# Distributed Systems and Peer-to-Peer Systems
# SDR 3.6

Martin Quinson <martin.quinson@loria.fr>

LORIA – M2R

2009-2010
(compiled on: January 19, 2010)

# Introduction

## Course Goals

- Introduce existing distributed systems, from a theoretical point of view
  - Basic concepts
  - Main issues, problems and solutions

## Prerequisite

- Notions of Theoretical Distributed Algorithmic (models, some algos)
- Notions of Distributed Programming (BSD sockets, CORBA, java RMI, J2EE)

## Motivations

- Distributed Systems more and more mainstream
- Interesting algorithmic issues
- Very active research area

# Administrativae

## Contents

- Quick recap of distributed algorithmic and Internet
- Present several innovative distributed systems
- Introduce some current research issues in distributed computing

## Evaluation: test on desk *(partiel)*

- What: quiz about the lectures
  - *Know* the algorithms introduced in lectures
  - Be able to *recognize* principle of classical algorithm designs
  - Be able to *discuss* the validity of an approach to a problem
- When: someday in feb or march (check ADE agenda)
- Allowed material during test: one A4 sheet of paper only
  - Hand-written (not typed)
  - From you (no photocopy)

# About me

## Martin Quinson

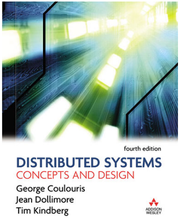
- Study: Université de Saint Étienne, France
- PhD: Grids and HPC in 2003 (team Graal of INRIA / ENS-Lyon, France)
- Since 2005:
  - Assistant professor at ESIAL (Univ. Henri Poincaré–Nancy I, France)
  - Researcher of AlGorille team of LORIA/INRIA
- Research interests:
  - Context: Distributed Systems
  - Main: Simulation of Distributed Applications (SimGrid project)
  - Others: Experimental Methodology, Model-Checking, ...
- More infos:
  - `http://www.loria.fr/~quinson`
  - `Martin.Quinson@loria.fr`

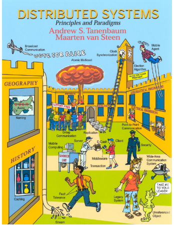
# References: Courses on Internet

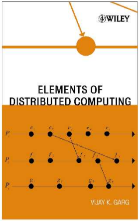
- Algorithmique et techniques de base des systèmes répartis (S. Krakowiak)
  Foundations of distributed systems (in French).
  `http://sardes.inrialpes.fr/~krakowia/Enseignement/M2R-SL/SR/`
- Distributed Systems (Shenker, Stoica; University of California, Berkley)
  A bit of everything, emphasis on Brewer's conjecture.
  `http://inst.eecs.berkley.edu/~cs194`
- Peer-to-Peer Networks (Jussi Kangasharju)
  Peer-to-peer systems.
  `http://www.cs.helsinki.fi/u/jakangas/Teaching/p2p-08f.html`
- Advanced Operating Systems (Neeraj Mittal)
  Very good presentation of the theoretical foundations.
  `http://www.utdallas.edu/~neerajm/cs6378f09`
- Grid Computing WS 09/10 (E. Jessen, M. Gerndt)
  Grid and Cloud computing.
  `http://www.lrr.in.tum.de/~gerndt/home/Teaching/WS2009/GridComputing/GridComputing.htm`

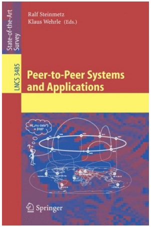
# References: Books

- Coulouris, Jean et Kindberg. Distributed Systems: Concepts and Design.
- Tannenbaum, Steen. Distributed Systems: Principles and Paradigms.
- V. K. Garg. Elements of Distributed Computing.
- Ralf Steinmetz, Klaus Wehrle (Eds): Peer-to-Peer Systems and Applications. http://www.peer-to-peer.info/



# Table of Contents

## Part I: History of Distributed Systems

1. Introduction to Distributed Systems
   - What is it? Research Agendas and Communities; Examples.
2. Distributed Algorithmic
   - Time and state; Ordering events; Abstract Clocks; Classical Algorithms.
3. Internet
   - OSI and TCP/IP; Design of Internet; Some Mecanisms; Brewer revisited.

## Part II: Innovative Distributed Systems

1. Peer-to-Peer Systems
   - Introduction; Unstructured Overlays; DHTs; Applications; Hot Research Topics.
2. SensorNets
   - Presentation; State of the field.

# Chapter 1

## Introduction

- What is a Distributed System?
- Example of Distributed Systems
- Limit between Computers and Distributed Systems

# What is a distributed system?

## Definition

*A distributed system is a collection of independent computers that appear to the users of the system as a single computer.*

— *A. Tanenbaum.*

⇝ Set of elements (CPU, storage) interconnected by the network


- The set is more than the sum of its parts (elements do collaborate)
- Intuitive examples not from CS
  - Ant nest
  - Driving rules (cars share the road)

# What is a distributed system?

## Definition (optimistic)

*A distributed system is a collection of independent computers that appear to the users of the system as a single computer.*

— *A. Tanenbaum.*

$\rightsquigarrow$ Set of elements (CPU, storage) interconnected by the network


- The set is more than the sum of its parts (elements do collaborate)
- Intuitive examples not from CS
  - Ant nest
  - Driving rules (cars share the road)

## Definition (pessimistic)

*You know you have one when the crash of a computer you never heard of stops you from getting any work done.* — *L. Lamport.*

- Interdepending behavior of elements
  - That's not that easy
  - Failures do happen and must be dealt with

# Why would you distribute your computer system??

Application needs: you sometimes have to

- Collaborative work (between human beings, between corporate facilities)
- Distributed electronic devices ⇒ *Ubiquitous Computing* and *SensorNets*
- Application integration (multi-physics simulation) ⇒ *Grid Computing*

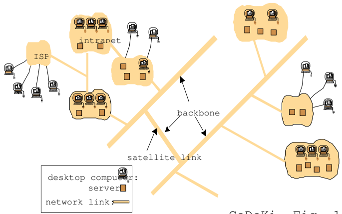
Technical possibility creates the need

- Cost effectiveness
  - A set of PC is less expensive than a big mainframe ⇒ *Cluster Computing*
  - Scale savings of mesocenter (wrt than several clusters) ⇒ *Cloud Computing*
- Generalized interconnections (TV, Internet, phone are converging)
  - Share storage resources ⇒ *Peer-to-Peer systems*
  - Share (otherwise unused) computational resources ⇒ *Volunteer Computing*

# Example of Distributed Systems (1/2)

## The Internet: the network of networks

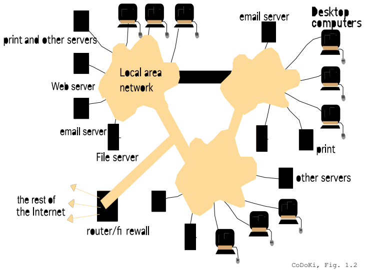
- Enormous (open ended)
- No single authority
  (mapping internet is a research agenda)
- Data, audio, video; Requests, push, streams.



CoDoKi, Fig. 1.1

## Intranets

- A single authority
- Protected access
  (firewall, encrypted channels, total isolation)
- May be worldwide

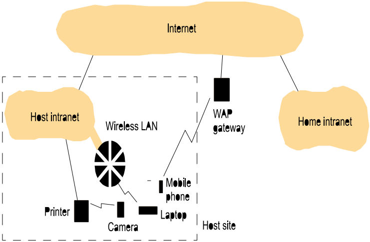


CoDoKi, Fig. 1.2

# Example of Distributed Systems (2/2)

## Mobile and Ubiquitous Computing

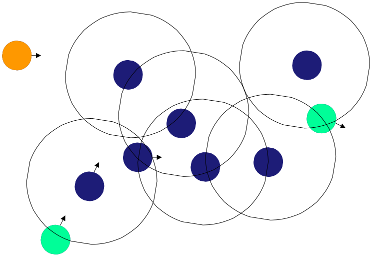
- Portable devices
  - laptops, notebook
  - handheld, wearable devices
  - devices embedded in appliances
- Mobile computing
- Connected to Internet through fixed infrastructure



( CoDoKi Fig 1.3 )

## Mobile Ad-hoc Networks (Manets)

- No fix infrastructure
  - wireless communication
  - multi-hop networking
  - long, non deterministic delays
  - ⤳ nodes part of infrastructure
- Nodes come and go



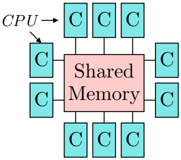
# Limit between Computers and Distributed Systems

## Why is this limit blurred?

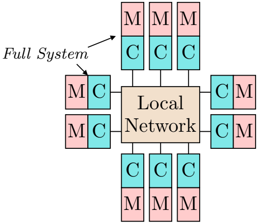
- Motivation: endless need for power (modeling/game realism, server scalability)
- Past solution: Increase clock speed, more electronic gates
  (but reaching physical limits + speed linear vs. energy quadratic)
- Current trend: Multi-many (Multiply cores, processors and machines)

## Multi-processors systems

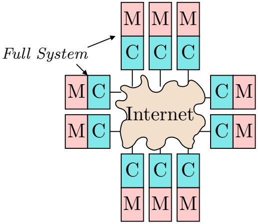
### Shared Memory Processor (SMP)




### Cluster System

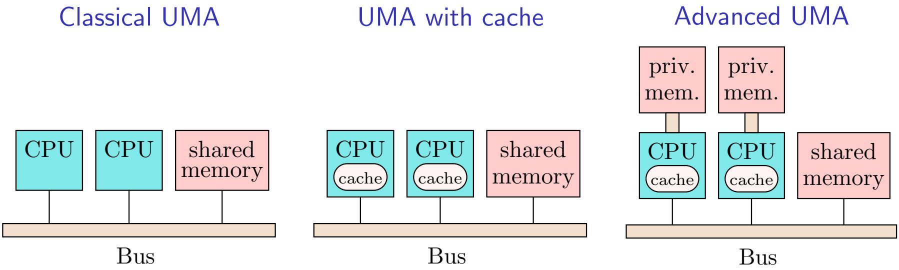



### Distributed Systems




- SMP communicate through shared memory
- Clusters and DS communicate through classical network

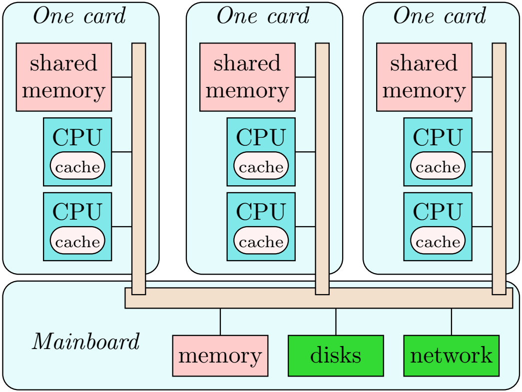
# Some SMPs are UMA (Uniform Memory Access)




- Every processor access the memory at the same speed
- But memory to slow in classical design, thus adding a cache
- Can go further by adding a private memory to each processor

# NUMA: NON-uniform Memory Access

- Biggest challenge: feed CPU with data (memory slower than CPU)
- Idea: Put several CPU per board, and plug boards on mainboard



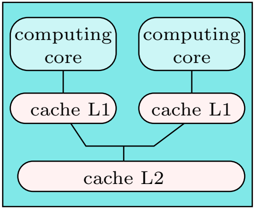

## Issue

- Memory access is non-uniform (slower when far away)
  Need specific programming approach to keep efficient
- Cache consistency can turn into a nightmare

# Multi-core: Parallelism on Chip

- Idea: Reduce distance to elements (thus latency)
- How: Put several computing elements on the same chip

AMD/Intel bicore chips



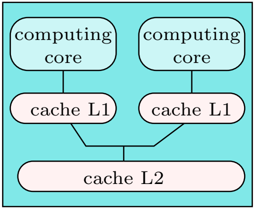

## Current and Future Trends

- Put more and more cores on chip
  (80 cores already prototyped, full Cluster-On-Chip envisioned)
- Increase Architecture Hierarchy (Clusters of NUMA of multi-cores)

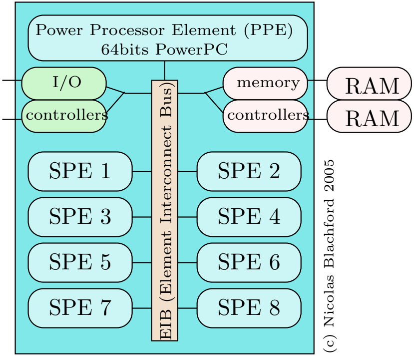
# Multi-core: Parallelism on Chip

- Idea: Reduce distance to elements (thus latency)
- How: Put several computing elements on the same chip

AMD/Intel bicore chips




Cell Processor




## Current and Future Trends

- Put more and more cores on chip
  (80 cores already prototyped, full Cluster-On-Chip envisioned)
- Increase Architecture Hierarchy (Clusters of NUMA of multi-cores)
- Even put non-symmetric cores: PPE is classical RISC, SPE are SIMD

# Distributed, Parallel or Concurrent??

## Distributed Algorithm: $\frac{\textit{computation time}}{\textit{communication time}} \leadsto 0$

- Computation negligible wrt to communications
- Classical metric: amount of messages (as a function of amount of nodes)

## Parallel Algorithm: $\frac{\textit{computation time}}{\textit{communication time}} \approx 1$

- Computation and Communication comparable
- Classical metric: makespan (time to completion of last processor)

## Concurrent Algorithm: $\frac{\textit{computation time}}{\textit{communication time}} \leadsto \infty$

- Communication negligible wrt computation ($\textit{comm time} = 0 \Rightarrow$, multi-threading)
- Classical metric: speedup (how faster when using N cpus)

# Distributed, Parallel or Concurrent??

Distributed Algorithm: $\frac{\textit{computation time}}{\textit{communication time}} \leadsto 0$

- Computation negligible wrt to communications
- Classical metric: amount of messages (as a function of amount of nodes)

Parallel Algorithm: $\frac{\textit{computation time}}{\textit{communication time}} \approx 1$

- Computation and Communication comparable
- Classical metric: makespan (time to completion of last processor)

Concurrent Algorithm: $\frac{\textit{computation time}}{\textit{communication time}} \leadsto \infty$

- Communication negligible wrt computation ($\textit{comm time} = 0 \Rightarrow$, multi-threading)
- Classical metric: speedup (how faster when using N cpus)

Focus of this course: distributed systems (some content applies to others)

# Distributed, Parallel or Concurrent??

**Distributed Algorithm:** $\frac{\textit{computation time}}{\textit{communication time}} \rightsquigarrow 0$

- Computation negligible wrt to communications
- Classical metric: amount of messages (as a function of amount of nodes)
- Current research agenda: P2P, consistency (distributed DB)

**Parallel Algorithm:** $\frac{\textit{computation time}}{\textit{communication time}} \approx 1$

- Computation and Communication comparable
- Classical metric: makespan (time to completion of last processor)
- Current research agenda: Cluster & Grid & Cloud Computing, interoperability

**Concurrent Algorithm:** $\frac{\textit{computation time}}{\textit{communication time}} \rightsquigarrow \infty$

- Communication negligible wrt computation ($\textit{comm time} = 0 \Rightarrow$, multi-threading)
- Classical metric: speedup (how faster when using N cpus)
- Current research agenda: Lock-free, wait-free, correctness (model-checking)

**Focus of this course: distributed systems** (some content applies to others)

- Each domain constitutes a huge research area
- Current trend: intermixing, but strong historical heritage

# What to expect from a distributed system?

## Expected characteristics

- Scalability: deal with large amount of work
- Failure tolerance:
  - Deal with the failure of elements
  - Deal with message loss, or element performance degradation
- Security: Deal with malicious users (Privacy, Integrity, Deny-of-Services)
- Adaptability: deal with environment changes

## Expected difficulties

- Absence of Global Clock: there is no common notion of time
- Absence of Shared Memory: no process has up-to-date global knowledge
- Failures (fail-stop or malicious): that *will* happen
- Delays (asynchronous): harder to detect failures
- Dynamism: global knowledge even harder to get
- Human brain is (somehow) sequential. Thinking distributed is harder.

# Chapter 2

## Theoretical foundations

- Time and State of a Distributed System
- Ordering of events
- Abstract Clocks
  - Global Observer
  - Logical Clocks
  - Vector Clocks
- Some Distributed Algorithms
  - Mutual Exclusion
    - Coordinator-based Algorithm
    - Lamport's Algorithm
    - Ricart and Agrawala's Algorithm
    - Roucairol and Carvalho's Algorithm
    - Token-Ring algorithm
    - Suzuki and Kasami's Algorithm
  - Leader Election
  - Consensus
  - Ordering Messages
  - Group Protocols
- Conclusion on distributed algorithmic

# Time and State of a Distributed System

Fundamental Goal: think about a system or an application

## What do we need

- Define a state: for example to define predicates
- Define an order: to coordinate the activities

## Why is it harder for Distributed Systems? (Inherent Limitations)

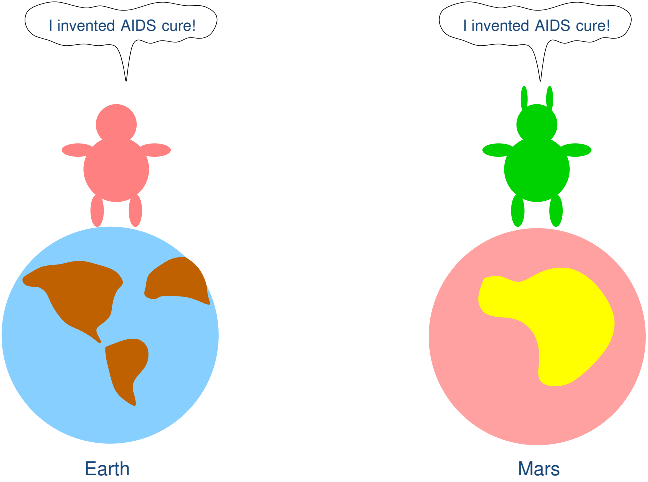
- Absence of Global Clock: There is no common notion of time
- Absence of Shared Memory: No process has up-to-date global knowledge
- Asynchronous communications and computations (generally speaking)
  - Ie, comm/comp time has no maximum
  - Because dynamically changing load and resources not exclusively allocated
  - Synchronous systems (real time, phone) more rare because more expansive

## Goal now

- Define an order relation (used later for global state)
- At the end, that's quite simple, but it needed several years of research

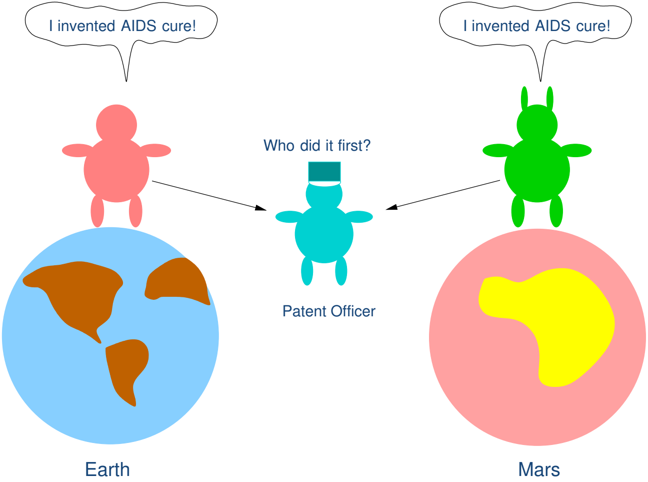
# Absence of Global Clock

Different processes may have different notions of time



# Absence of Global Clock

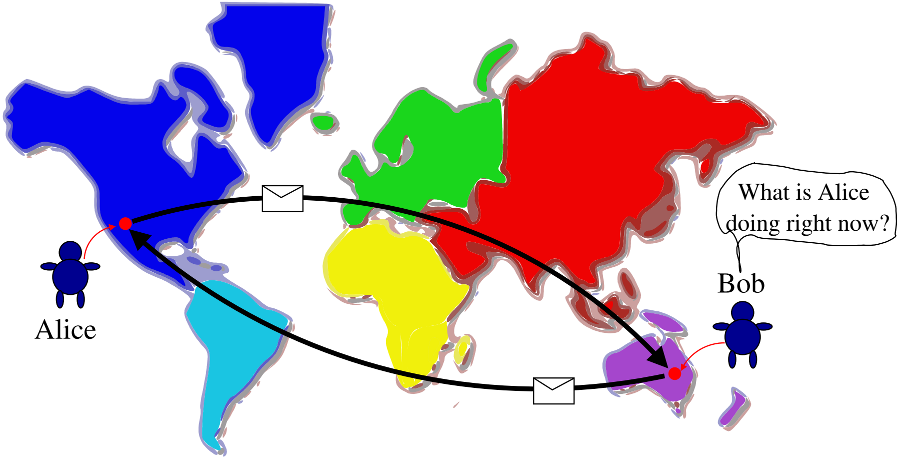
Different processes may have different notions of time




- Problem: How do we order events on different processes?

# Absence of Shared Memory

A process does not know current state of other processes

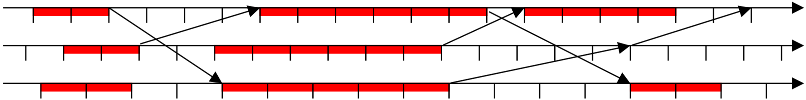



- Problem: How do we obtain a coherent view of the system?

