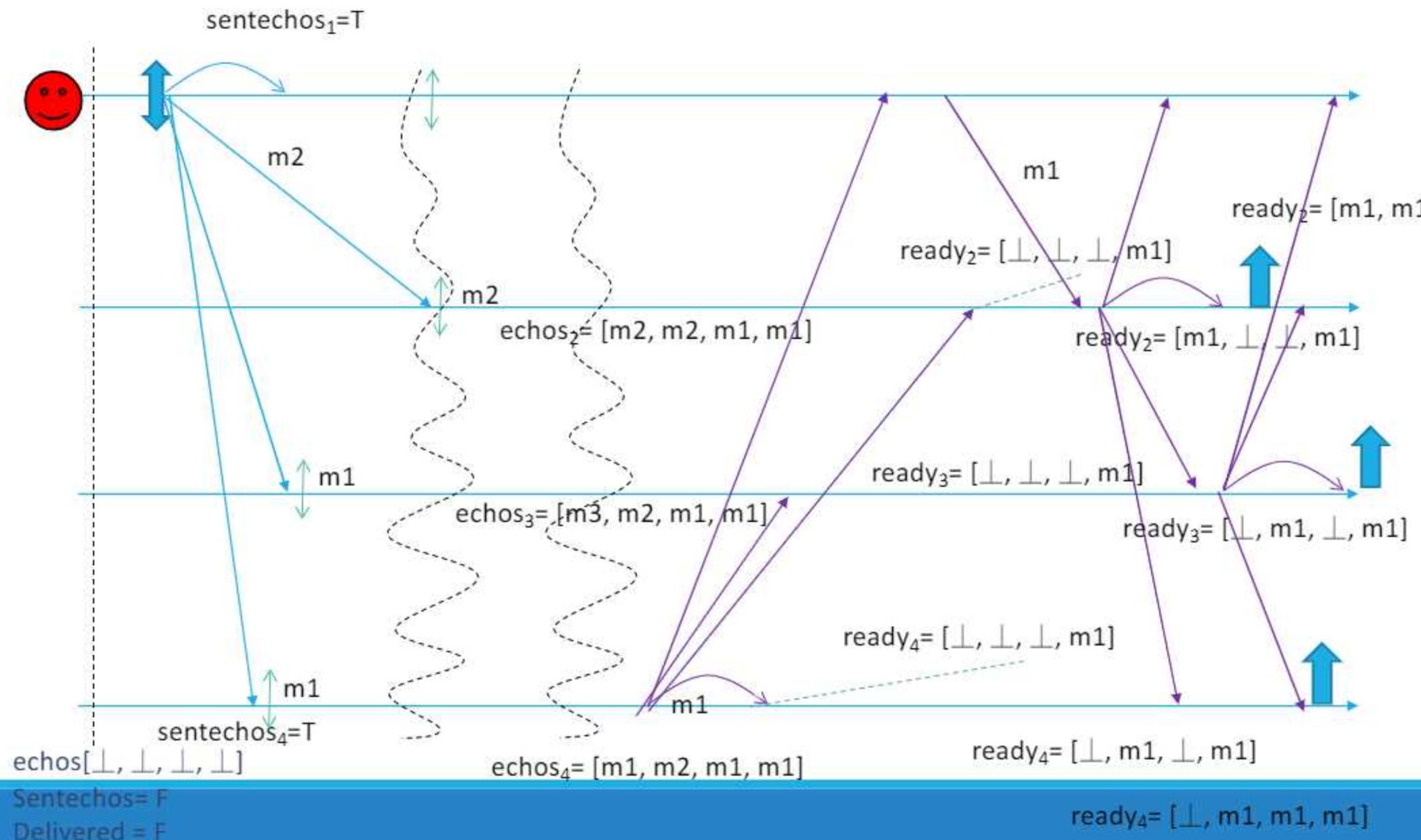
Assumption for Correctness:  $N > 3f \rightarrow f=1$ ,  $N=4$  with  $p_1$  acting as sender (faulty sender)



Assumption for Correctness:  $N > 3f \rightarrow f=1$ ,  $N=4$  with  $p_1$  acting as sender (faulty sender)



Assumption for Correctness:  $N > 3f \rightarrow f=1$ ,  $N=4$  with  $p_1$  acting as sender (faulty sender)



# References

---

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

- Chapter 2 – Section 2.4.6
- Chapter 3 – Section 3.10 (except 3.10.4), Section 3.11

# Dependable Distributed Systems

## Master of Science in Engineering in

## Computer Science

AA 2022/2023

---

LECTURE 27: RELIABLE COMMUNICATION IN PRESENCE  
OF BYZANTINE PROCESSES

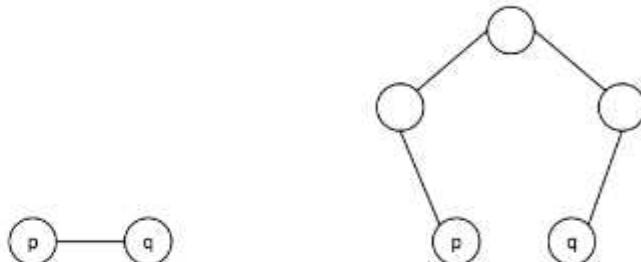
# From Perfect Links to Reliable Communication

A link model the capability of a pair processes to exchange messages

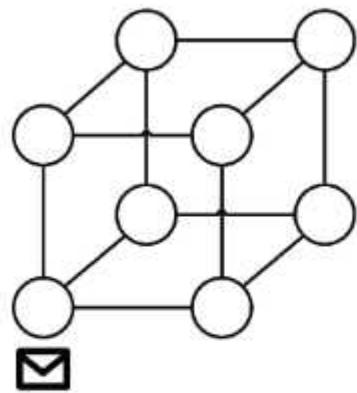
## Perfect Point-to-Point Link:

- **Reliable delivery:** If a correct process  $p$  sends a message  $m$  to a correct process  $q$ , then  $q$  eventually delivers  $m$ .
- **No duplication**
- **No creation**

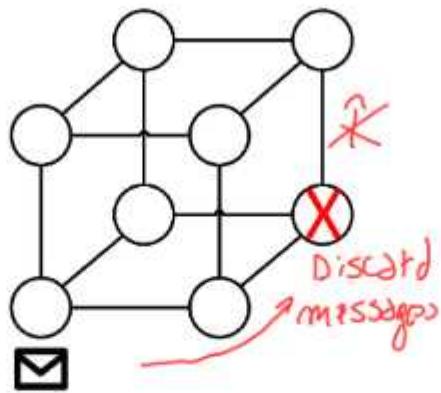
Can processes  $p$  and  $q$  exchange messages?



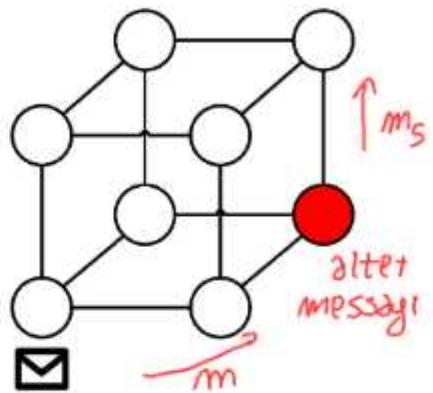
# From Perfect Links to Reliable Communication



Reliable Delivery

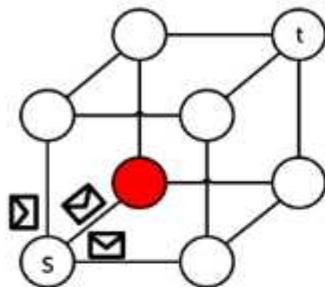


Discard message



No Creation  
No Duplication

# Reliable Communication Specification



- **Safety** = if  $p_t$  is a correct process and it delivers a **content c** from  $p_s$ , then  $p_s$  previously sent c (*message authorship*)
- **Liveness** = if  $p_s$  is a correct process and it sends a **content c** to a correct process  $p_t$ , then  $p_t$  eventually delivers c from  $p_s$  (*message delivery*)

# Reliable Communication

Various assumptions we can consider: and solution vary with these

What kind of faults may occur?

How many faults may occur?

How faults are distributed?

Which facilities are available?

What knowledge the processes have

about the system?

...

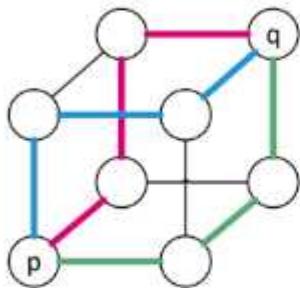


we consider some assumptions  
for solve a problem and analize the  
complexity of a solution

# Menger Theorem

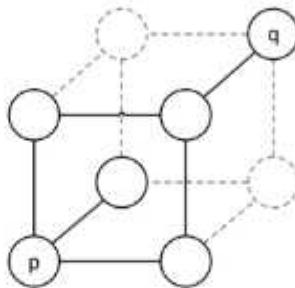
**Menger Theorem - Vertex Cut VS Disjoint Paths:** Let  $G = (V, E)$  be a graph and  $p, q \subseteq V$ . Then the minimum number of vertexes separating  $p$  from  $q$  in  $G$  is equal to the maximum number of disjoint  $p - q$  paths in  $G$ .

[https://en.wikipedia.org/wiki/Menger%27s\\_theorem](https://en.wikipedia.org/wiki/Menger%27s_theorem)



Disjoint Paths

3 disjoint path help



Min-Cut

minimum number of node  
to remove for disconnect  
the graph, here three.

NOTE: the min-cut can  
be computed with a  
polynomial algorithm  
in the size of the graph

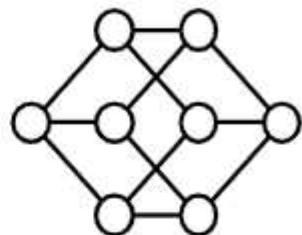
↓  
maximum number of disjoint  
path between between  $p - q$

$| \text{min-cut} | = | \text{disjoint-path} |$   
( $p, q$ )

# Globally Bounded Fault Model - Crashes

## System model:

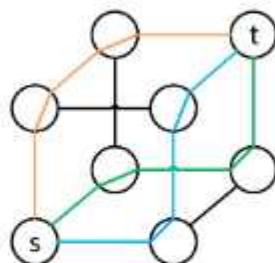
- at most  $f$  processes can be faulty, crash faults
- $n$  processes
- perfect point-to-point links



## Correctness Conditions:

**RC:** At least  $f+1$  node-disjoint paths must exist between the source and the target processes

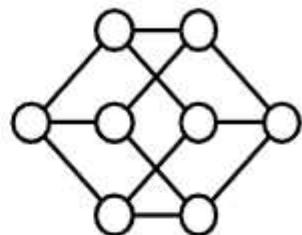
**BEB:** node-connectivity  $k > f+1$



# Globally Bounded Fault Model - Crashes

## Protocol for (BEB):

- the source P2P-sends the content over all of its links



- every process that P2P-receives a content for the first time delivers the content and relais it (P2P-sends it) over all of its links

# Globally Bounded Fault Model – Byzantine Faults, Authenticated Messages

## System model:

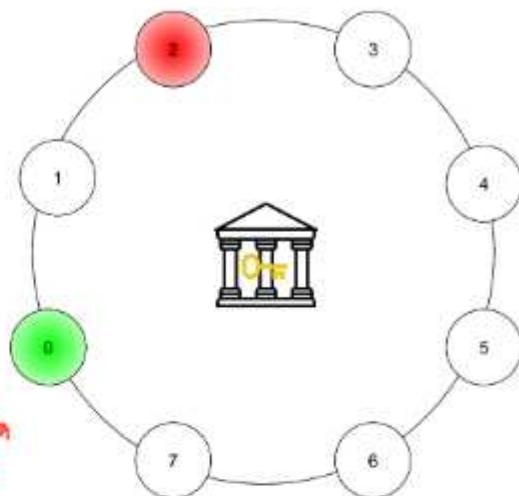
- at most  $f$  processes can be faulty, Byzantine faults
- $n$  processes
- perfect point-to-point links
- digitally signed messages – authenticated messages  
(every process can sign ONLY its messages  
and verify the signatures generated by every process)

## Correctness Conditions (same as the crash case):

RC: At least  $f+1$  node-disjoint paths must exist between the source and the target processes

BEB: node-connectivity  $k > f+1$

Similar protocol as the crash case,  
contents attached with a digital signature,  
only contents with a valid digital signature are considered



# Globally Bounded Fault Model – Byzantine Faults, Authenticated Links

System model:

- at most  $f$  processes can be faulty, Byzantine faults

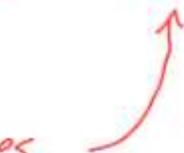
-  $n$  processes

- authenticated perfect point-to-point links, <sup>not messages</sup>

- processes know the topology of the communication

\_> every process can compute (deterministically) node-disjoint paths between all pairs of processes

only pair of processes  
can verify messages

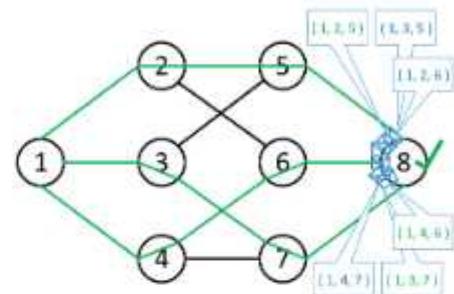


in messages all  
processes

# Globally Bounded Fault Model – Byzantine Faults, Autenticated Links

Reliable communication

```
1: upon DolevR_send( $t, c$ ) do           i: sender  
2:   for  $\pi \in \Pi_{i,t}$  do                  c: destination  
3:     | send( $\langle i, t, c, \pi \rangle, \pi[1]$ )           <:  
4: upon receive( $\langle s, t, c, \pi \rangle, j$ ) do            $\Pi$ :  
5:   | if  $\exists \pi \in \Pi_{s,t}, \exists m \in \mathbb{N}^0, \pi[m-1] = j, \pi[m] = i$  then           Optimal message complexity:  $O(n)$   
6:     | | if  $|\pi| = m$  then           Optimal delivery complexity:  $O(f)$   
7:       | | |  $Paths_{(s,c)} \leftarrow Paths_{(s,c)} \cup \pi$   
8:     | | else  
9:       | | | send( $\langle s, t, c, \pi \rangle, \pi[m+1]$ )  
10: upon  $|Paths_{(s,c)}| > f$  do  
11:   | DolevU_deliver( $\langle s, c \rangle$ )
```



Correctness: node-connectivity  $> 2f$

Question: why  $2f$  is enough?

# Globally Bounded Fault Model – Byzantine Faults, Autenticated Links

---

System model:

- at most  $f$  processes can be faulty, Byzantine faults

-  $n$  processes

- authenticated perfect point-to-point links

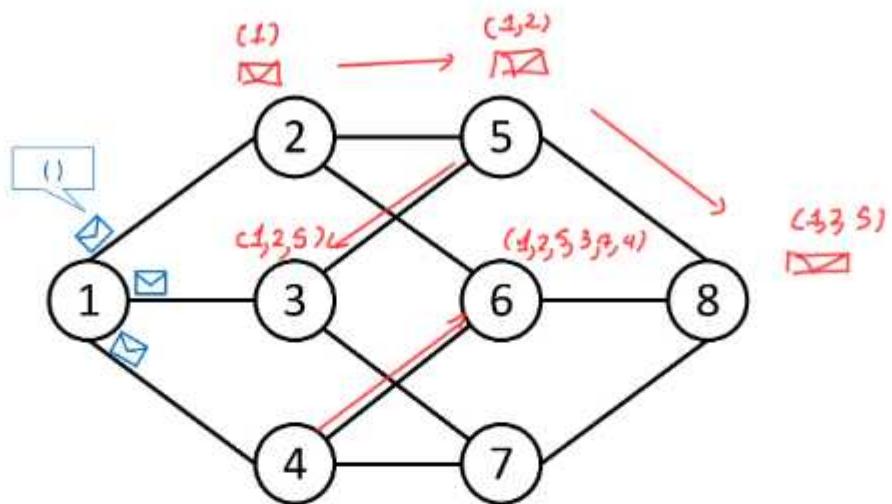
— processes know the topology of the communication

↳ no knowledge of topology now

—> Flooding

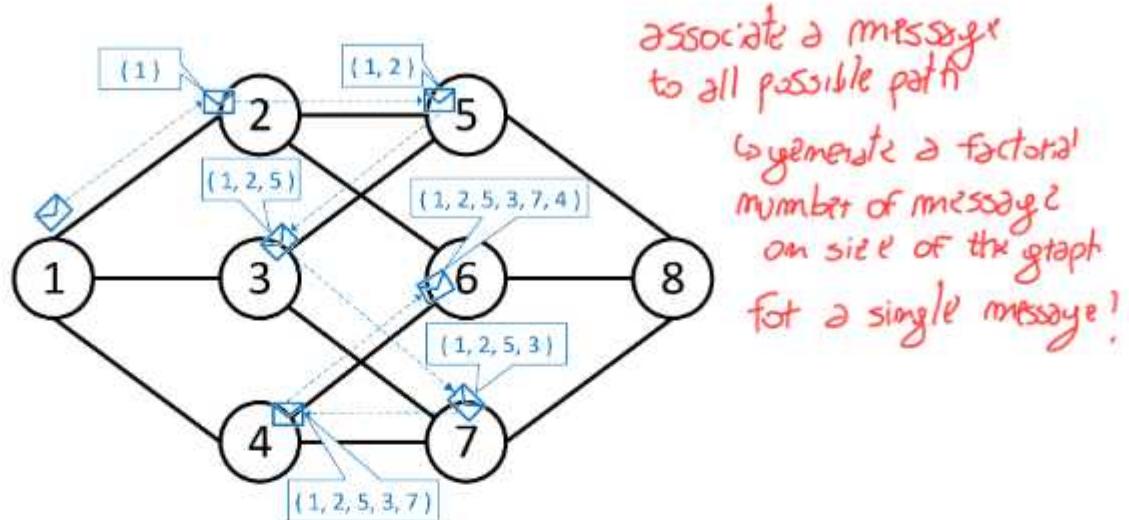
↳ lot gossip

# DolevU – Propagation

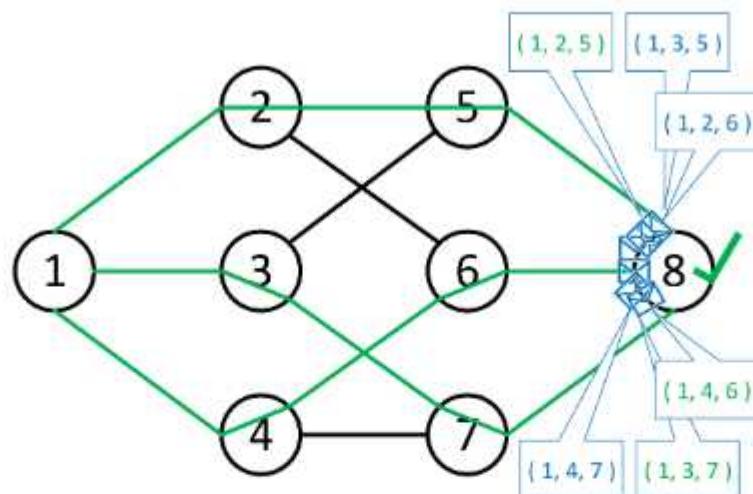


Note: only part of the messages exchanged are shown

# DolevU - Issue



# DolevU - Delivery



$f+1$  mode disjoint path (in green)  
i am allowed to deliver

# Globally Bounded Fault Model – Byzantine Faults, Autenticated Links

```
1: upon DolevU_send( $c$ ) do
2:   for  $j \in \Gamma(i)$  do
3:     | send( $\langle i, *, c, \emptyset \rangle, j$ )
4: upon receive( $\langle s, *, c, path \rangle, j$ ) do
5:   |  $path \leftarrow path \cup \{j\}$ 
6:   |  $Paths_{(s,c)} \leftarrow Paths_{(s,c)} \cup \{path\}$ 
7:   for  $j \in \Gamma(i)$  do
8:     | if  $j \notin path$  then
9:       |   | send( $\langle s, c, path \rangle, j$ )
10: upon max_disjoint_paths( $Paths_{(s,c)}$ )  $> f$  do
11:   | DolevU_deliver( $\langle s, c \rangle$ )
```

message complexity:  $O(n!)$

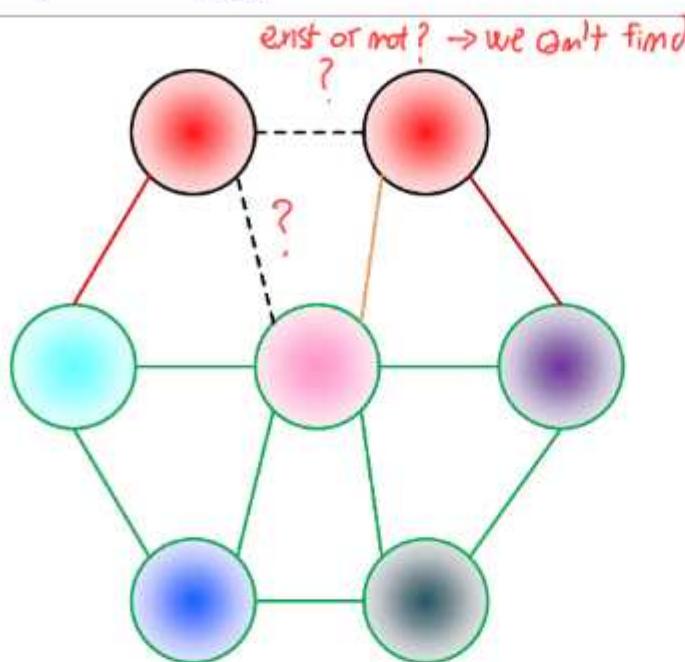
delivery complexity: NP Complete problem!

Correctness: node-connectivity  $> 2f$

Shit

↗ factorial, too much

# (EXTRA) Byzantine Tolerant Topology Reconstruction



It is possible to define a RC protocol with optimal message complexity and delivery complexity if stronger assumptions (about the links or about the node-connectivity of the network) are assumed

every process want to find topology of system and use the first protocol, but given that we have byzantine Faulty processes, we can only discover a partial topology of the system.

# Locally Bounded Fault Model – Byzantine Faults, Autenticated Links

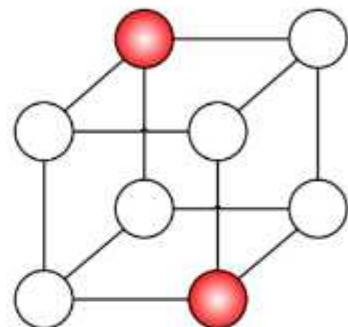
System model:

- at most  $f$  processes can be faulty in the neighborhood of every node,

Byzantine faults

-  $n$  processes

- authenticated perfect point-to-point links



1-local distribution

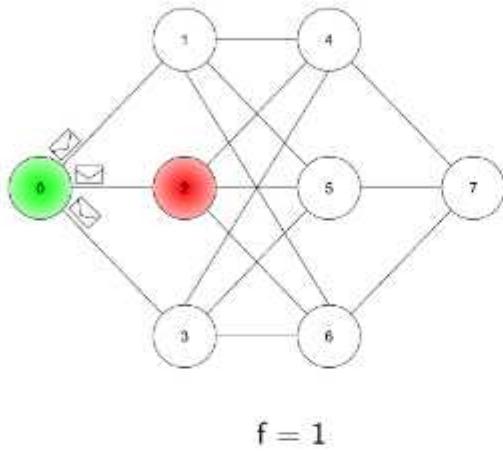
2-global distribution

Question: can you solve RC on that topology?

# Locally Bounded Fault Model – Byzantine Faults, Autenticated Links

## Certified Propagation Algorithm (CPA)

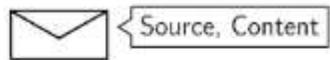
more simple  
protocol



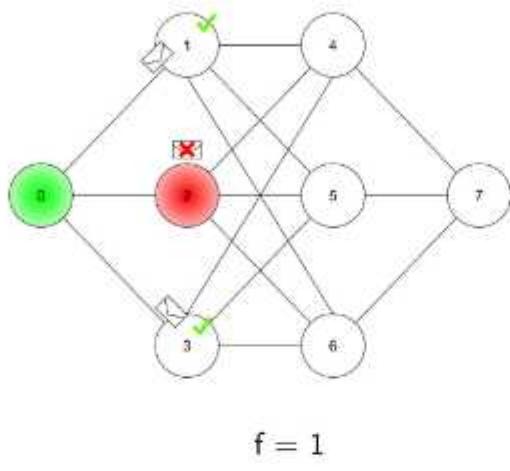
- ▶ the source broadcasts the message;  $s$
- ▶ a neighbor of the source directly accepts and relays the message;
- ▶ a process that receives the same message from  $f + 1$  distinct neighbors accepts and relays the message.

# Locally Bounded Fault Model – Byzantine Faults, Autenticated Links

## Certified Propagation Algorithm (CPA)



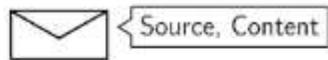
Message Format



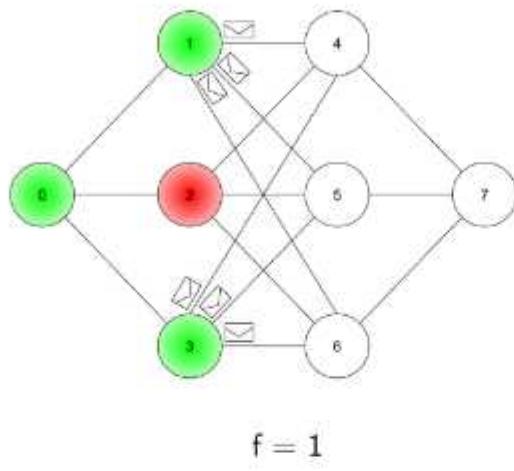
- ▶ the source broadcasts the message;
- ▶ a **neighbor of the source directly accepts and relays the message**;
- ▶ a process that receives the same message from  $f + 1$  distinct neighbors accepts and relays the message.

# Locally Bounded Fault Model – Byzantine Faults, Autenticated Links

## Certified Propagation Algorithm (CPA)



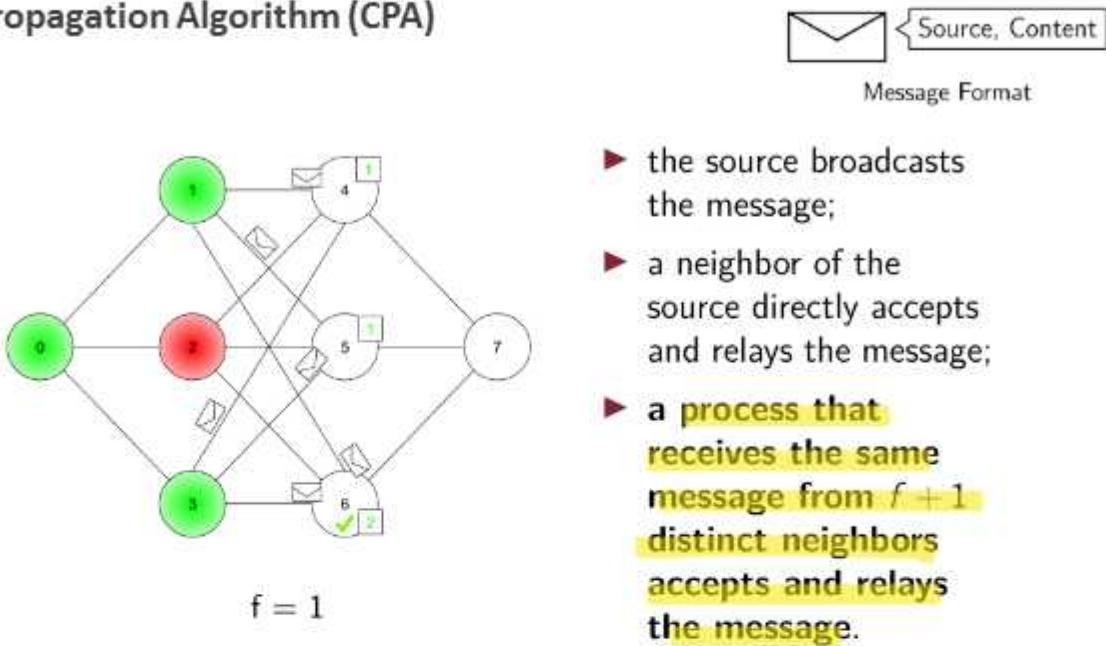
Message Format



- ▶ the source broadcasts the message;
- ▶ a **neighbor of the source directly accepts and relays the message**;
- ▶ a process that receives the same message from  $f + 1$  distinct neighbors accepts and relays the message.

# Locally Bounded Fault Model – Byzantine Faults, Autenticated Links

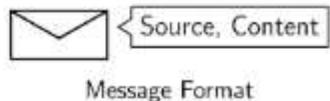
## Certified Propagation Algorithm (CPA)



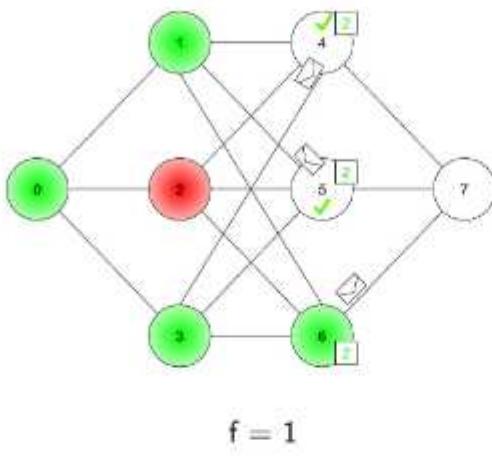
- ▶ the source broadcasts the message;
  - ▶ a neighbor of the source directly accepts and relays the message;
  - ▶ a process that receives the same message from  $f + 1$  distinct neighbors accepts and relays the message.

# Locally Bounded Fault Model – Byzantine Faults, Autenticated Links

## Certified Propagation Algorithm (CPA)



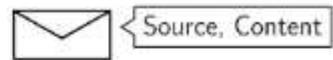
Message Format



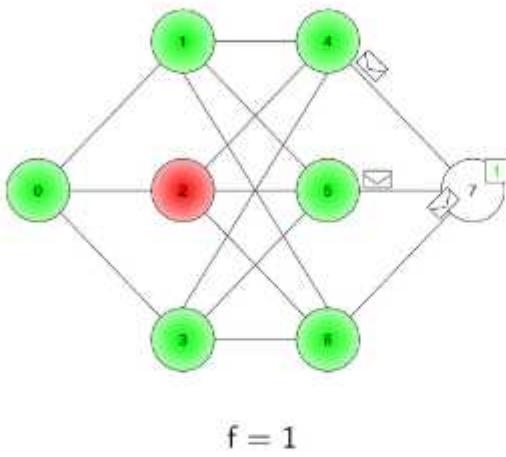
- ▶ the source broadcasts the message;
- ▶ a neighbor of the source directly accepts and relays the message;
- ▶ **a process that receives the same message from  $f + 1$  distinct neighbors accepts and relays the message.**

# Locally Bounded Fault Model – Byzantine Faults, Autenticated Links

## Certified Propagation Algorithm (CPA)



Message Format



- ▶ the source broadcasts the message;
- ▶ a neighbor of the source directly accepts and relays the message;
- ▶ a process that receives the same message from  $f + 1$  distinct neighbors accepts and relays the message.

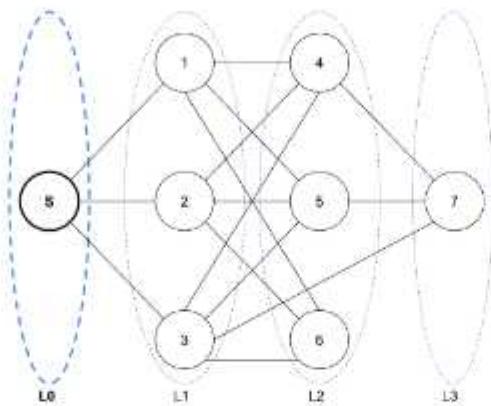
Now a  
quadratic  
number of  
messages?

# Locally Bounded Fault Model – Byzantine Faults, Autenticated Links

CPA Correctness (from a specific source) – Minimum k-level ordering (MKLO)

MKLO = Partition of the nodes in levels

► The source is placed in  $L_0$ ;

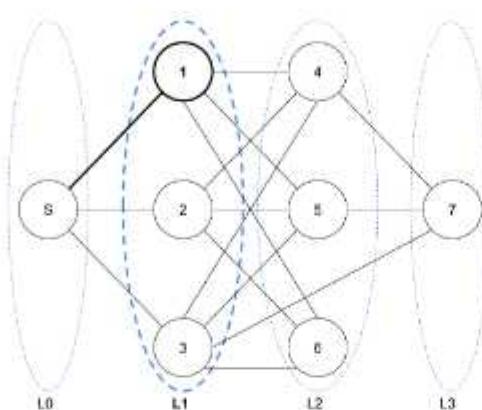


$k = 3$

# Locally Bounded Fault Model – Byzantine Faults, Autenticated Links

CPA Correctness (from a specific source) – Minimum k-level ordering (MKLO)

MKLO = Partition of the nodes in levels



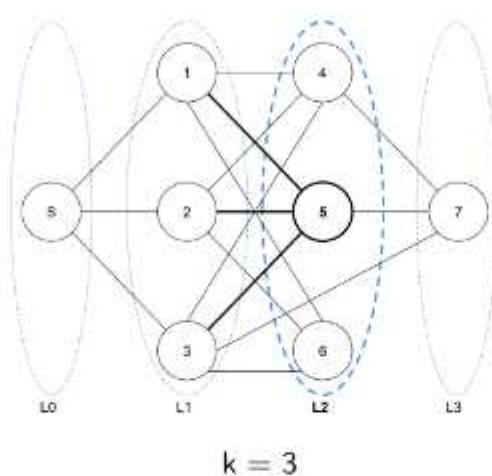
- ▶ The source is placed in  $L_0$ ;
- ▶ **The neighbors of the source are placed in level  $L_1$** ;

$k = 3$

# Locally Bounded Fault Model – Byzantine Faults, Autenticated Links

CPA Correctness (from a specific source) – Minimum k-level ordering (MKLO)

MKLO = Partition of the nodes in levels

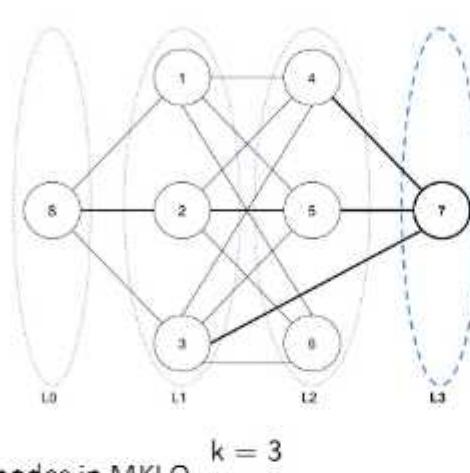


- ▶ The source is placed in  $L_0$ ;
- ▶ The neighbors of the source are placed in level  $L_1$ ;
- ▶ **Any other node is placed in the first level such that it has at least  $k$  neighbors in the previous levels.**

# Locally Bounded Fault Model – Byzantine Faults, Autenticated Links

CPA Correctness (from a specific source) – Minimum k-level ordering (MKLO)

MKLO = Partition of the nodes in levels



- ▶ The source is placed in  $L0$ ;
- ▶ The neighbors of the source are placed in level  $L1$ ;
- ▶ **Any other node is placed in the first level such that it has at least  $k$  neighbors in the previous levels.**

NOTE: you must include all nodes in MKLO

# Locally Bounded Fault Model – Byzantine Faults, Autenticated Links

---

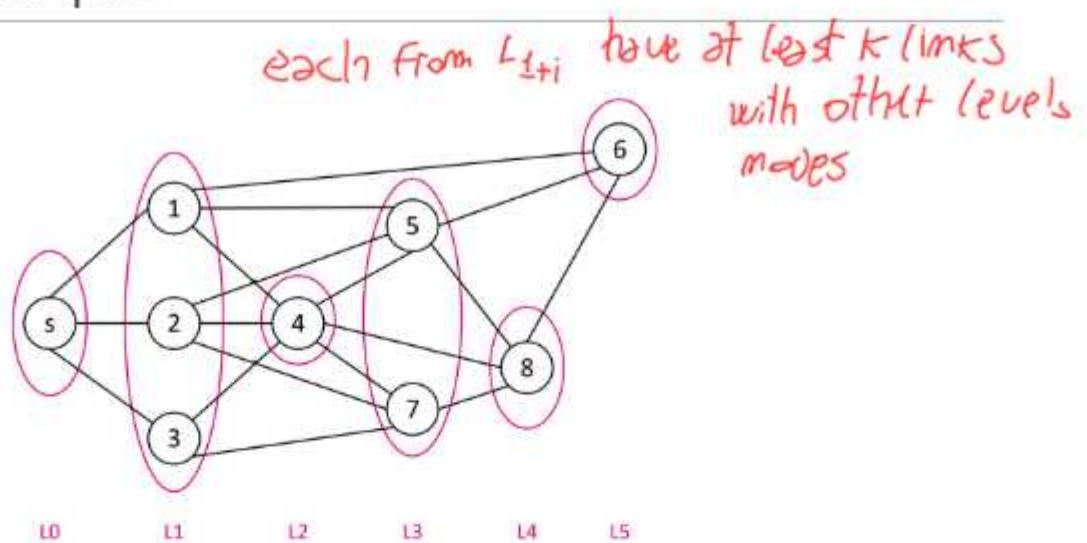
CPA Correctness (from a specific source) – Minimum k-level ordering (MKLO)

*Necessary condition: MKLO with  $k = f+1$*

*Sufficient condition: MKLO with  $k = 2f+1$*

*Strict condition: MKLO with  $k = f+1$  removing any possible placement of the Byzantine processes (NP-Complete Problem)*

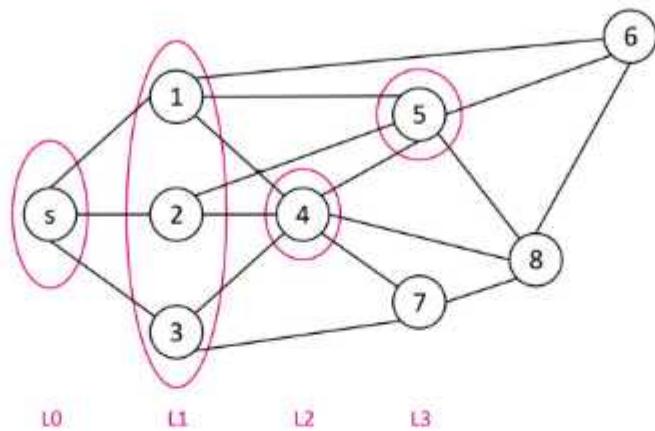
## MKLO Example



MKLO with  $K=3$

# MKLO Example

---

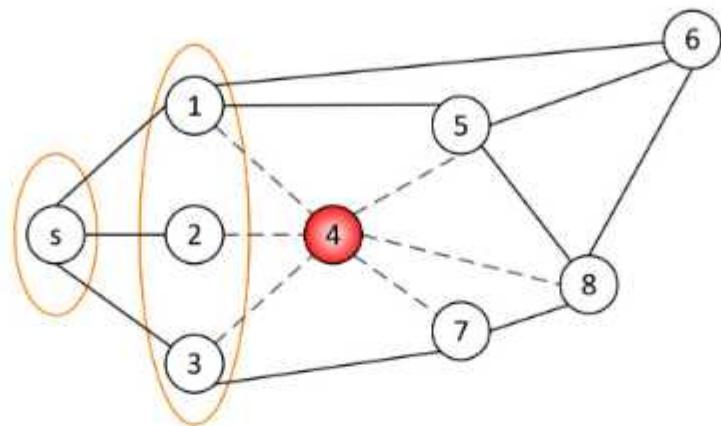


MKLO with  $K=3$  is not defined

MKLO with  $K=2$  is defined for every 1-local removal

## MKLO Example

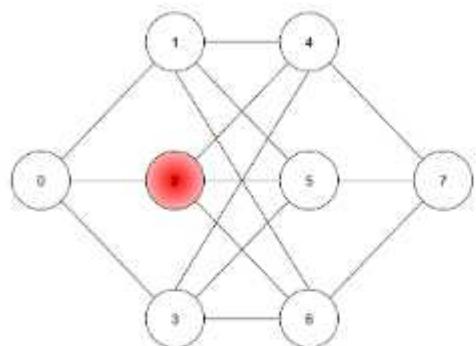
---



MKLO with  $K=2$  is not defined removing node 4

# CPA Recap

---



CPA:  $O(n^2)$  messages, delivery  $O(f)$

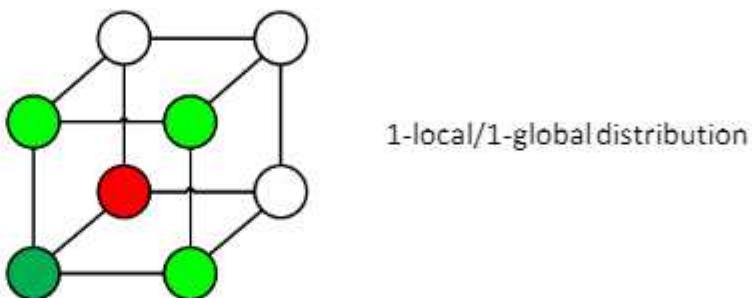
Exact Solvability: NP-Complete

# Globally vs Locally Bounded

**f-global distribution => f-local distribution (for the same value of f)**

The vice-versa is not true (1-local may imply >1-global)

\_> If the topology is unknown, you may attempt to use CPA to solve RC in the globally bounded failure model, but a «more dense» topology is required



# BRB in General Networks

**Question:** it is possible for a faulty source using an ByzRC primitive to send different contents to distinct processes?

System model:

- n processes
- at most f Byzantine faulty processes
- authenticated links
- not fully connected topology, general topology

How you can solve the problem?

What are the correctness conditions?

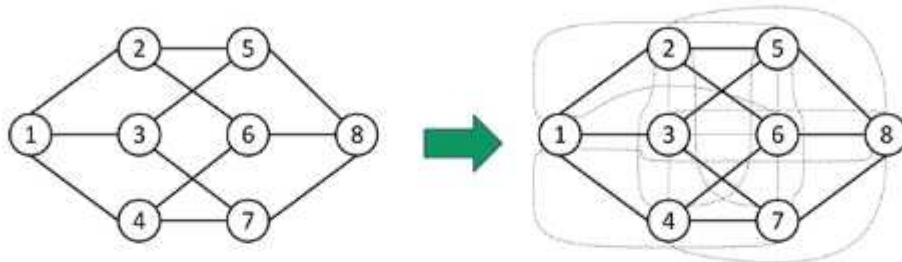
is possible to solve BRB  
in this situation?

yes it is possible

# Why reliable communication is useful?

Simulate a **complete communication network**

\_> use all the solutions defined for fully-connected distributed systems



**Communication Network**  
(made of P2P-link)

**Overlay Network**

# References

---

- Danny Dolev. *Unanimity in an unknown and unreliable environment* <https://doi.org/10.1109/SFCS.1981.53>
- Andrzej Pelc and David Peleg. *Broadcasting with locally bounded byzantine faults* <https://doi.org/10.1016/j.ipl.2004.10.007>
- Chris Litsas, Aris Pagourtzis, and Dimitris Sakavalas. *A graph parameter that matches the resilience of the certified propagation algorithm* [https://doi.org/10.1007/978-3-642-39247-4\\_23](https://doi.org/10.1007/978-3-642-39247-4_23).
- Giovanni Farina. *Tractable Reliable Communication in Compromised Networks* <https://tel.archives-ouvertes.fr/tel-03118108>

# Dependable Distributed Systems

## Master of Science in Engineering in Computer Science

AA 2021/2022

---

LECTURE 21: REGISTERS IN PRESENCE OF BYZANTINE  
PROCESSES

→ weaker validity respect to regular.  
concurrent processes can return any value in the domain

# Safe Register Specification

---

**Module 4.5:** Interface and properties of a  $(1, N)$  Byzantine safe register

---

**Module:**

**Name:**  $(1, N)$ -ByzantineSafeRegister, **instance**  $bonsr$ , with writer  $w$ .

**Events:**

**Request:**  $\langle bonsr, \text{Read} \rangle$ : Invokes a read operation on the register.

**Request:**  $\langle bonsr, \text{Write} \mid v \rangle$ : Invokes a write operation with value  $v$  on the register.  
Executed only by process  $w$ .

**Indication:**  $\langle bonsr, \text{ReadReturn} \mid v \rangle$ : Completes a read operation on the register  
with return value  $v$ .

**Indication:**  $\langle bonsr, \text{WriteReturn} \rangle$ : Completes a write operation on the register.  
Occurs only at process  $w$ .

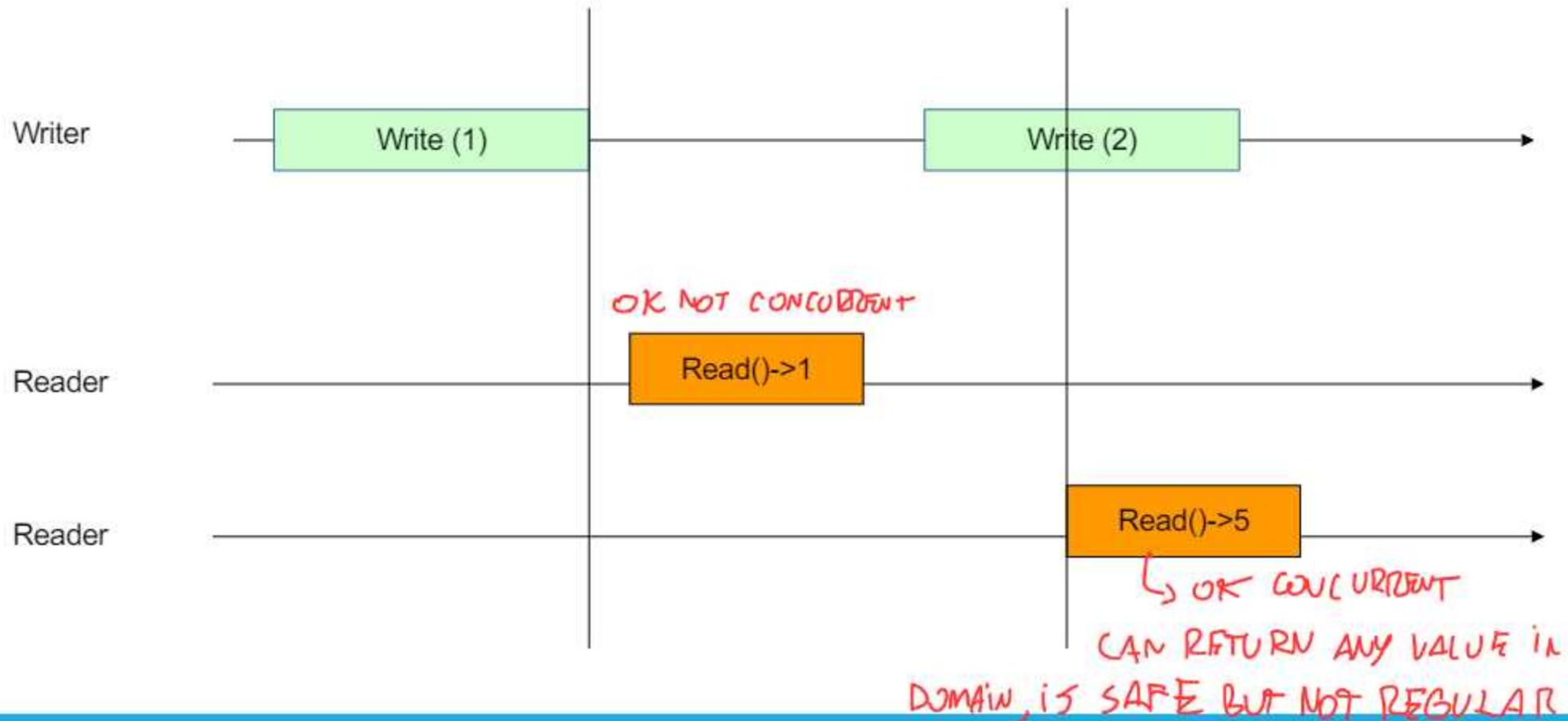
**Properties:**

**BONS1: Termination:** If a correct process invokes an operation, then the operation  
eventually completes.

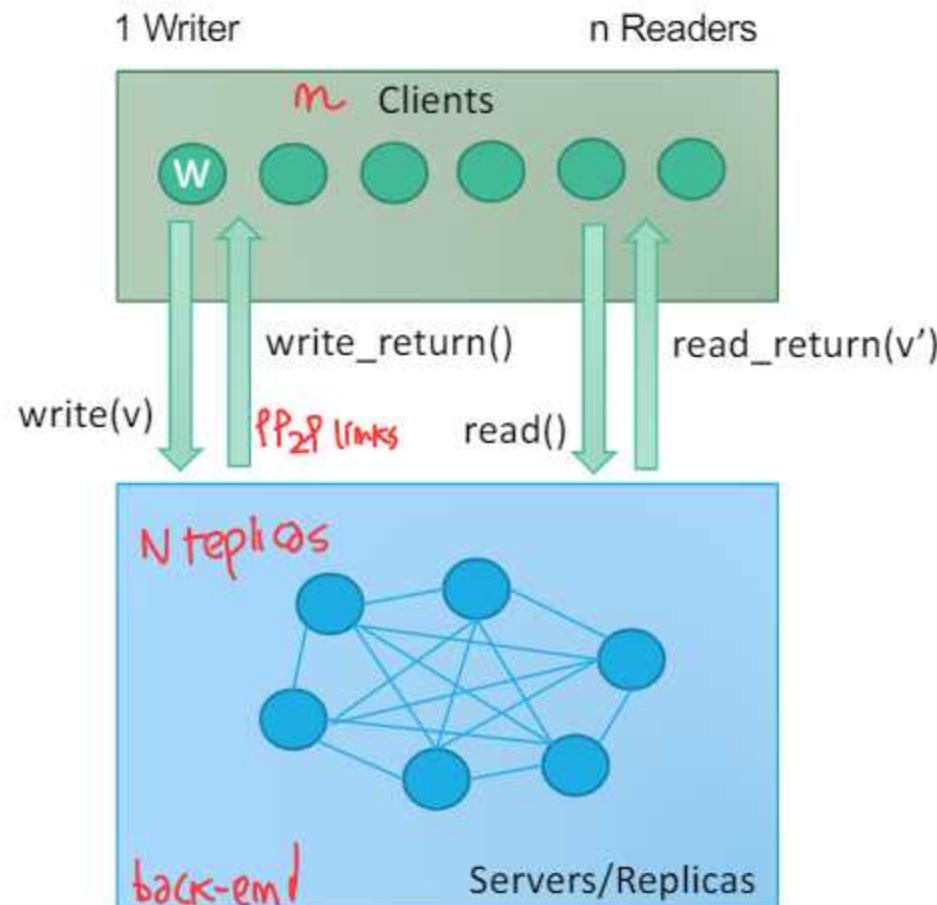
**BONS2: Validity:** A read that is not concurrent with a write returns the last value  
written.

→ concurrent ANY VALUES IN DOMAIN

# Safe Register



## Byzantine Tolerant Safe Register (1,n)



- Client-Server paradigm *→ client not fake info, can not be byzantine*
  - Asynchronous System
  - authenticated perfect point-to-point link
  - $f$  servers may be Byzantine
  - any client may crash but they cannot be Byzantine

n number of readers on client side

N number of replicas on back-end

↳ S2HVB+S

Do not confuse  $n$  with  $N$

# Safe Register Intuition

---

We have to assure that once writer returns from a write operation, then any following read operation returns the last written value.

- **Write operation:** sends  $\langle v, wts \rangle$  to servers and waits for ACK messages.

How many ACK messages?