# The Reliable Asynchronous Model

## That's the weaker (reliable) model

- Very strong constraints from the system
  - No upper bound on communication or computation
  - Algorithms working here work also in more friendly models
  - Models made more friendly by removing constraints (setup upper bounds)
  - (that's not the worst model: it is reliable)
- This model is often used for Bounding costs or Impossibility results



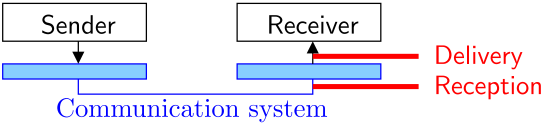
- Each site has a clock (not synchronized, with relative drifts)
- Processes only communicate by message exchanges
- Possible events:
  - Local (process internal state change)
  - Emission or Reception of messages

# About messages

## Properties of the communication system

1. No loss: Every sent message arrives (no upper bound on transit time)
   - How to achieve this: failure detection (with timeout) and resending
2. Messages are not altered
   - How to achieve this: Mechanisms for detection and correction of errors
3. FIFO channel between processes
   - How: message numbering
   - Assumption sometimes removed (⇒ even harder)

## Distinguish message reception and delivering



# Process Execution and Synchronization

## Process Execution

- That's a suite of events (its history, its trace)

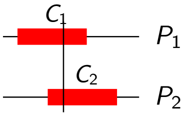
  Recall: kind of possible events = {local, sending, receiving}
- Suite ordered by the local clock
- For $P_1$: $e_1^1, e_1^2, e_1^3, e_1^4, \ldots e_1^k, \ldots$

## "Synchronizing processes"?!

$\rightsquigarrow$ force an order to the events of these processes

- Example: mutual exclusion

$$\text{either} \begin{cases} end(C_2) \textbf{ precedes } begin(C_1) \\ end(C_1) \textbf{ precedes } begin(C_2) \end{cases}$$



# When is it possible to order two events?

## Causality Principle

- The Cause comes before the Effect

## Three Cases:

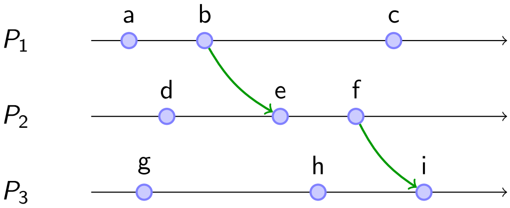
1. Events executed on the same process:
   - if $e$ and $f$ are events on the same process and $e$ occurred before $f$, then $e$ *happened-before* $f$
2. Communication events of the same message:
   - if $e$ is the send event of a message and $f$ is the receive event of the same message, then $e$ *happened-before* $f$
3. Events related by transitivity:
   - if event $e$ happened-before event $g$ and event $g$ happened-before event $f$, then $e$ *happened-before* $f$

# Happened-Before Relation

## Notation

- Happened-before relation is denoted by $\longrightarrow$

## Illustration



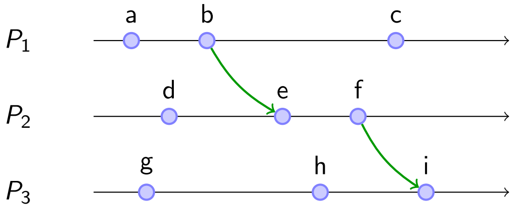

- Events on the same process
  $a \longrightarrow b$, $b \longrightarrow c$, $d \longrightarrow f$
- Events of the same message
  $b \longrightarrow e$, $f \longrightarrow i$
- Transitivity
  $a \longrightarrow c$, $a \longrightarrow e$, $a \longrightarrow i$

# Happened-Before Relation

## Notation

- Happened-before relation is denoted by $\longrightarrow$

## Illustration




- Events on the same process
  $a \longrightarrow b$, $b \longrightarrow c$, $d \longrightarrow f$
- Events of the same message
  $b \longrightarrow e$, $f \longrightarrow i$
- Transitivity
  $a \longrightarrow c$, $a \longrightarrow e$, $a \longrightarrow i$

## Concurrent events

- Events not related by the happened-before relation
- Concurrency relation is denoted by $\parallel$
- Examples: $a \parallel d$, $e \parallel h$, $c \parallel i$,
- Concurrency is not transitive: $a \parallel d$ and $d \parallel c$ but $a \not\parallel c$

# Theoretical foundations

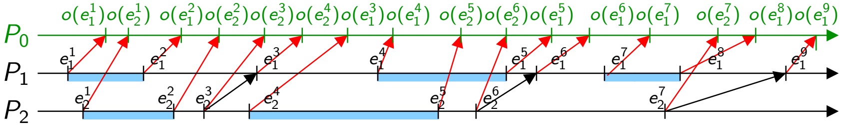

- Time and State of a Distributed System
- Ordering of events
- Abstract Clocks
  - Global Observer
  - Logical Clocks
  - Vector Clocks
- Some Distributed Algorithms
  - Mutual Exclusion
    - Coordinator-based Algorithm
    - Lamport's Algorithm
    - Ricart and Agrawala's Algorithm
    - Roucairol and Carvalho's Algorithm
    - Token-Ring algorithm
    - Suzuki and Kasami's Algorithm
  - Leader Election
  - Consensus
  - Ordering Messages
  - Group Protocols
- Conclusion on distributed algorithmic

# Dating System (for sake of global ordering)

Goal: Dating System compatible with Causality

## First Approach: notion of observation

- A "observer" process $P_0$ is informed by message of every event
- The suite of events as observed by $P_0$ is a global observation
- Later: each process is observer, and observations match



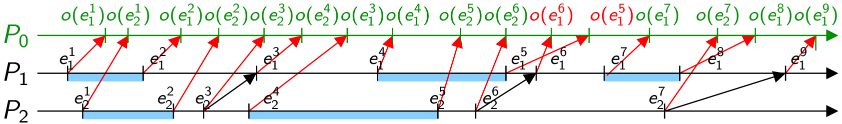
# Validity of Observations

## Definition

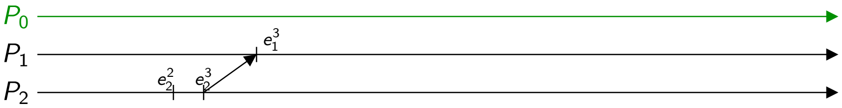
- Observation said valid iff $(e \rightarrow f) \Rightarrow (o(e) \rightarrow o(f))$

## Examples

1. $(e_1^5 \rightarrow e_1^6)$ but $o(e_1^6)$ precedes $o(e_1^5)$




2. $(e_2^2 \rightarrow e_1^3)$ because $(e_2^2 \rightarrow e_2^3)$ and $(e_2^3 \rightarrow e_1^3)$



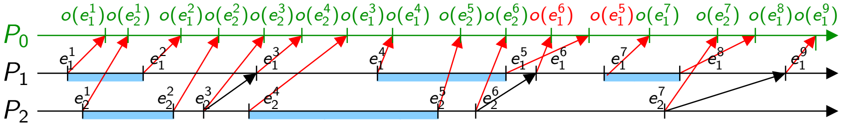
# Validity of Observations

## Definition

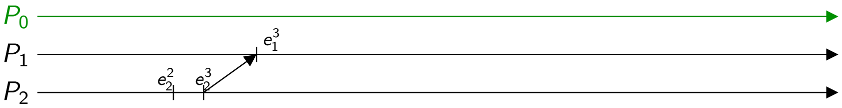
- Observation said valid iff $(e \rightarrow f) \Rightarrow (o(e) \rightarrow o(f))$

## Examples

1. $(e_1^5 \rightarrow e_1^6)$ but $o(e_1^6)$ precedes $o(e_1^5)$ $(P_1 \rightarrow P_0$ is not FIFO$)$




2. $(e_2^2 \rightarrow e_1^3)$ because $(e_2^2 \rightarrow e_2^3)$ and $(e_2^3 \rightarrow e_1^3)$



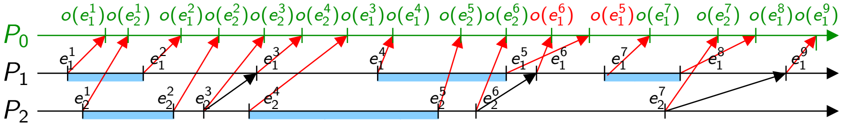
# Validity of Observations

## Definition

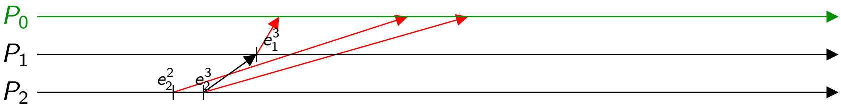
- Observation said valid iff $(e \to f) \Rightarrow (o(e) \to o(f))$

## Examples

1. $(e_1^5 \to e_1^6)$ but $o(e_1^6)$ precedes $o(e_1^5)$ $\qquad$ $(P_1 \to P_0$ is not FIFO$)$




2. $(e_2^2 \to e_1^3)$ because $(e_2^2 \to e_2^3)$ and $(e_2^3 \to e_1^3)$
   but $o(e_1^3)$ can not precede $o(e_2^2)$ even if channels are fifo



# Abstract Clocks

## Setting up an observer is suboptimal

- Expensive: A huge amount of messages must be sent to the observer
- Not robust: What if the observer fails?
- Not reliable: invalid observations are still possible

## Abstract Clocks

- Why: (try to) solve absence of global clock
- How: processes timestamp events locally so that they get globally ordered

## Different kind of abstract clocks

- Each offers differing abilities, associated to differing complexities
- Logical clock: used to totally order all events
- Vector Clocks: used to track happened-before relation
- Matrix Clocks: used to track what other processes know about other processes
- Direct Dependency Clocks: used to track direct causal dependencies

# Logical Clocks (or Lamport's Clock)

## General idea

- Implements the notion of virtual time
- Can be used to totally order all events
- Assigns timestamp $C(e)$ to each event $e$
- Compute $C(e)$ in a way that is consistent with the happened-before relation:

$$\boxed{e \rightarrow f \Rightarrow C(e) < C(f)}$$

- (Note that this is $\Rightarrow$, not $\Leftrightarrow$)

*Time, Clocks and the Ordering of Events in a Distributed System*, Leslie Lamport, 1978.

# Implementing Logical Clocks

- Each process $i$ has a local scalar counter $C_i$ $(\in \mathbb{N})$
- Each even $e$ local to $i$ is dated by the current value of $C_i$
- Each message $m$ sent from $i$ is also annoted with $C_i$ (sending time)

## Computation rules on process $i$

| | |
|---|---|
| Initialization : | $C_i \leftarrow 0$ |
| Local event : | $C_i += 1$ |
| Sending message $(m)$ : | $C_i += 1$ then send $(m, C_i)$ |
| Receiving message $(m, E_m)$: | $C_i \leftarrow max(C_i, E_m) + 1$ |

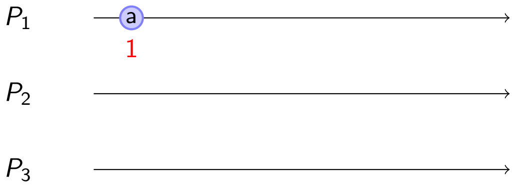
# Implementing Logical Clocks

- Each process $i$ has a local scalar counter $C_i$ ($\in \mathbb{N}$)
- Each even $e$ local to $i$ is dated by the current value of $C_i$
- Each message $m$ sent from $i$ is also annoted with $C_i$ (sending time)

## Computation rules on process $i$

| | |
|---|---|
| Initialization : | $C_i \leftarrow 0$ |
| Local event : | $C_i += 1$ |
| Sending message $(m)$ : | $C_i += 1$ then send $(m, C_i)$ |
| Receiving message $(m, E_m)$: | $C_i \leftarrow max(C_i, E_m) + 1$ |

## Example




- $a$ first event of $P_1 \rightsquigarrow C(a) = 1$

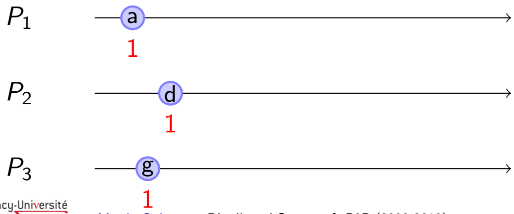
# Implementing Logical Clocks

- Each process $i$ has a local scalar counter $C_i$ ($\in \mathbb{N}$)
- Each even $e$ local to $i$ is dated by the current value of $C_i$
- Each message $m$ sent from $i$ is also annoted with $C_i$ (sending time)

## Computation rules on process $i$

| | |
|---|---|
| Initialization : | $C_i \leftarrow 0$ |
| Local event : | $C_i += 1$ |
| Sending message $(m)$ : | $C_i += 1$ then send $(m, C_i)$ |
| Receiving message $(m, E_m)$: | $C_i \leftarrow max(C_i, E_m) + 1$ |

## Example




- $a$ first event of $P_1 \rightsquigarrow C(a) = 1$

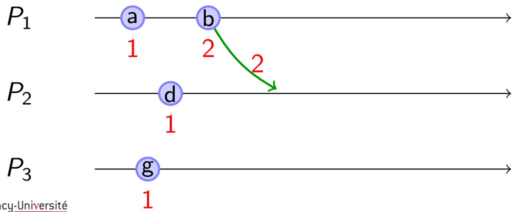
# Implementing Logical Clocks

- Each process $i$ has a local scalar counter $C_i$ ($\in \mathbb{N}$)
- Each even $e$ local to $i$ is dated by the current value of $C_i$
- Each message $m$ sent from $i$ is also annoted with $C_i$ (sending time)

## Computation rules on process $i$

| | |
|---|---|
| Initialization : | $C_i \leftarrow 0$ |
| Local event : | $C_i += 1$ |
| Sending message $(m)$ : | $C_i += 1$ then send $(m, C_i)$ |
| Receiving message $(m, E_m)$: | $C_i \leftarrow max(C_i, E_m) + 1$ |

## Example




- $a$ first event of $P_1 \rightsquigarrow C(a) = 1$
- $b$: send $\rightsquigarrow C_1 := C_1 + 1$; send 2

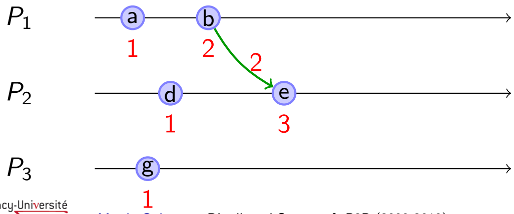
# Implementing Logical Clocks

- Each process $i$ has a local scalar counter $C_i$ ($\in \mathbb{N}$)
- Each even $e$ local to $i$ is dated by the current value of $C_i$
- Each message $m$ sent from $i$ is also annoted with $C_i$ (sending time)

## Computation rules on process $i$

$$\begin{aligned} \text{Initialization}: &\quad C_i \leftarrow 0 \\ \text{Local event}: &\quad C_i += 1 \\ \text{Sending message } (m): &\quad C_i += 1 \text{ then send } (m, C_i) \\ \text{Receiving message } (m, E_m): &\quad C_i \leftarrow max(C_i, E_m) + 1 \end{aligned}$$

## Example




- $a$ first event of $P_1 \rightsquigarrow C(a) = 1$
- $b$: send $\rightsquigarrow C_1 := C_1 + 1$; send 2
- $e$:recv $C(e) = max(1, 2) + 1 = 3$

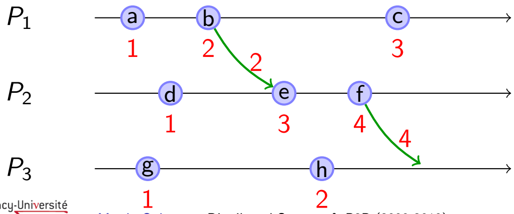
# Implementing Logical Clocks

- Each process $i$ has a local scalar counter $C_i$ ($\in \mathbb{N}$)
- Each even $e$ local to $i$ is dated by the current value of $C_i$
- Each message $m$ sent from $i$ is also annoted with $C_i$ (sending time)

## Computation rules on process $i$

| | |
|---|---|
| Initialization : | $C_i \leftarrow 0$ |
| Local event : | $C_i += 1$ |
| Sending message ($m$) : | $C_i += 1$ then send $(m, C_i)$ |
| Receiving message ($m, E_m$): | $C_i \leftarrow max(C_i, E_m) + 1$ |

## Example




- $a$ first event of $P_1 \rightsquigarrow C(a) = 1$
- $b$: send $\rightsquigarrow C_1 := C_1 + 1$; send 2
- $e$:recv $C(e) = max(1, 2) + 1 = 3$
- $c$, $h$: local events; $f$: send

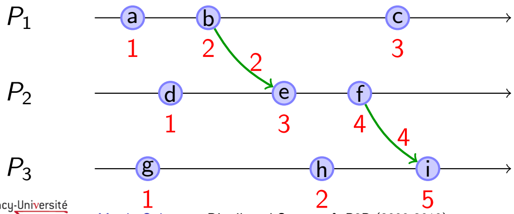
# Implementing Logical Clocks

- Each process $i$ has a local scalar counter $C_i$ ($\in \mathbb{N}$)
- Each even $e$ local to $i$ is dated by the current value of $C_i$
- Each message $m$ sent from $i$ is also annoted with $C_i$ (sending time)

## Computation rules on process $i$

| | |
|---|---|
| Initialization : | $C_i \leftarrow 0$ |
| Local event : | $C_i += 1$ |
| Sending message ($m$) : | $C_i += 1$ then send $(m, C_i)$ |
| Receiving message ($m, E_m$): | $C_i \leftarrow max(C_i, E_m) + 1$ |

## Example




- $a$ first event of $P_1 \rightsquigarrow C(a) = 1$
- $b$: send $\rightsquigarrow C_1 := C_1 + 1$; send 2
- $e$:recv $C(e) = max(1, 2) + 1 = 3$
- $c$, $h$: local events; $f$: send
- i:recv; $C(i) = max(4, 2) + 1 = 5$

# Conclusion on Logical Clocks

## Possible Applications

- Distributed waiting queue (mutual exclusion; replicas update)
- Determine least access (cache coherence, DSM)

# Conclusion on Logical Clocks

## Possible Applications

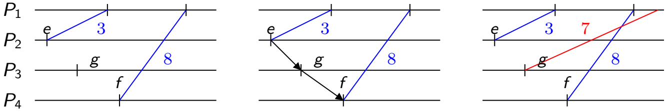
- Distributed waiting queue (mutual exclusion; replicas update)
- Determine least access (cache coherence, DSM)

## Limits of the Logical Clocks

- Cannot be used to determine events concurrency

$$(e \parallel f) \text{ does not imply } (C(e) = C(f))$$

- Some missing events may go undetected:
  - If $C(e) < C(f)$, is there any $g$ so that $e \rightarrow g \rightarrow f$?
  - Impossible to answer with logical clocks only




$P_3$ will synchronize later — $P_3$ synchronized — $P_3$ desynchronized!

# Conclusion on Logical Clocks

## Possible Applications

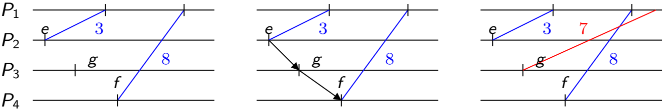
- Distributed waiting queue (mutual exclusion; replicas update)
- Determine least access (cache coherence, DSM)

## Limits of the Logical Clocks

- Cannot be used to determine events concurrency

  $(e \parallel f)$ does not imply $(C(e) = C(f))$

- Some missing events may go undetected:
  - If $C(e) < C(f)$, is there any $g$ so that $e \to g \to f$?
  - Impossible to answer with logical clocks only




$P_3$ will synchronize later — $P_3$ synchronized — $P_3$ desynchronized!

- All is because $e \to f \Rightarrow C(e) < C(f)$ is no $\Leftrightarrow$

# Vector Clocks

## General idea

- Captures the happened-before relation
- Assigns timestamp to each events such that

$$\boxed{e \longrightarrow f \Leftrightarrow C(e) < C(f)}$$

- Like the name says, values $C(e)$ are not scalars but vectors ($\in \mathbb{N}^{\#processes}$)
  $V_i[j]$: What $i$ knows of the clock of $j$

# Vector Clocks

## General idea

- Captures the happened-before relation
- Assigns timestamp to each events such that

$$\boxed{e \rightarrow f \Leftrightarrow C(e) < C(f)}$$

- Like the name says, values $C(e)$ are not scalars but vectors ($\in \mathbb{N}^{\#processes}$)
  $V_i[j]$: What $i$ knows of the clock of $j$

## Comparing two vectors: component-wise

- Equality: $V = W$ iff $\forall i, V_i = W_i$
- Comparison: $V < W$ iff $\forall i, V_i \leq W_i$ and $\exists i, V_i < W_i$
- Examples: $\begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} < \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}$ and $\begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} < \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$ but $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \not< \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$

# Implementing Vector Clocks

- Each process $i$ has a local scalar vector $C_i$ ($\in \mathbb{N}^{\#processes}$)

## Computation rules on process $i$

| | |
|---|---|
| Initialization : | $C_i \leftarrow \{0, ..., 0\}$ |
| Local event : | $C_i[i] += 1$ |
| Sending message ($m$) : | $C_i[i] += 1$ then send $(m, C_i)$ |
| Receiving message $(m, E_m)$: | $\forall k, C_i[k] \leftarrow max(C_i[k], E_m[k])$ <br> $C_i[i] += 1$ |

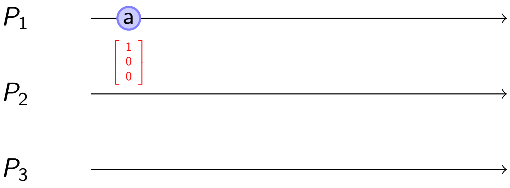
# Implementing Vector Clocks

- Each process $i$ has a local scalar vector $C_i$ ($\in \mathbb{N}^{\#processes}$)

## Computation rules on process $i$

| | |
|---|---|
| Initialization : | $C_i \leftarrow \{0, ..., 0\}$ |
| Local event : | $C_i[i] += 1$ |
| Sending message $(m)$ : | $C_i[i] += 1$ then send $(m, C_i)$ |
| Receiving message $(m, E_m)$: | $\forall k, C_i[k] \leftarrow max(C_i[k], E_m[k])$ |
| | $C_i[i] += 1$ |

## Example




- $a$ first event of $P_1 \leadsto C(a) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

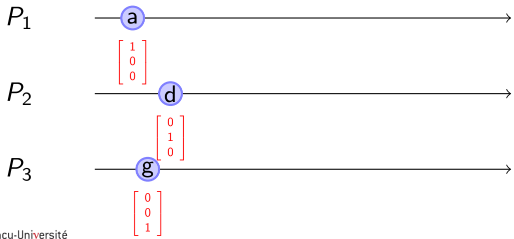
# Implementing Vector Clocks

- Each process $i$ has a local scalar vector $C_i$ ($\in \mathbb{N}^{\#processes}$)

## Computation rules on process $i$

| | |
|---|---|
| Initialization : | $C_i \leftarrow \{0, ..., 0\}$ |
| Local event : | $C_i[i] += 1$ |
| Sending message $(m)$ : | $C_i[i] += 1$ then send $(m, C_i)$ |
| Receiving message $(m, E_m)$: | $\forall k, C_i[k] \leftarrow max(C_i[k], E_m[k])$ <br> $C_i[i] += 1$ |

## Example




- $a$ first event of $P_1 \leadsto C(a) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

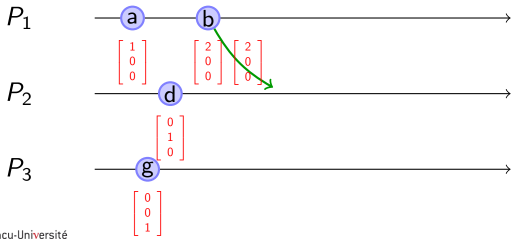
# Implementing Vector Clocks

- Each process $i$ has a local scalar vector $C_i$ $(\in \mathbb{N}^{\#processes})$