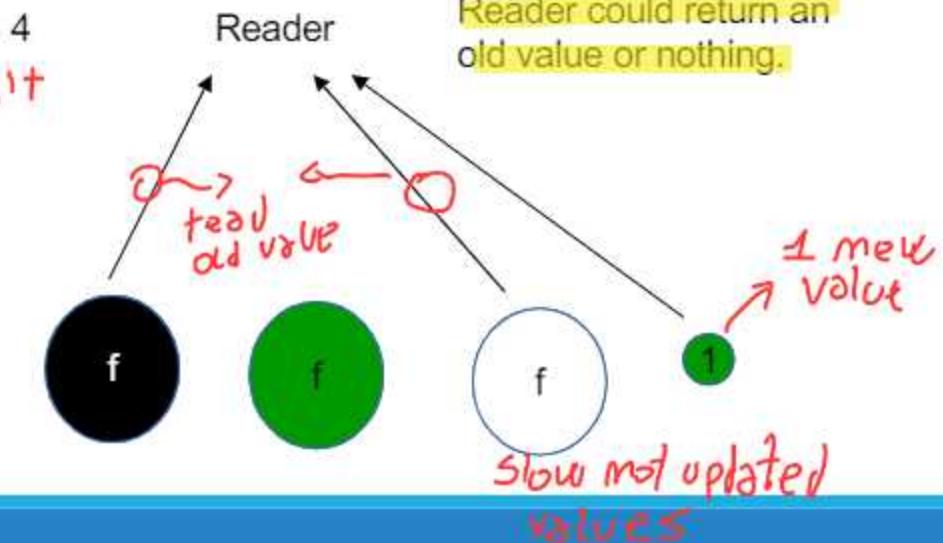
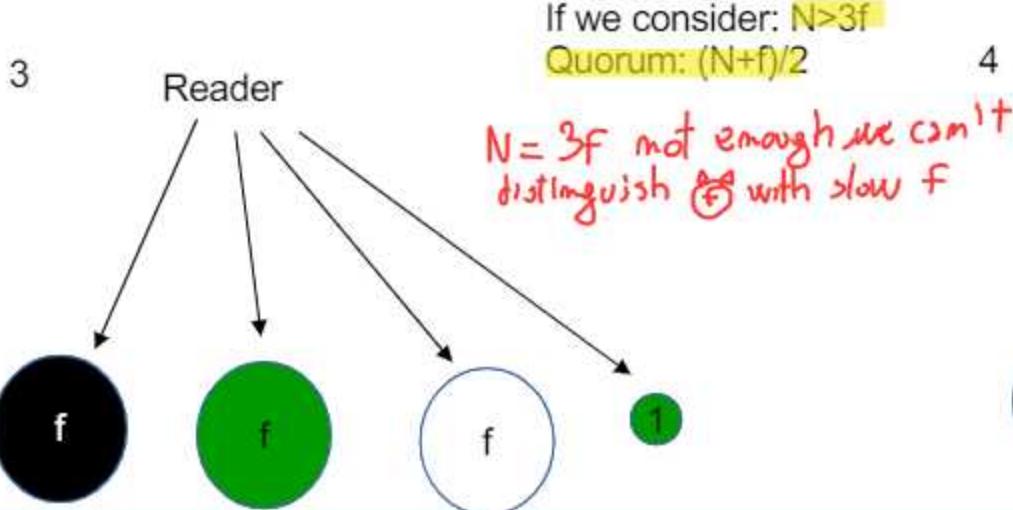
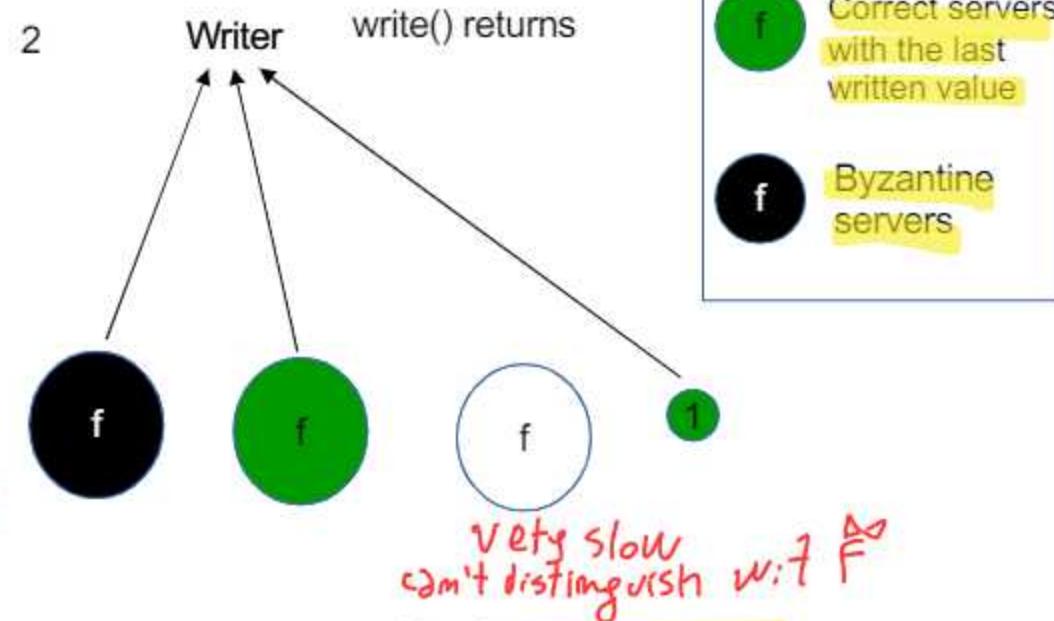
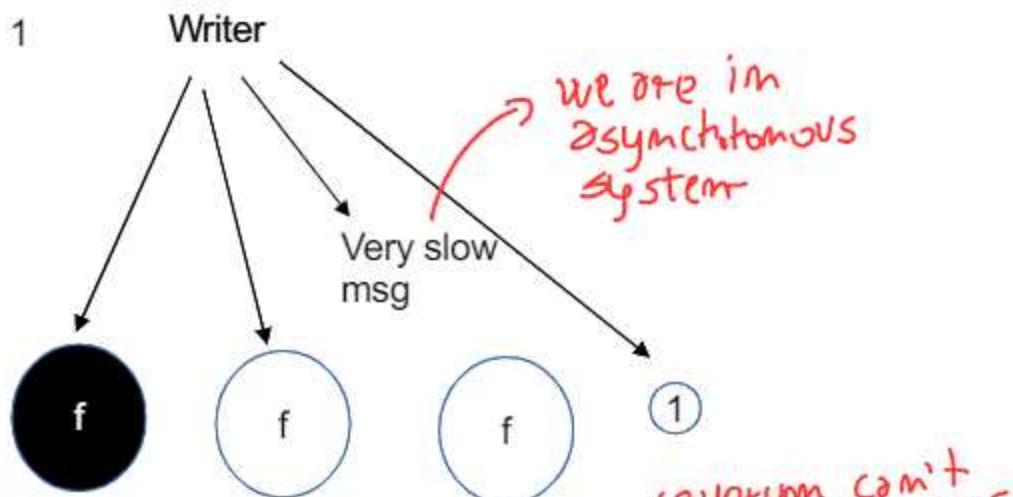
Enough to be sure that enough correct servers deliver  $\langle v, wts \rangle$

- **Read operation:** sends a read request and waits for reply messages.

How many reply messages?

Enough to be able to read newest value, not an old value or never written

# How large should the quorum be?



# Masking Quorum

*larger quorum with more replicas*

The kind of quorum working for Byzantine Broadcast here is not enough.

Safe Register has a stronger semantic with respect to broadcast. We require that a write operation is visible to all once it terminates.

To implement safe registers we use **Masking Quorums**:

$N > 4f$

*number of replicas four times the number of failures*

**Quorum:  $(N+2f)/2$  (i.e.,  $3f+1$ )**

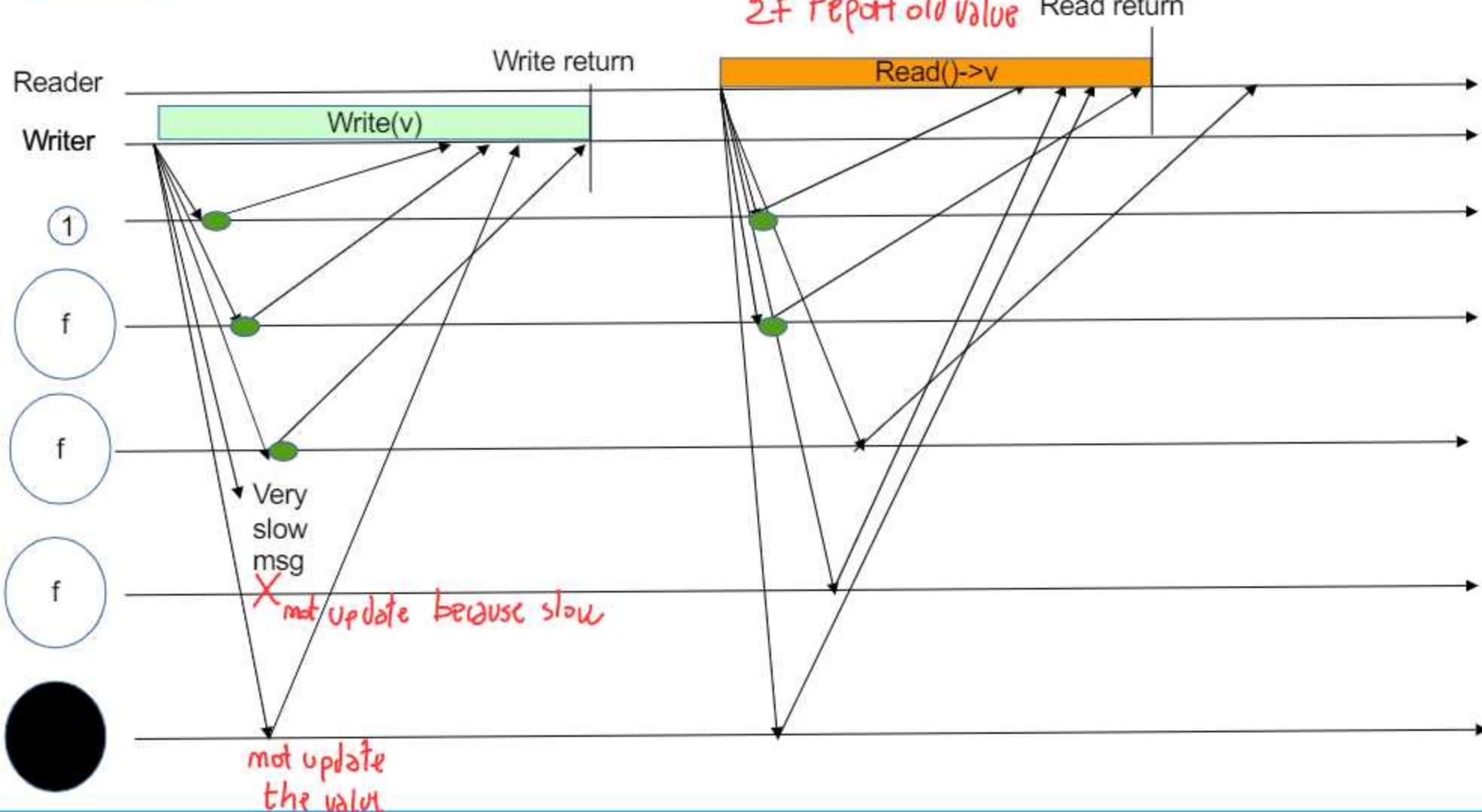
3f  $\rightarrow$  f fake, f old, f+1 correct

N > 4f

Quorum: (N+2f)/2

f+1 processes report new value  $\Rightarrow$  OK

2f report old value



**Algorithm 4.14: Byzantine Masking Quorum****Implements:** $(1, N)$ -ByzantineSafeRegister, **instance**  $bonsr$ , with writer  $w$ .**Uses:****AuthPerfectPointToPointLinks**, **instance**  $al$ .*prevent spoofing attack*

```

upon event (  $bonsr$ , Init ) do
   $(ts, val) := (0, \perp)$ ;
   $wts := 0$ ;
   $acklist := [\perp]^N$ ;
   $rid := 0$ ;
   $readlist := [\perp]^N$ ;

upon event (  $bonsr$ , Write |  $v$  ) do // only process  $w$ 
   $wts := wts + 1$ ;
   $acklist := [\perp]^N$ ;
  forall  $q \in \Pi$  do
    trigger (  $al$ , Send |  $q$ , [WRITE,  $wts, v$ ] );
    Propagate value to all servers

upon event (  $al$ , Deliver |  $p$ , [WRITE,  $ts', v'$ ] ) such that  $p = w$  do
  if  $ts' > ts$  then
     $(ts, val) := (ts', v')$ ;
    trigger (  $al$ , Send |  $p$ , [ACK,  $ts'$ ] );
    Get a write

```

*same data we saw  
in asynchronous case**we have that f+1 are correct*

**byzhighestval** (  $\cdot$  ): selects the value from the pair that occurs more than  $f$  time and with the highest timestamp. If no pair exists, the reader selects a default value  $v_0$  from the domain of the register.

**Assumption** $N > 4f$ **upon event** (  $al$ , Deliver |  $q$ , [ACK,  $ts'$ ] ) **such that**  $ts' = wts$  **do**

1  $acklist[q] := \text{ACK}$ ;  
**if**  $\#(acklist) > (N + 2f)/2$  **then** *3F+1 acknowledgement*  
 $acklist := [\perp]^N$ ;  
 trigger (  $bonsr$ , WriteReturn );

**upon event** (  $bonsr$ , Read ) **do**

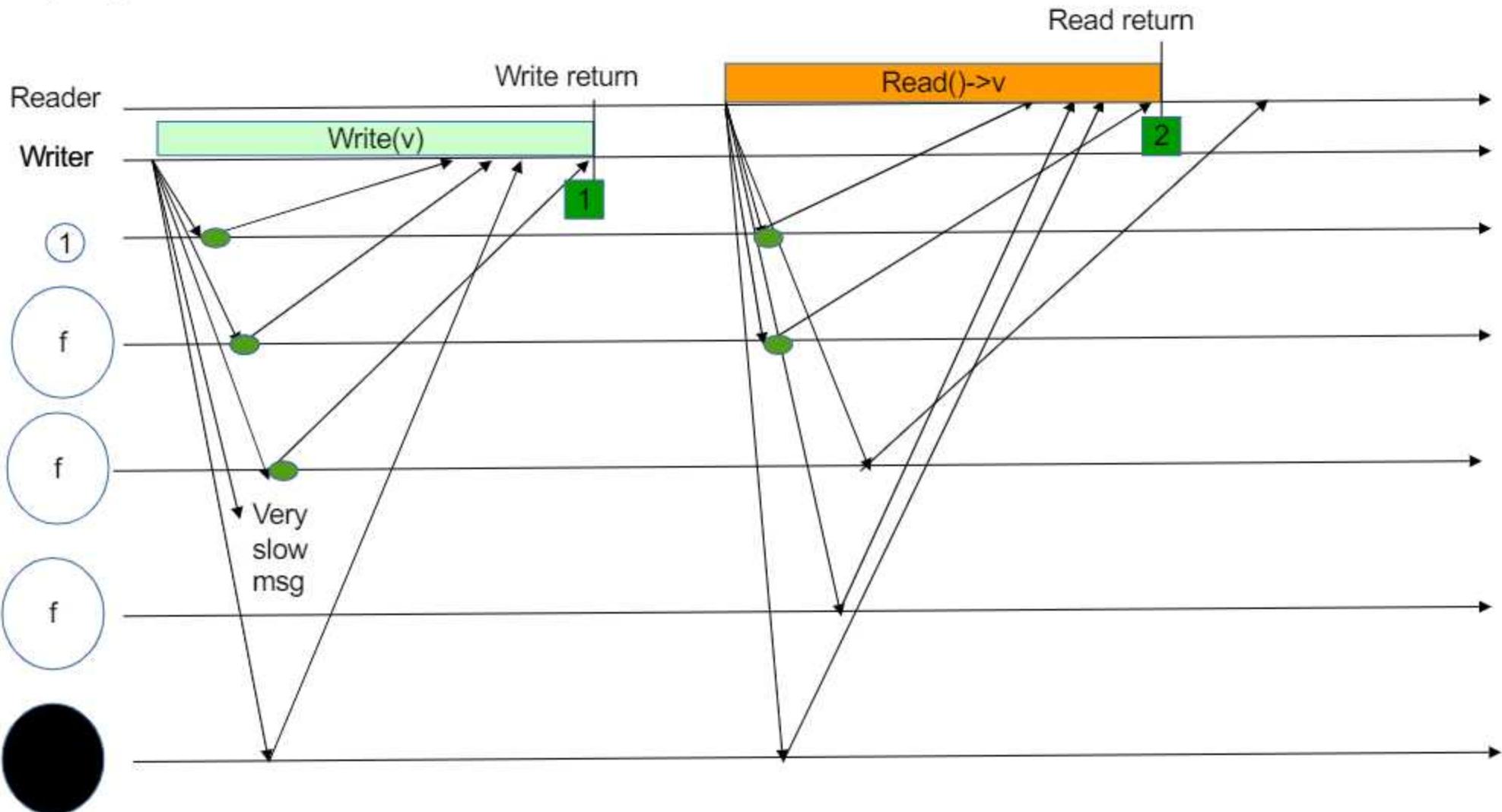
$rid := rid + 1$ ;  
 $readlist := [\perp]^N$ ;  
**forall**  $q \in \Pi$  **do**  
 trigger (  $al$ , Send |  $q$ , [READ,  $rid$ ] );
 *to everybody*

**upon event** (  $al$ , Deliver |  $p$ , [READ,  $r$ ] ) **do**  
 trigger (  $al$ , Send |  $p$ , [VALUE,  $r, ts, val$ ] );**upon event** (  $al$ , Deliver |  $q$ , [VALUE,  $r, ts', v'$ ] ) **such that**  $r = rid$  **do**

2  $readlist[q] := (ts', v')$ ;  
**if**  $\#(readlist) > \frac{N+2f}{2}$  **then**  
 $v := \text{byzhighestval}(readlist)$ ;  
 $readlist := [\perp]^N$ ;  
 trigger (  $bonsr$ , ReadReturn |  $v$  );
 *store newer value and that occurred more time*

$N > 4f$

Quorum:  $(N+2f)/2$



# Regular Registers

---

The specification does not change

---

We will discuss two implementations:

1. Using cryptography
2. Without cryptography

SET VETS save signatures, these reduce  
number of set vets needed

## Regular Register

### Implementation with cryptographic assumptions

BASIC IDEA -> evolution of Majority voting Algorithm

- the writer signs the timestamp/value pair
- Processes store it together with the signature
- The reader verifies the signature on each timestamp/value pair received in a VALUE message and ignores those with invalid signatures



A Byzantine process is prevented from returning an arbitrary timestamp value in the VALUE message, although it may include a signed value with an outdated timestamp

# Regular Register Implementation with cryptographic assumptions

## Algorithm 4.15: Authenticated-Data Byzantine Quorum

### Implements:

$(1, N)$ -ByzantineRegularRegister, instance  $bonrr$ , with writer  $w$ .

### Uses:

AuthPerfectPointToPointLinks, instance  $al$ .

```

upon event ( bonrr, Init ) do
   $(ts, val, \sigma) := (0, \perp, \perp)$ ;
   $wts := 0$ ; ↳ store also signature
   $acklist := [\perp]^N$ ;
   $rid := 0$ ;
   $readlist := [\perp]^N$ ;
  // only process w

upon event ( bonrr, Write |  $v$  ) do
   $wts := wts + 1$ ;
   $acklist := [\perp]^N$ ;
   $\sigma := \text{sign}(\text{self}, bonrr \parallel \text{self} \parallel \text{WRITE} \parallel wts \parallel v)$ ;
  forall  $q \in \Pi$  do
    trigger (  $al$ , Send |  $q$ , [WRITE,  $wts$ ,  $\sigma$ ] );
  if  $ts' > ts$  then
     $(ts, val, \sigma) := (ts', v', \sigma')$ ;
  trigger (  $al$ , Send |  $p$ , [ACK,  $ts'$ ] );

upon event (  $al$ , Deliver |  $p$ , [WRITE,  $ts'$ ,  $v'$ ,  $\sigma'$ ] ) such that  $p = w$  do
  if  $ts' > ts$  then
     $(ts, val, \sigma) := (ts', v', \sigma')$ ;
  trigger (  $al$ , Send |  $p$ , [ACK,  $ts'$ ] );

upon event (  $al$ , Deliver |  $q$ , [ACK,  $ts'$ ] ) such that  $ts' = wts$  do
   $acklist[q] := \text{ACK}$ ;
  if #(acklist) > (N + f) / 2 then
     $acklist := [\perp]^N$ ;
  trigger (  $bonrr$ , WriteReturn );
  i.e. > 2f
  ↳ reduced number of signatures

```

## Assumption

$N > 3f$

*↳ now 3 not 4, because more server is not able to generate a proof for a false value we can trust also one f.*

```

upon event ( bonrr, Read ) do
   $rid := rid + 1$ ;
   $readlist := [\perp]^N$ ;
  forall  $q \in \Pi$  do
    trigger (  $al$ , Send |  $q$ , [READ,  $rid$ ] );

upon event (  $al$ , Deliver |  $p$ , [READ,  $r$ ] ) do
  trigger (  $al$ , Send |  $p$ , [VALUE,  $r$ ,  $ts$ ,  $val$ ,  $\sigma$ ] );

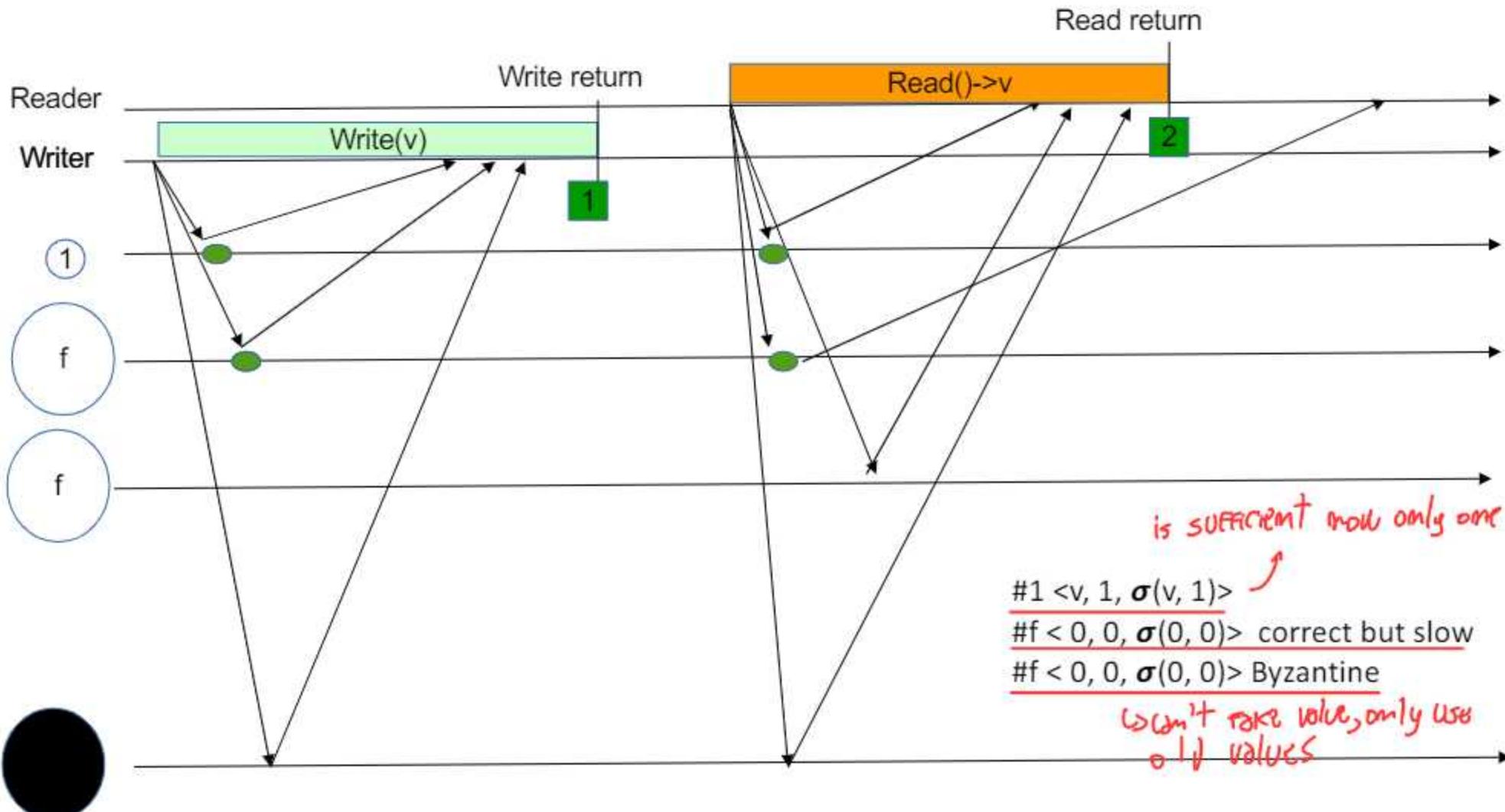
upon event (  $al$ , Deliver |  $q$ , [VALUE,  $r$ ,  $ts'$ ,  $v'$ ,  $\sigma'$ ] ) such that  $r = rid$  do
  if verifysig( $q$ ,  $bonrr \parallel w \parallel \text{WRITE} \parallel [ts' \parallel v', \sigma']$ ) then
     $readlist[q] := (ts', v')$ ;
    if #(readlist) >  $\frac{N+f}{2}$  then
       $v := \text{highestval}(readlist)$ ;
       $readlist := [\perp]^N$ ;
    trigger (  $bonrr$ , ReadReturn |  $v$  );
  i.e. > 2f

```

N>3f

Quorum:  $(N+2f)/2$

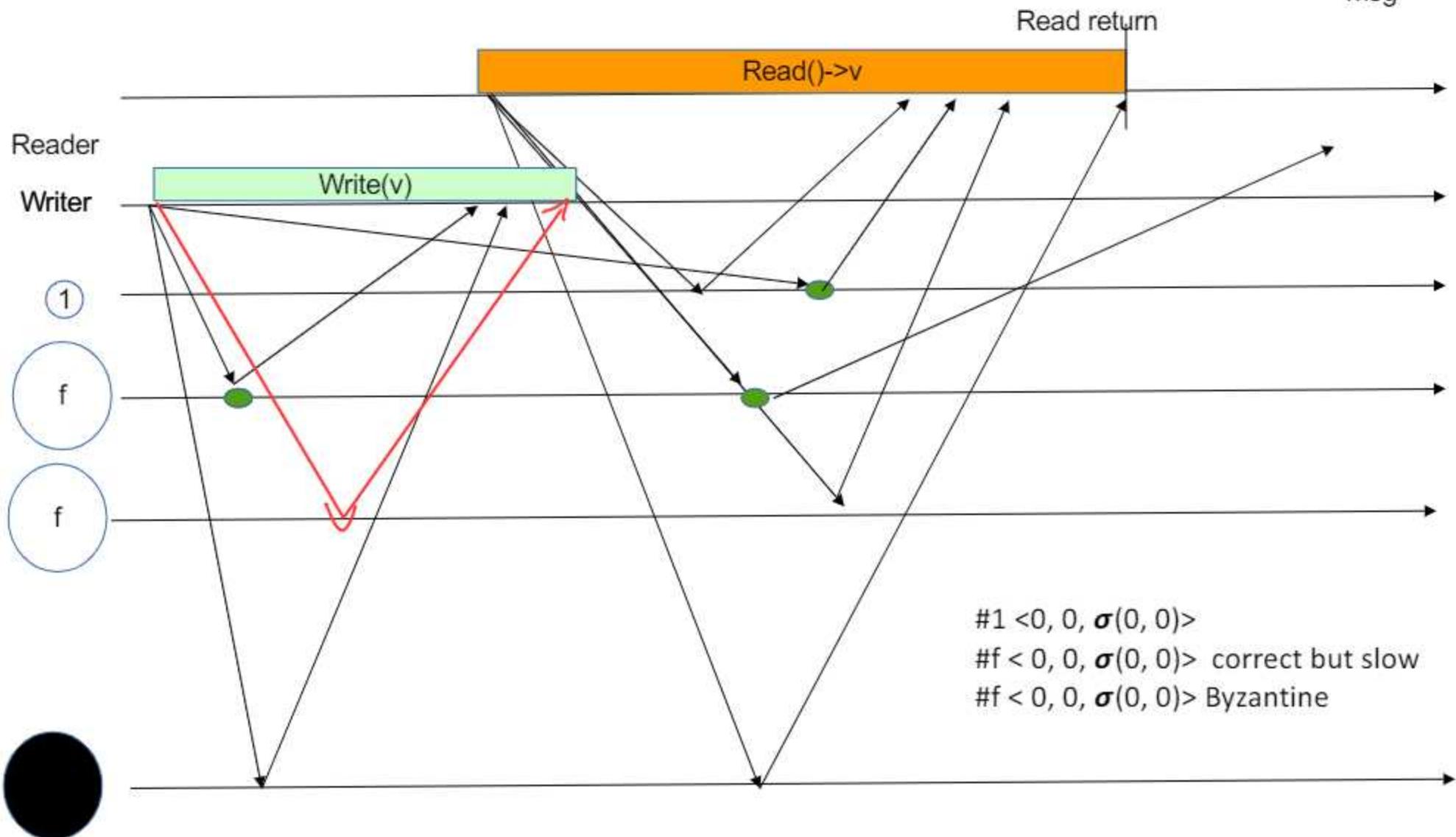
Very slow msg



N>3f

Quorum:  $(N+2f)/2$

Very slow msg



# Regular Register

## Implementation without cryptographic assumptions

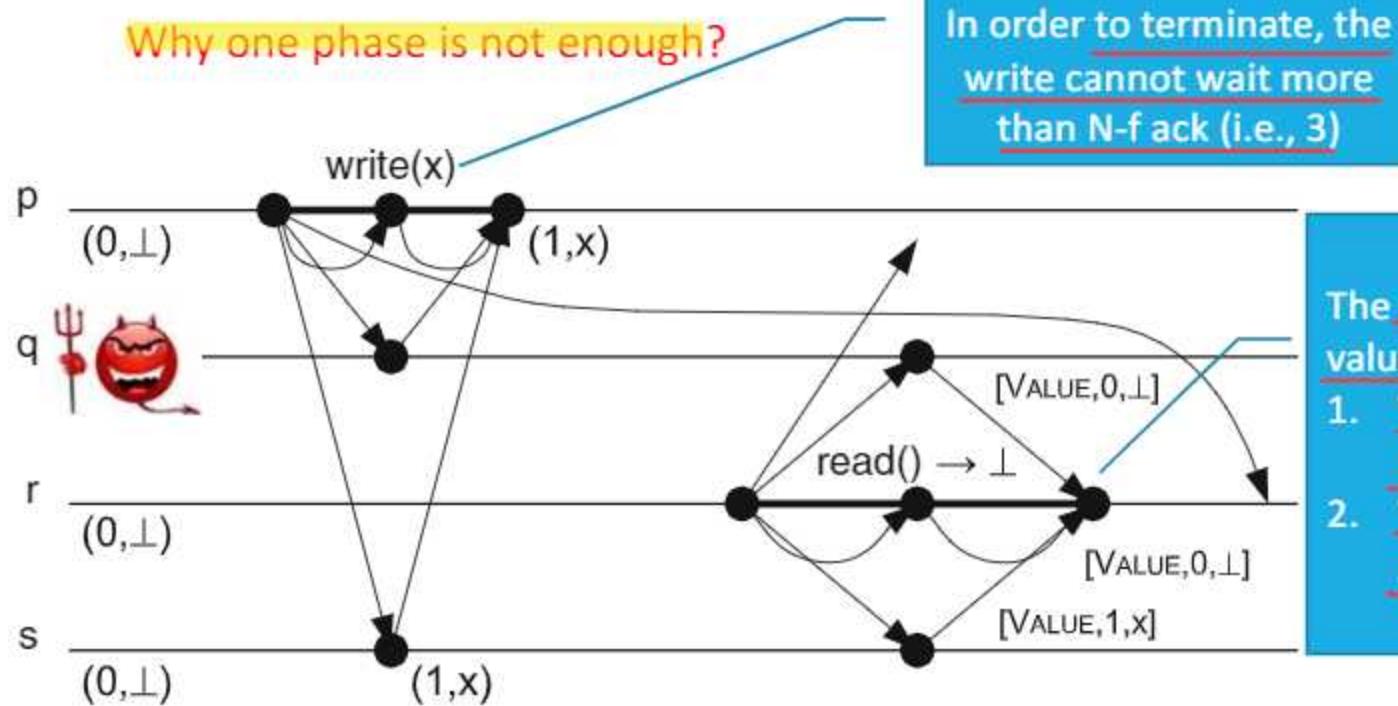
the writer process  $p$  uses two phases to write a new

- a pre-write phase and
- a write phase

Assumption

$$N > 3f$$

Why one phase is not enough?