## Computation rules on process $i$

| | |
|---|---|
| Initialization : | $C_i \leftarrow \{0, ..., 0\}$ |
| Local event : | $C_i[i] += 1$ |
| Sending message $(m)$ : | $C_i[i] += 1$ then send $(m, C_i)$ |
| Receiving message $(m, E_m)$: | $\forall k, C_i[k] \leftarrow max(C_i[k], E_m[k])$ <br> $C_i[i] += 1$ |

## Example




- $a$ first event of $P_1 \rightsquigarrow C(a) = \begin{bmatrix}1\\0\\0\end{bmatrix}$
- $b$: send $\rightsquigarrow C_1[1] += 1$; send $C_1$

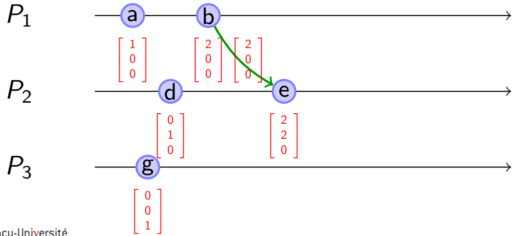
# Implementing Vector Clocks

- Each process $i$ has a local scalar vector $C_i$ ($\in \mathbb{N}^{\#processes}$)

## Computation rules on process $i$

$$\begin{aligned}
\text{Initialization :} \quad & C_i \leftarrow \{0, ..., 0\} \\
\text{Local event :} \quad & C_i[i] += 1 \\
\text{Sending message } (m) \text{ :} \quad & C_i[i] += 1 \text{ then send } (m, C_i) \\
\text{Receiving message } (m, E_m)\text{:} \quad & \forall k, C_i[k] \leftarrow max(C_i[k], E_m[k]) \\
& C_i[i] += 1
\end{aligned}$$

## Example




- $a$ first event of $P_1 \rightsquigarrow C(a) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$
- $b$: send $\rightsquigarrow C_1[1] += 1$; send $C_1$
- $e$: recv

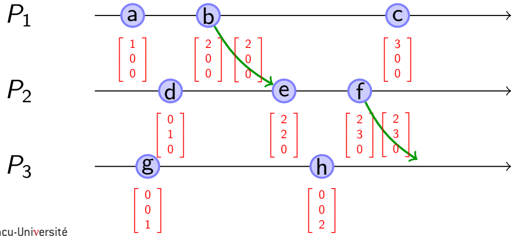
# Implementing Vector Clocks

- Each process $i$ has a local scalar vector $C_i$ ($\in \mathbb{N}^{\#processes}$)

## Computation rules on process $i$

$$
\begin{aligned}
\text{Initialization}: &\quad C_i \leftarrow \{0, ..., 0\} \\
\text{Local event}: &\quad C_i[i] += 1 \\
\text{Sending message } (m): &\quad C_i[i] += 1 \text{ then send } (m, C_i) \\
\text{Receiving message } (m, E_m): &\quad \forall k, C_i[k] \leftarrow max(C_i[k], E_m[k]) \\
&\quad C_i[i] += 1
\end{aligned}
$$

## Example




- $a$ first event of $P_1 \rightsquigarrow C(a) = \begin{bmatrix}1\\0\\0\end{bmatrix}$
- $b$: send $\rightsquigarrow C_1[1] += 1$; send $C_1$
- $e$: recv
- $c$, $h$: local events; $f$: send

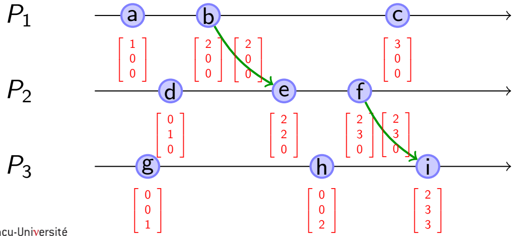
# Implementing Vector Clocks

- Each process $i$ has a local scalar vector $C_i$ ($\in \mathbb{N}^{\#processes}$)

## Computation rules on process $i$

| | |
|---|---|
| Initialization : | $C_i \leftarrow \{0, ..., 0\}$ |
| Local event : | $C_i[i] += 1$ |
| Sending message $(m)$ : | $C_i[i] += 1$ then send $(m, C_i)$ |
| Receiving message $(m, E_m)$: | $\forall k, C_i[k] \leftarrow max(C_i[k], E_m[k])$ <br> $C_i[i] += 1$ |

## Example




- $a$ first event of $P_1 \rightsquigarrow C(a) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$
- $b$: send $\rightsquigarrow C_1[1] += 1$; send $C_1$
- $e$: recv
- $c$, $h$: local events; $f$: send
- i: recv

# Conclusion on Vector Clocks

## Possible Applications

- Distributed system monitoring (event dating, distributed debugging)
- Computation of global state; Distributed simulation

## Limits of Vector Clocks

- Comparing two vectors can require up to N comparison
- Processes don't know whether the others are up-to-date or lag behind
  - Matrix clocks solve that issue
  - $MC_i[j, k]$: what $i$ knows of the knowledge of $j$ about $k$'s clock
  - This allows causal delivery
  - But matrix clocks are even more expensive ($O(n^2)$)

## Theoretical foundations


- Time and State of a Distributed System
- Ordering of events
- Abstract Clocks
  - Global Observer
  - Logical Clocks
  - Vector Clocks
- Some Distributed Algorithms
  - Mutual Exclusion
    - Coordinator-based Algorithm
    - Lamport's Algorithm
    - Ricart and Agrawala's Algorithm
    - Roucairol and Carvalho's Algorithm
    - Token-Ring algorithm
    - Suzuki and Kasami's Algorithm
  - Leader Election
  - Consensus
  - Ordering Messages
  - Group Protocols
- Conclusion on distributed algorithmic

# Some Distributed Algorithms

## Goals of this section

### Present some basic algorithms

- Mutual exclusion
- Election
- Consensus
- Group protocols

### Present general approaches

- Ordering events (with abstract clocks)
- Applicative topologies (ring, tree, graph without circuit)

# Some Distributed Algorithms

## Goals of this section

### Present some basic algorithms

- Mutual exclusion
- Election
- Consensus
- Group protocols
- Sequential equivalents
  - Sorting, Shortest path
  - Classical data structures (stack, list, hashing, trees)

### Present general approaches

- Ordering events (with abstract clocks)
- Applicative topologies (ring, tree, graph without circuit)
- Sequential equivalents
  - Recursion, Divide&Conquer, Greedy algorithms

# Mutual Exclusion

## Problem Statement

- Force an order on the execution of critical sections
- Fairness (no infinite starvation of any process); Liveness (no deadlock)

## Approaches

- Centralized coordinator: ask lock to coordinator, get lock, release lock
- Use a global order: using abstract clocks
  Ask everyones, and concurrent requests are handled "in order"
- Using quorums: Ask only members of specific groups
- Force a topology: virtual ring, virtual tree
  Gives an order on nodes, not only on requests

## Algorithms

- A whole load of such algorithms in literature
- #messages$\in [O(log(n)); O(n)]$ (ask everyone, or distributed waiting queue)

## What's coming now: Details of some algorithms

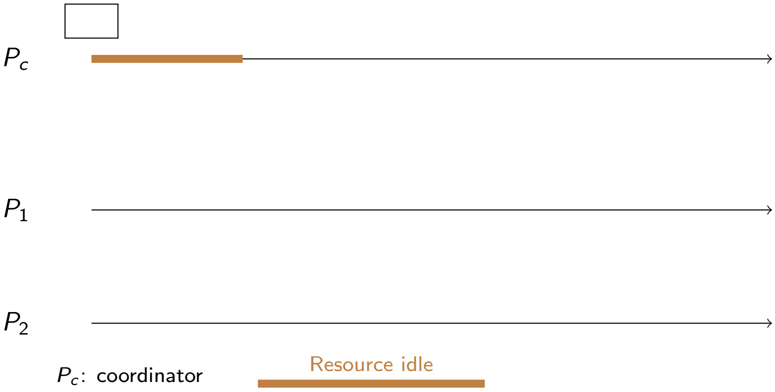
- For culture and to get a grip on distributed algorithms development approach

# Centralized: Coordinator Based Algorithm

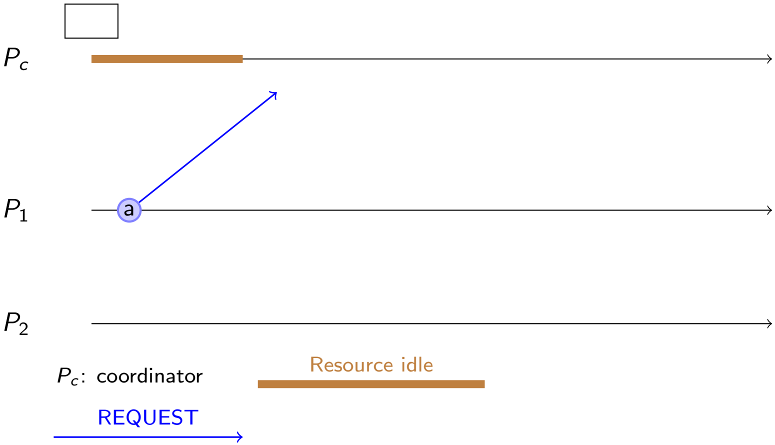
## Main Idea

- One of the processes acts as coordinator (cf. Leader Election Algorithm)
  Coordinator decides the order in which critical section requests are fulfilled
- Processes send requests to coordinator and wait permission
  Requests are fulfilled in FIFO order at the coordinator
- Coordinator grants permission to requests one at a time
  All other requests are queued in a FIFO queue.

# Coordinator Based Algorithm for Mutual Exclusion



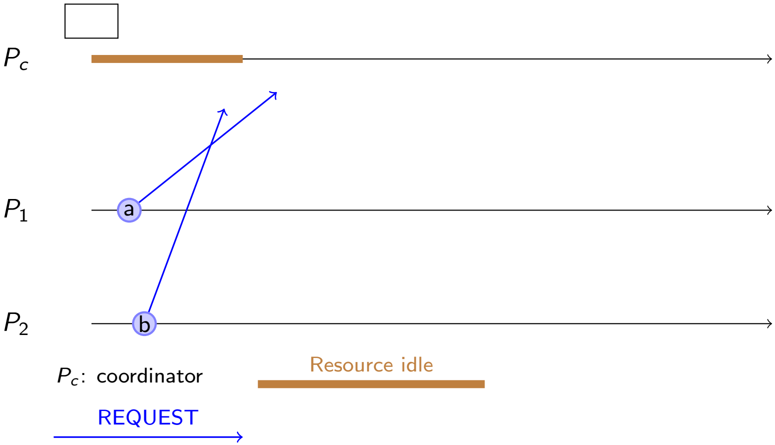
# Coordinator Based Algorithm for Mutual Exclusion




## Event explanation

a. $P_1$ requests the CS to coordinator

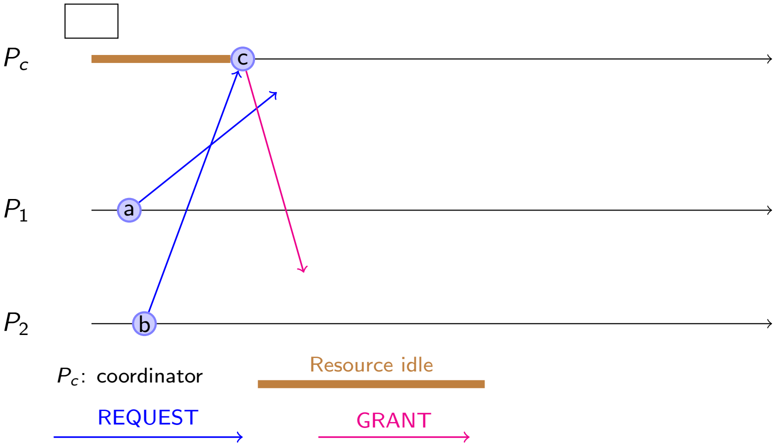
# Coordinator Based Algorithm for Mutual Exclusion




## Event explanation

b. $P_2$ requests the CS to coordinator

# Coordinator Based Algorithm for Mutual Exclusion

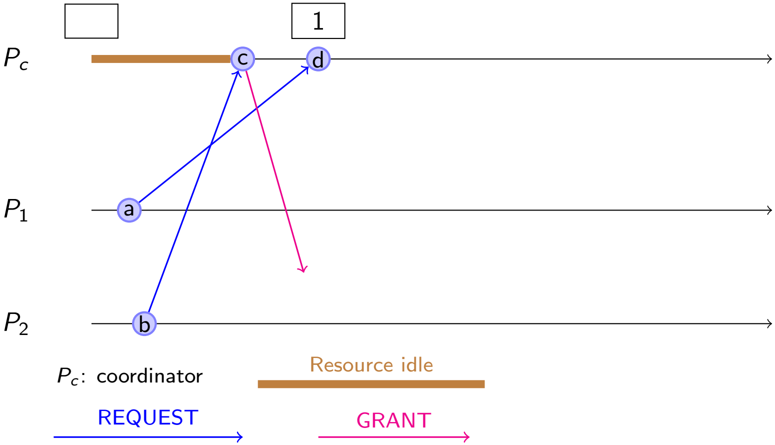



## Event explanation

c. coordinator receives the request from $P_2$

- Idle token, so send reply back

# Coordinator Based Algorithm for Mutual Exclusion

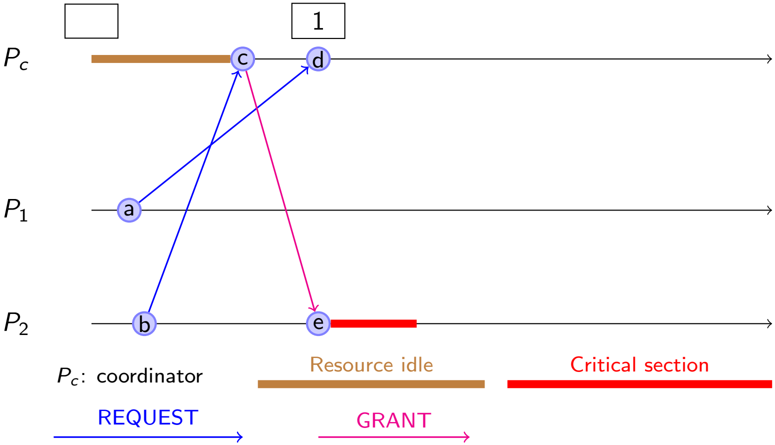



## Event explanation

d. coordinator receives the request from $P_1$

- Token not there, so enqueue the request

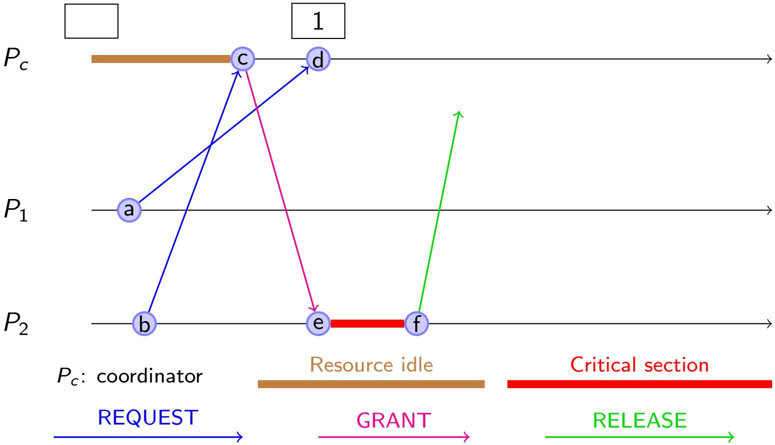
# Coordinator Based Algorithm for Mutual Exclusion




## Event explanation

e. $P_2$ receives the grant
   - Enters the CS

# Coordinator Based Algorithm for Mutual Exclusion

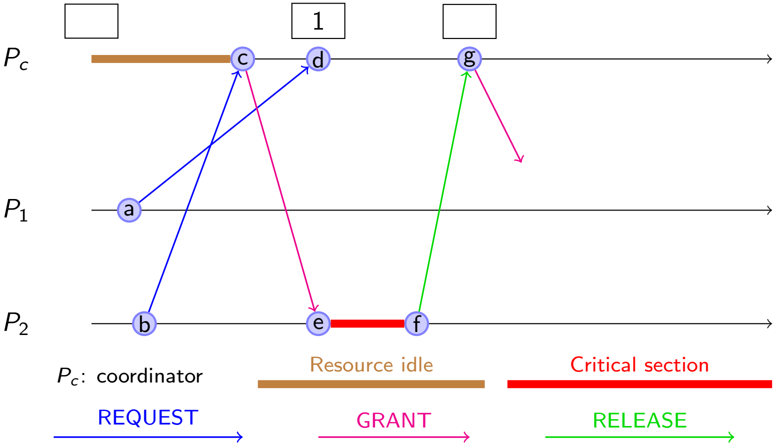



## Event explanation

f. $P_2$ exits the CS

- Send release to coordinator

# Coordinator Based Algorithm for Mutual Exclusion

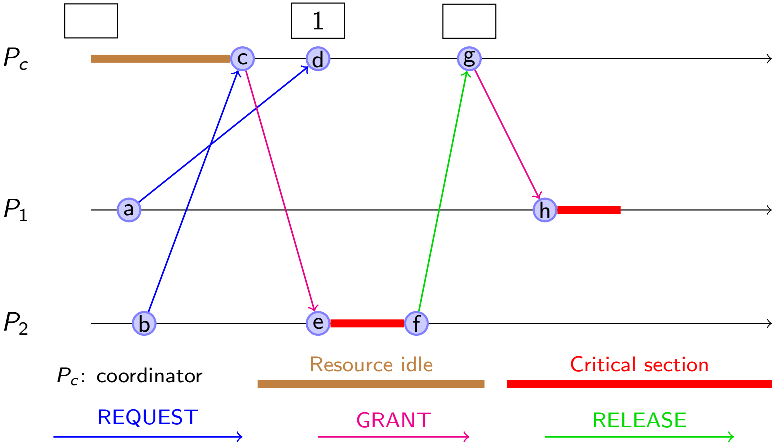



## Event explanation

g. coordinator receives the release

- Someone ($P_1$) is waiting in the queue
- Unqueue $P_1$
- Send grant to $P_1$

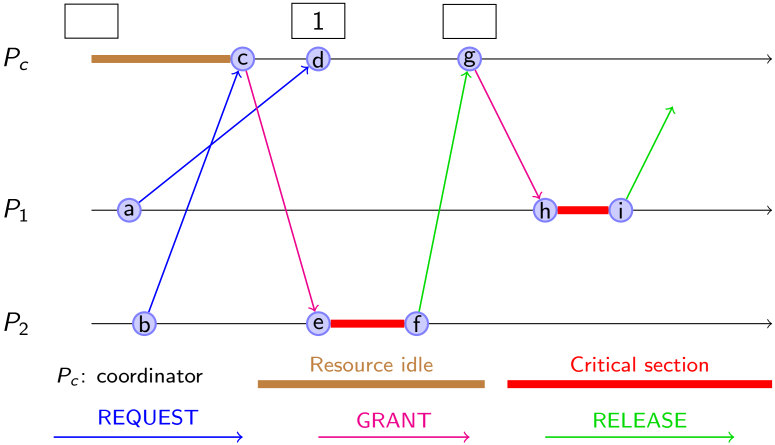
# Coordinator Based Algorithm for Mutual Exclusion




## Event explanation

h. $P_1$ receives the grant
   - Enters the CS

# Coordinator Based Algorithm for Mutual Exclusion

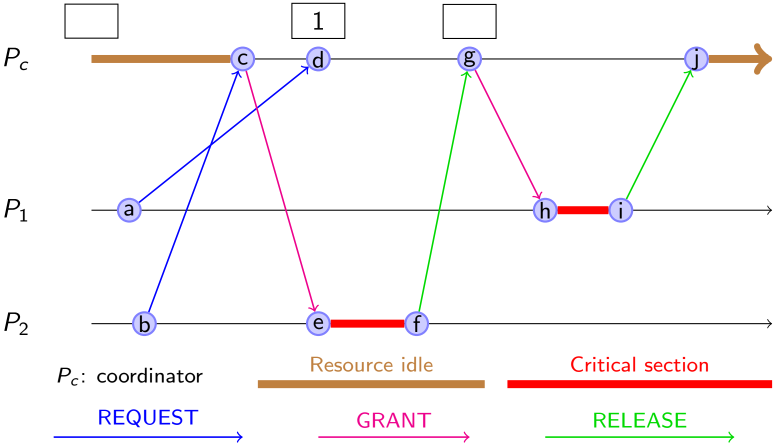



## Event explanation

f. $P_1$ exits the CS

- Send release to coordinator

# Coordinator Based Algorithm for Mutual Exclusion




## Event explanation

g. coordinator receives the release

- Nobody in queue, nothing to do
- Let the token idling

# Centralized Mutual Exclusion: Complexity Analysis

## Parameters

N Number of processes in the system

T Message transmission time

E Critical section execution time

## Message complexity: 3

- 1 REQUEST message + 1 GRANT message + 1 RELEASE message
- Message-size complexity: O(1)

## Time complexity

- Response time (under light load): 2T + E
- Synchronization delay (under heavy load): 2T

# Lamport's Algorithm for Mutual Exclusion

## Assumptions

- Channels are FIFO
- Processes run a Lamport's Logical Clock

## Main Idea

- Requests are timestamped using logical clocks, and fulfilled in timestamp order
- Processes maintain a priority queue of all requests they know about
- Lots of broadcasts to get the timestamps propagate to peers

# Lamport's Mutual Exclusion: Steps for process $P_i$

## On generating a critical section request

- Insert the request into the priority queue
- Broadcast the request to all processes

## On receiving a critical section request from another process:

- Insert the request into the priority queue.
- Send a REPLY message to the requesting process.