The reader is not able to choose a value:

1. Value ⊥ could be old given that process s provided value x with ts 1
2. Value x could be a fake value generated by a Byzantine process

# Regular Register

## Implementation without cryptographic assumptions

---

the writer process p uses two phases to write a new

- a pre-write phase: the writer sends PREWRITE messages with the current timestamp/value pair.  
Then it waits until it receives PREACK messages from  $N - f$  processes
- a write phase: the writer sends ordinary WRITE messages, again containing the current time-  
stamp/value pair, and then waits until it receives ACK messages from  $N - f$  processes

Every process stores two timestamp/value pairs, one from the pre-write phase and one  
from the write phase

## Intuition

two pair store  
locally  
(0,1)

$$(0, \perp)$$

(0,  $\perp$ )

p

write (x)

r can infer that one of the three processes is faulty (the writer received  $N - f$  ack for the pre-write phase for  $(1, x)$  hence, either s sent the wrong written pair or q sent the wrong pre-written pair, as r itself is obviously correct).

q 

(0,  $\perp$ )

$$r = (0, \perp)$$

(0, 1)

(0, 1)

1, x)

$\theta, \perp$

1

1

(0, 1)

$$\sqrt{(0, \pm)}$$

1)

1)

(1, x)  
(0,  $\perp$ )

please write

(1, x)  
(1, x)

(1, x) ↳ write  
↳ consistent  
same value and  
timestamp?

# Regular Register

## write Implementation without cryptographic assumptions

Algorithm 4.16: Double-Write Byzantine Quorum (part 1, write)

Implements:

(1,  $N$ )-ByzantineRegularRegister, instance  $bonrr$ , with writer  $w$ .

Uses:

AuthPerfectPointToPointLinks, instance  $al$ .

```

upon event { bonrr, Init } do
   $(pts, pval) := (0, \perp);$ 
   $(ts, val) := (0, \perp);$ 
   $(wts, wval) := (0, \perp);$ 
   $preacklist := [\perp]^N;$ 
   $acklist := [\perp]^N;$ 
   $rid := 0;$ 
   $readlist := [\perp]^N;$ 

upon event { bonrr, Write |  $v$  } do
   $(wts, wval) := (wts + 1, v);$ 
   $preacklist := [\perp]^N;$ 
   $acklist := [\perp]^N;$ 
  forall  $q \in \Pi$  do
    trigger { al, Send |  $q$ , [PREWRITE,  $wts, wval$ ] };

upon event { al, Deliver |  $p$ , [PREWRITE,  $pts'$ ,  $pval'$ ] }
  such that  $p = w \wedge pts' = pts + 1$  do
     $(pts, pval) := (pts', pval');$ 
    trigger { al, Send |  $p$ , [PREACK,  $pts$ ] };

upon event { al, Deliver |  $q$ , [PREACK,  $pts'$ ] } such that  $pts' = wts$  do
   $preacklist[q] := PREACK;$ 
  if  $\#(preacklist) \geq N - f$  then
     $preacklist := [\perp]^N;$ 
  forall  $q \in \Pi$  do
    trigger { al, Send |  $q$ , [WRITE,  $wts, wval$ ] };
  
```

*→ p = real sender, consistent of sequence of write set of pts' = pts + 1*

*only on client side*

*↑*

*→ two pairs instead of one*

```

upon event { al, Deliver |  $p$ , [WRITE,  $ts'$ ,  $val'$ ] }
  such that  $p = w \wedge ts' = pts \wedge ts' > ts$  do
     $(ts, val) := (ts', val');$ 
    trigger { al, Send |  $p$ , [ACK,  $ts$ ] };

upon event { al, Deliver |  $q$ , [ACK,  $ts'$ ] } such that  $ts' = wts$  do
   $acklist[q] := ACK;$ 
  if  $\#(acklist) \geq N - f$  then
     $acklist := [\perp]^N;$ 
    trigger { bonrr, WriteReturn };
  
```

- $p = w$ : it's  $t$
- $ts' =$  PreACK phase
- $ts' = ts$  exactly in serom stay

# Regular Register

WORKS if not too many writes, termination is conditioned from frequency of writes

## read Implementation without cryptographic assumptions

### Algorithm 4.17: Double-Write Byzantine Quorum (part 2, read)

```
upon event < bonrr, Read > do
    rid := rid + 1;
    readlist := [⊥]N;
    forall q ∈ Π do
        trigger < al, Send | q, [READ, rid] >;
    both part of value written
```

```
upon event < al, Deliver | p, [READ, r] > do
    trigger < al, Send | p, [VALUE, r, pts, pval, ts, val] >;
```

```
upon event < al, Deliver | q, [VALUE, r, pts', pval', ts', val'] > such that r = rid do
    if pts' = ts' + 1 ∨ (pts', pval') = (ts', val') then
        readlist[q] := (pts', pval', ts', val');
    if exists (ts, v) in an entry of readlist such that authentic(ts, v, readlist) = TRUE
        and exists Q ⊆ readlist such that
            #(Q) >  $\frac{N+f}{2}$  ∧ selectedmax(ts, v, Q) = TRUE then
                readlist := [⊥]N;
                trigger < bonrr, ReadReturn | v >;
            else
                trigger < al, Send | q, [READ, r] >;
                Restart
```

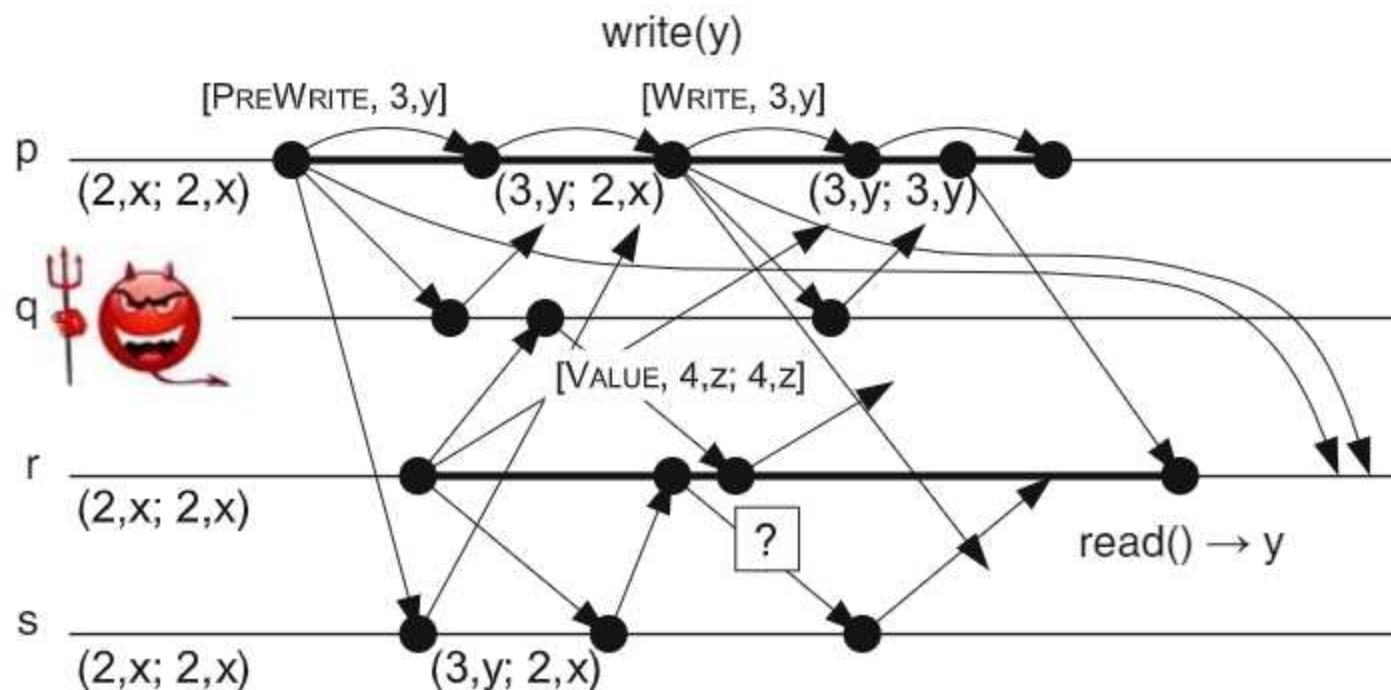
TRUE if *readlist* contains a pair (ts, v) that is found in the entries of more than f processes

*readlist* contains a pair (ts, v) such there is a Byzantine quorum (Q) of entries in *readlist* whose highest timestamp/value pair, selected among the pre-written or written pair of the entries, is (ts, v)

one pair authentic  
↓  
reported from  
at least f+1  
servers

# Example

---



# Regular Register without cryptographic assumptions: Observations

---

Termination property must be relaxed in to *finite-write termination*

- Instead of requiring that every operation of a correct process eventually terminates, a read operation that is concurrent with infinitely many write operations may not terminate

It has been shown that such a relaxation is necessary

# Exercise *- Exam question*

---

Does the Regular Register implementation without cryptographic assumption satisfy also the atomic specification?

# References

---

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

- Chapter 4 – Sections 4.6 and 4.7

Dependable Distributed Systems  
Master of Science in Engineering in  
Computer Science

AA 2022/2023

---

LECTURE 29: CONSENSUS IN PRESENCE OF BYZANTINE  
PROCESSES

# Byzantine Tolerant Consensus

---

Ideally, we would like to obtain the same properties we get in the crash prone environment

- **Termination**: Every correct process eventually decides some value
- **Validity**: If a process decides  $v$ , then  $v$  was proposed by some process
- **Integrity**: No process decides twice
- **Agreement**: No two correct processes decide differently

Consensus in Byzantine world is impossible

However...

- We cannot require anything from Byzantine
- The validity property must be adapted as Byzantine processes may invent values or claim to have proposed different values

Thus...

- We restrict the specification only to correct processes
- We define two different versions of validity (weak and strong)

# Weak Byzantine Consensus Specification

if there is only one Byzantine don't know what happens to consensus.

## Module 5.10: Interface and properties of weak Byzantine consensus



NOTE: Weak Validity allows to decide an arbitrary value if some process is Byzantine.

### Properties:

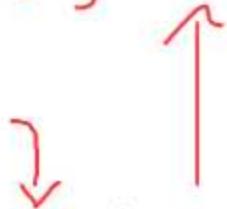
WBC1: *Termination*: Every correct process eventually decides some value.

WBC2: *Weak validity*: If all processes are correct and propose the same value  $v$ , then no correct process decides a value different from  $v$ ; furthermore, if all processes are correct and some process decides  $v$ , then  $v$  was proposed by some process.

WBC3: *Integrity*: No correct process decides twice.

WBC4: *Agreement*: No two correct processes decide differently.

value could not come from correct but by a Byzantine



you can decide on arbitrary value, but with agreement

→ change set of permissible value, here can not decide an arbitrary value, only one from correct set  $\square$   
default

# Strong Byzantine Consensus

---

**Module 5.11:** Interface and properties of (strong) Byzantine consensus

---

**Module:**

**Name:** ByzantineConsensus, instance  $bc$ .

**Events:**

**Request:**  $\langle bc, \text{Propose} \mid v \rangle$ : Proposes value  $v$  for consensus.

**Indication:**  $\langle bc, \text{Decide} \mid v \rangle$ : Outputs a decided value  $v$  of consensus.

**Properties:**

**BC1 and BC3-BC4:** Same as properties WBC1 and WBC3-WBC4 in weak Byzantine consensus (Module 5.10).

**BC2: Strong validity:** If all correct processes propose the same value  $v$ , then no correct process decides a value different from  $v$ ; otherwise, a correct process may only decide a value that was proposed by some correct process or the special value  $\square$ .

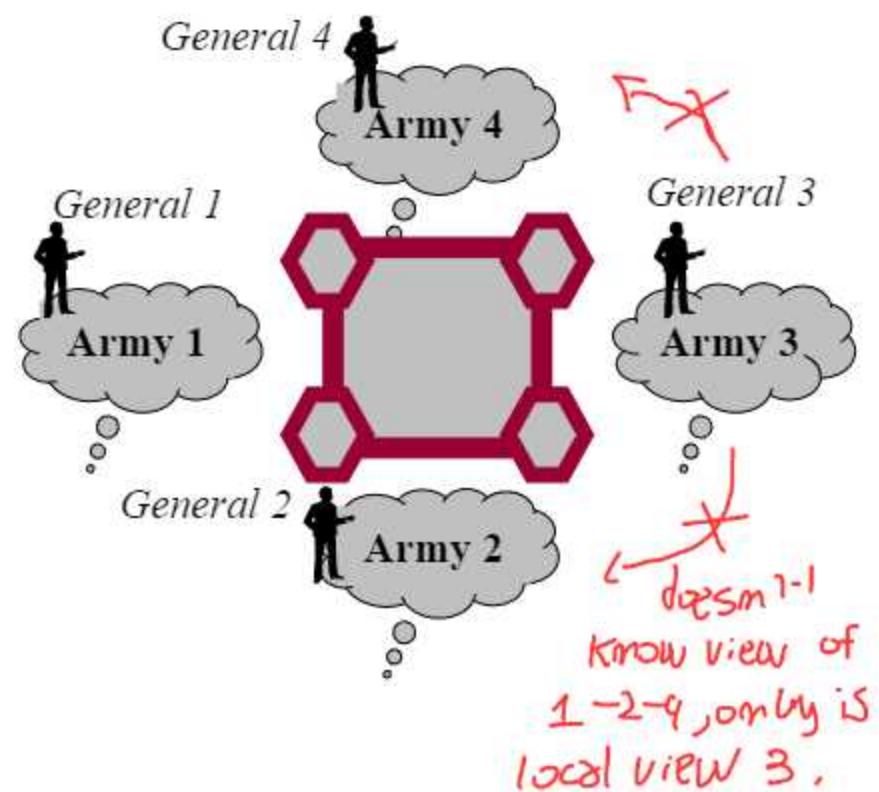


# The Byzantine Generals Problem

Abstract the consensus problem in presence of Byzantine processes

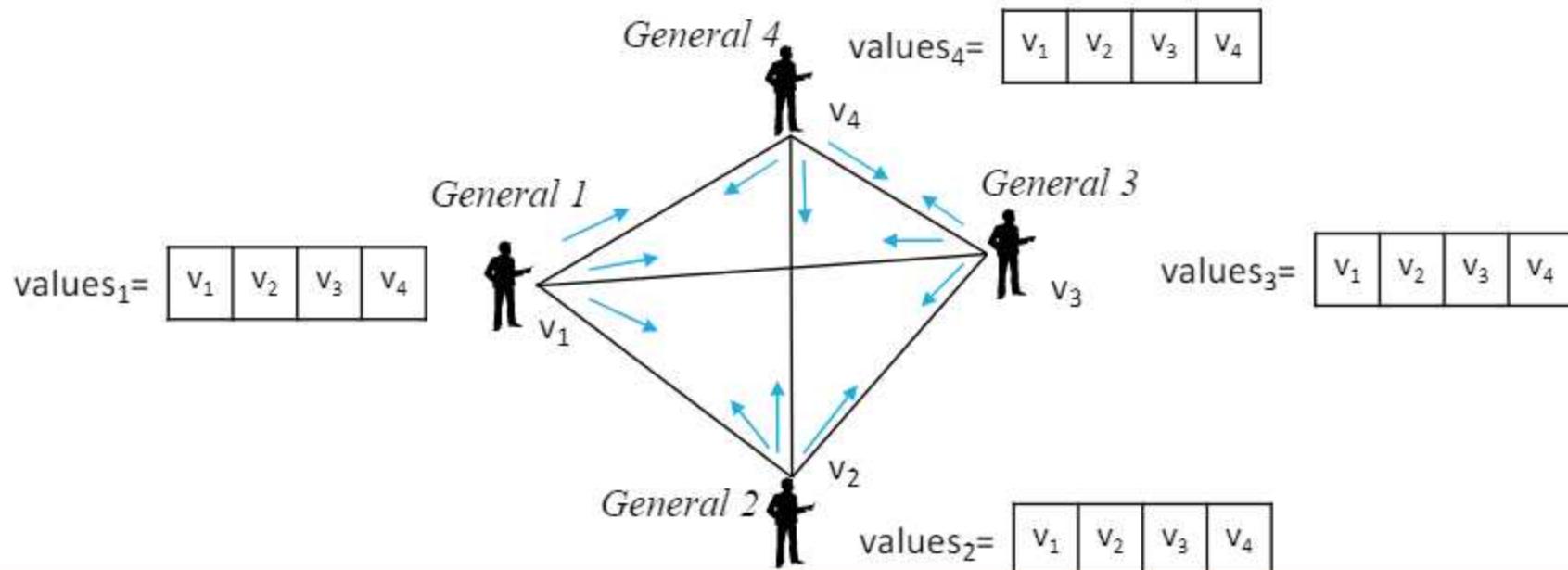
- several divisions of the Byzantine army are camped outside of an enemy city
- generals at the head of each division can communicate by messengers
- observing the enemy each general can propose to attack or retreat
- some generals are traitors *byzantine*
- the army wins only if all loyal generals attack or retreat

↳ only correct  
tech consensus



# An Intuitive Algorithm

1. Each general starts with its own value  $v(i)$
2.  $v(i)$  must be communicated by the  $i$ -th general to others
3. Each general uses some method for combining the values  $v(1), \dots, v(n)$  into a single plan of action, where  $n$  is the number of generals
  - o We need to define a function  $f(values_i)$



# The Byzantine Generals Problem

**Goal:**  $f(\text{values}_i)$  must be defined in a way that:

- A. all loyal generals decide upon the same plan of action
- B. a small number of traitors cannot cause the loyal generals to adopt a bad plan

What does it mean  
“bad plan”?

↳ what it is?

# An Intuitive Algorithm

---

## OBSERVATIONS

- Condition **A** is achieved by having all generals use the **same method for combining the information**
  - $f(\text{values}_i)$  must be **deterministic**
- Condition **B** is achieved by using a **robust method**
  - E.g., **if the only decision to be made is whether to attack or retreat, then  $v(i)$  can be General  $i$ 's opinion of which option is best, and the final decision can be based upon a majority vote among them.**

# The Byzantine Generals Problem

Each general  $i$  sends its opinion represented by the value  $v(i)$  to all

Rephrasing the goal: ideal algorithm

↗ each general obtain  
same vector

1. Every loyal general must obtain the same information  $v(1), \dots, v(n)$
2. If the  $i$ -th general is loyal, then the value that he sends must be used by every loyal general as the value of  $v(i)$

Therefore 1. can be rephrased:

1. Any two loyal generals use the same value of  $v(i)$

# The Byzantine Generals Problem

Both conditions 1 and 2 are expressed on the single value sent by the i-th general. Therefore, we can restrict the problem to how a general communicate its value to loyal generals.

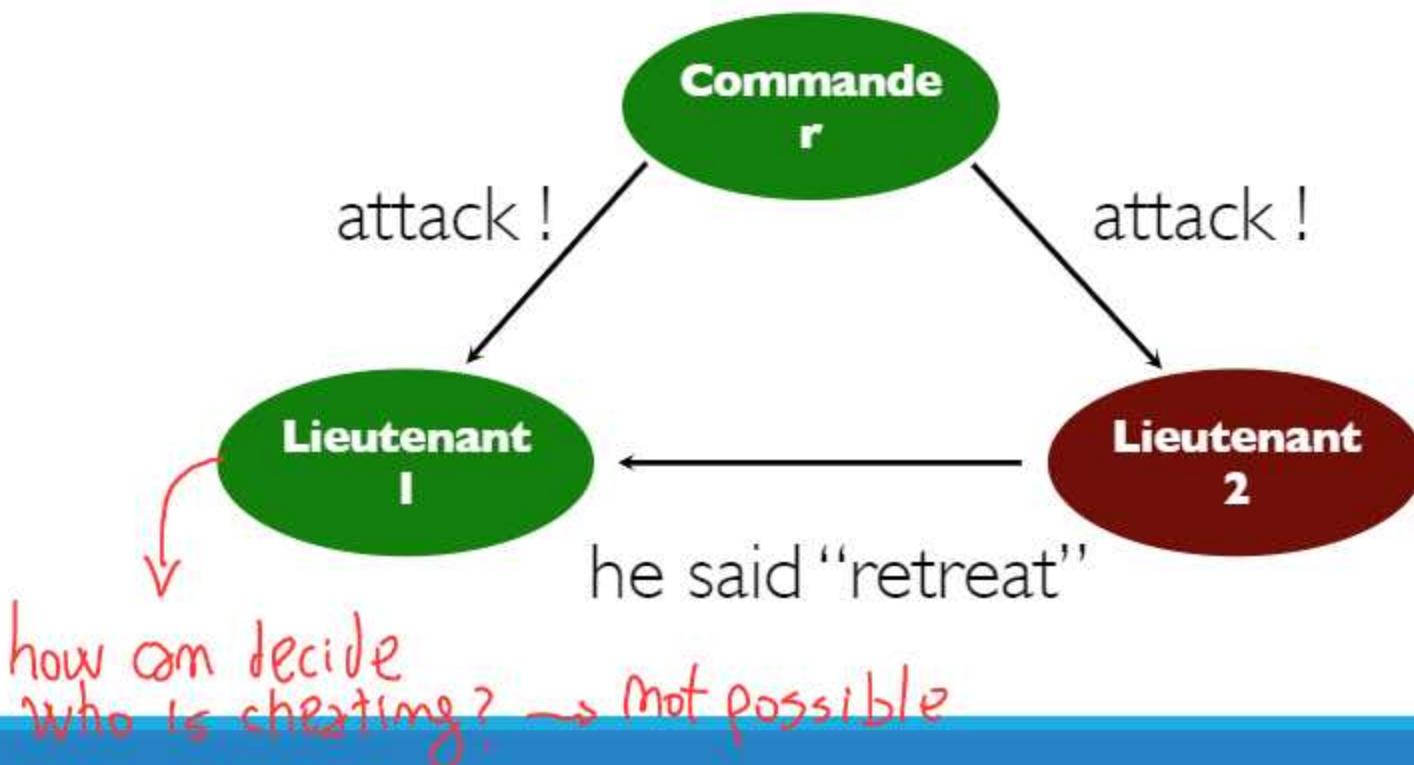
Interactive  
consistency

**Byzantine Generals Problem:** a commanding general must send his order to  $n-1$  lieutenant generals such that:

- All loyal lieutenants obey the same order (IC1)
- If the commanding general is loyal, then every loyal general obeys the order he sends (IC2)
- The order is “Use  $v(i)$  as my value”.

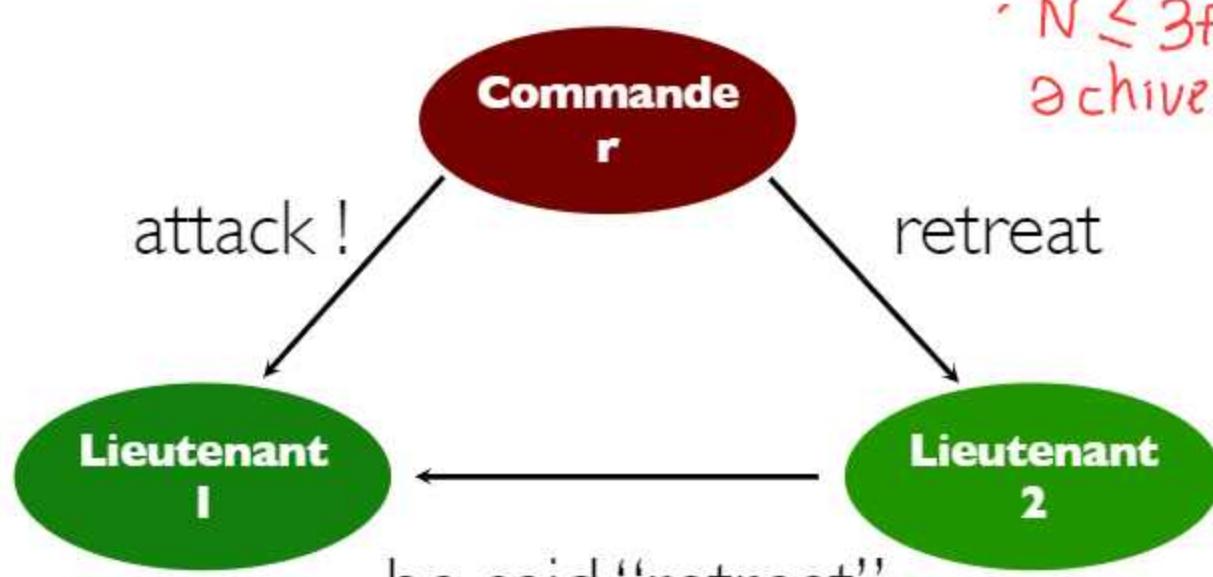
# The Byzantine Generals Problem

**Impossibility result:** if the generals can send only plain text messages no solution will work unless more than two-thirds of the generals are loyal.



# The Byzantine Generals Problem

**Impossibility result:** if the generals can send only plain text messages no solution will work unless more than two-thirds of the generals are loyal.



•  $N \leq 3F$  is impossible to achieve consensus

↓  
must have that  
 $N \geq 3F + 1$

↳ can't distinguish from previous situation

# The Byzantine Generals Problem

---

This **abstract problem captures many of the issues we face if we want to implement a RSM in a byzantine failure model:**

- How can correct replicas agree on a common order for client requests ?
- How can correct replicas decide what is the correct answer to a client request ?
- How can correct replicas maintain their state consistent ?

# Byzantine reliable consensus

## System model:

- Up to  $f$  processes can be Byzantine
- $N \geq 3f + 1$
- The “Oral Message” communication model is assumed
  - Every Message that is sent is delivered correctly  $\xrightarrow{\text{PP2P}}$
  - Message source is known to the receiver  $\xrightarrow{\text{authenticated links}}$
  - Message omissions can be detected  $\xrightarrow{\text{system is synchronous}}$

The default decision for Lieutenants is RETREAT

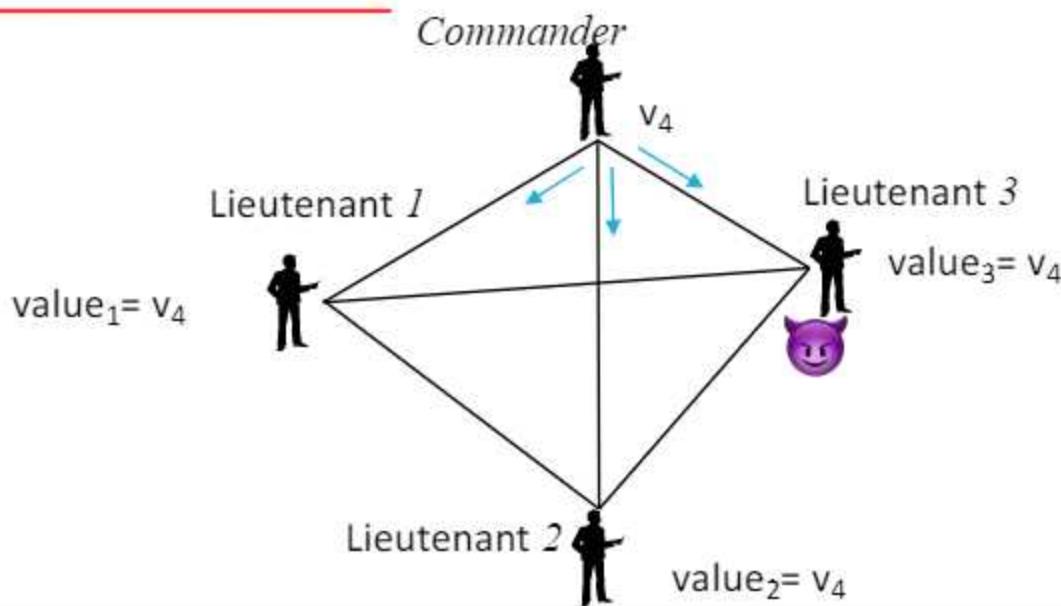
$\hookrightarrow$  we have a default value

## Implementing the "Oral Message" communication abstraction

We define inductively a set of protocols  $\text{OM}(f)$ :  
 $\text{OM}(0)$

$f+1$  rounds are iterate

1. The commander sends his value to every lieutenant.
  2. Each lieutenant uses the value he receives from the commander, or uses the value RETREAT (i.e.,  $\perp$ ) if he receives no value.



majority (  $\square$  <sub>general</sub>  $\square$  <sub>Lieutenant i</sub>  $\cdots$   $\square$  )  
ROUND 1   ROUND 2   ROUND f

# Byzantine reliable consensus

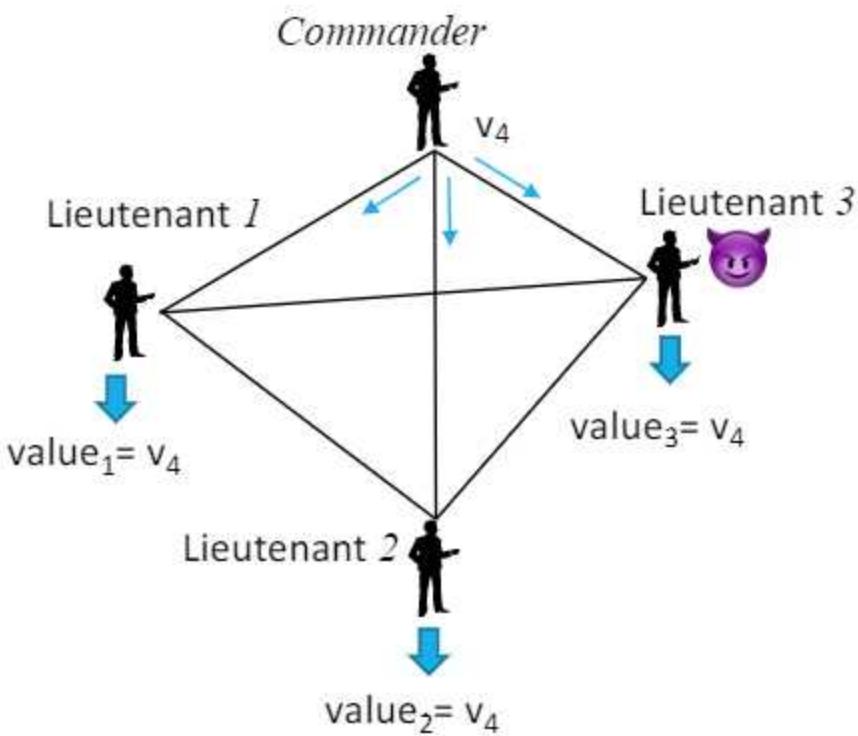
We define inductively a set of protocols  $\text{OM}(f)$ :

## $\text{OM}(f)$ with $f > 0$

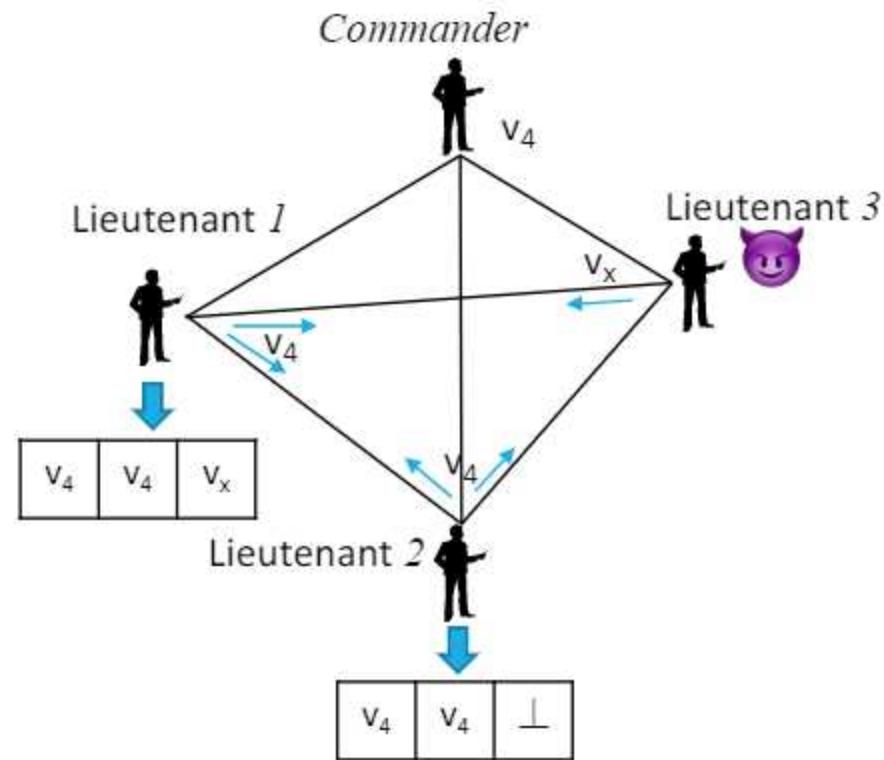
1. The commander sends his value to every lieutenant.
2. For each  $i$ , let  $v_i$  be the value Lieutenant  $i$  receives from the commander, or else be RETREAT if he receives no value:
  - Lieutenant  $i$  acts as the commander in algorithm  $\text{OM}(f - 1)$  to send the value  $v_i$  to each of the  $N - 2$  other lieutenants.
3. For each  $i$  and  $j$  ( $j \neq i$ ), let  $v_j$  be the value Lieut.  $i$  received from Lieut.  $j$  in step (2), or else RETREAT if he received no value:
  - Lieutenant  $i$  uses the value  $\text{majority}(v_1, \dots, v_{N-1})$ .

# Implementing the "Oral Message" communication abstraction

OM(1)



STEP 1 → store locally value received from commander

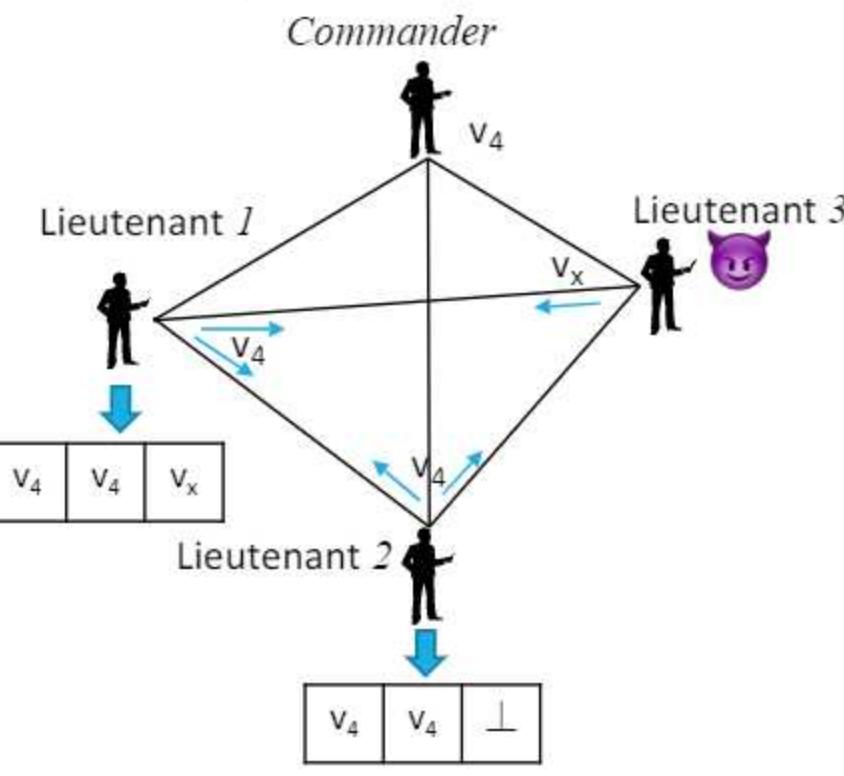


STEP 2

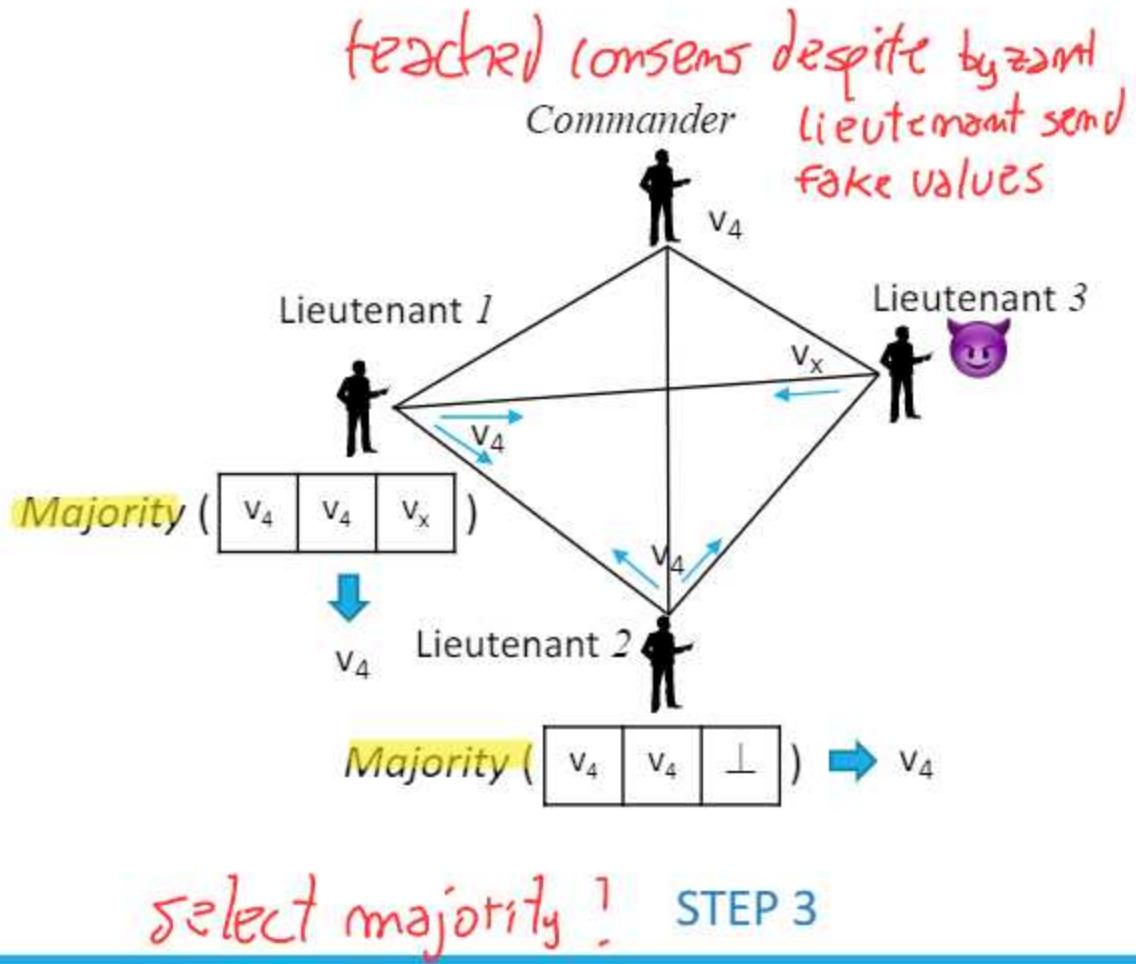
Each Lieutenant starts OM(0)

# Implementing the "Oral Message" communication abstraction

OM(1)



STEP 2

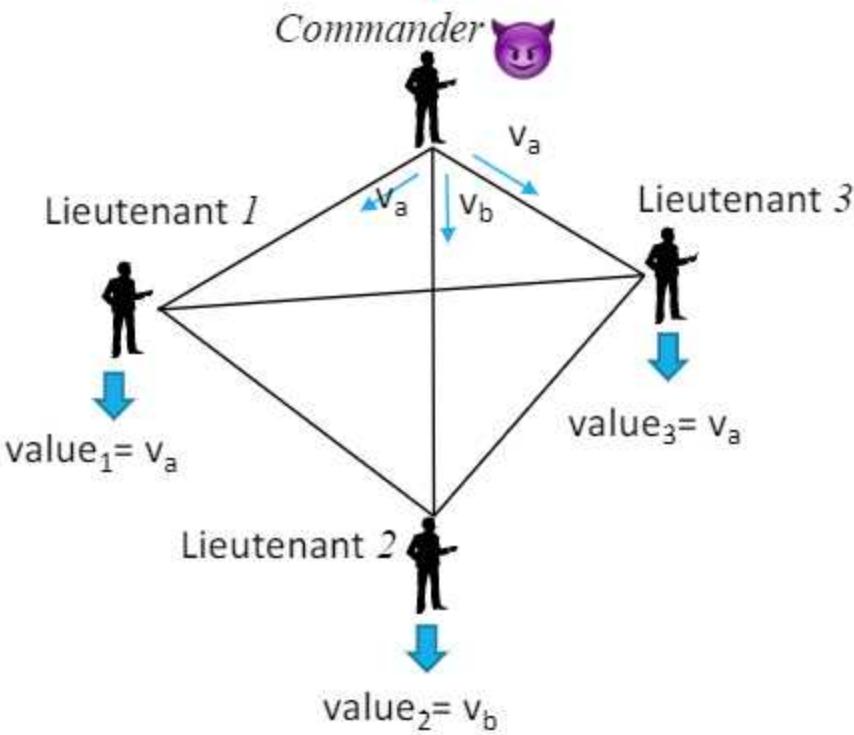


Each Lieutenant starts OM(0)

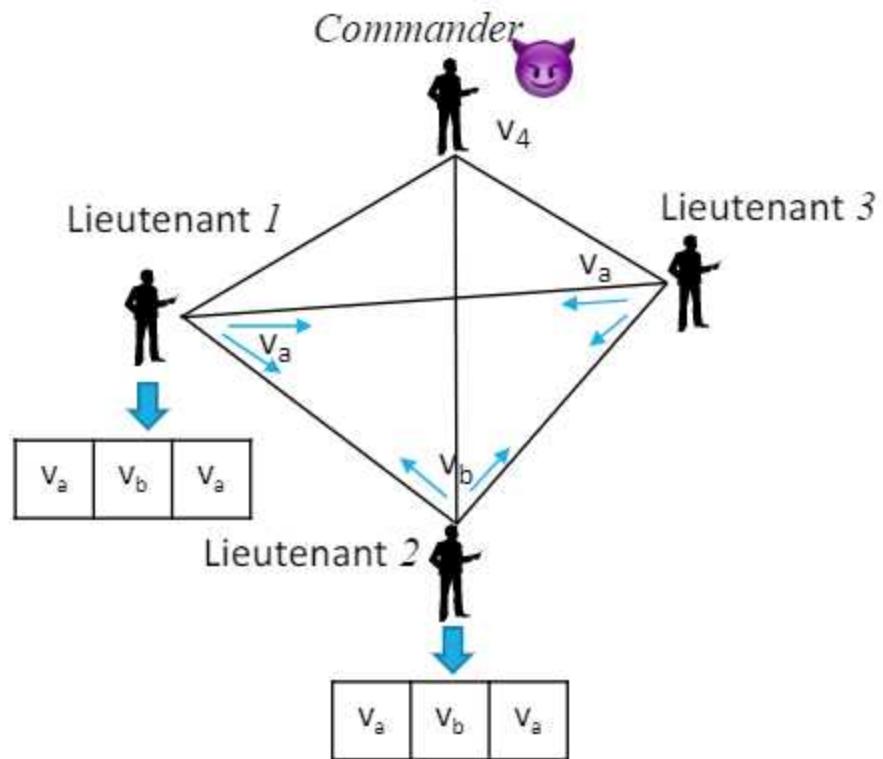
# Implementing the "Oral Message" communication abstraction

OM(1)

→ heterogeneous communication



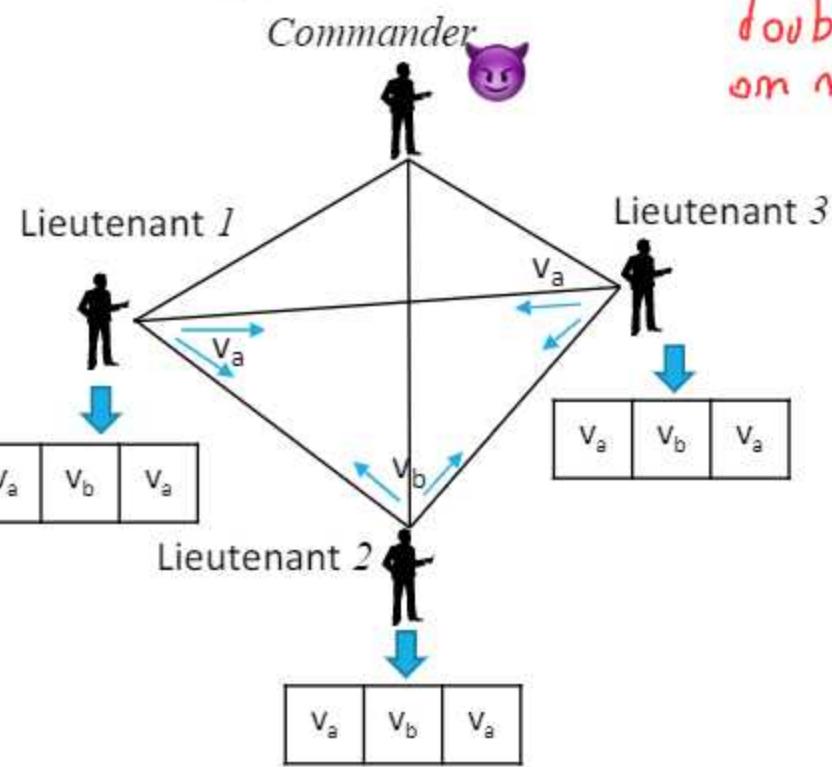
STEP 1



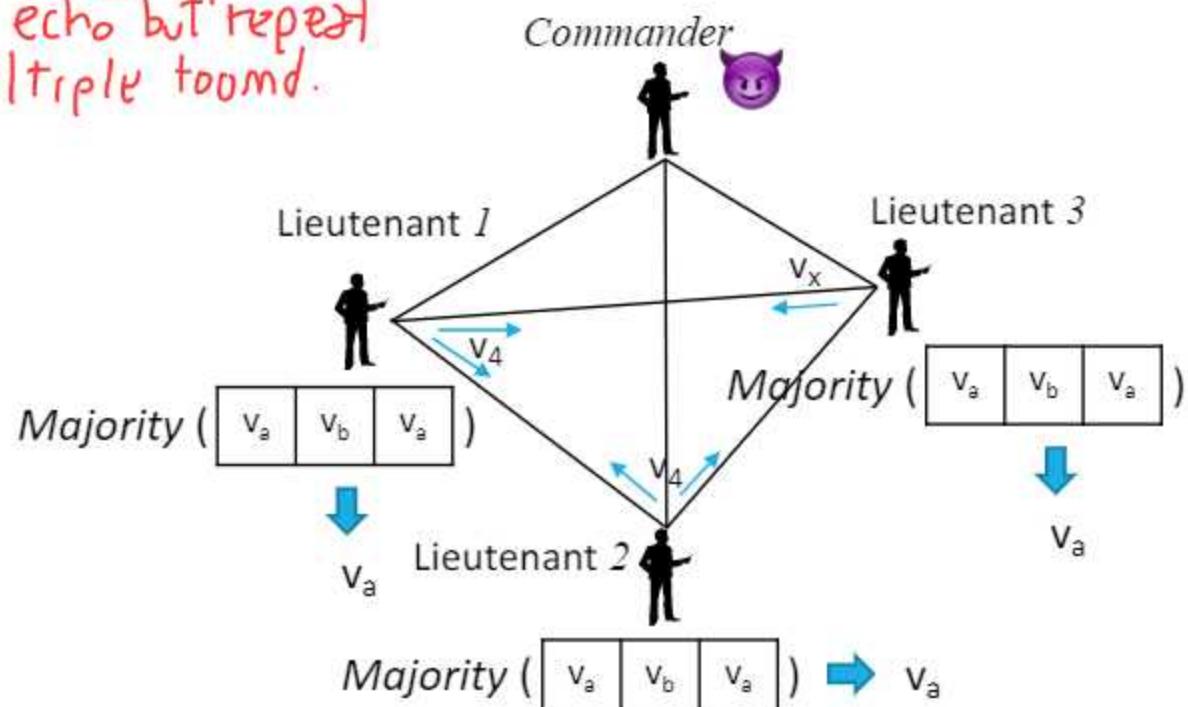
STEP 2

# Implementing the "Oral Message" communication abstraction

OM(1)



same philosophy of  
double echo but repeat  
on multiple round.



achieve a consensus  
but on a fake value

# Byzantine reliable consensus

Strongly inefficient algorithm

- Recursion creates a large number of messages

!

The complexity lies in the fact that a Byzantine process can easily change the content of messages

↳ we can achieve better result identifying the traitor

What happens if we use message authentication codes ?

- A solution for 3 processes exists !

↓  
attacker can't  
forge signature of sender

We can use less than  $3f$  processes, decreases complexity  
we not need  $f$  rounds, we simply need a majority

# Byzantine reliable consensus

The commander signs and sends his value ( $v:0$ ) to every lieutenant.

For each  $i$ :

↳ sign value before send it

- If Lieutenant  $i$  receives a message of the form  $v:0$  from the commander and he has not yet received any order, then
  - $V_i = \{v\}$
  - Sends message  $v:0:i$  to every other lieutenant
- If Lieutenant  $i$  receives a message of the form  $v:0:j_1:j_2:j_k$  and  $v$  is not in  $V_i$  then
  - adds  $v$  to  $V_i$ ;
  - if  $k < f$  sends the message  $v:0:j_1:j_2:j_k:i$  to every lieutenant other than  $j_1, \dots, j_k$

For each  $i$ : when Lieutenant  $i$  receives no more messages, he obeys the order choice( $V_i$ ).

# References

---

Leslie Lamport, Robert Shostak, and Marshall Pease "The Byzantine Generals Problem " in ACM TOPLAS 1982

Available at <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/The-Byzantine-Generals-Problem.pdf>

Dependable Distributed Systems  
Master of Science in Engineering in  
Computer Science

AA 2022/2023

---

LECTURE 30,31: DLT AND BLOCKCHAIN

Blockchain is a distributed storage, that has some kind of security and performance constraint, maintaining by a distributed set of processes

## What is a Distributed Ledger?

### Distributed Ledger

(particular type of distributed storage system, repository of data, taking temporal perspective over the data.

book over  
which are recording  
transaction

A ledger is a written or computerized record of all the transactions a business has completed.

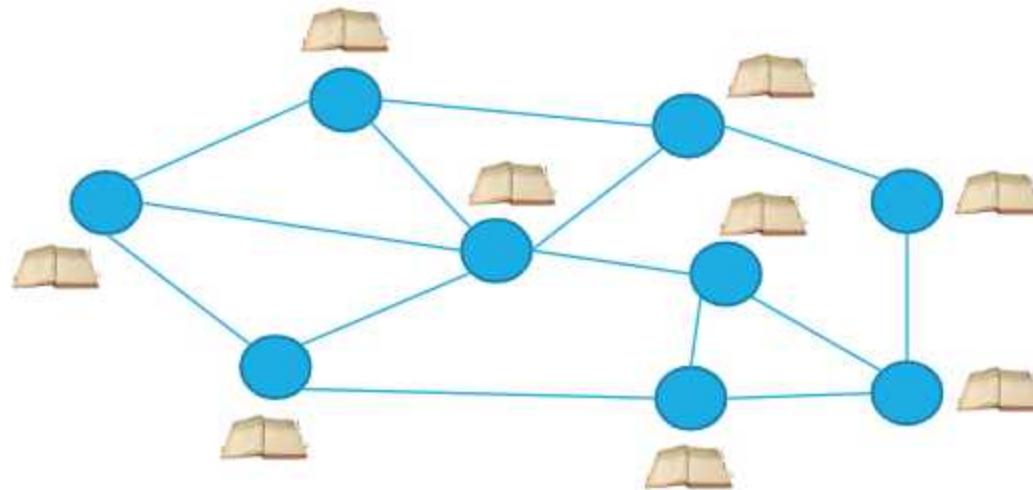


A distributed ledger is a database replicated across several locations or among multiple participants used to store record of transactions

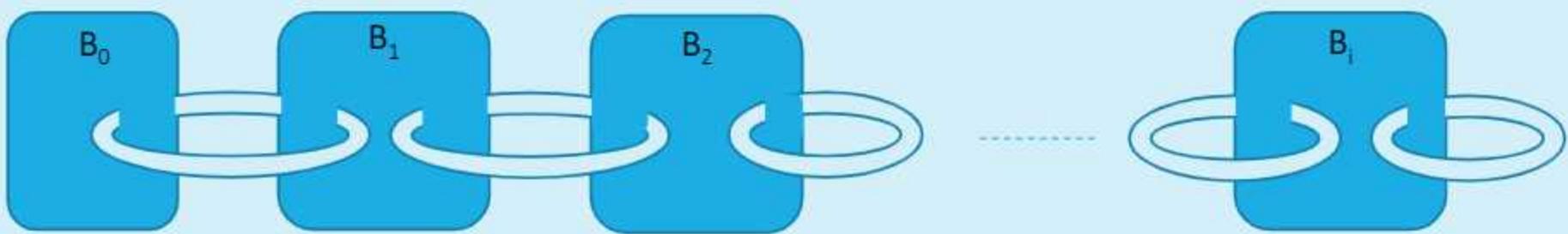
# What is a Distributed Ledger?