## Conditions to enter critical section:

- L1: $P_i$ has received a REPLY message from all processes.
  Any request received in future will have larger timestamp than own request
- L2: $P_i$'s own request is at the top of its queue.
  I have the smallest timestamp among all already received requests

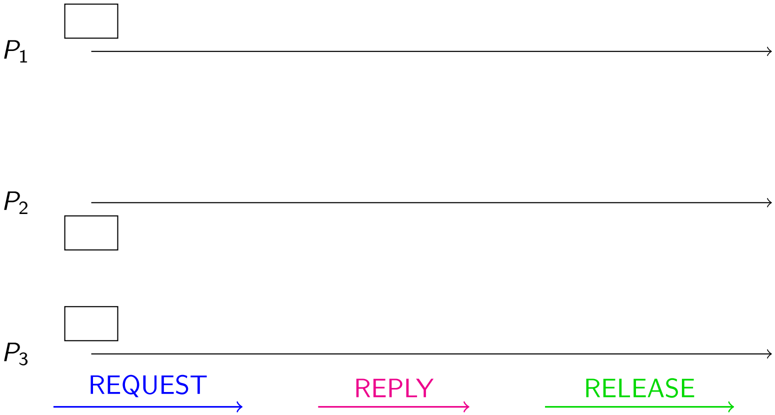
## On leaving the critical section

- Remove the request from the queue
- Broadcast a RELEASE message to all processes

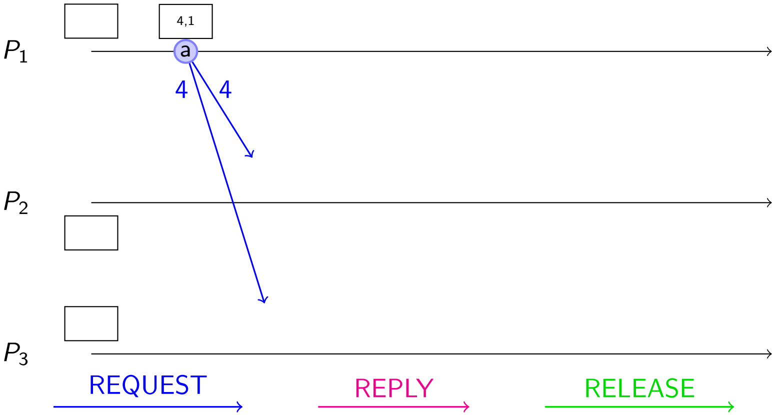
## On receiving a RELEASE message from another process

- Remove the request of that process from the queue

# Lamport's Mutual Exclusion: Illustration



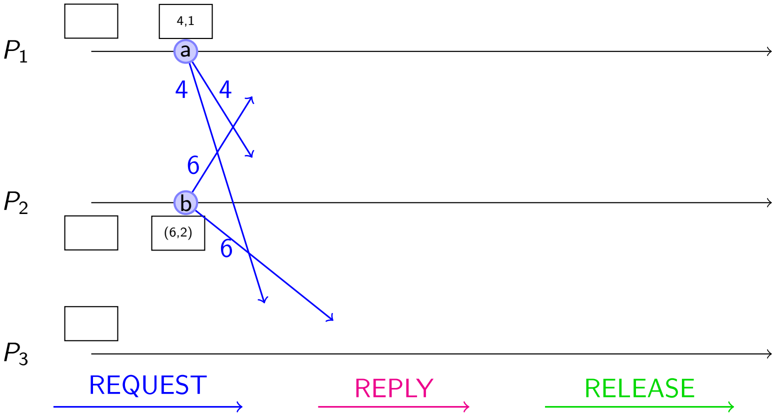
# Lamport's Mutual Exclusion: Illustration




a. $P_1$ requests the CS (timestamp=4)

- Broadcast the request
- Enqueue the request locally

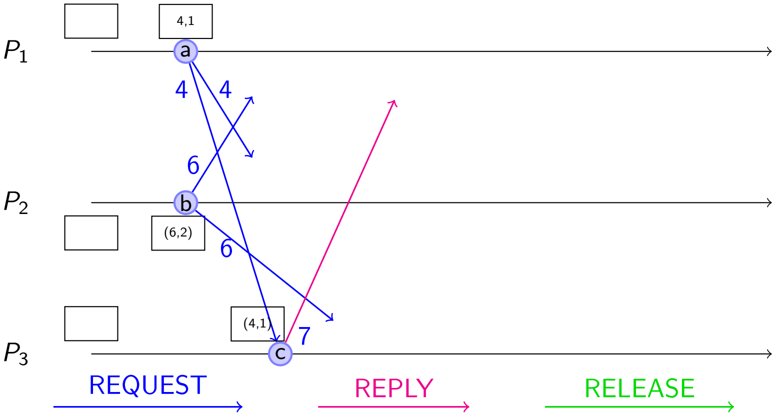
# Lamport's Mutual Exclusion: Illustration




b. $P_2$ requests the CS (timestamp=6)

- Broadcast the request
- Enqueue the request locally

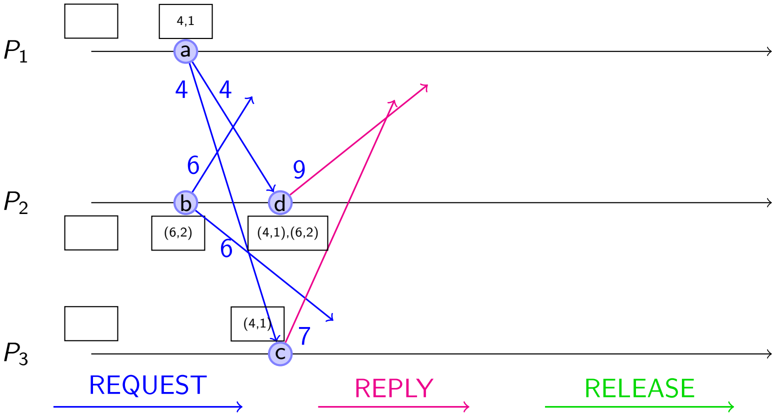
# Lamport's Mutual Exclusion: Illustration




c. $P_3$ receives the request from $P_1$

- Answer REPLY with timestamp 7
- Enqueue the request locally

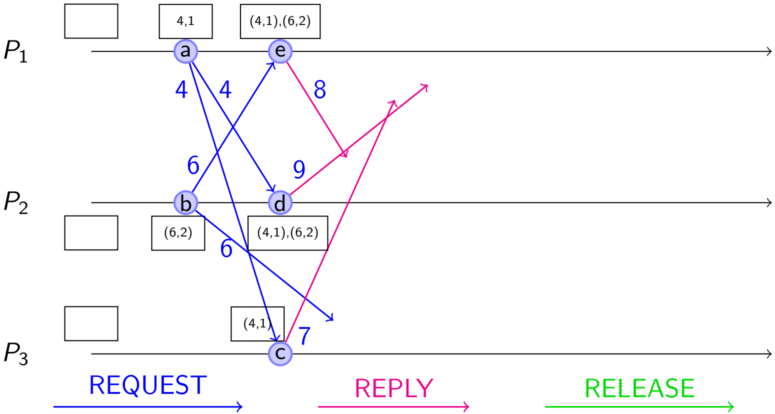
# Lamport's Mutual Exclusion: Illustration




d. $P_2$ receives the request from $P_1$

- Answer REPLY with timestamp (max(6,7)+1)+1=9
- Enqueue the request locally (sorting on Lamport's clock)

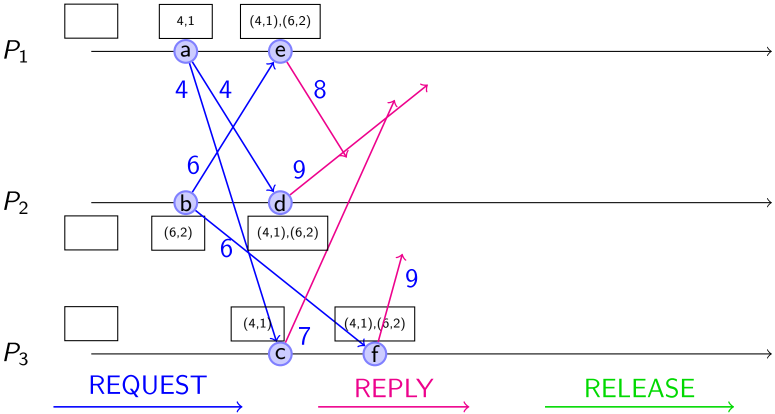
# Lamport's Mutual Exclusion: Illustration




e. $P_1$ receives the request from $P_2$

- Answer REPLY with timestamp max(4,6)+1=8
- Enqueue the request locally (sorting on Lamport's clock)

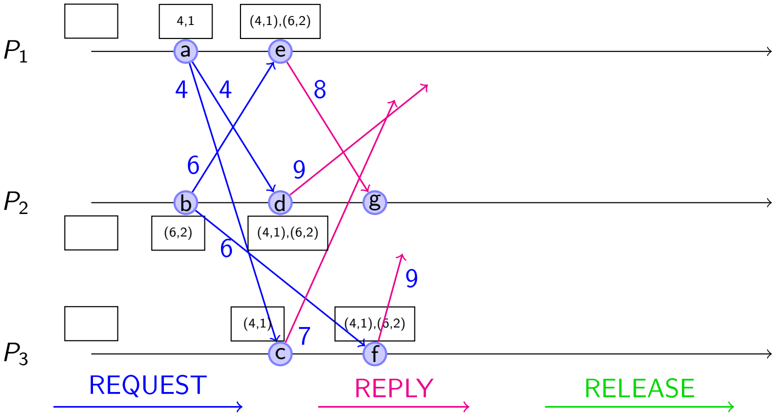
# Lamport's Mutual Exclusion: Illustration




f. $P_3$ receives the request from $P_2$

- Answer REPLY with timestamp (max(4,6)+1)+1=9
- Enqueue the request locally

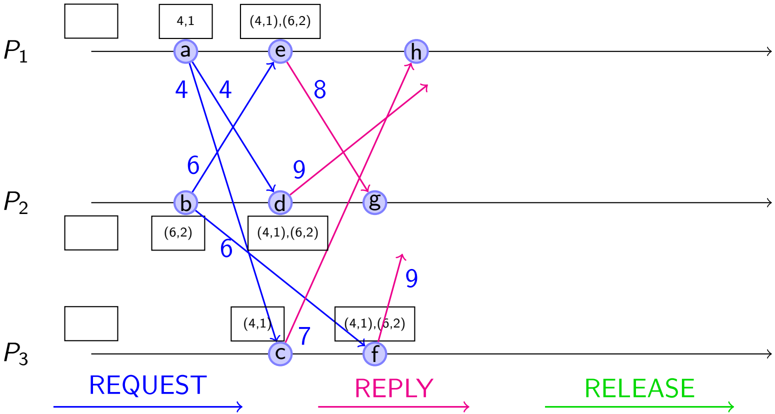
# Lamport's Mutual Exclusion: Illustration




g. $P_2$ receives the reply from $P_1$

- (nothing to do, one request still missing)

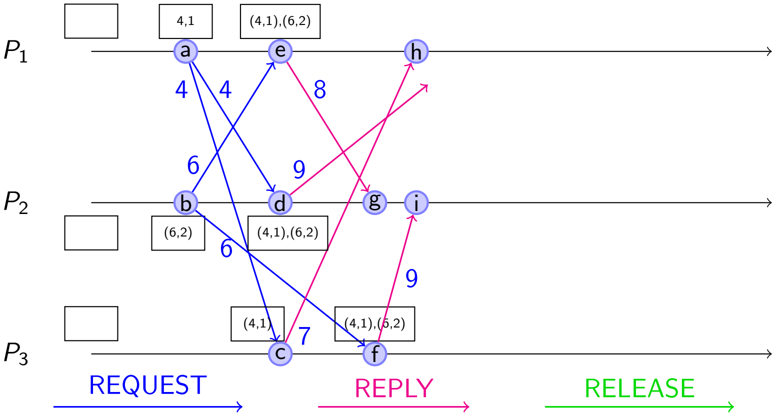
# Lamport's Mutual Exclusion: Illustration




h. $P_1$ receives the reply from $P_3$

- (nothing to do, one request still missing)

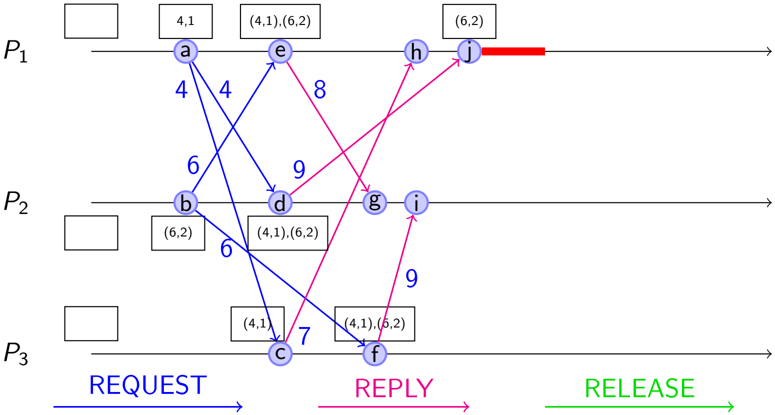
# Lamport's Mutual Exclusion: Illustration




i. $P_2$ receives the reply from $P_3$

- Every request received, but not first in queue
- Thus nothing to do

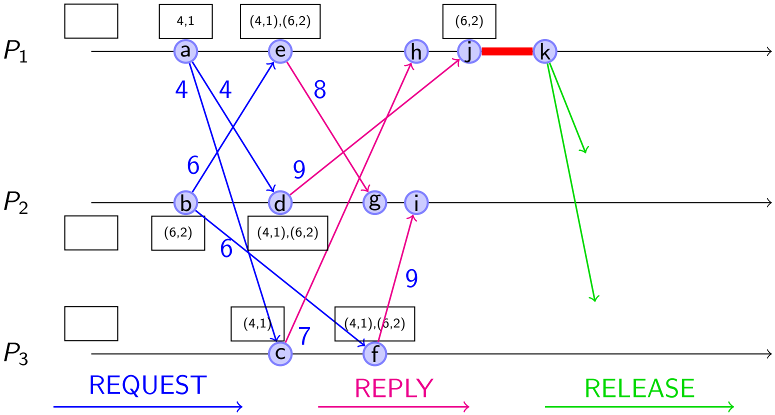
# Lamport's Mutual Exclusion: Illustration




j. $P_1$ receives the reply from $P_2$

- Every request received, and first in queue
- Thus dequeuing self request and entering CS

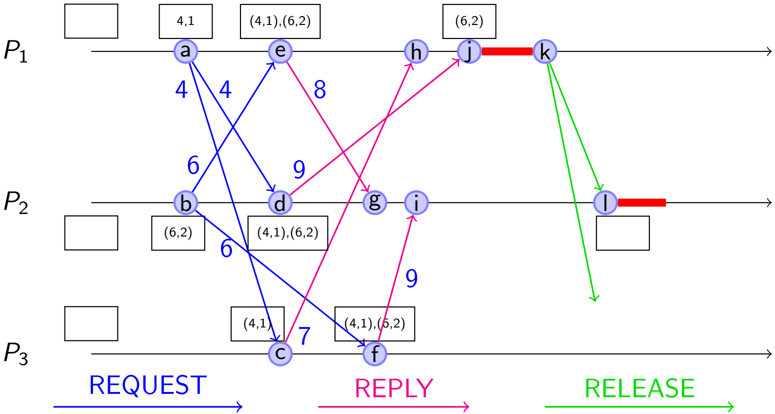
# Lamport's Mutual Exclusion: Illustration




k. $P_1$ exits CS

- Broadcast RELEASE

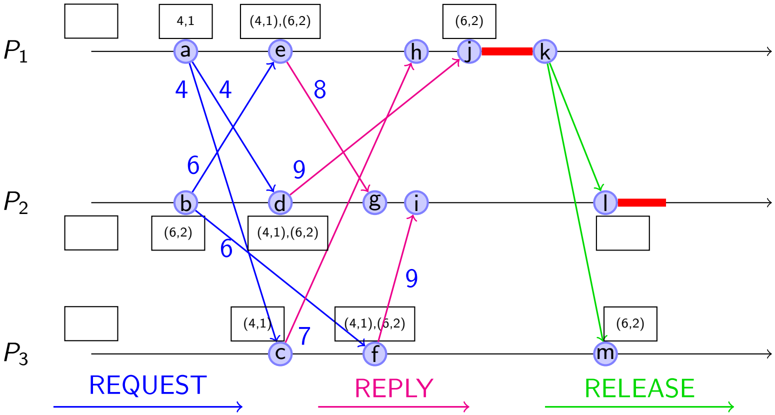
# Lamport's Mutual Exclusion: Illustration




l. $P_2$ receives RELEASE from $P_1$

- Remove (4,1) from queue
- Every replies received and first of queue
- Thus entering CS (after removing myself from queue)

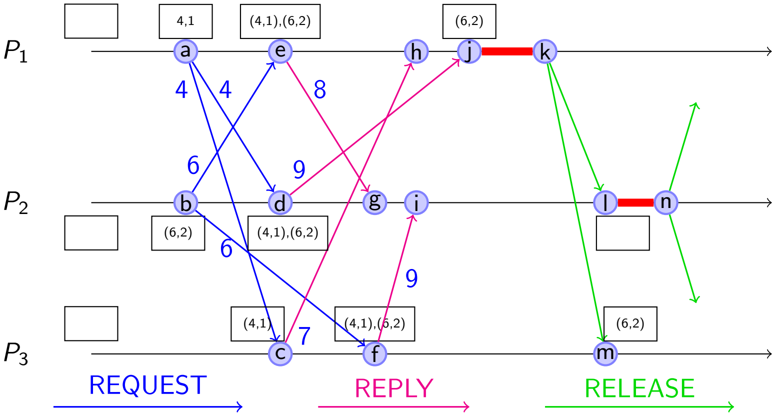
# Lamport's Mutual Exclusion: Illustration




m. $P_3$ receives RELEASE from $P_1$

- Update the queue

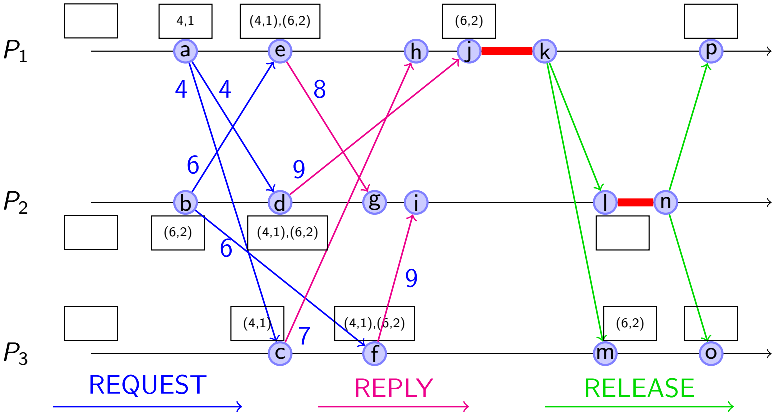
# Lamport's Mutual Exclusion: Illustration




n. $P_2$ exits its CS

- Broadcast RELEASE

# Lamport's Mutual Exclusion: Illustration




o&p. $P_1$ and $P_2$ receive RELEASE from $P_2$

- Update queues

# Lamport's Mutual Exclusion: Optimization

Recap Conditions to enter critical section:

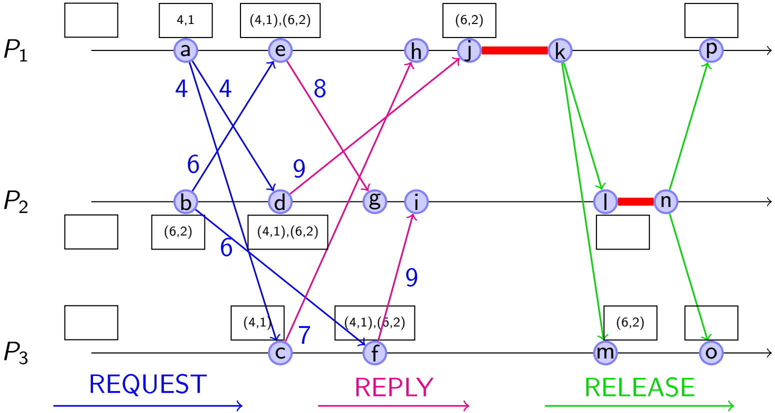
- L1: $P_i$ has received a REPLY message from all processes.
  Any request received in future will have larger timestamp than own request
- L2: $P_i$'s own request is at the top of its queue.
  I have the smallest timestamp among all already received requests

L1 is too restrictive wrt the wanted property

- Wait for any messages with higher timestamp from all processes is enough
  Any request received in future will *still* have larger timestamp than own request

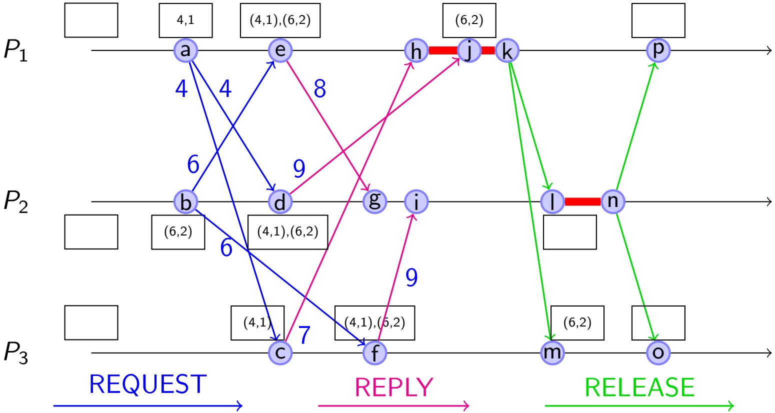
# Lamport's Mutex Optimization: Illustration

## Without the optimization



# Lamport's Mutex Optimization: Illustration

## With the optimization



# Lamport's Mutex Algorithm: Complexity Analysis

## Parameters

N Number of processes in the system

T Message transmission time

E Critical section execution time

## Message complexity: 3(N - 1)

- $N - 1$ REQUEST messages + $N - 1$ REPLY messages + $N - 1$ RELEASE messages
- Message-size complexity: O(1)

## Time complexity

- Response time (under light load): 2T + E
- Synchronization delay (under heavy load): T

# Ricart and Agrawala's Algorithm

## Inefficiencies in Lamport's Algorithm

- Scenario 1
  - Situation: $P_i$ and $P_j$ concurrently request CS and $C(P_i) < C(P_j)$
  - Lamport: $P_i$ first send REPLY and later RELEASE.
    $P_j$ only acts on RELEASE
  - Improvement: $P_i$'s REPLY can be ommited
- Scenario 2
  - Situation: $P_i$ requests CS and $P_j$ don't for some time
  - Lamport: $P_i$ send RELEASE to $P_j$ on exiting CS
  - Improvement: That message can be ommited
    (if $P_j$ requests CS, it will contact $P_i$ anyway)

## Main ideas of Ricart and Agrawala's Algorithm

- Combine REPLY and RELEASE messages
- On leaving CS, only REPLY/RELEASE to processes with unfulfilled CS requests
- Eliminate priority queue

# Ricart and Agrawala Mutex: Steps for process $P_i$

## On generating a critical section request

- Broadcast the request to all processes

## On receiving a critical section request from another process:

- Send a REPLY if any of these condition is true
  - $P_i$ has no unfulfilled request of its own
  - $P_i$ unfulfilled request has larger timestamp than that of the received request
- Else, defer sending the REPLY message