## Key properties

- **Consistency** → not modified by failures of adversary
- **Integrity**
- **Availability** → client can access immediately



# What is a blockchain?

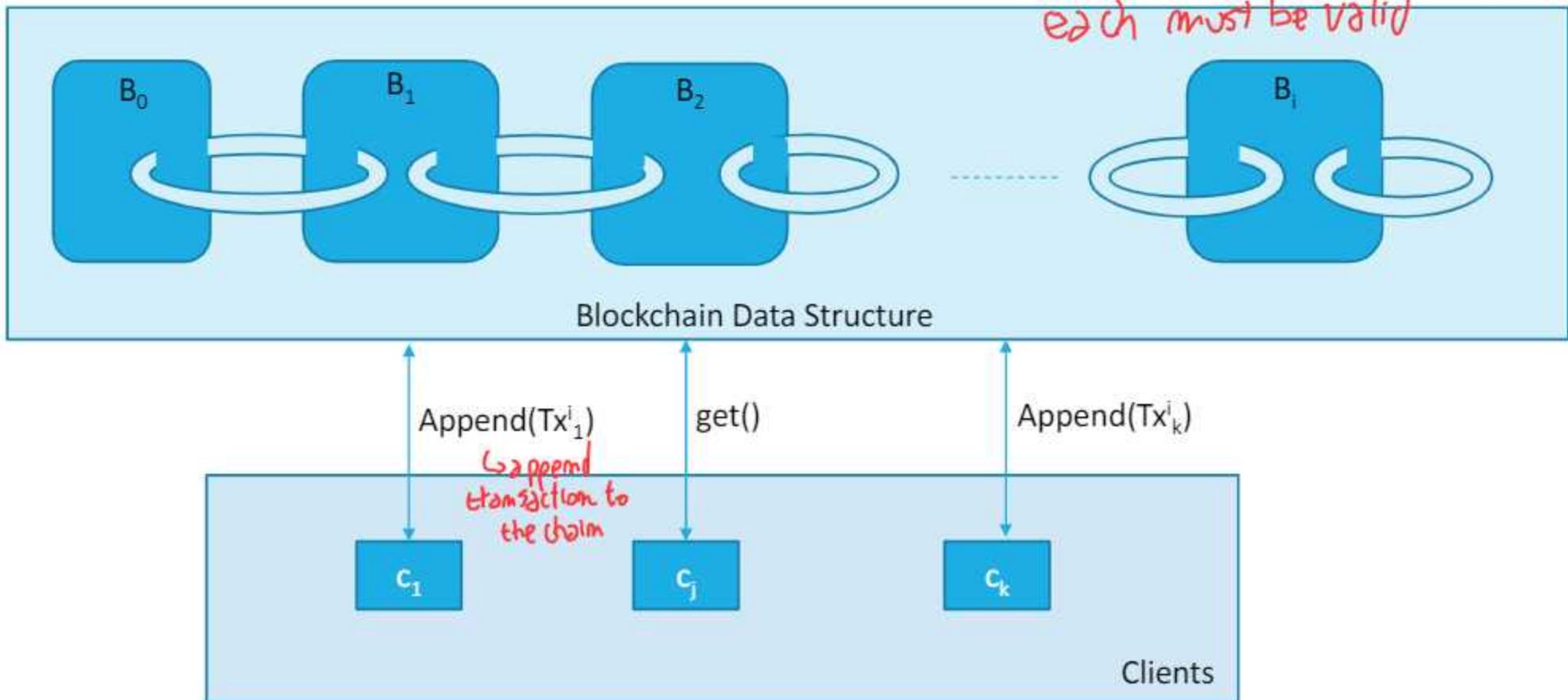


↳ list of blocks linked, list of transactions

A blockchain is a decentralized, distributed data structure used to record transactions (aggregated in blocks) across many computers.

# What is a blockchain?

collection of blocks, each contains transactions and each must be valid

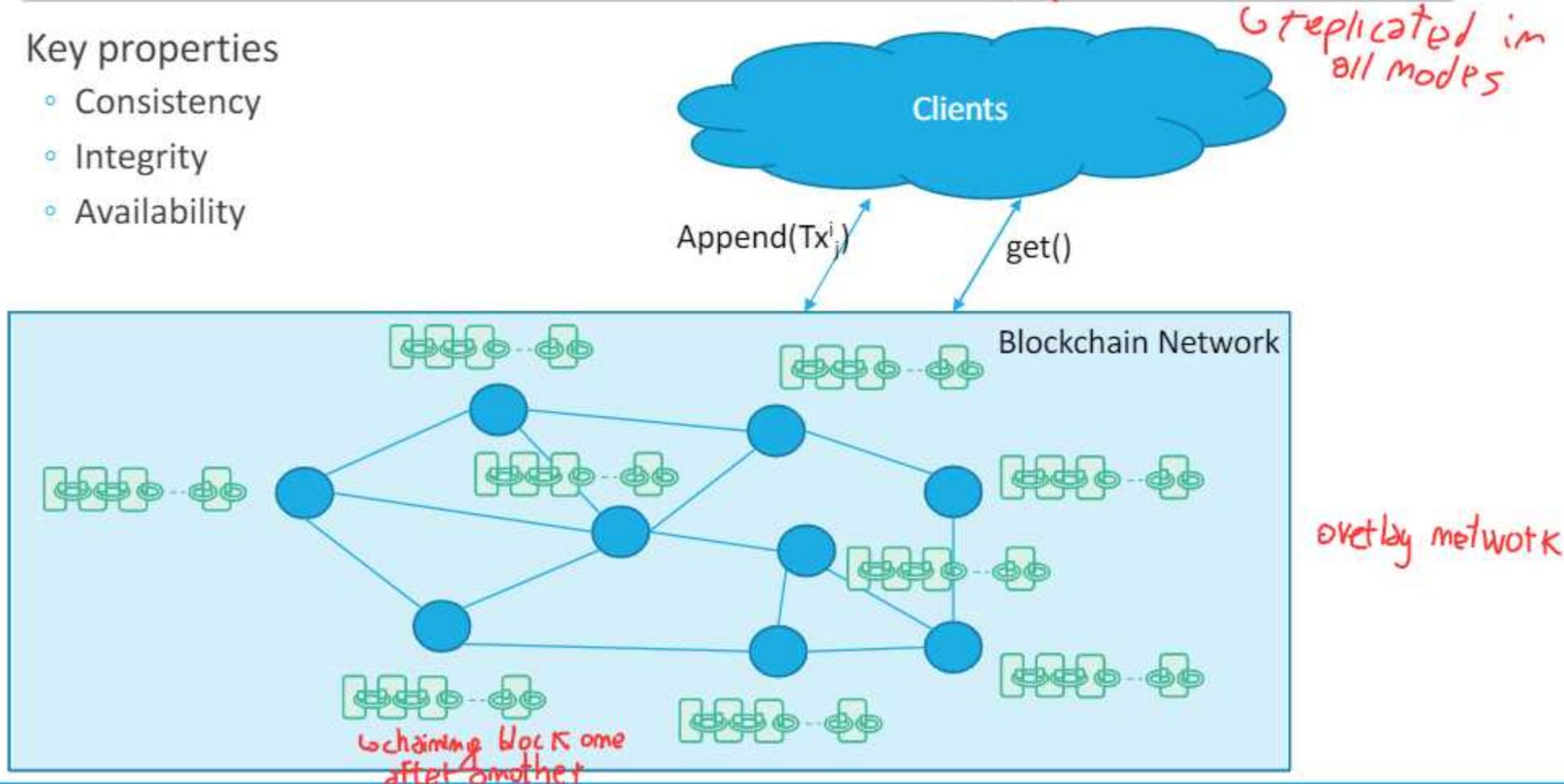


# What is a Blockchain?

From abstract you have a single object, but in reality it is distributed to all nodes that are part of blockchain

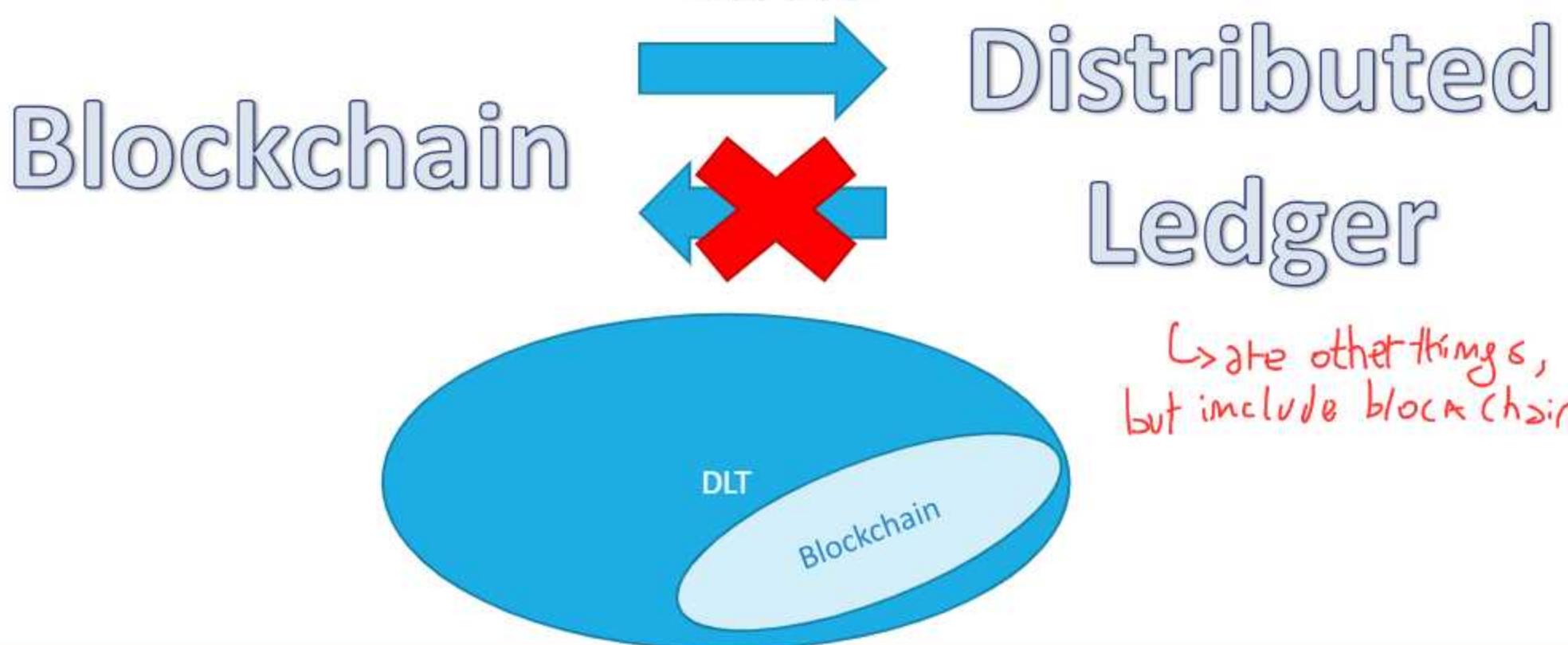
## Key properties

- Consistency
- Integrity
- Availability



# Blockchain vs DL

*Every blockchain represents a distributed ledger but the vice versa is not true*



in ledger state the history of a transaction, in blockchain history of crypto currency

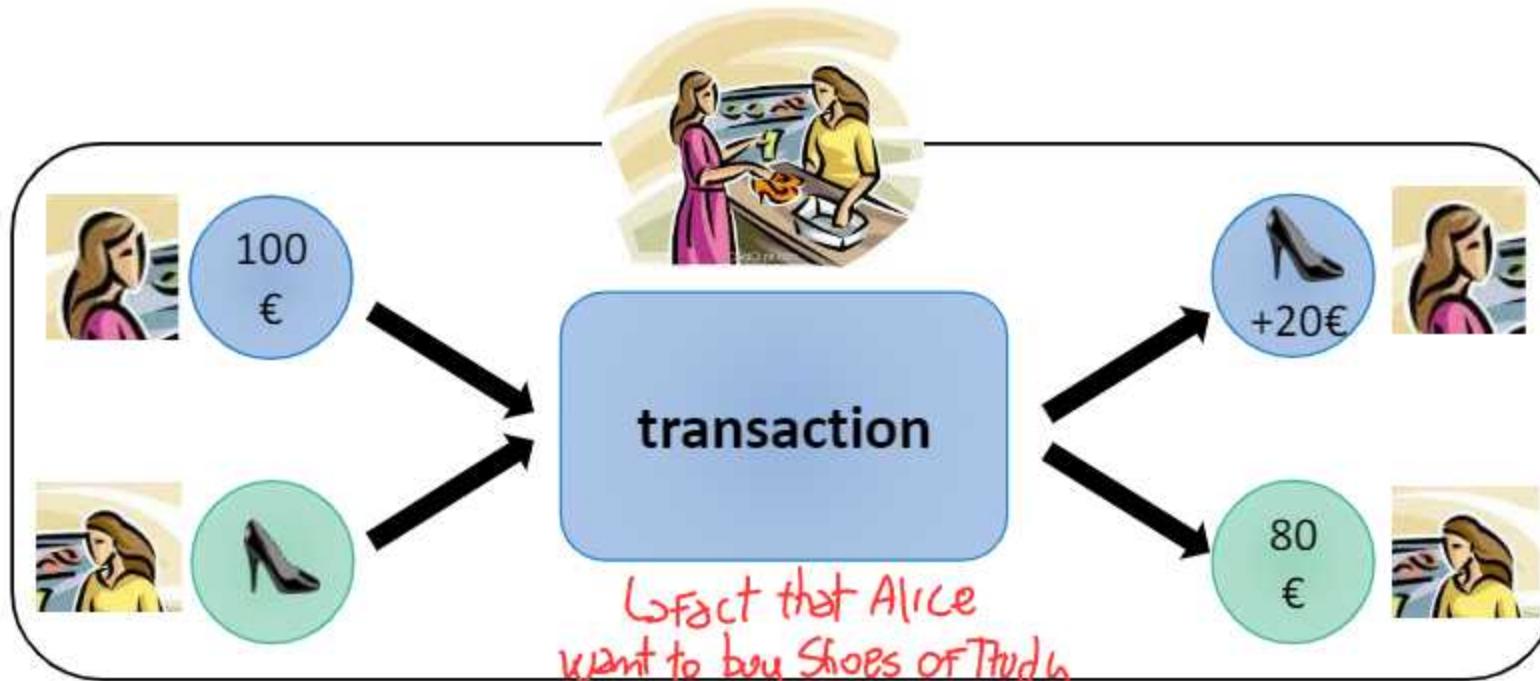
# What is a transaction?

A transaction is an atomic representation of exchanging of goods

- an instance of buying or selling something
- an exchange or interaction between people

entity 1  
Alice

entity 2  
Tudy



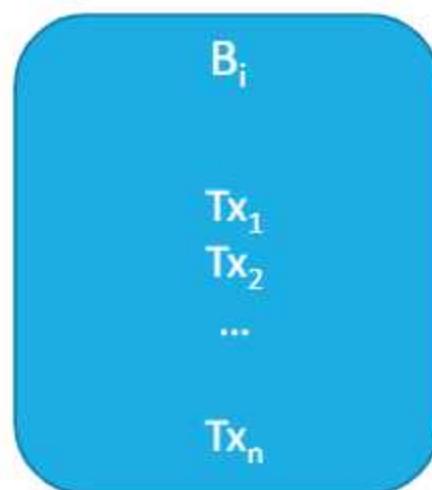
→ organize transaction in block in blockchain

# What is a Block?

---

Every block in a blockchain contains a set of transaction

- Clients generate transactions and submit them to nodes implementing the blockchain



# How to create a Block?

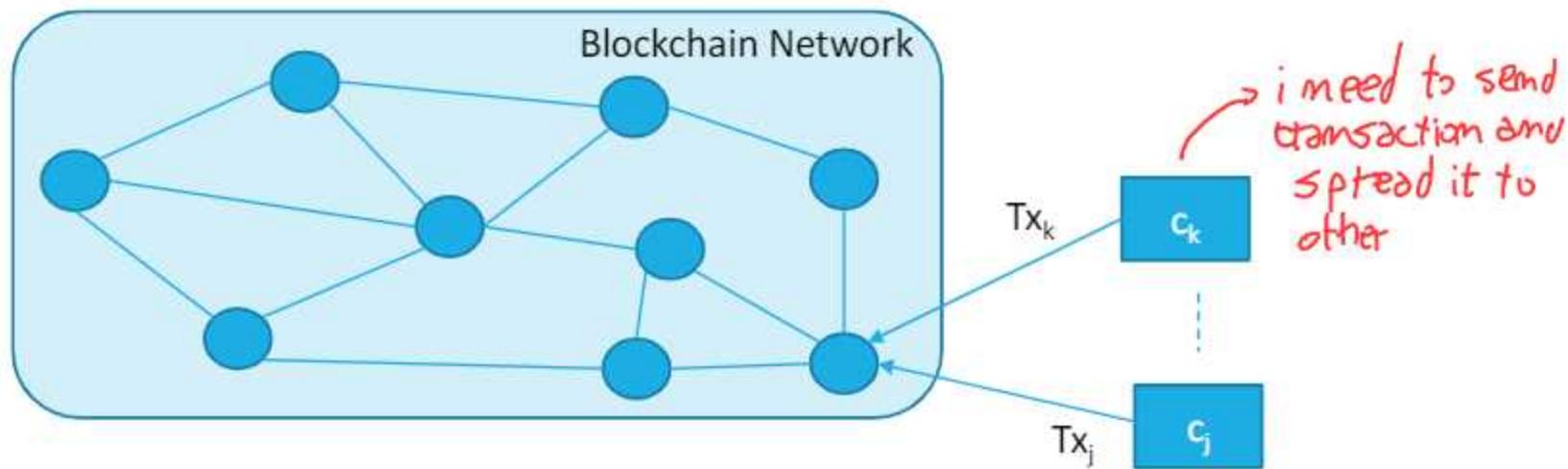
Transactions are collected by processes

Transactions need to be validated i.e., they need to be valid with respect to the ledger specification

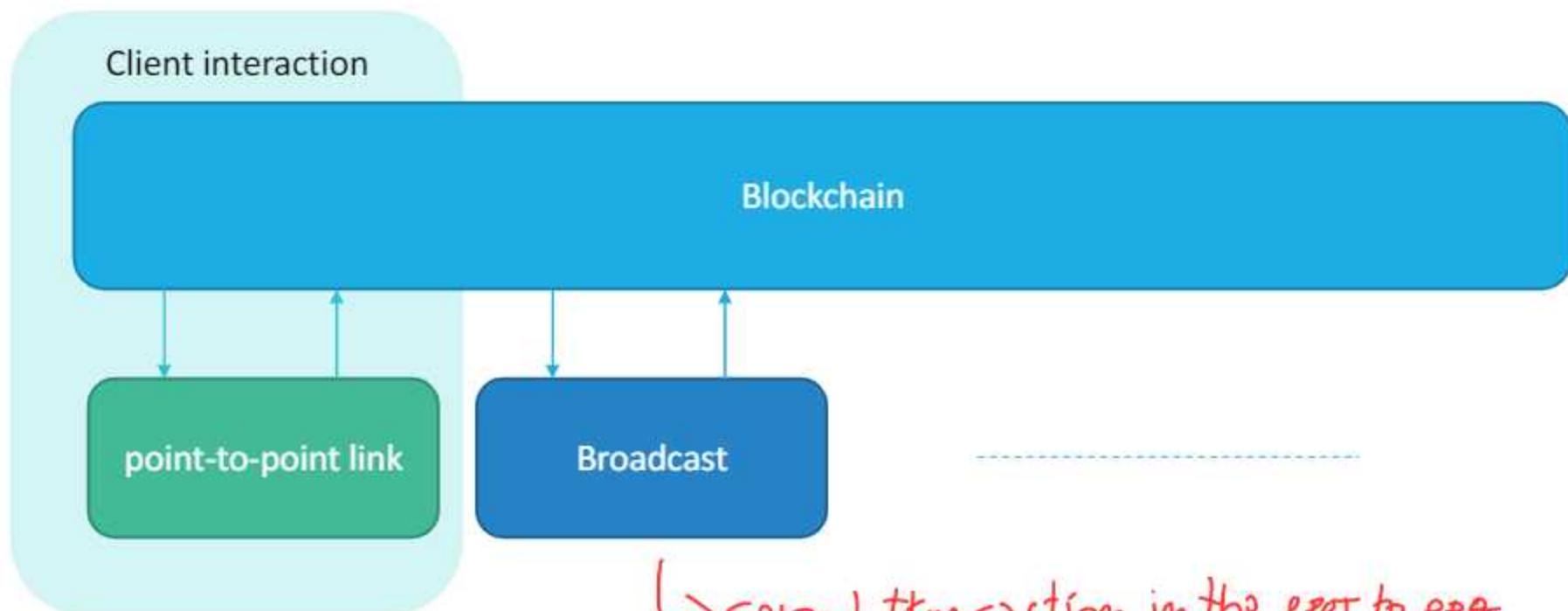
- E.g., in a financial ledger a transaction cannot spend money that are not in the account

Transactions remain *unconfirmed* until they are inserted in a block that is successfully attached to the chain

When "enough" transactions have been collected a block can be created and attached



# Key ingredients



↳ spread transaction in the peer to peer network

# How to attach a block to the chain?

---

## Validate transaction

- In order to validate a transaction, nodes need to read the “last” state of the ledger and check that the current transaction is correct with respect to the semantics of the ledger
  - E.g. if in the last block we found that Alice’s account has 300\$, then any transaction spending at most 300\$ will be valid

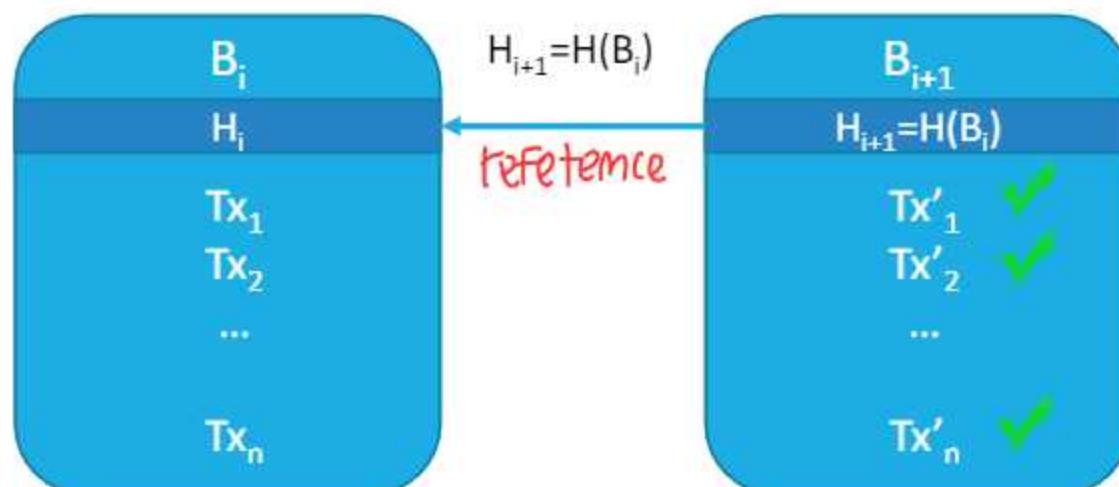
## OBSERVATION

- Ordering of transactions in the blocks is fundamental to check and guarantee the validity of the ledger

# How to attach a block to the chain?

## Chaining blocks

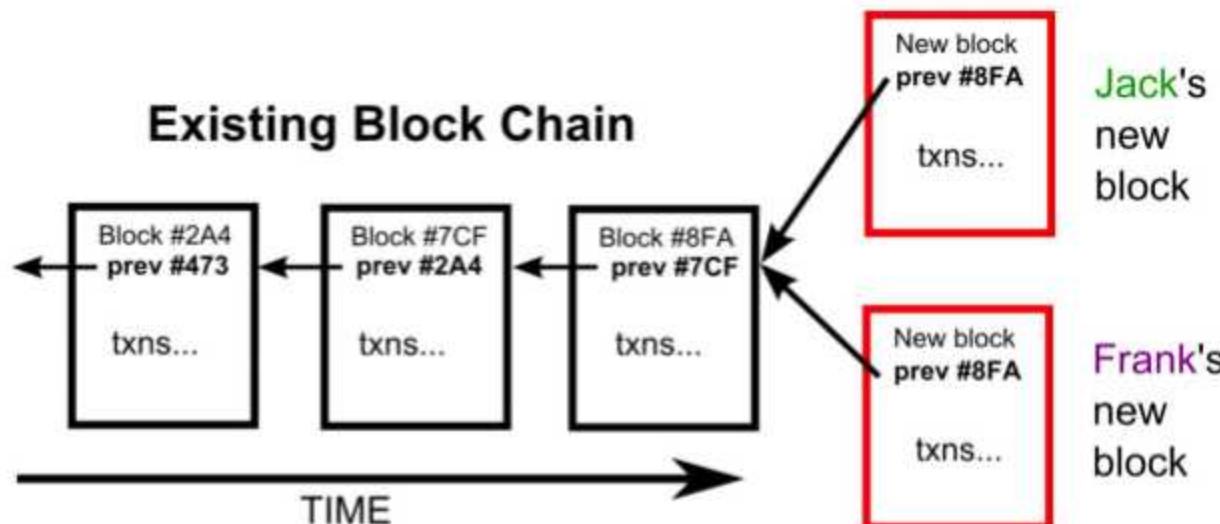
1. Check that all the transactions in the block are valid
2. Compute the hash  $H()$  of the last block attached to the chain
3. Include this hash in the current block to generate a pointer and attach the current block



# How to attach a block to the chain?

## ISSUES

- Concurrency in the block creation process
- Asynchronous systems



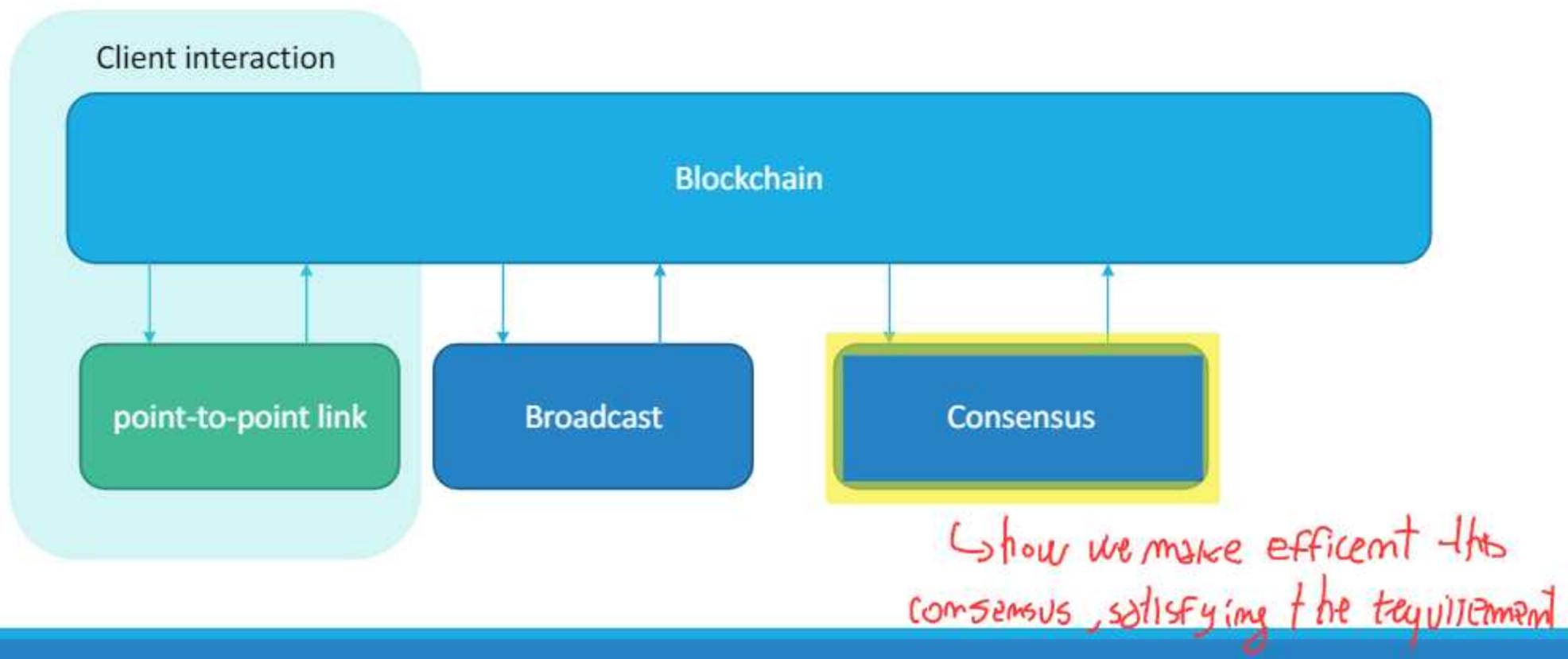
# How to create and chain blocks

---

**Attaching a new block  
requires agreement  
between nodes**

for having the same  
view of network

# Key ingredients



# How to create and chain blocks

---

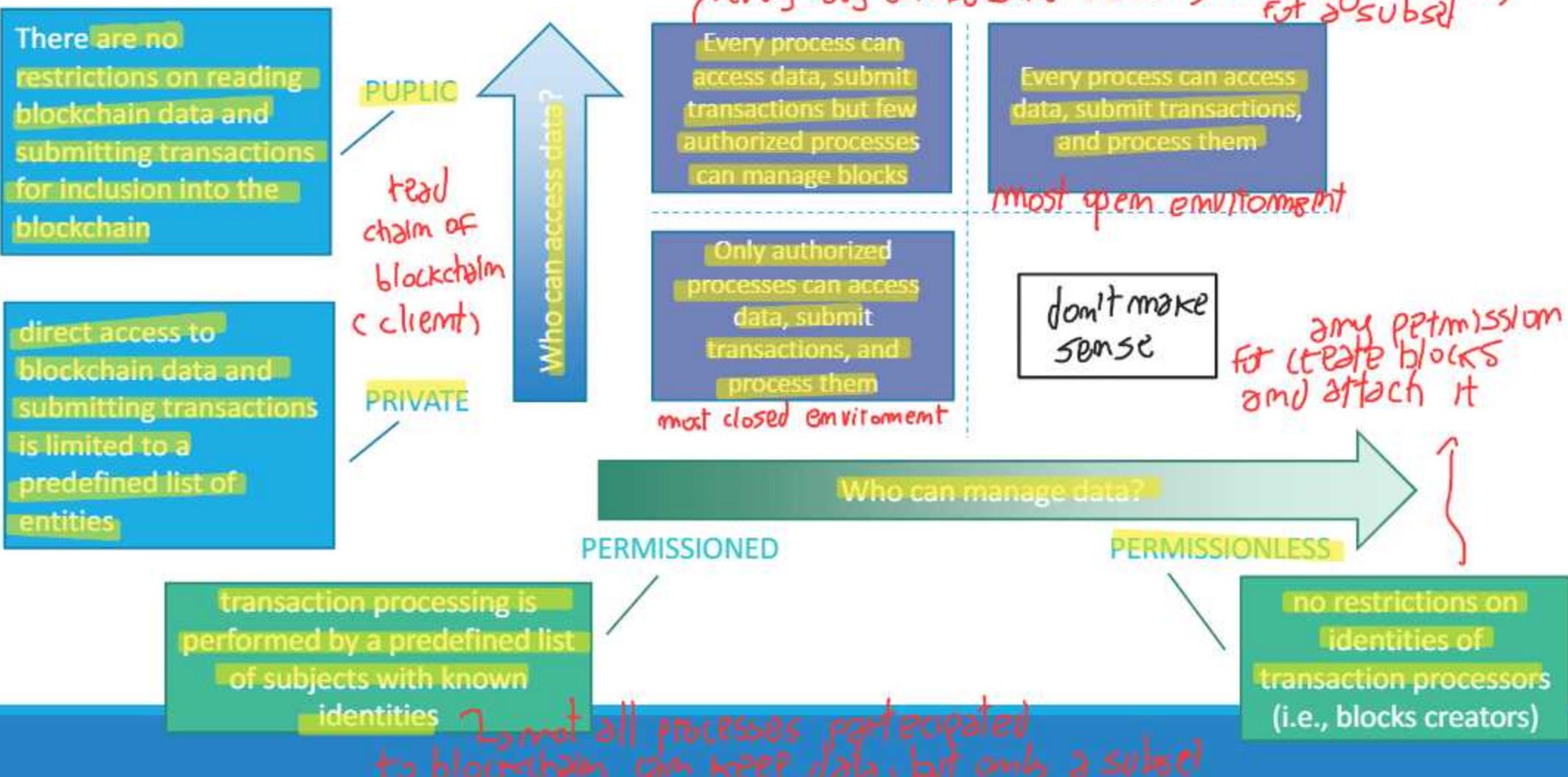
**Attaching a new block  
requires agreement  
between nodes**



**Which Consensus  
protocol should I use?**

→ different options, depends on  
characteristic you want for your data structure

# Blockchain Classification



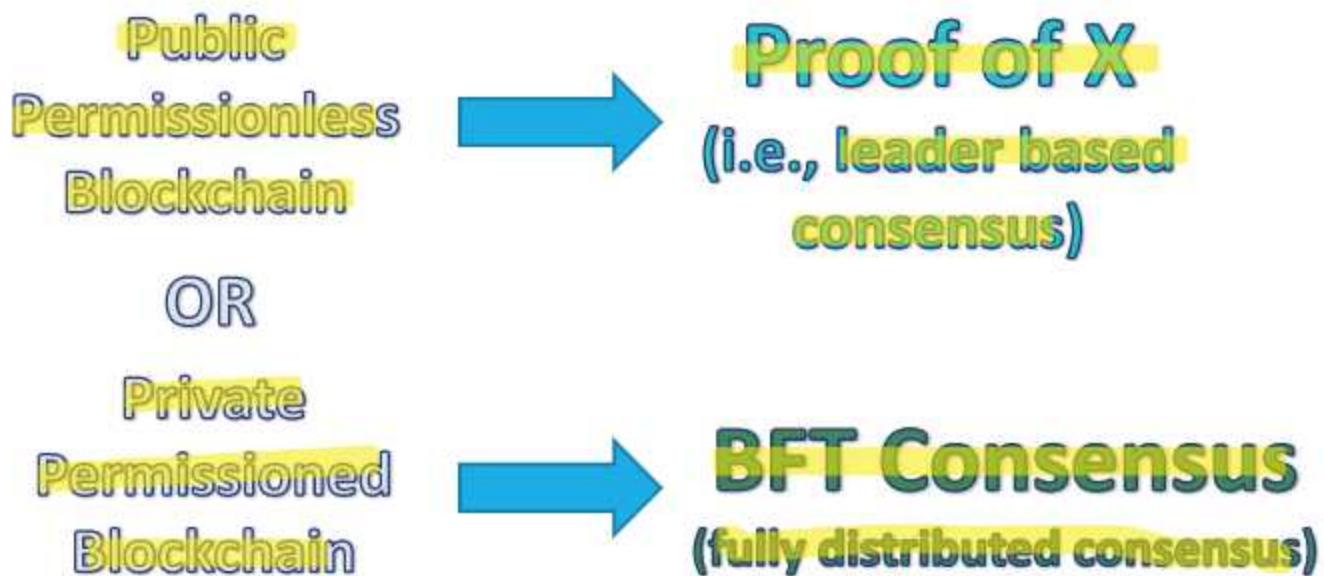
# How to create and chain blocks

---

## Attaching a new block requires Consensus

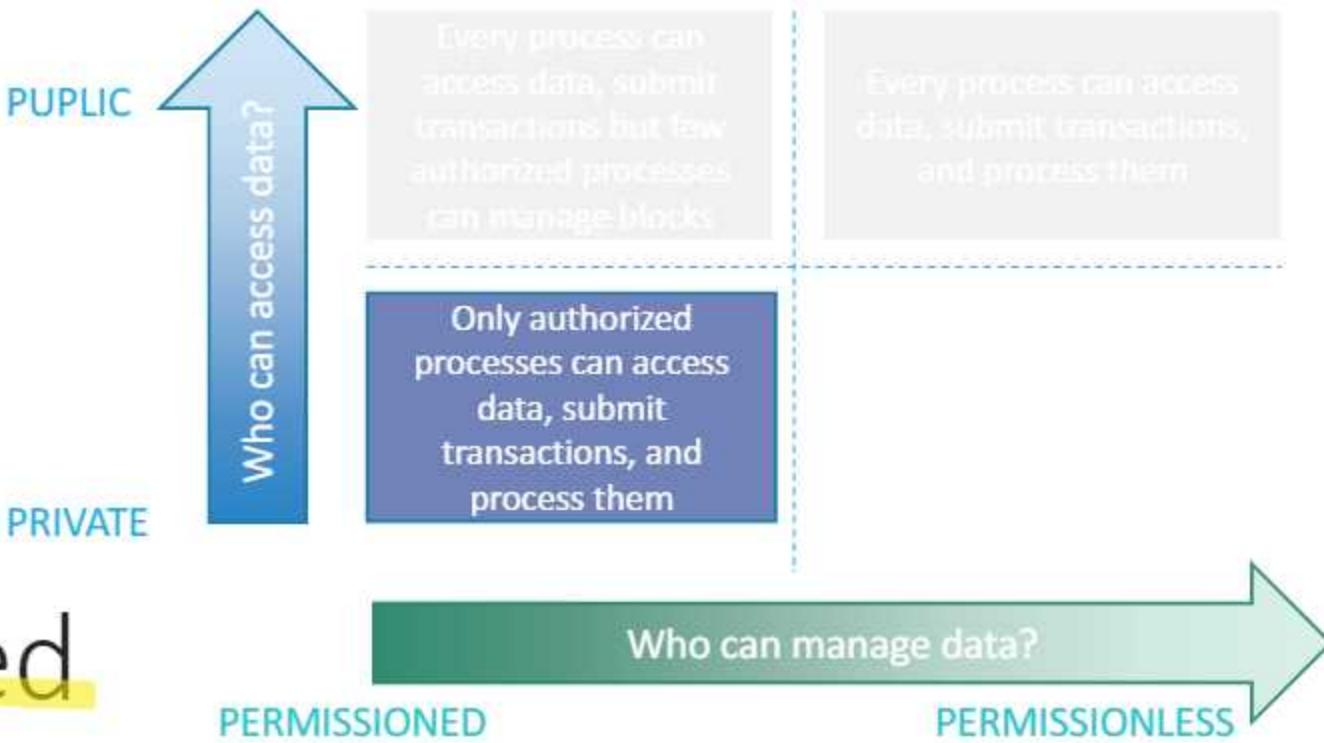


**Which Consensus  
protocol should I use?**



# Private Permissioned Blockchain

THE PBFT PROTOCOL



# Practical Byzantine Fault Tolerance

---

In private permissioned blockchain we have that

- Transactions are submitted only by known clients and only known clients can access them
- Blocks are created and attached by known and authorized processes

## IDEA

- Processes maintain a copy of the Blockchain locally and run a BFT consensus protocol to agree on the next block
- PBFT is currently the adopted protocol
  - It has been designed to support replication
  - It merges the primary-backup approach with the Byzantine General Consensus approach

# Practical Byzantine Fault Tolerance

## System model:

- Asynchronous distributed system
  - nodes connected by network
  - messages can be lost, delayed, duplicated, no order
- Independent node failures (byzantine)



Basically the same assumed to solve the Byzantine Generals Problem

# Practical Byzantine Fault Tolerance

---

System model:

- Cryptographic techniques (public key signatures, MAC, message digest produced by collision resistant hash functions)
  - needed to avoid spoofing, replay attacks and corruption
  - All replicas know each other's public keys to verify signatures

, Authenticated P2P link

· complete crypto

# Practical Byzantine Fault Tolerance

---

System model:

- Strong adversary that can
  - coordinate the action of faulty nodes
  - delay communication
  - delay correct nodes (not indefinitely)
- The adversary cannot subvert the cryptographic techniques cited above

# Practical Byzantine Fault Tolerance

---

## Properties:

- Safety
- satisfies linearizability – like a centralized system that executes operations atomically one at a time
- regardless of faulty clients
- but faulty clients can do strange (legal) operations (e.g., write garbage in a file system)
- limit damage by access control mechanisms

# Practical Byzantine Fault Tolerance

---

Properties:

- Liveness

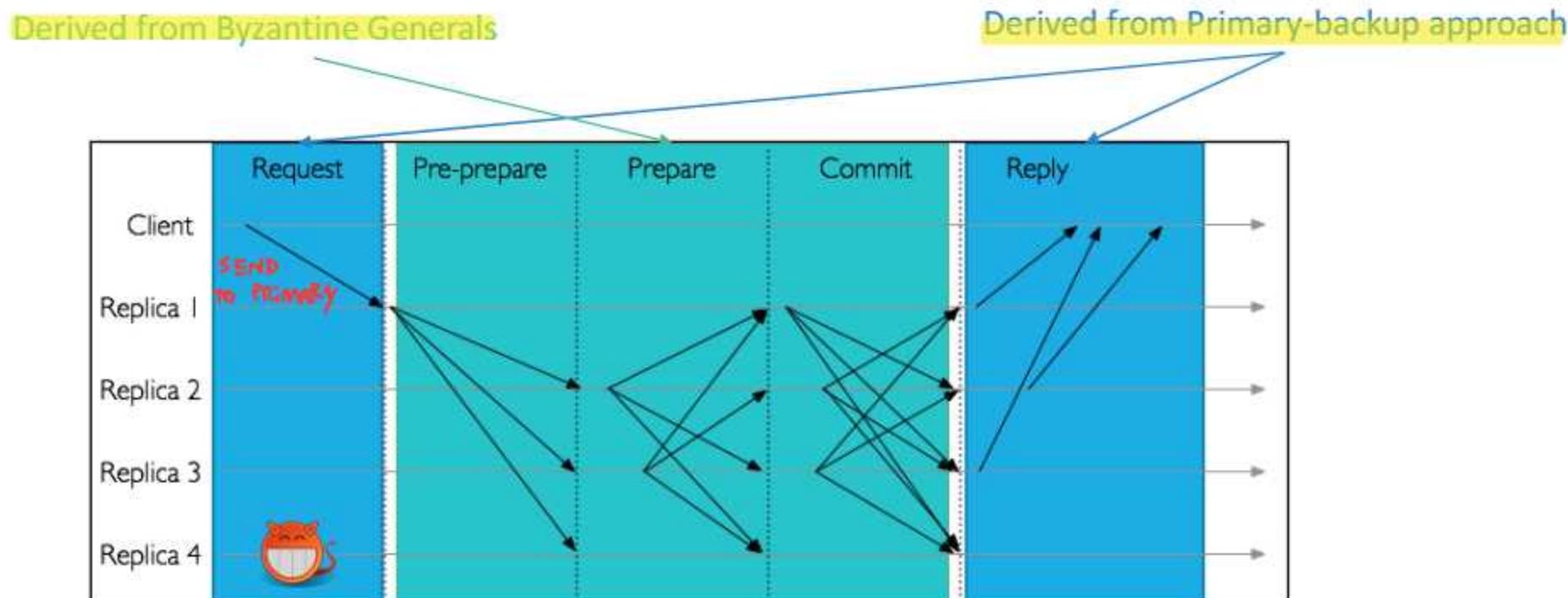
$$N > 3F + 1$$

- clients receive responses to their requests if at most  $(n-1)/3$  replicas are faulty and  $\text{delay}(t)$  does not grow faster than  $t$  indefinitely

# Practical Byzantine Fault Tolerance

The protocol is divided in five phases:

- Request, pre-prepare, prepare, commit and reply



# Practical Byzantine Fault Tolerance

---

## Basic idea:

- A client sends a request to invoke an operation to the primary
- The primary multicasts the request to the backups
- Replicas execute the request and send a reply to the client
- The client waits for  $f+1$  replies from different replicas with the same result; this is the result of the operation

# Practical Byzantine Fault Tolerance

---

If the primary is not correct it is substituted with a new one as soon as a misbehavior is detected

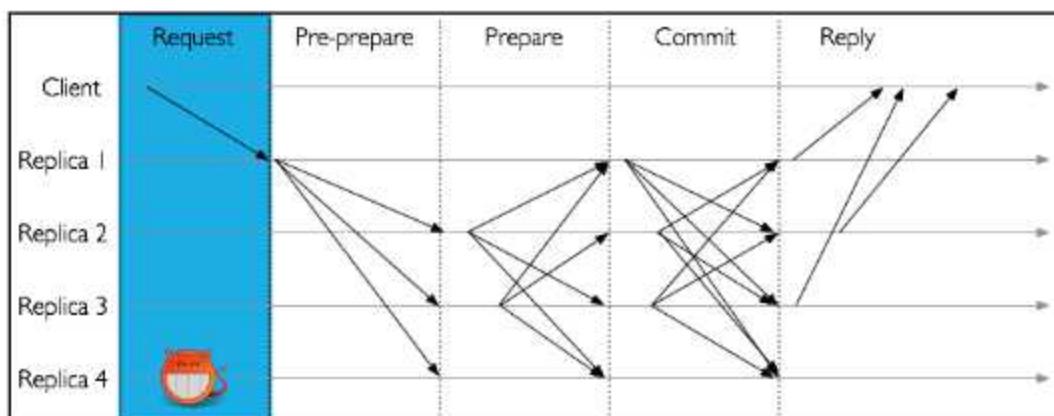
The system lifetime is characterized by a series of views each with a different primary

Views are numbered

# Practical Byzantine Fault Tolerance

## Request phase:

- Client  $c$  requests the execution of operation  $o$  issuing the request  $\langle \text{REQUEST}, o, t, c \rangle_{dc}$  to the primary
- The request is sent to the last primary known by the client
  - If a response is not received by a timeout the request is multicast to all replicas
- $t$  is a timestamp used to guarantee *exactly-once* semantic, for guarantee liveness



# Practical Byzantine Fault Tolerance

---

Replica internal state:

- replica id  $i$  (between 0 and  $N-1$ )
- service state
- view number  $v$  (initially 0) *≈ for progress in iteration and round of consensus*
- log of accepted messages

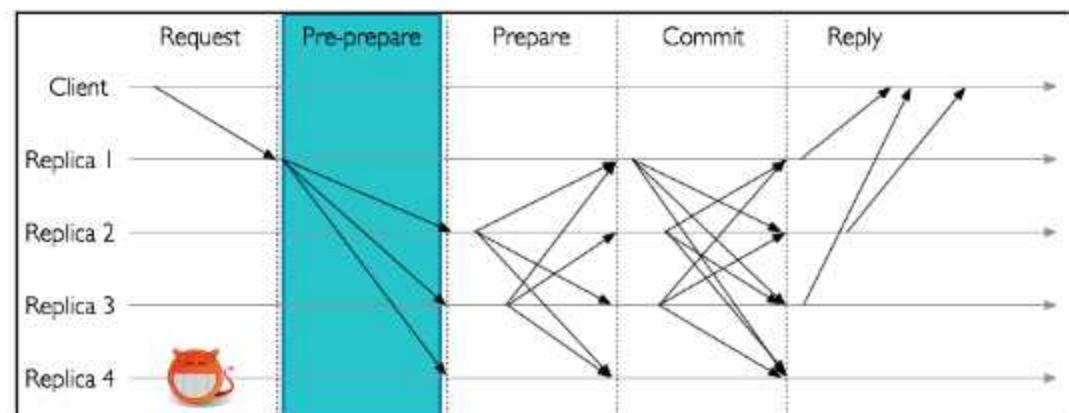
*↳ no do the same operation twice*

# Practical Byzantine Fault Tolerance

## Pre-prepare phase:

- the primary  $p$  sends to all backups the message  $\langle\!\langle \text{PRE\_PREPARE}, c, n, d \rangle\!\rangle_{op, m}$
- $c$  is the client id
- $n$  is a sequence number for the current view assigned by the primary
- $d$  is a digest of the client request
- $m$  is the client request

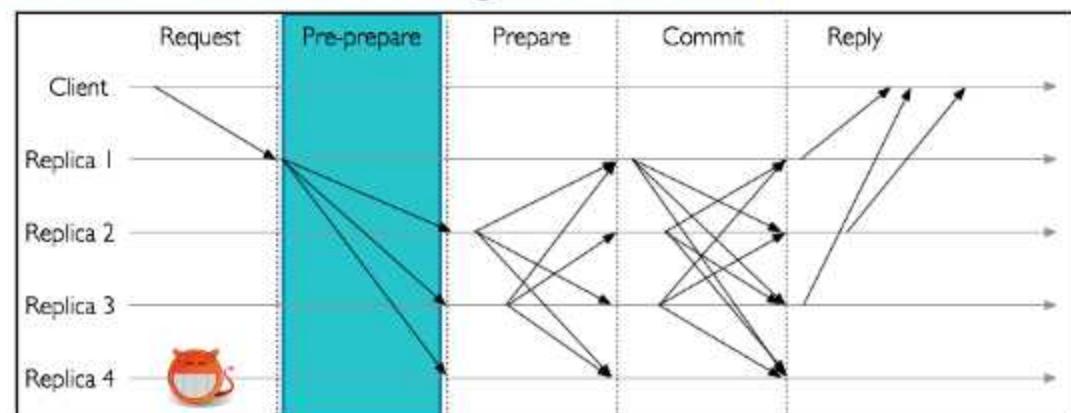
broadcast the block  
to all the replicas



# Practical Byzantine Fault Tolerance

Pre-prepare phase:

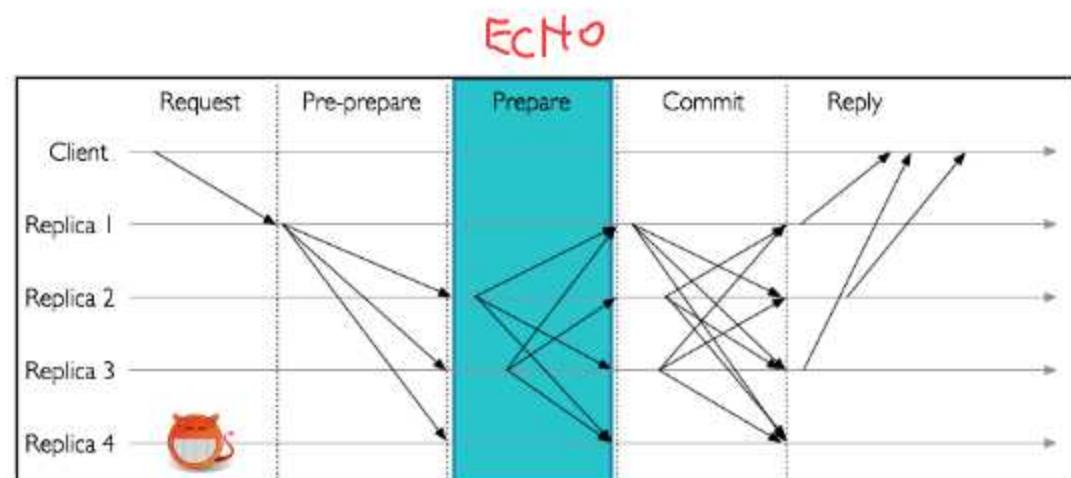
- a backup accepts (puts in the log) the pre-prepare msg iff:
  1. the signature in the pre-prepare and request messages are verified, and  $d$  is  $m$ 's digest
  2. it is currently in view  $v$
  3. it has not accepted a pre-prepare message for view  $v$  and sequence number  $n$  containing a different digest
  4. the sequence number  $n$  is between a low watermark  $h$  and a high watermark  $H$



# Practical Byzantine Fault Tolerance

## Prepare phase:

- a backup  $i$  sends to all nodes the message  $\langle \text{PREPARE}, v, n, d, i \rangle_{\sigma_i}$
- another node accepts a prepare message (and puts it in the log) iff:
  1. signatures are correct
  2. it is currently in view  $v$
  3.  $n$  is between  $h$  and  $H$



# Practical Byzantine Fault Tolerance

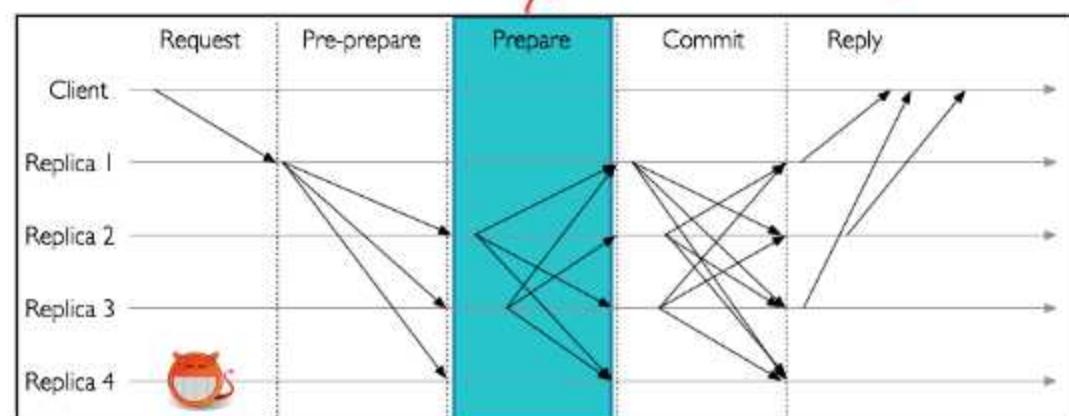
Replicas remain blocked until the predicate  $prepared(m, v, n, i)$  becomes true

This happen iff replica  $i$  has in its log:

1. the message  $m$
2. a pre-prepare for  $m$
3. More than  $2f$  prepare messages from different backups that match the pre-prepare

matching among messages is done checking  $v$ ,  $n$  and  $d$

quorum of enough ErMO



# Practical Byzantine Fault Tolerance

---

The pre-prepare and prepare phase guarantee that non-faulty replicas agree on a total order of execution for requests in a same view

More precisely, they ensure that if  $\text{prepared}(m, v, n, i)$  is true then  $\text{prepared}(m', v, n, j)$  is false for any non faulty replica  $j$  and any  $m'$  such that the digest of  $m$  is different from the digest of  $m'$

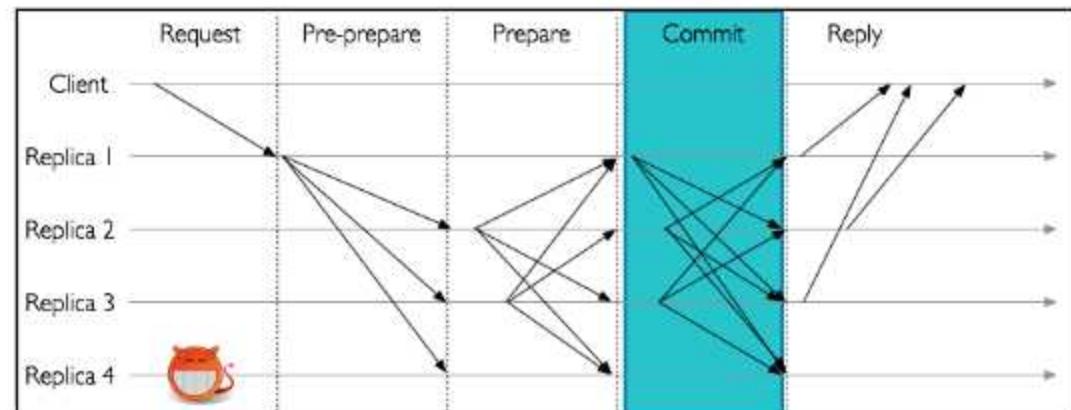
This is due to the intersection among quorums of nodes that accept the prepare messages

# Practical Byzantine Fault Tolerance

## Commit phase:

- as soon as  $prepared(m, v, n, i)$  becomes true replica  $i$  sends to all the message  $\langle COMMIT, v, n, d, i \rangle_o$
- another node accepts a commit message (and puts it in the log) iff:
  - signatures are correct
  - it is in view  $v$
  - $n$  is between  $h$  and  $H$

Double Echo



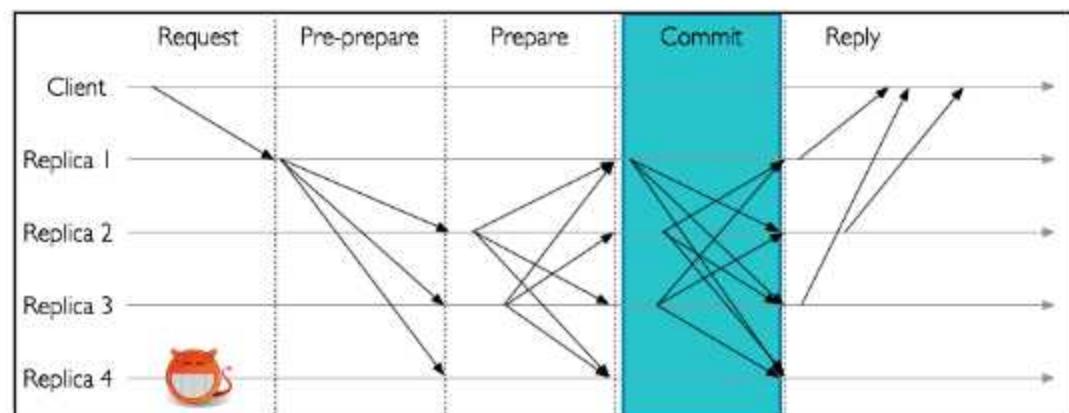
# Practical Byzantine Fault Tolerance

The predicate  $committed(m, v, n)$  is true iff  $prepared(m, v, n, i)$  is true for all  $i$  in a set of  $f+1$  non faulty replicas.