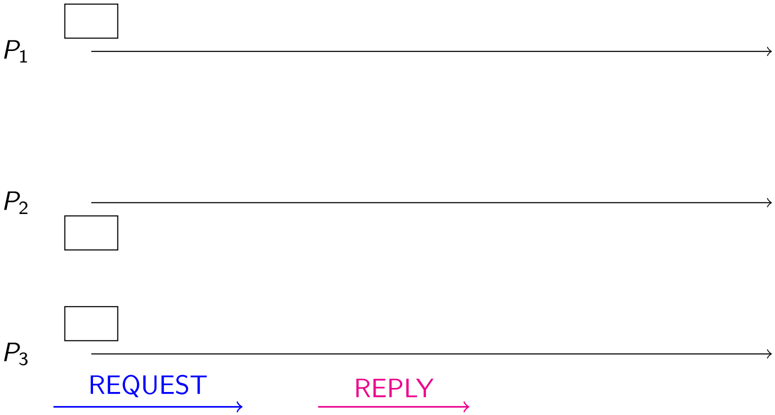
## Conditions to enter critical section:

- $P_i$ has received a REPLY message from all processes

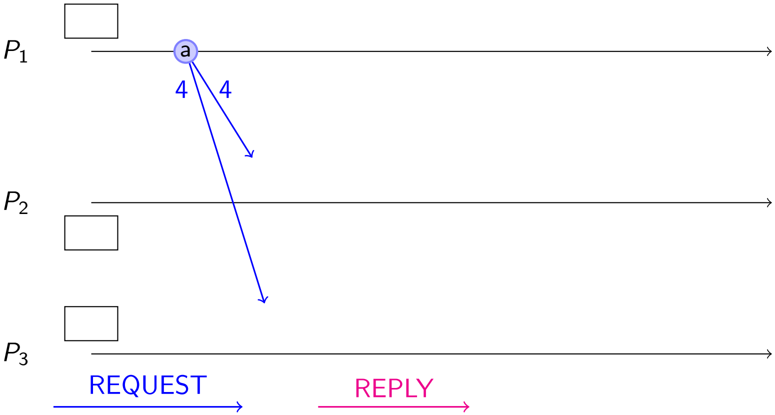
## On leaving the critical section

- Send all defered REPLY messages

# Ricart and Agrawala Mutex: Illustration



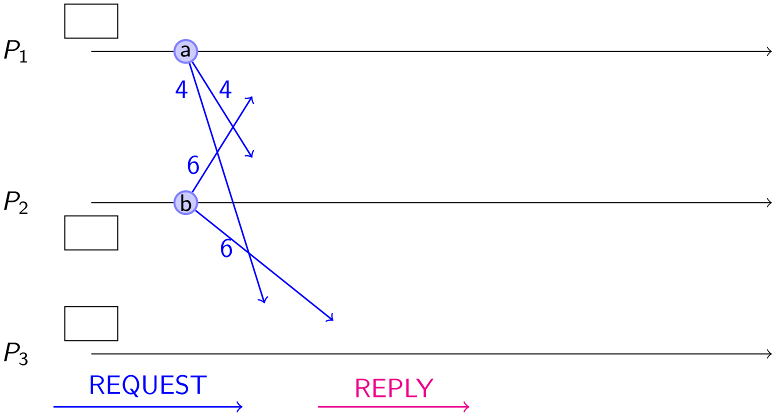
# Ricart and Agrawala Mutex: Illustration




a. $P_1$ requests the CS (timestamp=4)

- Broadcast the request

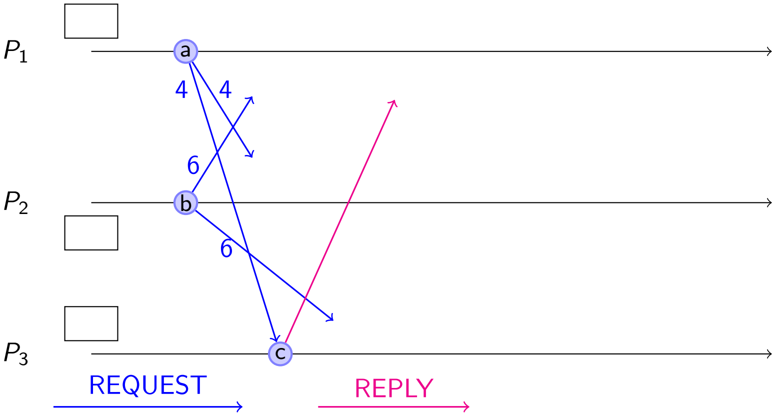
# Ricart and Agrawala Mutex: Illustration




b. $P_2$ requests the CS (timestamp=6)

- Broadcast the request

# Ricart and Agrawala Mutex: Illustration

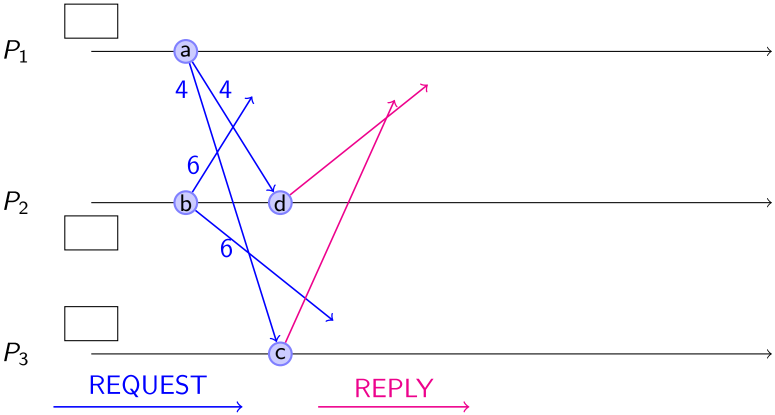



c. $P_3$ receives the request from $P_1$

- No unfulfilled request itself

⇝ Returns a REPLY

# Ricart and Agrawala Mutex: Illustration

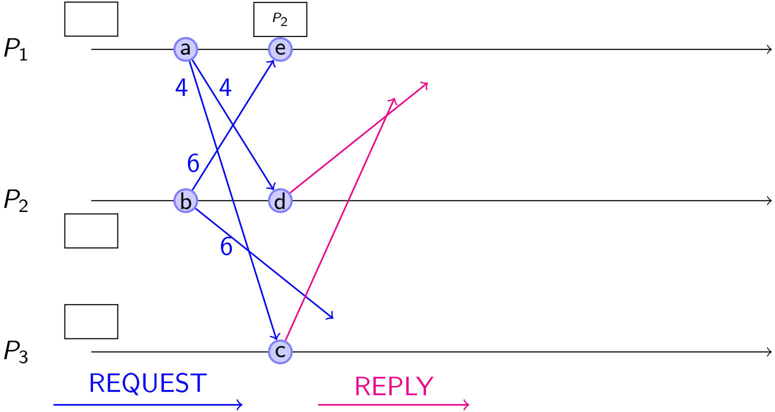



d. $P_2$ receives the request from $P_1$

- Own unfulfilled request has larger timestamp

⇝ Returns a REPLY

# Ricart and Agrawala Mutex: Illustration

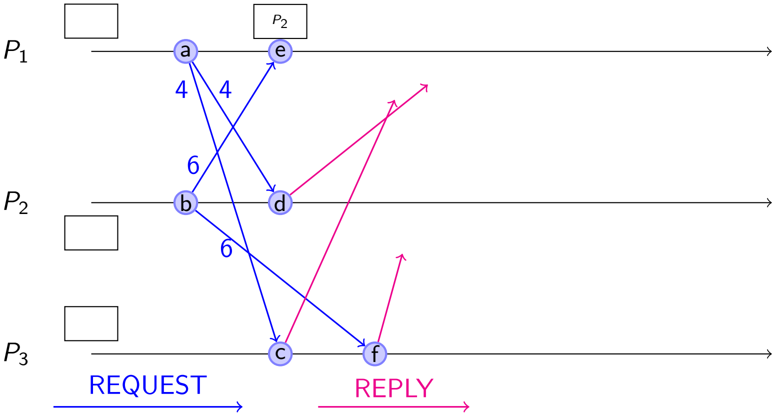



e. $P_1$ receives the request from $P_2$

- Own unfulfilled request has smaller timestamp

$\leadsto$ Defer the sending of REPLY

# Ricart and Agrawala Mutex: Illustration

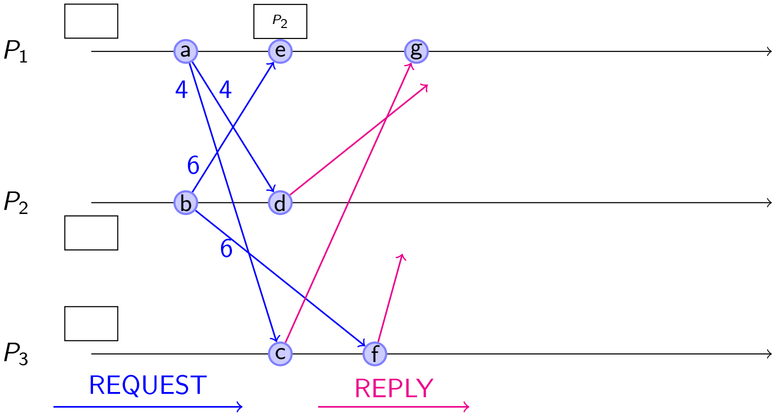



f. $P_3$ receives the request from $P_2$

- No unfulfilled request itself

⇝ Returns a REPLY

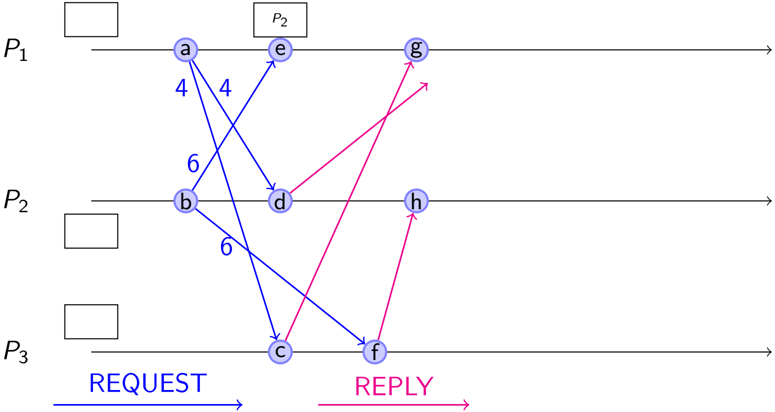
# Ricart and Agrawala Mutex: Illustration




g. $P_1$ receives the reply from $P_3$

- Nothing to do, one request still missing

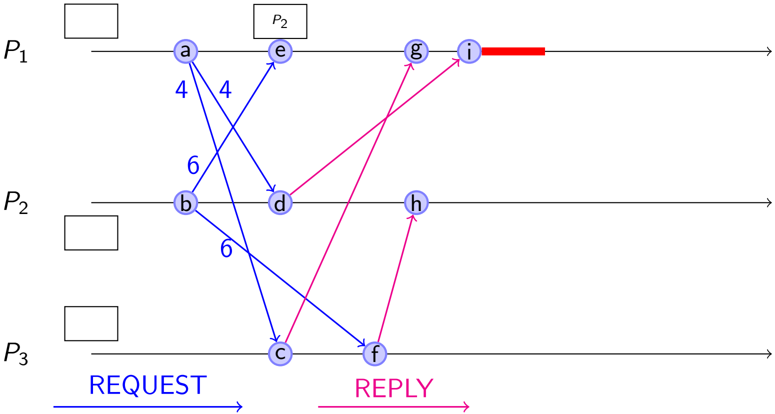
# Ricart and Agrawala Mutex: Illustration




h. $P_2$ receives the reply from $P_3$

- Nothing to do, one request still missing (since it's delayed)

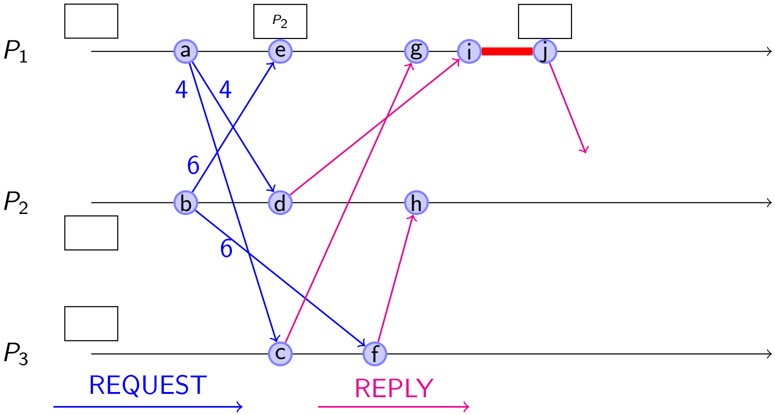
# Ricart and Agrawala Mutex: Illustration




i. $P_1$ receives the reply from $P_2$

- Every request received
- Thus entering CS

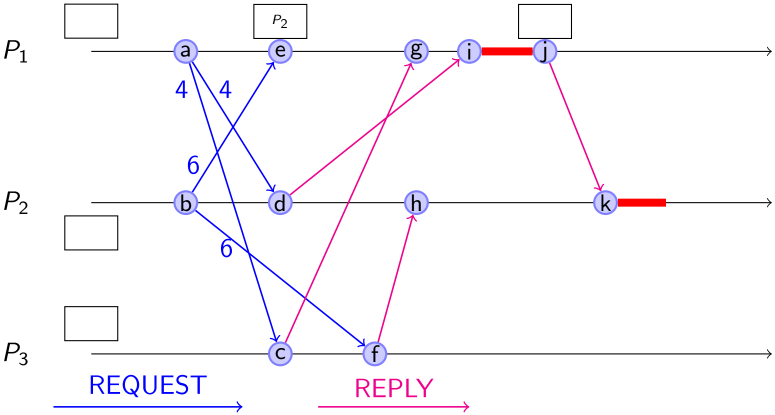
# Ricart and Agrawala Mutex: Illustration




j. $P_1$ exits CS

- Send delayed REPLY to $P_2$

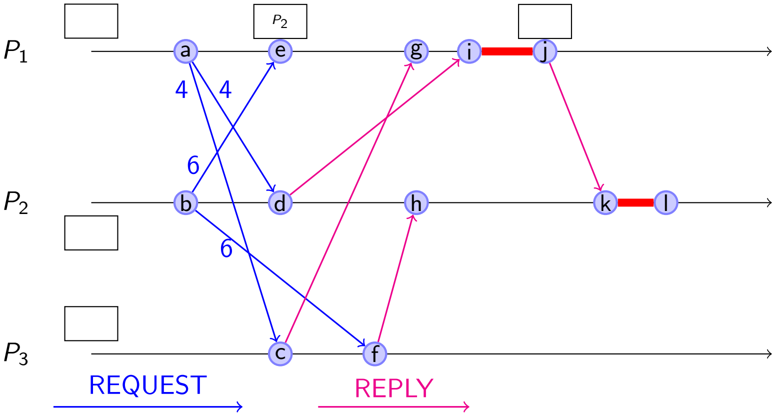
# Ricart and Agrawala Mutex: Illustration




k. $P_2$ receives RELEASE from $P_1$

- Every replies received
- Thus entering CS

# Ricart and Agrawala Mutex: Illustration




l. $P_3$ receives RELEASE from $P_1$

- No delayed REPLY, nothing to do

# Ricart and Agrawala: Complexity Analysis

## Parameters

N Number of processes in the system

T Message transmission time

E Critical section execution time

## Message complexity: 2(N - 1)

- $N-1$ REQUEST messages + $N-1$ REPLY messages
- Message-size complexity: O(1)

## Time complexity

- Response time (under light load): 2T + E
- Synchronization delay (under heavy load): T

# Roucairol and Carvalho's Algorithm

## Inefficiency in Ricart and Agrawala's Algorithm

- Every process handles every critical section request.

## Goal of this new algorithm for conflict resolution

- Change algorithm so that only active processes (requesting CS) interact
- Process not requesting the CS will eventually stop receiving messages

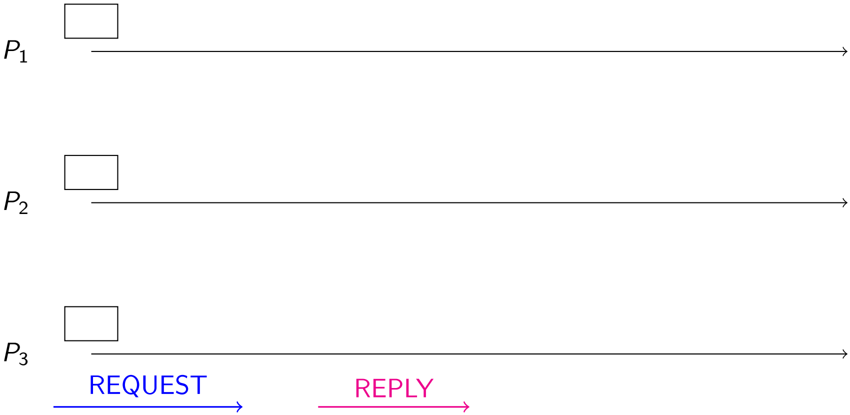
## Main idea

- REPLY from $P_j$ to $P_i$ means: $P_j$ grants permission to $P_i$ to enter CS
- $P_i$ keeps that permission until it send REPLY to someone else

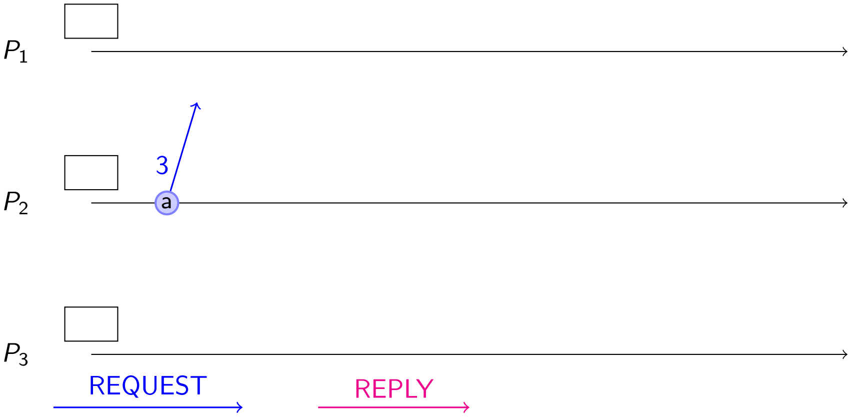
## Modification to Ricart and Agrawala's Algorithm

- To enter CS, $P_i$ asks for permission from $P_j$ if either:
  - ($P_i$ sent REPLY to $P_j$) AND ($P_i$ didn't got REPLY from $P_j$ since then)
  - (It's $P_i$'s first request) AND ($i > j$)

# Roucairol and Carvalho's Mutex: Illustration events



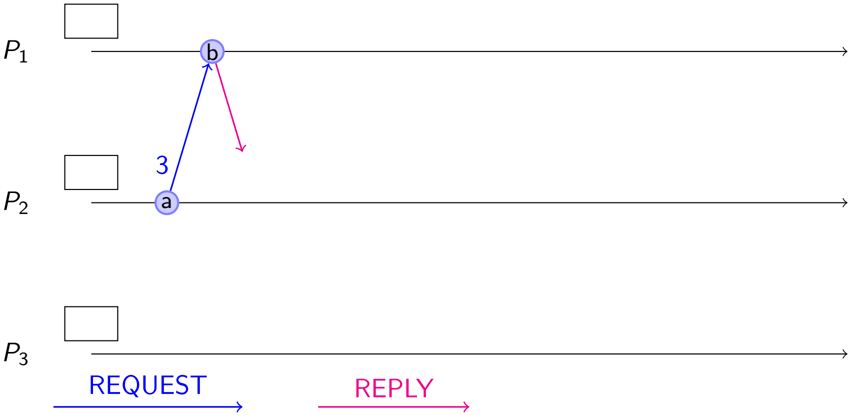
# Roucairol and Carvalho's Mutex: Illustration events




a. $P_2$ requests the CS (timestamp=3)

$\leadsto$ Send the request to $P_1$ only $(1 < 2)$

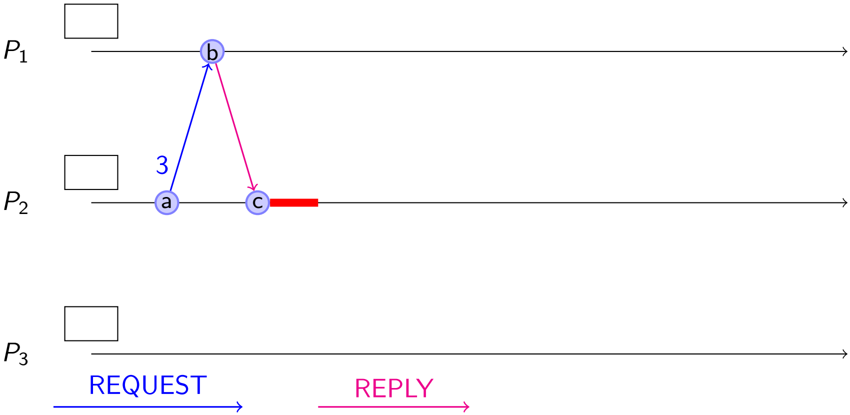
# Roucairol and Carvalho's Mutex: Illustration events




b. $P_1$ receives $P_2$'s REQUEST

⇝ returns REPLY

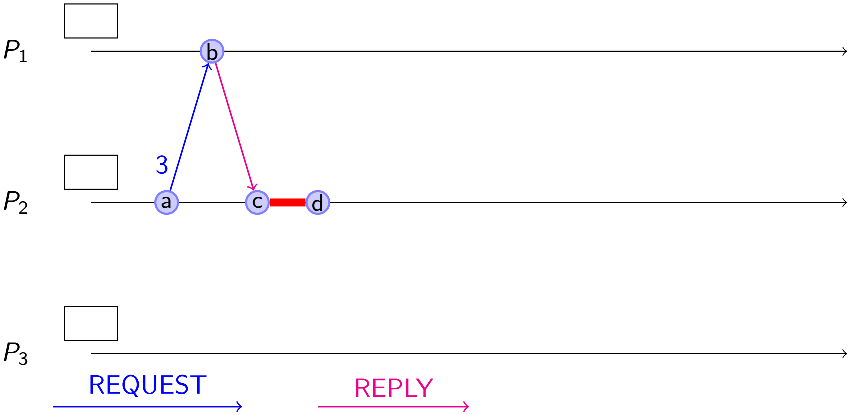
# Roucairol and Carvalho's Mutex: Illustration events




c. $P_2$ receives REPLY from $P_1$.