The predicate  $committed-local(m, v, n, i)$  is true iff  $prepared(m, v, n, i)$  is true and  $i$  has accepted  $2f+1$  commits from different replicas that match the pre-prepare for  $m$

The commit phase guarantees that if  $committed-local(m, v, n, i)$  is true for some non faulty  $i$ , then  $committed(m, v, n)$  is true.

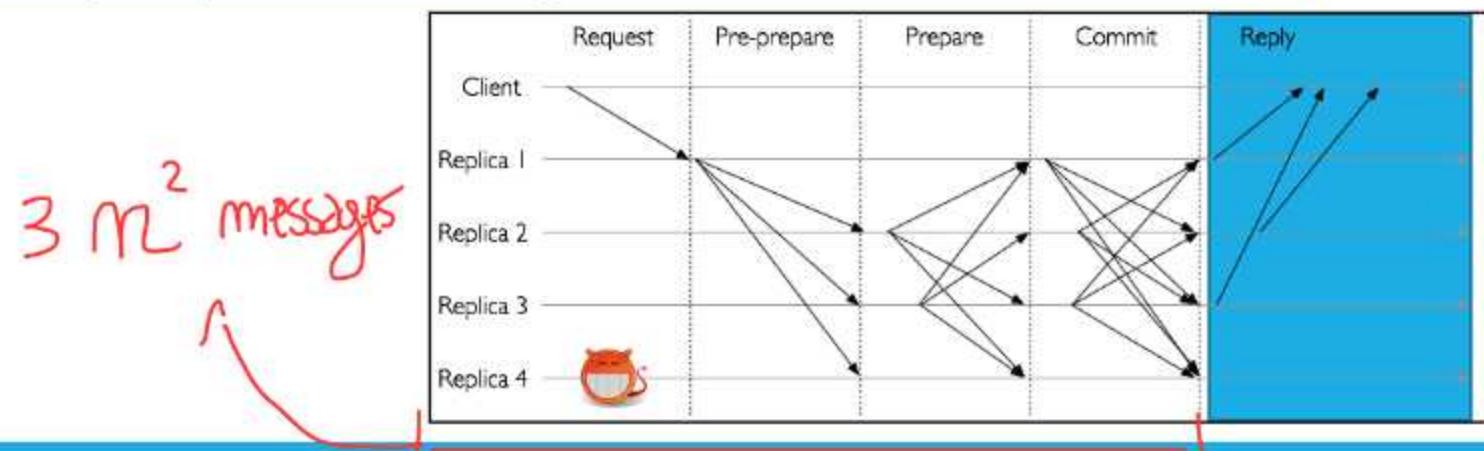
- because if  $i$  received  $2f+1$  commits,  $prepared(m, v, n, i)$  is true for at least  $f+1$  non-faulty nodes.



# Practical Byzantine Fault Tolerance

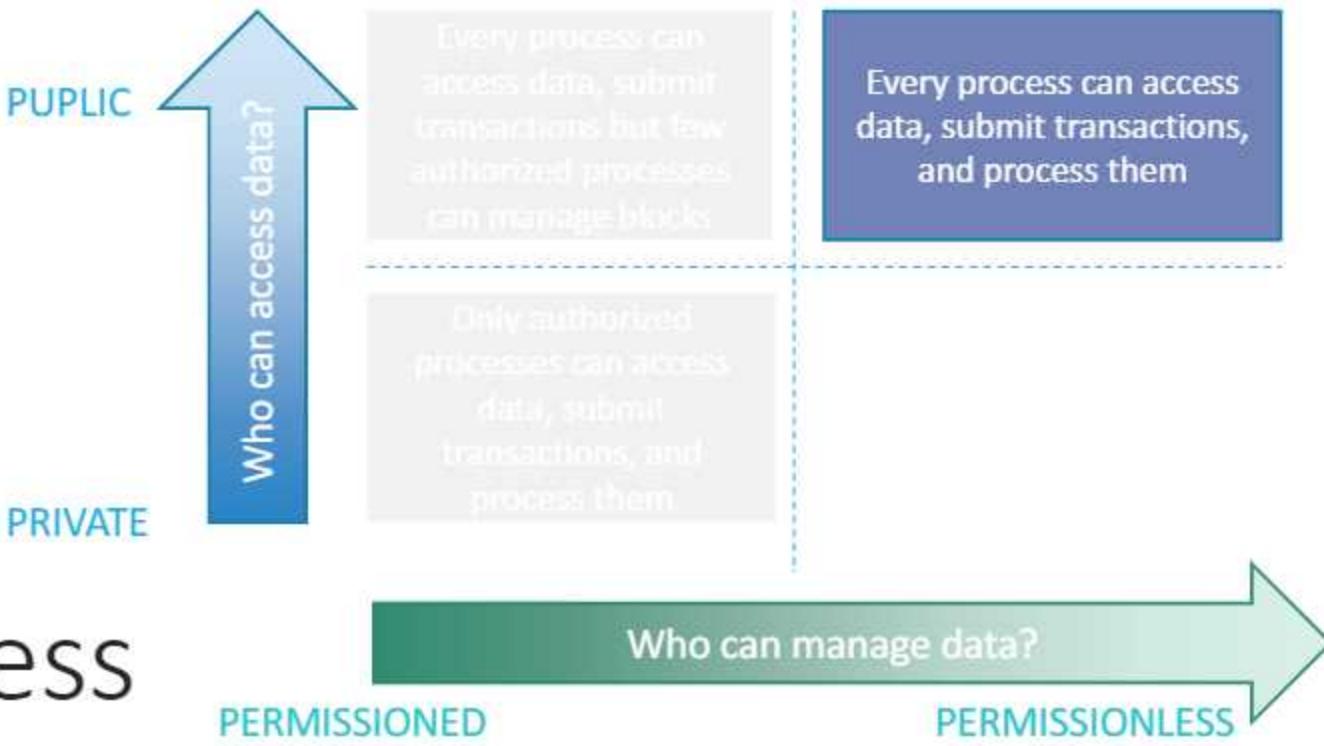
## Reply phase:

- A replica  $i$  executes the request as soon as
  - $\text{committed-local}(m, v, n, i)$  is true
  - $i$ 's state reflects the sequential execution of all requests
- The response is then sent to the client  $\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma_i}$
- The client waits for  $f+1$  replies with valid signatures and the same values for  $t$  and  $r$



# Public Permissionless Blockchain

PROOF OF X



→ Lack of trust, protocol studied before doesn't work, was based on collaboration, here number of adversary is not limited to a small

# Public Permissionless Blockchain

In public permissionless blockchain we have that

- Transactions can be submitted by everyone and every process can read the chain
- Blocks can be created and attached by every process in the system

→ can't estimate value of

limited number

can estimate  
a quorum

use a competition

## OBSERVATION

- Public permissioned blockchains are characterized by
  - Lack of trust in other processes
  - Possibly large scale

## CONSEQUENCE

- PBFT-like protocols do not work

# Public Permissionless Blockchain

## IDEA

- Processes start a competition and only the winner can attach its block to the blockchain (i.e., they are trying to elect a leader)
- They implement a "randomized leader election"

## ISSUES

- Two processes may win the competition

no guarantee that leader  
is unique

because of randomness



→ impossible to achieve a deterministic leader election, because of asynchronous and attackable, lack of trust

- Two blocks can be attached to the chain

low probabilities that more processes win the competition

For becoming a leader have to prove that i invested resources for solve block

# Proof-of-work (PoW) & Mining

- **Proof-of-work:** mathematical challenge to “solve” a block (proposed) in blockchain of blocks
- **Mining:** find a number s.t.  $\text{hash}(\text{block}) < \text{target}$
- The first node who “solves” a **block** can propose it as next in the blockchain
- Other nodes which receive the block re-compute the hash to check the validity

hashed with all  
blocks, contain  
a certain number of zeros.

New Block	
prev block:	#78A...
transactions:	txn 839... txn a76... txn 91c... txn 383...
...	
random number (guess):	30282937

}

execute an algorithm, cryptographic  
puzzle with a type of brute force  
that find the number

$$f(\text{block}) < \text{target}$$

Cryptographic Hash (SHA256)

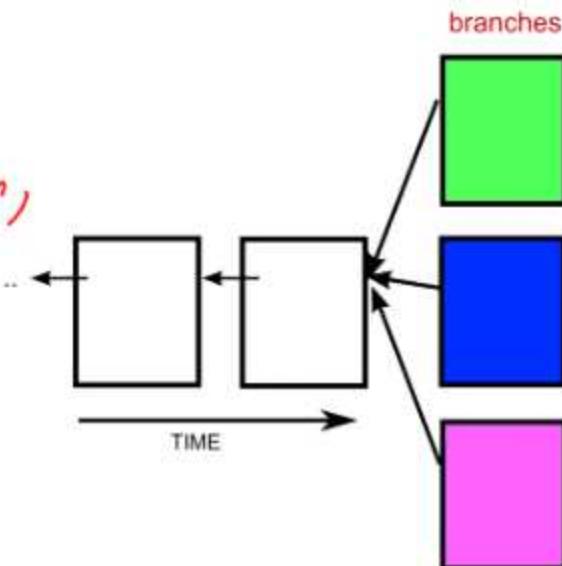
bad processes is treated equal to other,  
only depend on physical resources

# Proof-of-work & Mining: Branches Management

block chain fork problem

- Occasionally two or more blocks may arrive together → temporal disagree
  - The probability that two or more nodes mine a block at the same time is very low
- RULE:** in case of branches the network has to converge to the longest branch.

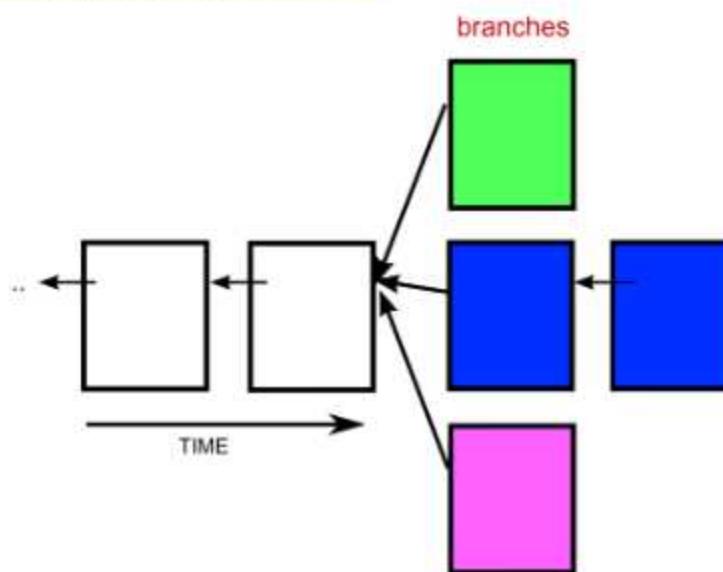
every process has all  
the last block of chain,  
start a competition



not all blocks are accepted  
by the nodes, all nodes  
will have different chain,  
cut all chain shorter in  
length

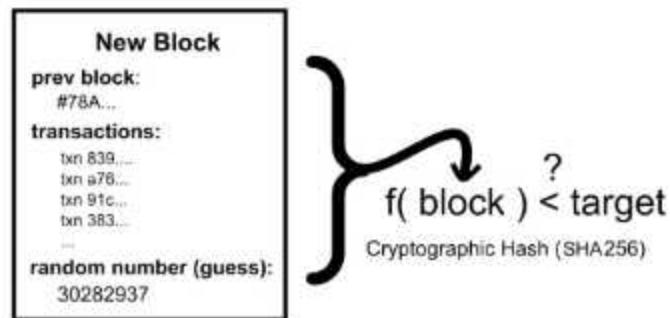
# Proof-of-work & Mining: Branches Management

- Occasionally two or more blocks may arrive together → **temporal disagree**
  - The probability that two or more nodes mine a block at the same time is very low
- **RULE:** in case of branches the network has to converge to the longest branch.
  - **Problem solved with next blocks: eventually one branch will become the longest (usually after one block) bringing convergence**



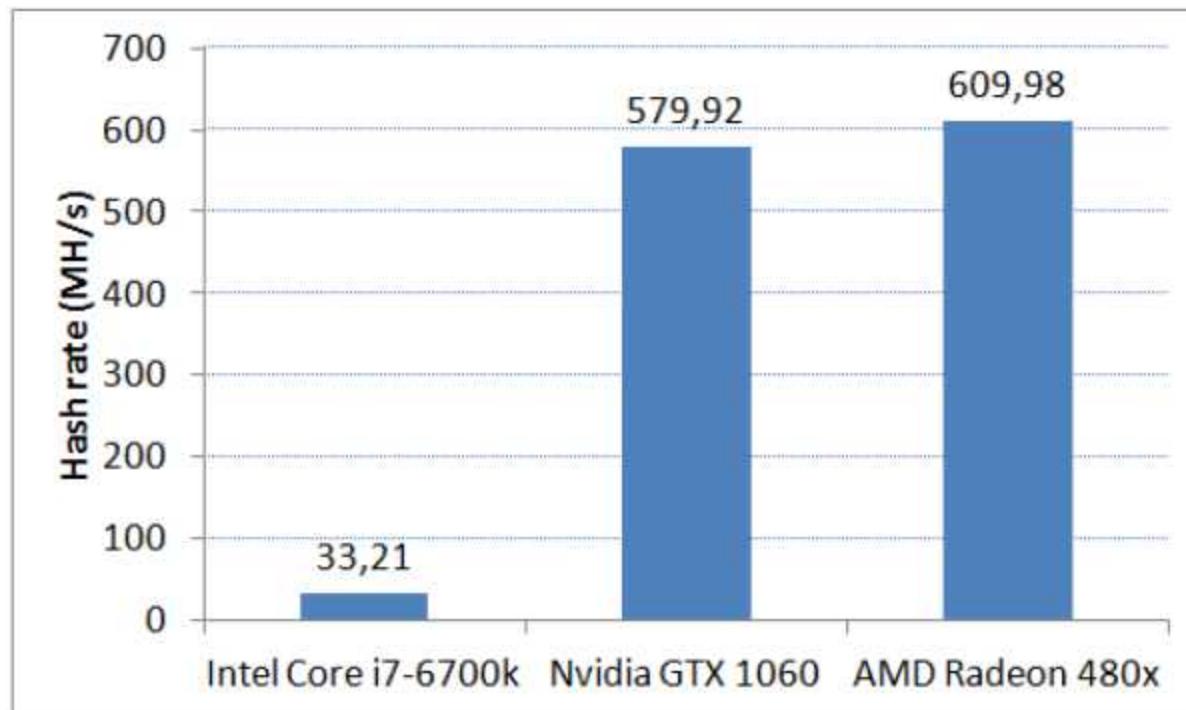
# Scalability Issues

- Transaction rate depends on two parameters
  - Block size: how many transactions to include in a block?
  - Block interval: how long to wait for a block to propagate to all the nodes?
    - Change run-time the difficult of the target to solve a block according to the computational power of the network
- Metrics
  - Throughput: how many transactions per second?
  - Latency: how long to wait for a transaction to complete?
- How parameters impact on metrics
  - Increasing block size improves throughput, but the resulting bigger blocks take longer to propagate in the network
  - Reducing the block interval reduces latency, but leads to instability where the system is in disagreement and the blockchain is subject to reorganization



# Miner Requirements

- A miner needs a high computational power to compute a thousands of hash per seconds (KH/s or MH/s)
- CPUs are ineffective (few cores) better using GPUs



# Miner Requirements

- Possible to improve performances by employing a cluster of GPU



**KADA 6.1 GPU Mining Rig  
Open Air Frame Case Chassis  
with 6 USB Risers - Ethereum**

99.6 % Positive feedback (★ ★ ★ ★ ★)

**235 \$**

**Listing Status: Completed**

**Country: US**

**Item condition: New**

**Buy Now!**

ebay



PayPal

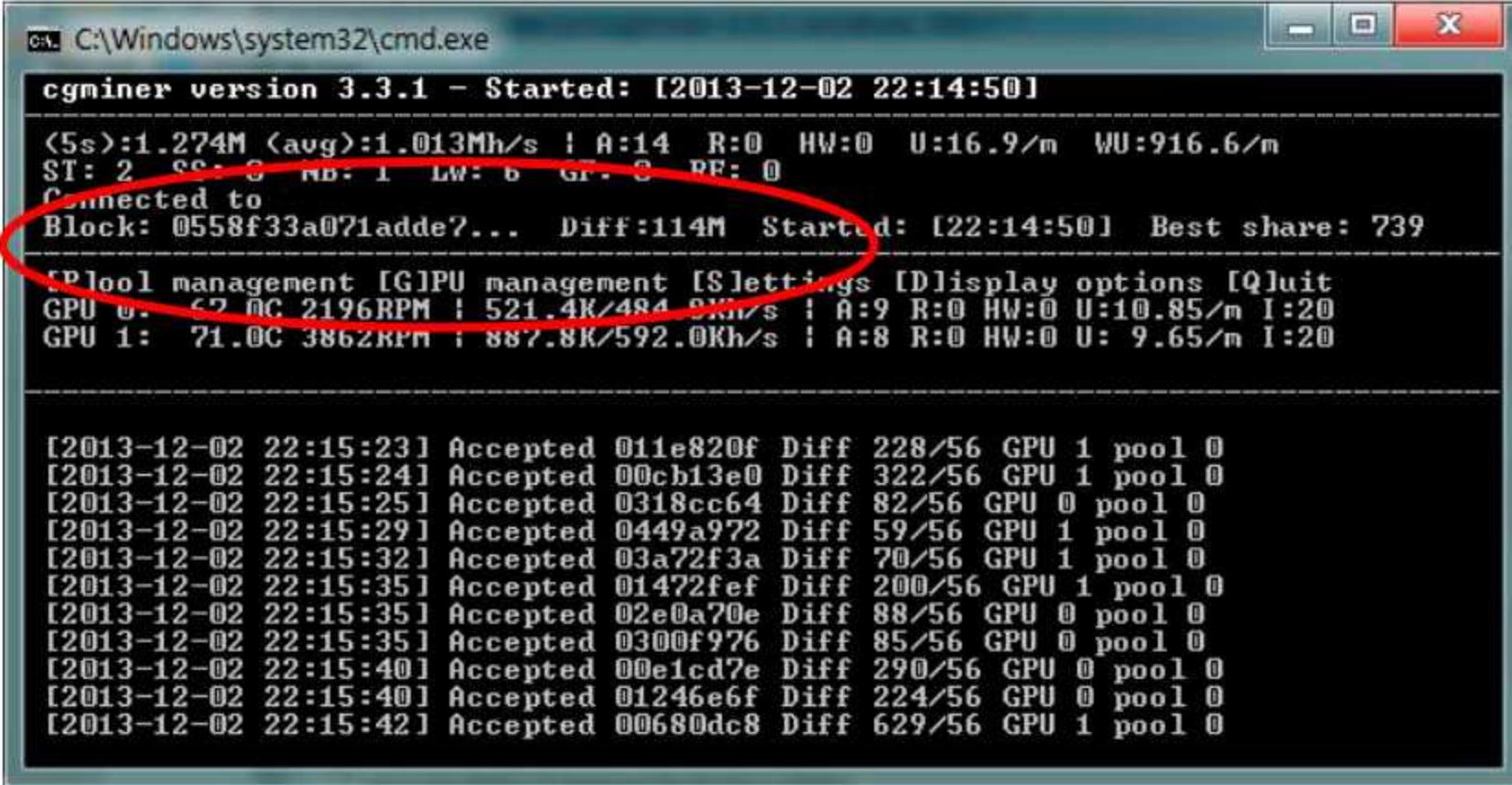
VISA

MasterCard

Discover

# Software for Mining

- CGminer, BFGminer, BitMiner, BTMiner, DiabloMiner, ...



```
C:\Windows\system32\cmd.exe
cgminer version 3.3.1 - Started: [2013-12-02 22:14:50]
<5s>:1.274M <avg>:1.013Mh/s | A:14 R:0 HW:0 U:16.9/m WU:916.6/m
ST: 2 SS: 3 MB: 1 LW: 6 GR: 3 PF: 0
Connected to
Block: 0558f33a071adde?... Diff:114M Started: [22:14:50] Best share: 739
[E]pool management [G]PU management [S]ettings [D]isplay options [Q]uit
GPU 0: 62 OC 2196RPM : 521.4K/484.0Kh/s | A:9 R:0 HW:0 U:10.85/m I:20
GPU 1: 71.0C 3862RPM : 887.8K/592.0Kh/s | A:8 R:0 HW:0 U: 9.65/m I:20

[2013-12-02 22:15:23] Accepted 011e820f Diff 228/56 GPU 1 pool 0
[2013-12-02 22:15:24] Accepted 00cb13e0 Diff 322/56 GPU 1 pool 0
[2013-12-02 22:15:25] Accepted 0318cc64 Diff 82/56 GPU 0 pool 0
[2013-12-02 22:15:29] Accepted 0449a972 Diff 59/56 GPU 1 pool 0
[2013-12-02 22:15:32] Accepted 03a72f3a Diff 70/56 GPU 1 pool 0
[2013-12-02 22:15:35] Accepted 01472fef Diff 200/56 GPU 1 pool 0
[2013-12-02 22:15:35] Accepted 02e0a70e Diff 88/56 GPU 0 pool 0
[2013-12-02 22:15:35] Accepted 0300f976 Diff 85/56 GPU 0 pool 0
[2013-12-02 22:15:40] Accepted 00e1cd7e Diff 290/56 GPU 0 pool 0
[2013-12-02 22:15:40] Accepted 01246e6f Diff 224/56 GPU 0 pool 0
[2013-12-02 22:15:42] Accepted 00680dc8 Diff 629/56 GPU 1 pool 0
```

# ASIC

---

- ASIC: alternative specific hardware for mining
- Example: AntMiner S5
  - Hashrate: 1.15GH/s
  - Price: \$370
  - Power consumption: 590W
    - Mining for 24 hours/day is expensive!

**Question:**  
*Why should a user mine?*



↳ speculation of support Blockchain

# Miner Incentive

---

- Reward: Solving a block gives coins to node that found the proof-of-work
  - Some txn may bring additional fee to node who mines the block containing that txns
  - Currently the txn fees are quasi-zero
  - Miners will be motivated to include in a block txn with a fee
- Rewards are an incentive for nodes to keep them supporting the blockchain and keep nodes honest
  - Invalid txns won't give to miner the reward!
- Furthermore, it is a way to distribute coins into circulation
  - There is no central authority issuing new coins
  - Each crypto-currency platform will not erogate new coins to miners forever → currency deflation
    - Is effective to mine whilst you earn more money than those spent for electric power

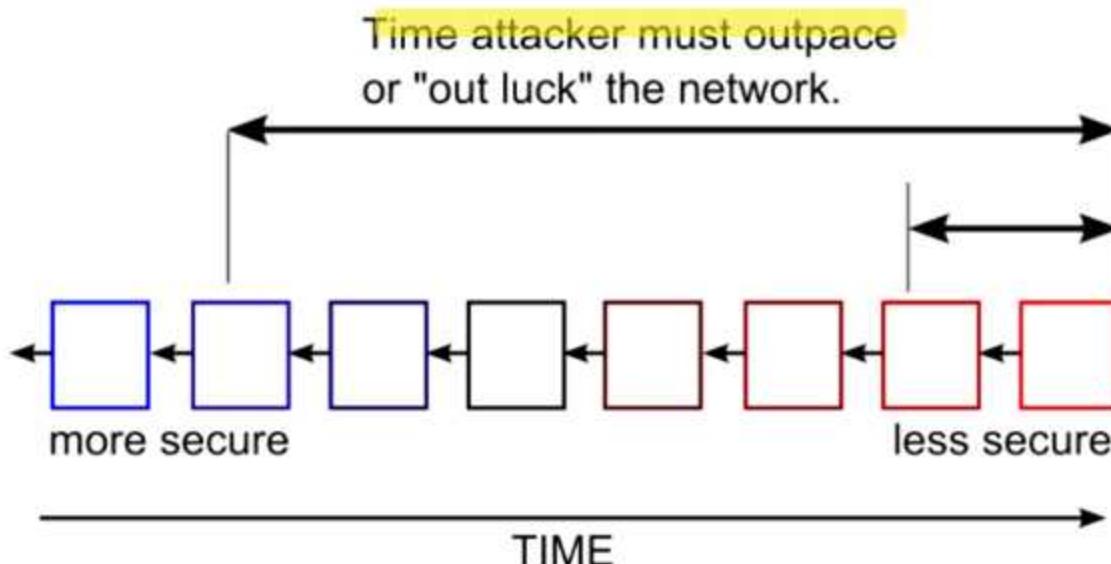
# How the PoW Ensures Integrity: Computational Power

---

- **Impossible to change a txn in a block  $b$  in the blockchain:** an attacker should be quicker than the rest of the whole network to mine a block.
- In that case he could be able to re-mine  $n+1$  blocks (i.e. all blocks next to  $b+1$ ) quicker than the rest of the network to mine a new block.
- If so, **the attacker could obtain the longest** (modified) **blockchain** and all network would converge to it.
- But for doing that, he would have the 50% of the computational power of the network to have a 50% probability to solve a block before another node.
- And he would have a higher percentage of computational power to solve more blocks sequentially.

# How Blockchain Ensures Integrity: Computational Power

- Last blocks are so less secure
- Wait for 5/6 blocks makes a success probability too low for an attacker
- This solution protects from both Integrity and Double Spending Fraud



# Alternative to PoW?

---

- PoW pro: very secure
- PoW cons: waste of electric power
- Proof-of-Stake (PoS) is the PoW alternative
  - Secure without mining → no energy wasted

# Proof-of-Stake

---

- Instead of mine a block, the creator of the next block is chosen in a deterministic way according to its wealth (i.e. stake)
- The reward are not related to the created block but according to your wallet
- The longer you keep the coin in the wallet, the more the reward is high
- The probability to *mint* (instead of mine) a block is proportional to your wallet
  - Minting process require a lot of coin to attack the network
  - If you have  $p\%$  of coins of the network you will mint  $p\%$  of the blocks
  - Very difficult to mint two consecutive blocks!