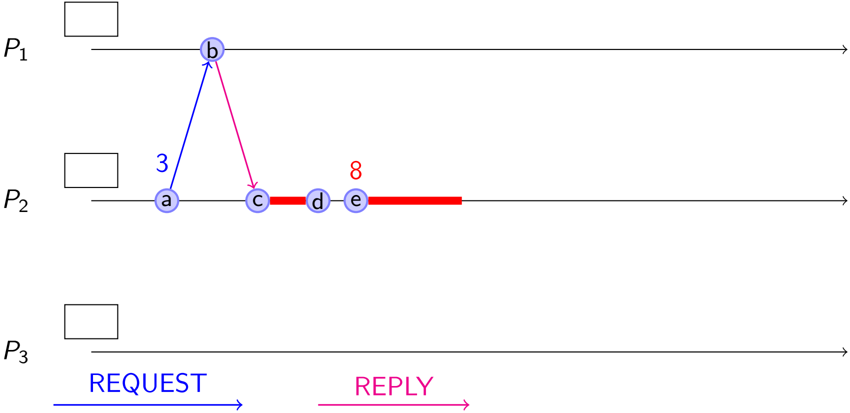
$\rightsquigarrow$ enters CS

# Roucairol and Carvalho's Mutex: Illustration events




d. $P_2$ exists CS

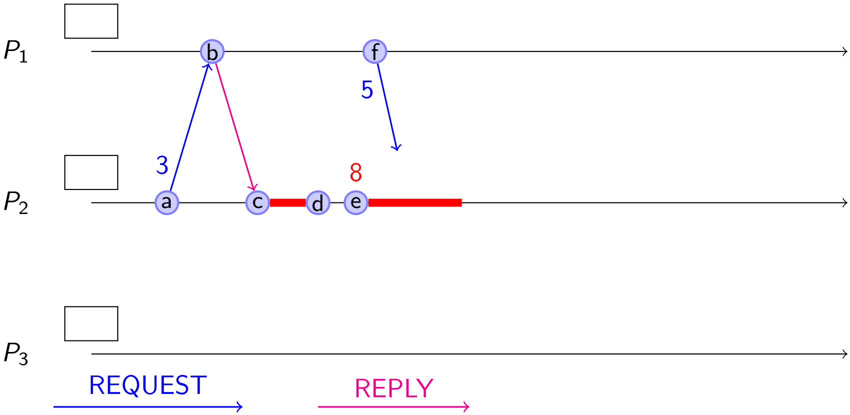
# Roucairol and Carvalho's Mutex: Illustration events




e. $P_2$ requests CS again (stamp=8)

⇝ re-enter CS without any new message

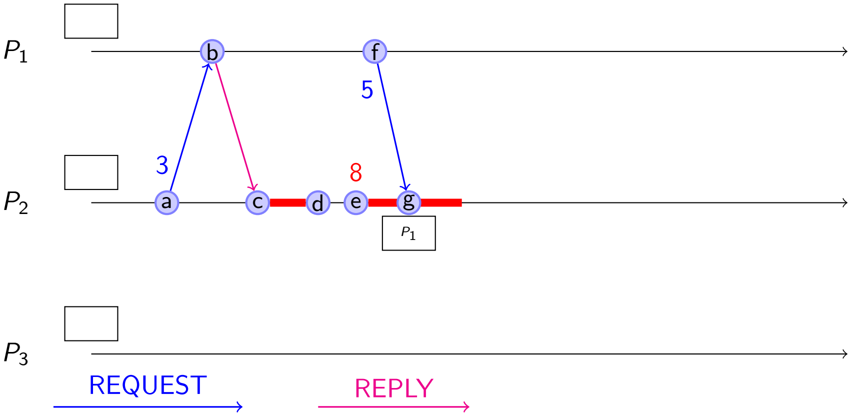
# Roucairol and Carvalho's Mutex: Illustration events




f. $P_1$ requests CS (stamp=5)

$\leadsto$ send REQUEST to $P_2$ only (active known peer)

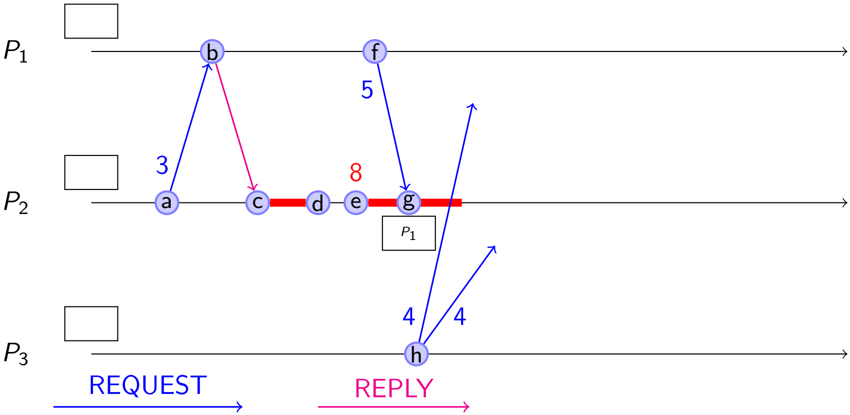
# Roucairol and Carvalho's Mutex: Illustration events




g. $P_2$ receives REQUEST from $P_1$

$\rightsquigarrow$ defers REPLY because in CS

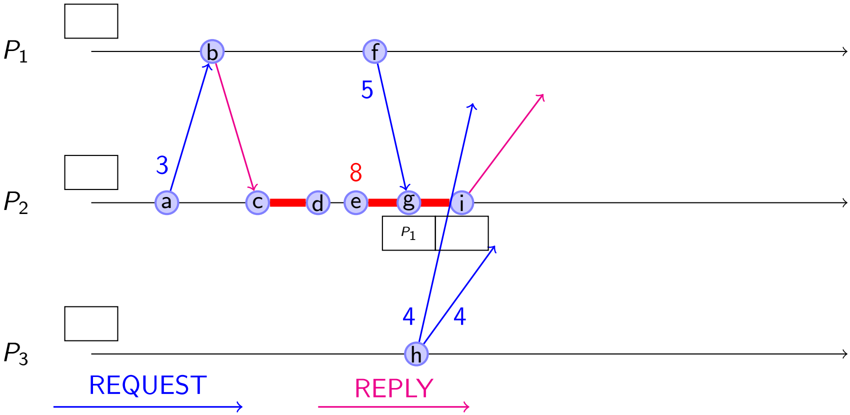
# Roucairol and Carvalho's Mutex: Illustration events




## h. $P_3$ requests the CS

$\rightsquigarrow$ broadcasts REQUEST to every processes

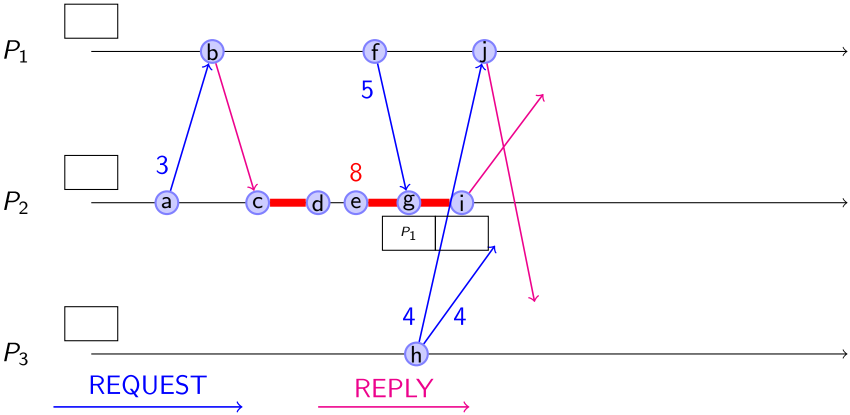
# Roucairol and Carvalho's Mutex: Illustration events




i. $P_2$ exists CS

⇝ send defered REPLY to $P_1$

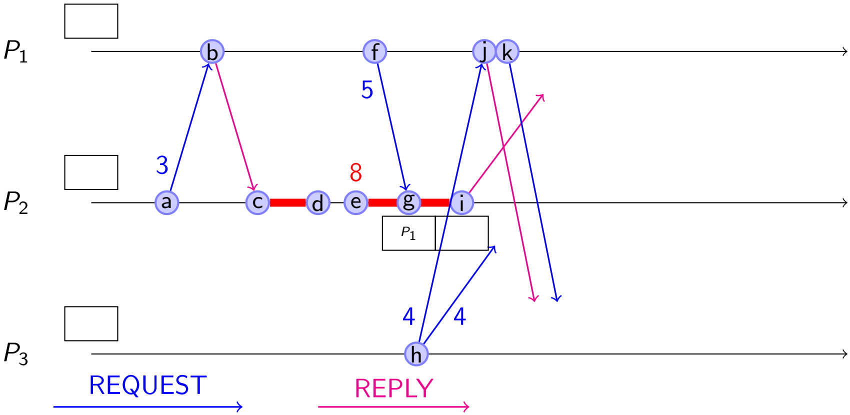
# Roucairol and Carvalho's Mutex: Illustration events




j. $P_1$ receives REQUEST from $P_3$

returns REPLY since stamp lower than own

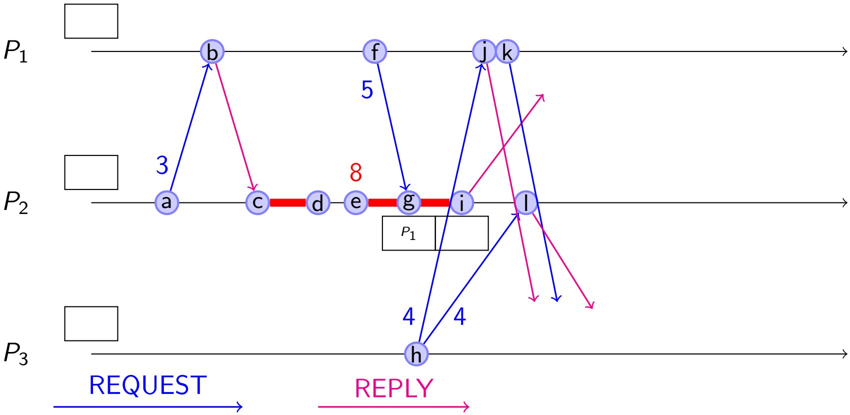
# Roucairol and Carvalho's Mutex: Illustration events




k. $P_1$ thought $P_3$ not active, until j.

$\leadsto$ send previous REQUEST now

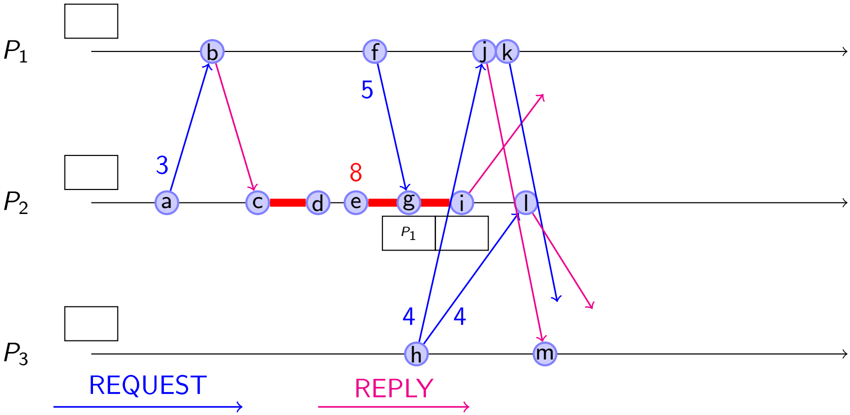
# Roucairol and Carvalho's Mutex: Illustration events




l. $P_2$ receives request from $P_3$

⤳ returns REPLY

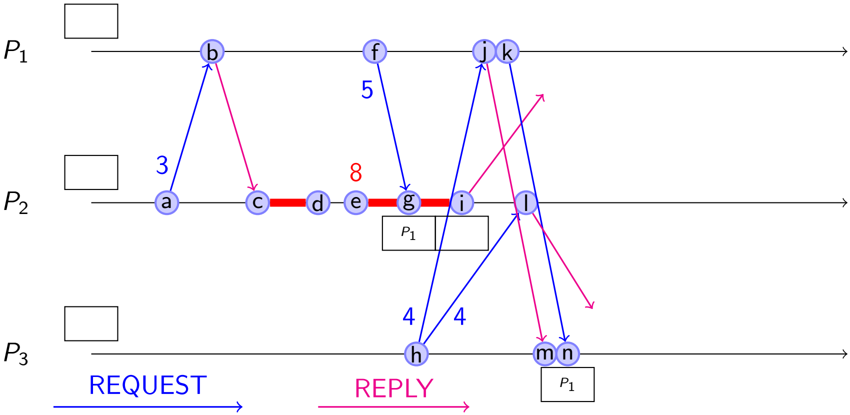
# Roucairol and Carvalho's Mutex: Illustration events




m. $P_3$ receives REPLY from $P_1$

(one missing)

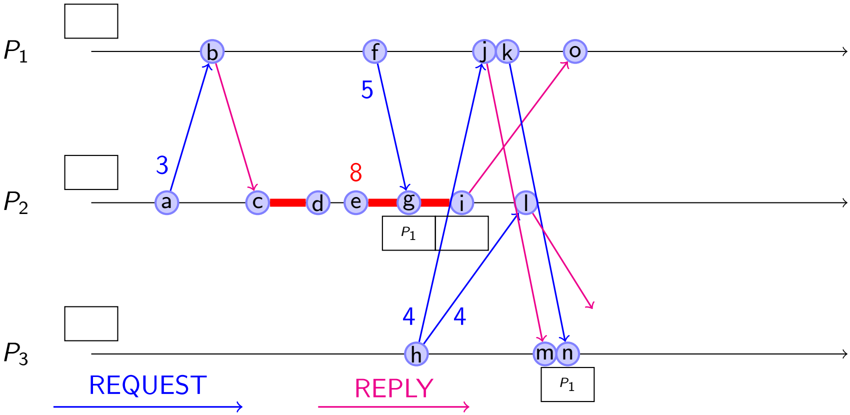
# Roucairol and Carvalho's Mutex: Illustration events




n. $P_3$ receives REQUEST from $P_1$

$\leadsto$ queues it because own timestamp lower

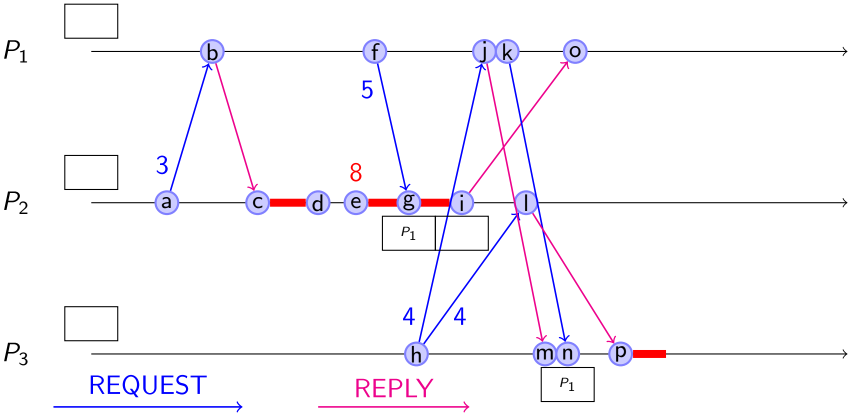
# Roucairol and Carvalho's Mutex: Illustration events




o. $P_1$ receives REPLY from $P_2$

(one missing)

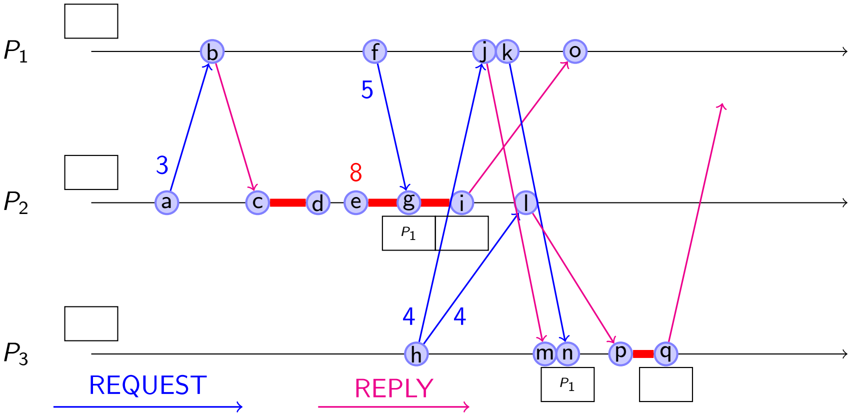
# Roucairol and Carvalho's Mutex: Illustration events




p. $P_3$ receives REPLY from $P_2$

everyone answered $\leadsto$ enters CS

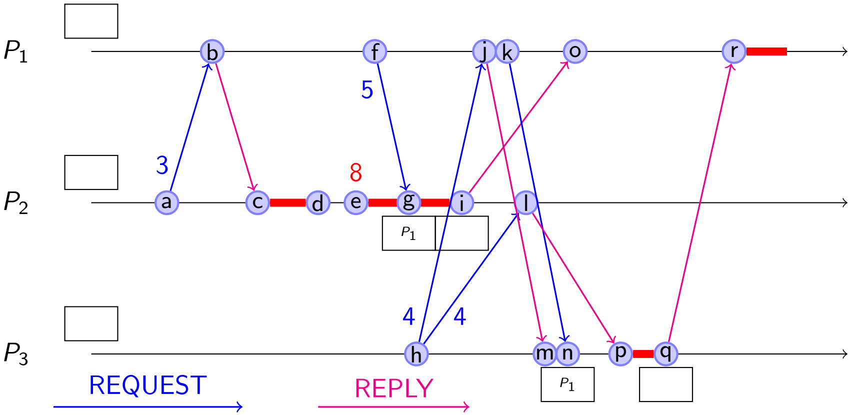
# Roucairol and Carvalho's Mutex: Illustration events




q. $P_3$ exits CS

$\rightsquigarrow$ send delayed REPLY to $P_1$

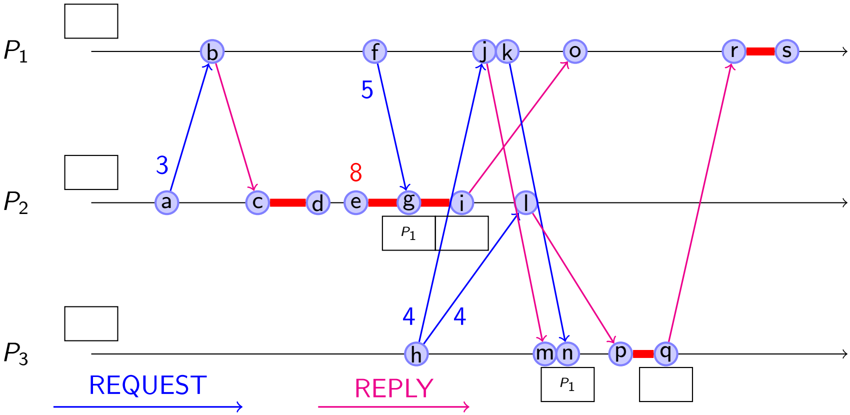
# Roucairol and Carvalho's Mutex: Illustration events




r. $P_1$ receives REPLY from $P_3$

everyone answered ⇝ enters CS

# Roucairol and Carvalho's Mutex: Illustration events




s. $P_1$ exits CS

(nothing to do)

# Roucairol and Carvalho's Mutex: Complexity Analysis

## Parameters

N Number of processes in the system

T Message transmission time

E Critical section execution time

## Message complexity:

- Best case: 0
- Worst case: 2(N-1): $N-1$ REQUEST messages + $N-1$ REPLY messages
- Message-size complexity: O(1)

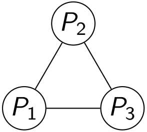
## Time complexity

- Response time (under light load):
  - Best case: E
  - Worst case: 2T+E
- Synchronization delay (under heavy load): T

# Token-Ring Algorithm

## Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring




## Illustration

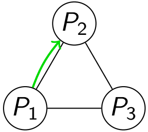

## Events

- Initially, $P_1$ has the token, and $P_2$ and $P_3$ want the CS. $P_1$ sends the token

# Token-Ring Algorithm

## Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring




## Illustration


## Events

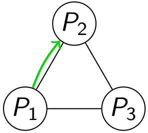
- Initially, $P_1$ has the token, and $P_2$ and $P_3$ want the CS. $P_1$ sends the token

d. $P_2$ gets the token $\leadsto$ enters CS.

# Token-Ring Algorithm

## Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring




## Illustration


## Events

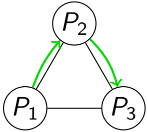
- Initially, $P_1$ has the token, and $P_2$ and $P_3$ want the CS. $P_1$ sends the token

d. $P_2$ gets the token $\rightsquigarrow$ enters CS. e. $P_2$ exits CS and send token to $P_3$

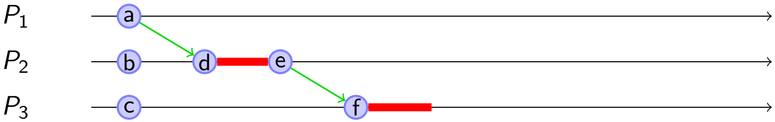
# Token-Ring Algorithm

## Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring




## Illustration




## Events

- Initially, $P_1$ has the token, and $P_2$ and $P_3$ want the CS. $P_1$ sends the token

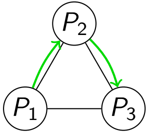
d. $P_2$ gets the token $\rightsquigarrow$ enters CS. e. $P_2$ exits CS and send token to $P_3$

f. $P_3$ gets the token $\rightsquigarrow$ enters CS.

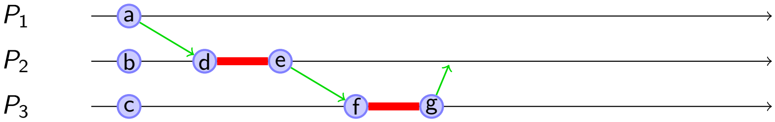
# Token-Ring Algorithm

## Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring




## Illustration




## Events

- Initially, $P_1$ has the token, and $P_2$ and $P_3$ want the CS. $P_1$ sends the token

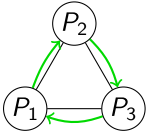
d. $P_2$ gets the token $\leadsto$ enters CS. e. $P_2$ exits CS and send token to $P_3$

f. $P_3$ gets the token $\leadsto$ enters CS. g. $P_3$ exits CS and send token to $P_1$

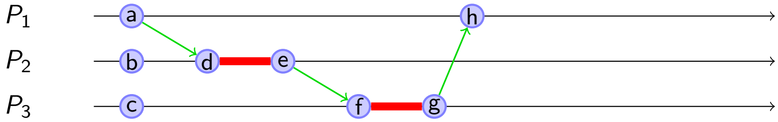
# Token-Ring Algorithm

## Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring




## Illustration




## Events

- Initially, $P_1$ has the token, and $P_2$ and $P_3$ want the CS. $P_1$ sends the token

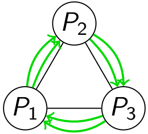
d. $P_2$ gets the token $\rightsquigarrow$ enters CS. e. $P_2$ exits CS and send token to $P_3$

f. $P_3$ gets the token $\rightsquigarrow$ enters CS. g. $P_3$ exits CS and send token to $P_1$

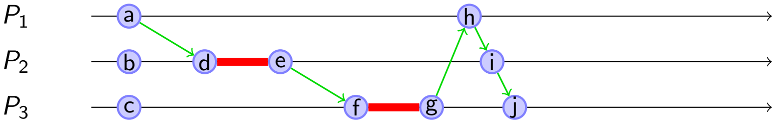
# Token-Ring Algorithm

## Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring




## Illustration




## Events

- Initially, $P_1$ has the token, and $P_2$ and $P_3$ want the CS. $P_1$ sends the token

d. $P_2$ gets the token $\leadsto$ enters CS. e. $P_2$ exits CS and send token to $P_3$

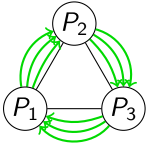
f. $P_3$ gets the token $\leadsto$ enters CS. g. $P_3$ exits CS and send token to $P_1$

- Seems interesting, but incredibly inefficient when nobody request the CS

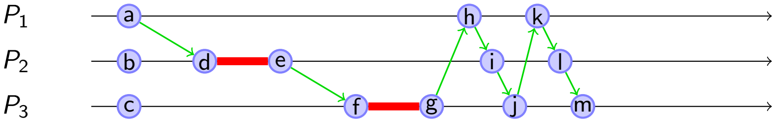
# Token-Ring Algorithm

## Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring




## Illustration




## Events

- Initially, $P_1$ has the token, and $P_2$ and $P_3$ want the CS. $P_1$ sends the token

d. $P_2$ gets the token $\leadsto$ enters CS. e. $P_2$ exits CS and send token to $P_3$

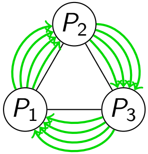
f. $P_3$ gets the token $\leadsto$ enters CS. g. $P_3$ exits CS and send token to $P_1$

- Seems interesting, but incredibly inefficient when nobody request the CS

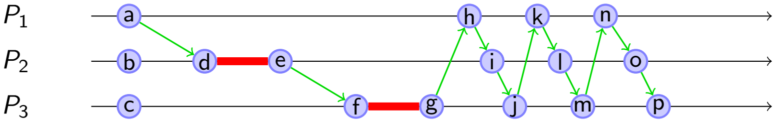
# Token-Ring Algorithm

## Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring




## Illustration




## Events

- Initially, $P_1$ has the token, and $P_2$ and $P_3$ want the CS. $P_1$ sends the token

d. $P_2$ gets the token $\rightsquigarrow$ enters CS. e. $P_2$ exits CS and send token to $P_3$

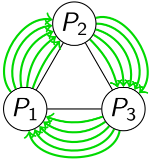
f. $P_3$ gets the token $\rightsquigarrow$ enters CS. g. $P_3$ exits CS and send token to $P_1$

- Seems interesting, but incredibly inefficient when nobody request the CS

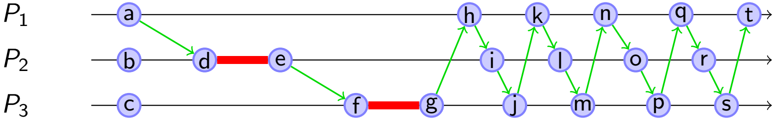
# Token-Ring Algorithm

## Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring




## Illustration




## Events

- Initially, $P_1$ has the token, and $P_2$ and $P_3$ want the CS. $P_1$ sends the token

d. $P_2$ gets the token $\leadsto$ enters CS. e. $P_2$ exits CS and send token to $P_3$

f. $P_3$ gets the token $\leadsto$ enters CS. g. $P_3$ exits CS and send token to $P_1$

- Seems interesting, but incredibly inefficient when nobody request the CS

# Suzuki and Kasami's Algorithm

## Main ideas

- Token-based (but not as inefficiently)
- The token is not passed automatically, but on request only

## Data structures

- Each process has a vector: v[i]=amount of CS request received from $P_i$
  This is a local variable
- The token contains 2 informations:
  - A vector: v[i]= amount of CS run for $P_i$
  - A FIFO: processes with unfulfilled requests

  This is a "global" variable, spead when possible
- These are not vector clocks

# Suzuki and Kasami's Algorithm Steps for $P_i$

## On requesting the CS

- If have token, enter CS
- If not, update request vector, then broadcast REQUEST to every processes

## On receiving a REQUEST from $P_j$

- Update request vector
- if (request is new) AND (have token) AND (token idle), then send token to $P_j$

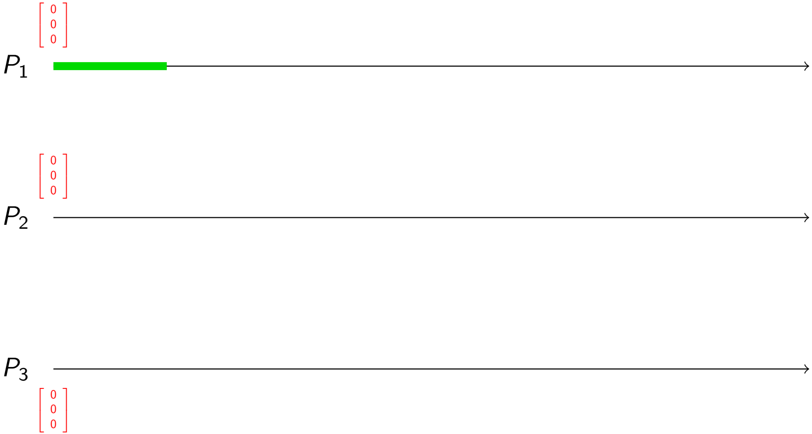
## On receiving the token

- Enter the CS

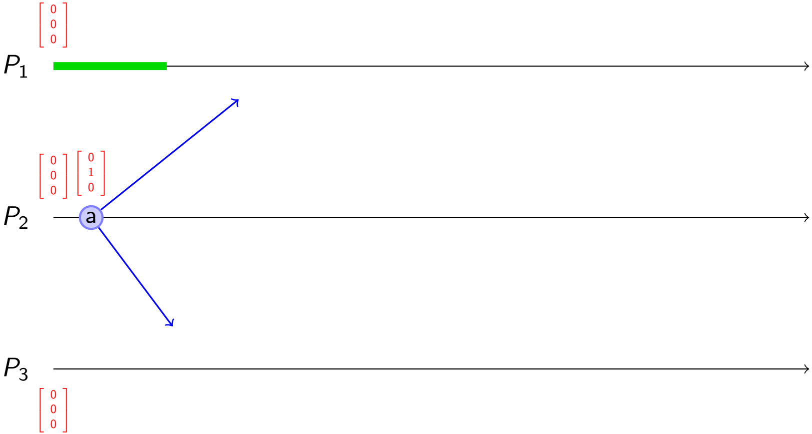
## On leaving the CS

- Update the token vector
- Add any unfulfilled requests from request vector to the token queue
- If token queue non-empty, then remove first and send the token that process

# Suzuki and Kasami's Algorithm: Illustration events



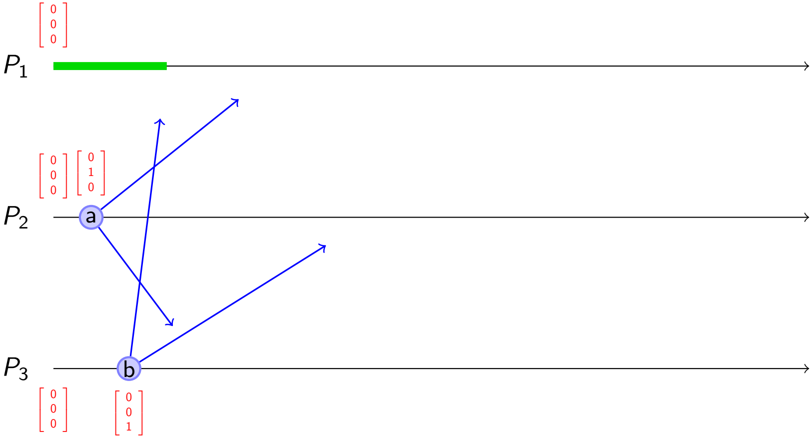
# Suzuki and Kasami's Algorithm: Illustration events




a. $P_2$ requests the CS

⇝ broadcasts the REQUEST

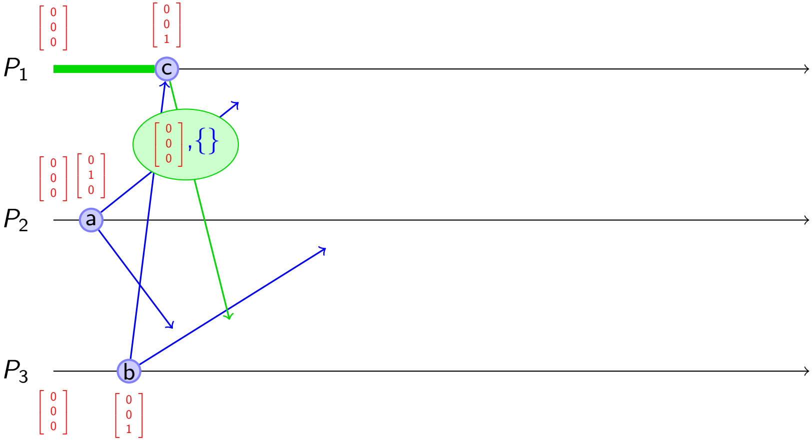
# Suzuki and Kasami's Algorithm: Illustration events




b. $P_3$ requests the CS

$\leadsto$ broadcasts the REQUEST

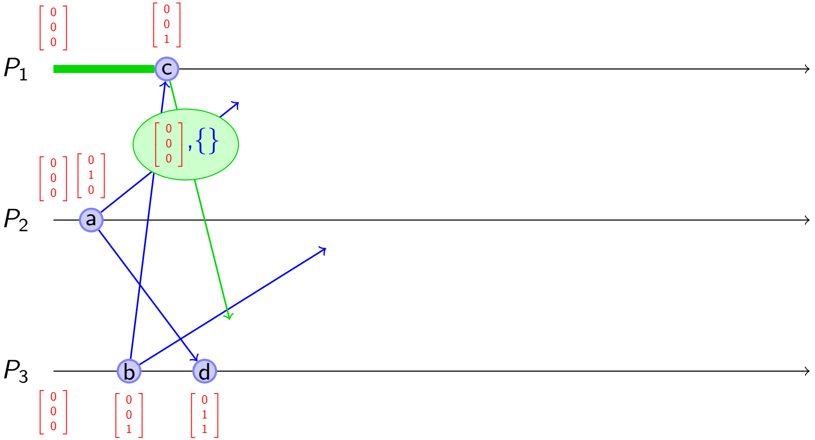
# Suzuki and Kasami's Algorithm: Illustration events




c. $P_1$ receives REQUEST from $P_3$.

$\rightsquigarrow$ Update request vector and send TOKEN

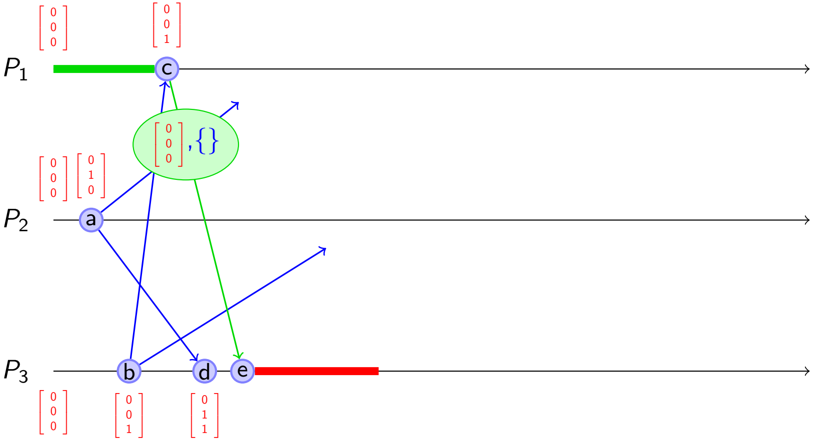
# Suzuki and Kasami's Algorithm: Illustration events




d. $P_1$ receives REQUEST from $P_3$.

$\rightsquigarrow$ update request vector

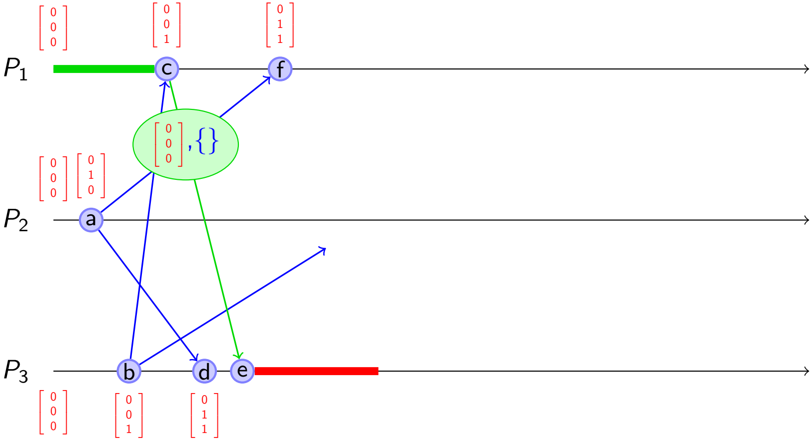
# Suzuki and Kasami's Algorithm: Illustration events




e. $P_3$ receives TOKEN

⤳ enters CS

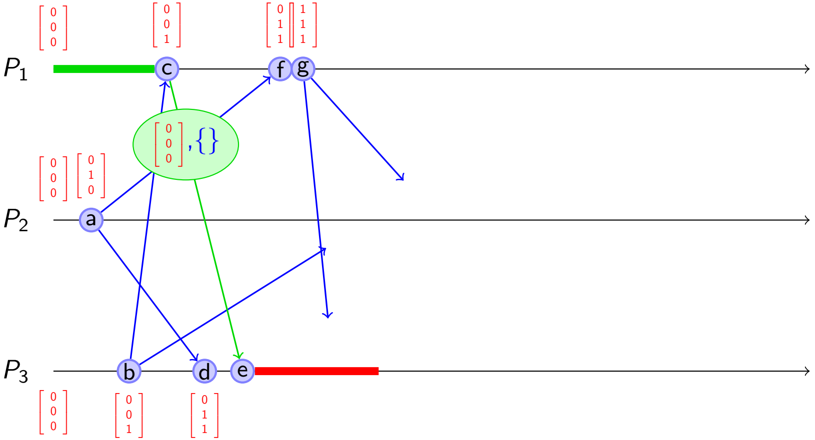
# Suzuki and Kasami's Algorithm: Illustration events




f. $P_1$ receives REQUEST from $P_2$

⇝ update request vector

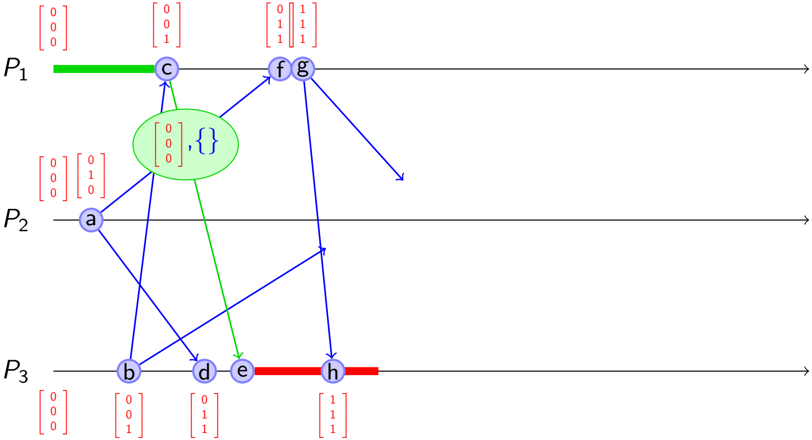
# Suzuki and Kasami's Algorithm: Illustration events




g. $P_1$ requests the CS

$\rightsquigarrow$ increment own entry, broadcast REQUEST to all

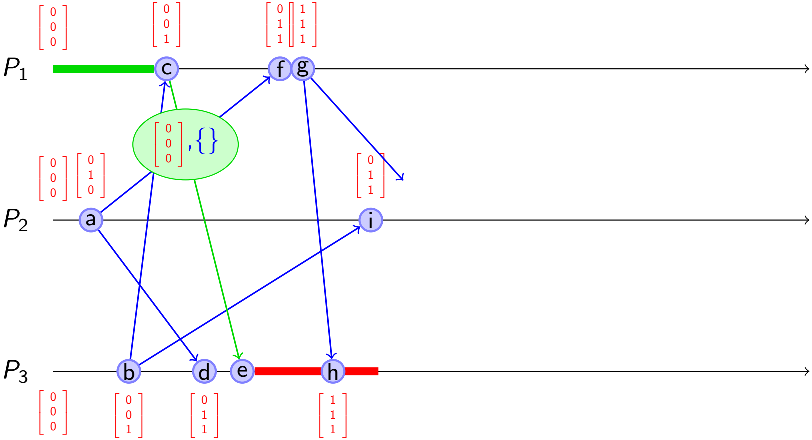
# Suzuki and Kasami's Algorithm: Illustration events




h. $P_3$ receives REQUEST from $P_1$

$\leadsto$ update request vector

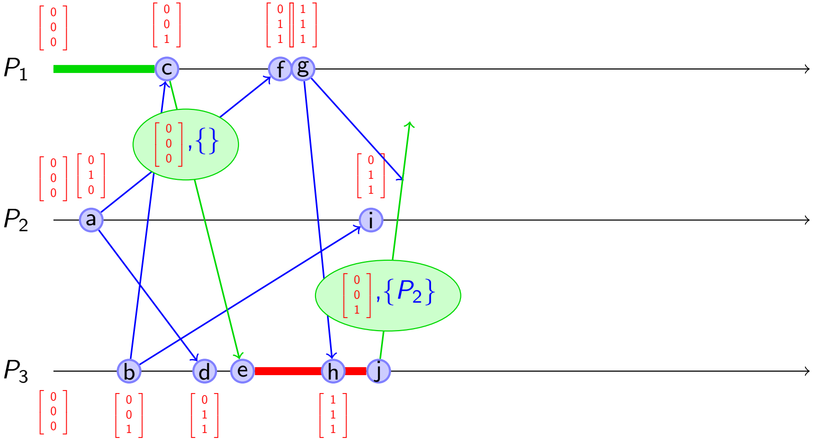
# Suzuki and Kasami's Algorithm: Illustration events




i. $P_2$ receives REQUEST from $P_3$

$\rightsquigarrow$ update request vector

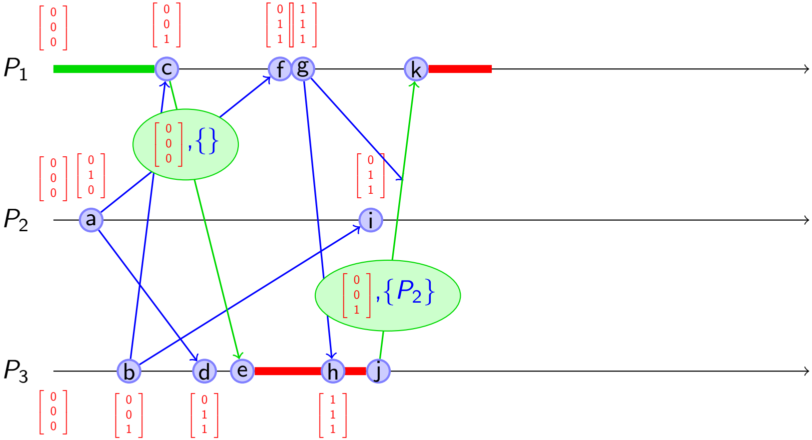
# Suzuki and Kasami's Algorithm: Illustration events




j. $P_3$ exits C.

- Update token vector to $\begin{bmatrix}0\\0\\1\end{bmatrix}$ since it just did a CS
- Compares request and token vectors. $\{P_1, P_2\}$: $\#req. > \#runs \rightsquigarrow$ Enqueue
- Send TOKEN to first of queue, $P_1$

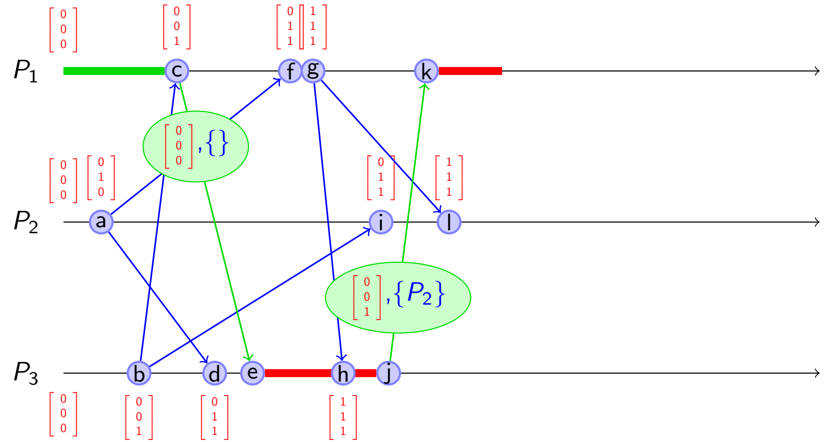
# Suzuki and Kasami's Algorithm: Illustration events




k. $P_1$ receives TOKEN

⇝ enters CS

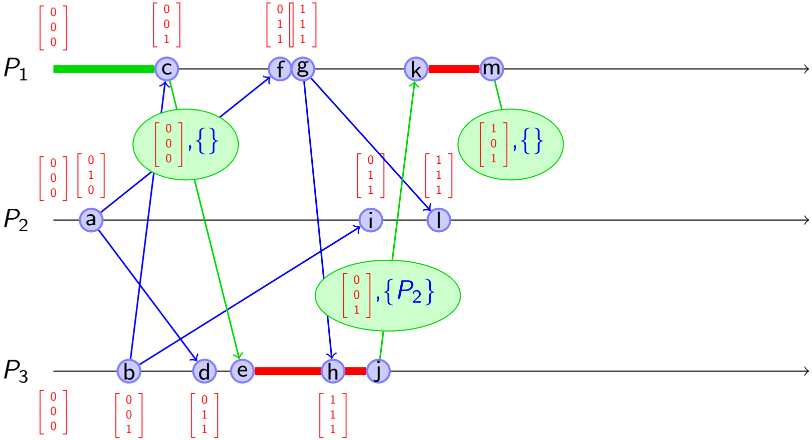
# Suzuki and Kasami's Algorithm: Illustration events




I. $P_2$ receives REQUEST from $P_1$

$\rightsquigarrow$ updates request vector

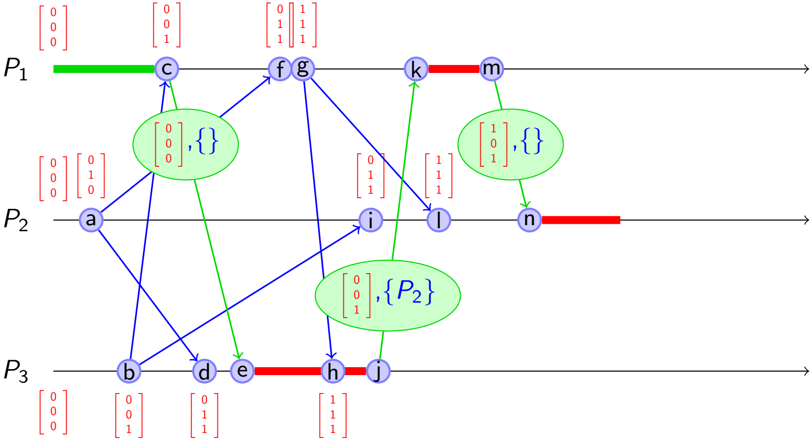
# Suzuki and Kasami's Algorithm: Illustration events




m. $P_1$ exits CS

Update token and send it to $P_2$

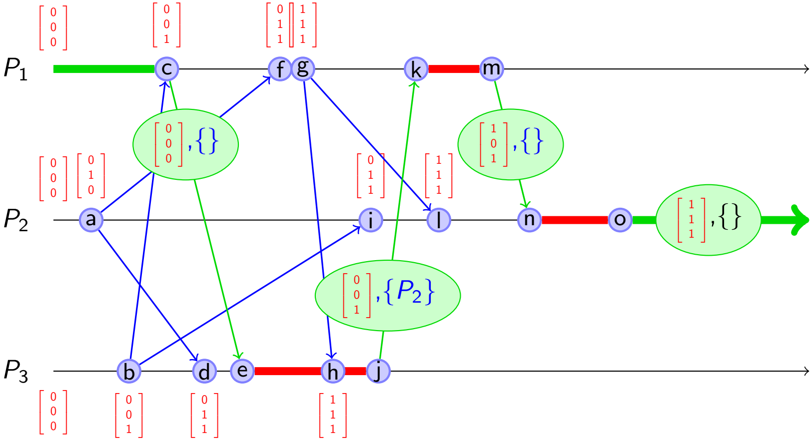
# Suzuki and Kasami's Algorithm: Illustration events




n. $P_2$ receives TOKEN

$\rightsquigarrow$ enters CS

# Suzuki and Kasami's Algorithm: Illustration events




o. $P_2$ exits CS

Update token and keep it

# Suzuki and Kasami's Algorithm: Complexity Analysis

## Parameters

N Number of processes in the system
T Message transmission time
E Critical section execution time

## Message complexity:

- Best case: 0
- Worst case: N= $(N-1)$ REQUEST + 1 TOKEN

## Message Size Complexity:

- Between 1 (REQUEST) and N (TOKEN)
- Average: O(1) (averaging over $(N-1)$ REQUEST and 1 TOKEN)

## Time complexity

- Response time (under light load): Best case: E; Worst case: 2T+E
- Synchronization delay (under heavy load): T

# (pedagical) Interest of this algorithm

## Builds a sort of distributed data structure

- Explicit list in token, which travels
- (built lazily by comparing local request vector to token vector)
- Request vectors are updated when receiving a REQUEST

## This concept is still somehow fuzzy

- List updated only when needed: when exiting the CS (lazy update)
- List updated by comparing local request vector to [global] token vector
- Request vectors are updated when receiving a REQUEST

## Other algorithm use distributed data structures more explicitely

- Raymond and Naimi-Trehel build a waiting queue, and a tree pointing to the waiting queue entry point

# Theoretical foundations


- Time and State of a Distributed System
- Ordering of events
- Abstract Clocks
  - Global Observer
  - Logical Clocks
  - Vector Clocks
- Some Distributed Algorithms
  - Mutual Exclusion
    - Coordinator-based Algorithm
    - Lamport's Algorithm
    - Ricart and Agrawala's Algorithm
    - Roucairol and Carvalho's Algorithm
    - Token-Ring algorithm
    - Suzuki and Kasami's Algorithm
  - Leader Election
  - Consensus
  - Ordering Messages
  - Group Protocols
- Conclusion on distributed algorithmic

# Leader Election

## Problem Statement

- The processes pick one and only one of them (and agree on which one)
- Use case: error recovery
  - Only one site recreates the (lost) token
  - Elect a new coordinator on need
- Election started by any process (maybe concurrent elections)
- Which one we pick is not important
- Difficulty: processes may fail during the election

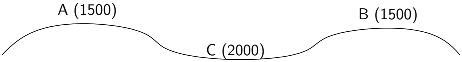
## Some approaches

- Bully Algorithm
  - Main idea
    - The one starting the election broadcasts its process number
    - Processes answer (take over) elections with a number smaller than their own
    - A process receiving no answer consider that he got elected
  - Remarks
    - Not very efficient algorithm ($O(n^2)$ messages at worst)
    - Robust to process failures, but not to asynchronism
- Ring $\Rightarrow$ Algorithm in $O(n \log(n))$ on average [Chang, Roberts]

# Consensus: First impossibility result

## Byzantin generals problem

- A and B want to attack C
- They must absolutely do it at the same time to succeed
- C can intercept messengers




A $\rightarrow$ B: Attack tomorrow
B $\rightarrow$ A: Got(Attack tomorrow)
A $\rightarrow$ B: Got(Got(Attack tomorrow))

A cannot be absolutely sure that B got his last message $\Rightarrow$ he does not attack

messages lost without detection $\rightsquigarrow$ consensus impossible (in finite amount of steps)

- Proof (reductio ad absurdum): Suppose $\exists$ such a protocol, consider $p = \{\ldots; A \rightarrow B : m_{n-1}; B \rightarrow A : m_n\}$ minimal in amout of messages.
  - B don't receive messages anymore $\Rightarrow$ casted its decision before $m_n$
  - Since $p$ works even if messages get lost, A casts its decision without $m_n$

  $\Rightarrow$ $m_n$ useless, and can be omitted from $p$. Contradiction with "$p$ is minimal"
- Only solution: detect message loss

# Consensus: An algorithm amongst others

## Lamport et al. (1982)

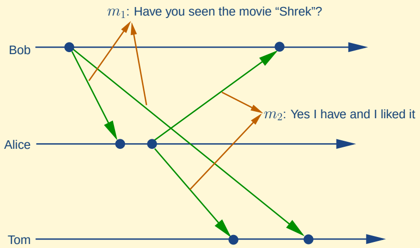
- Goal:
  - Generals want to inform each other of the present forces
- Assumptions:
  - Messages not corrupted (communication are *fail-stop*)
  - Receiver knows who sent the message
  - Communication time bounded (implementation: timestamp + timeouts + *fail-fast*)
- Result:
  - With $m$ malicious generals, need $2m + 1$ generals in total
  - Cannot identify malicious generals, only find correct values out
- Principe:
  1. Everyone broadcasts its own force to everyone
  2. Everyone broadcasts the vector of received values to everyone
  3. Everyone uses the vectors getting the majority of the casts

# Theoretical foundations


- Time and State of a Distributed System
- Ordering of events
- Abstract Clocks
  - Global Observer
  - Logical Clocks
  - Vector Clocks
- Some Distributed Algorithms
  - Mutual Exclusion
    - Coordinator-based Algorithm
    - Lamport's Algorithm
    - Ricart and Agrawala's Algorithm
    - Roucairol and Carvalho's Algorithm
    - Token-Ring algorithm
    - Suzuki and Kasami's Algorithm
  - Leader Election
  - Consensus
  - Ordering Messages
  - Group Protocols
- Conclusion on distributed algorithmic

# Ordering of Messages

- For many applications, messages should be delivered in certain order to be interpreted meaningfully
- Example:

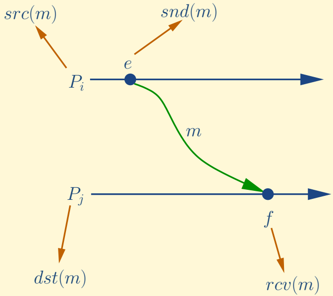



- $m_2$ cannot be interpreted until $m_1$ has been received
- Tom receives $m_2$ before $m_1$: an undesriable behavior


Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
❖ Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration
State of a Distributed System
Monitoring a Distributed System

# Useful Notations

- For a message $m$:
  - $src(m)$: source process of $m$
  - $dst(m)$: destination process of $m$
  - $snd(m)$: send event of $m$
  - $rcv(m)$: send event of $m$





Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
❖ Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration
State of a Distributed System
Monitoring a Distributed System

# Causal Delivery of Messages

- A message $w$ causally precedes a message $m$ if $snd(w) \rightarrow snd(m)$

- An execution of a distributed system is said to be causally ordered if the following holds for every message $m$:

  every message that *causally precedes* $m$ and is *destined for the same process* as $m$ is *delivered before* $m$

  Mathematically, for every message $w$:

$$(snd(w) \rightarrow snd(m)) \wedge (dst(w) = dst(m))$$
$$\Rightarrow$$
$$rcv(w) \rightarrow rcv(m)$$


Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
❖ Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration
State of a Distributed System
Monitoring a Distributed System

# A Causally Ordered Delivery Protocol

- Proposed by Birman, Schiper and Stephenson (BSS)
- Assumption:
  - communication is broadcast based: a process sends a message to every other process
- Each process maintains a vector with one entry for each process:
  - let $V_i$ denote the vector for process $P_i$
  - the $j^{th}$ entry of $V_i$ refers to the number of messages that have been broadcast by process $P_j$ that $P_i$ knows of


Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
❖ Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration
State of a Distributed System
Monitoring a Distributed System

# The BSS Protocol

- Protocol for process $P_i$:
  - On broadcasting a message $m$:

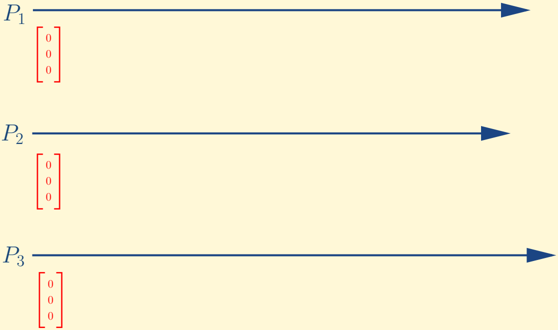
    piggyback $V_i$ on $m$
    $V_i[i] := V_i[i] + 1$
  - On arrival of a message $m$ from process $P_j$:

    let $V_m$ be the vector piggybacked on $m$
    deliver $m$ once $V_i \geq V_m$
  - On delivery of a message $m$ sent by process $P_j$:

    $V_i[j] := V_i[j] + 1$

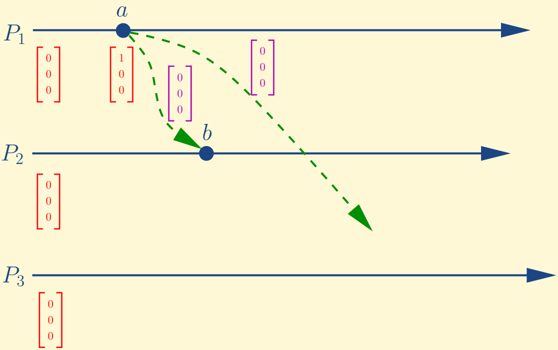

Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
❖ Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration
State of a Distributed System
Monitoring a Distributed System

# The BSS Protocol: An Illustration




Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
❖ Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration
State of a Distributed System
Monitoring a Distributed System

# The BSS Protocol: An Illustration





Inherent Limitations

Ordering of Events

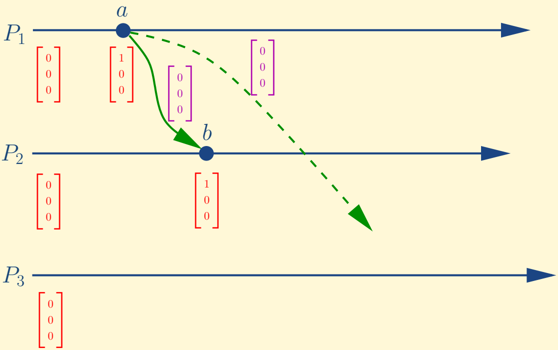
Abstract Clocks

Ordering of Messages
- Ordering of Messages
- Useful Notations
- Causal Delivery of Messages
- A Causally Ordered Delivery Protocol
- The BSS Protocol
- The BSS Protocol: An Illustration
- Another Causally Ordered Delivery Protocol
- When to Deliver a Message?
- The SES Protocol
- The SES Protocol (Continued)
- The SES Protocol: An Illustration

State of a Distributed System

Monitoring a Distributed System

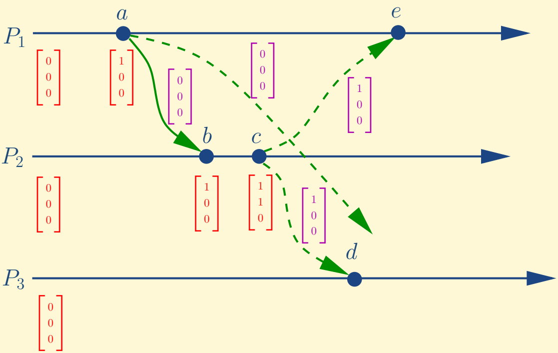
# The BSS Protocol: An Illustration





Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
❖ Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration
State of a Distributed System
Monitoring a Distributed System

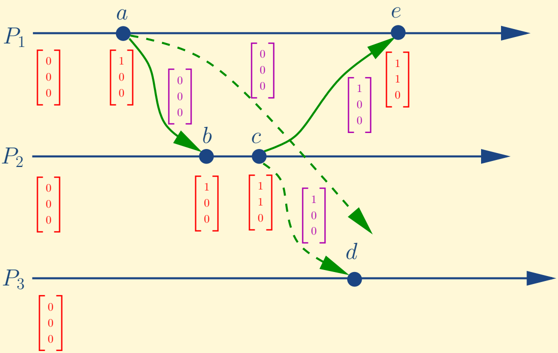
# The BSS Protocol: An Illustration





Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
❖ Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration
State of a Distributed System
Monitoring a Distributed System

# The BSS Protocol: An Illustration





Inherent Limitations

Ordering of Events

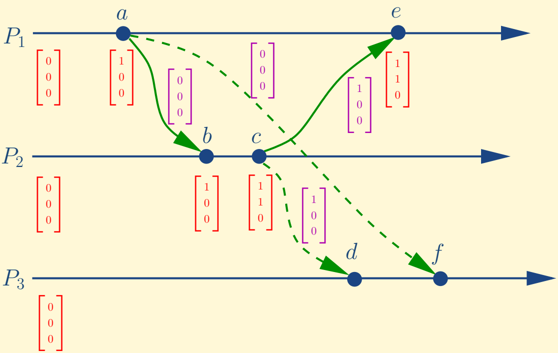
Abstract Clocks

Ordering of Messages
- Ordering of Messages
- Useful Notations
- Causal Delivery of Messages
- A Causally Ordered Delivery Protocol
- The BSS Protocol
- The BSS Protocol: An Illustration
- Another Causally Ordered Delivery Protocol
- When to Deliver a Message?
- The SES Protocol
- The SES Protocol (Continued)
- The SES Protocol: An Illustration

State of a Distributed System

Monitoring a Distributed System

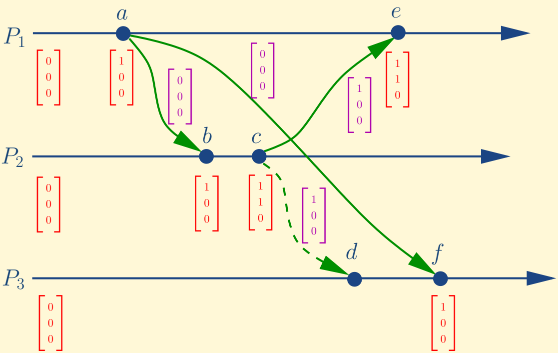
# The BSS Protocol: An Illustration





Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
❖ Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration
State of a Distributed System
Monitoring a Distributed System

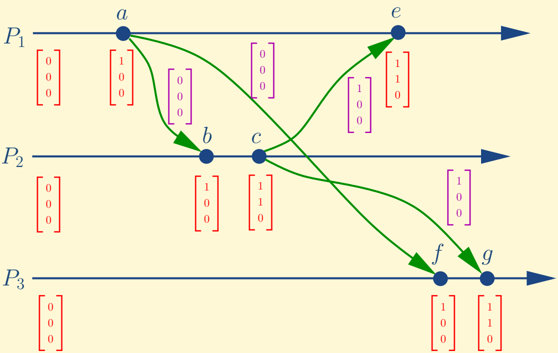
# The BSS Protocol: An Illustration





Inherent Limitations
Ordering of Events
Abstract Clocks
Ordering of Messages
❖ Ordering of Messages
❖ Useful Notations
❖ Causal Delivery of Messages
❖ A Causally Ordered Delivery Protocol
❖ The BSS Protocol
❖ The BSS Protocol: An Illustration
❖ Another Causally Ordered Delivery Protocol
❖ When to Deliver a Message?
❖ The SES Protocol
❖ The SES Protocol (Continued)
❖ The SES Protocol: An Illustration
State of a Distributed System
Monitoring a Distributed System

# The BSS Protocol: An Illustration





Inherent Limitations

Ordering of Events

Abstract Clocks

Ordering of Messages
- Ordering of Messages
- Useful Notations
- Causal Delivery of Messages
- A Causally Ordered Delivery Protocol
- The BSS Protocol
- The BSS Protocol: An Illustration
- Another Causally Ordered Delivery Protocol
- When to Deliver a Message?
- The SES Protocol
- The SES Protocol (Continued)
- The SES Protocol: An Illustration

State of a Distributed System

Monitoring a Distributed System

# Group Protocols

## Processes Group

- Definition: set of processes acting together
- Motivation:
  - Duplication (redundancy) of services
    Ex: servers group, duplicated data, clusters of computers
  - Cooperative work, Information sharing
- Problems:
  - Membership :
    dynamic knowledge of who's in the group (despite changes)
  - Broadcast and Multicast :
    communication between more than 2 processes (with specified properties)
    - Broadcast: send to every members
    - Multicast: send to some members
- Dynamic membership: arrival/departure, failures/restart

# Group Protocols: Main issues

## Specification difficulties

- Published specifications are often incomplete, incorrect, or ambiguous

## Algorithmic difficulties

- These protocols are difficult when taking failure into account
- Numerous impossible problems in asynchronous settings:
  (membership, atomic broadcast, synchronous views)
- Algorithmic instability:
  - Tiny specification changes can lead to huge change in implementation difficulty
  - Small change to protocol can lead to property violation
- Group protocols remains badly understood

⇒ Numerous researches (both theoretical and practical)

Chockler, et Al, *Group Communication Specifications : A Comprehensive Study*, 2001.

Meling and Helvik, *Performance Consequences of Inconsistent Client-side Membership Information in the Open Group Model*, IPCCC 2004.

# Group Protocols: Possible properties

## Properties on receivers

- Reliable diffusion: Message sent to every receivers, or to none
- Atomic diffusion (or totally ordered): Reliable+same order for all

## Properties on reception order

- FIFO: Messages from same sender are delivered in sending order
- Causal: Reception order respecting causal order on sending (implies FIFO)
  (forces an order of messages coming from differing senders)

## Time-related properties

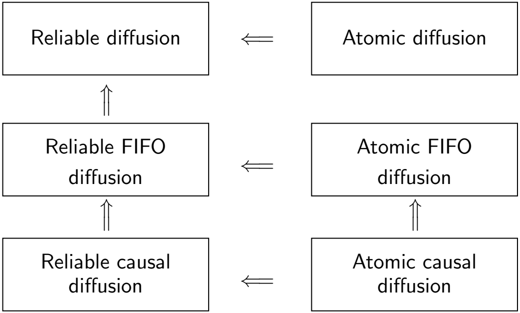
- Timed diffusion: No message is sent after a given delay
  (without underlying synchronous communication, you can only tend to this)

## Uniformity of a given property

- That property also apply to faulty processes

# Linking between these properties

- The four properties classes on group protocols are orthogonal
- Every combination exist
  - reliable without order, reliable FIFO, . . . , atomic causal
- Some combination imply other ones



# Conclusion

## What we saw

- Notion of distributed system (DS)
- Notion of time and state in a DS
- Main issues of faults in DS
- Expected properties of a DS:
  Safety, liveness (no deadlock, finishing), Scalability, Fault tolerance
- Classical problems in DS, and ideas of some algorithms
- Some classical approaches to solve these issues
  Order/abstract clocks, applicative topologies, Symmetry breaking (token, leader)

## What we didn't saw (because of lack of time)

- Notion of security in DS
- Every details of every algorithms
- A whole load of other problems, also quite classical:
  Wave algorithms; Distributed commits (2PC/3PC); Checkpointing; Ending detection

# What you should remember

## The models

- No shared time, no shared memory
- Asynchronism, Failures

## The tools

- Abstract clocks, applicative topologies, token-based

## The presented algorithms

- Mutex: Centralized, Lamport, Ricart/Agrawala, Roucairol/Carvalho, Suzuki/Kasami
  (you should be able to run them on a provided initial situation)
- The other ones (only the spirit)

## I hope you got the spirit of classical DS

- Even if I would need more time to get into real details

# Chapter 3

## Internet


- Theory vs. Practice
- The Models of Internet
- Internet Design
  - Brewer's Theorem
- Conclusion

# Theoretical Distributed Algorithmic vs. Internet

## Genesis

- At the beginning there were the mainframe
- Then came the PC and the local network (LAN)
- Then, people wanted clusters of PC to look alike the mainframe
- They proved theorems, builded file systems and distributed databases
- Then the Internet and the Web came, and *blowed* everything away

## Why DS failed?

- DS approach attracting, promising premises (theoretical and practical)
- Limitations of this approach (solved by the web):
  - Systems not autonomous: Domino effect on failures, co-configuration.
  - Complexity: In design, configuration and usage (and thus, cost)
  - Scalability: Impossible to use more than a few dozen servers, hundreds nodes

## The Internet and Web promise:

- Maximal autonomy, and scaling consequently
- Issues in data consistency (but who cares?)

# The Internet and the Web

## Inter-net

- This is the network of networks
- Assembled by interconnecting everything
- Started in 1969 with 4 nodes in one network
- Now, billions of elements

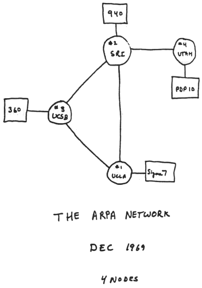
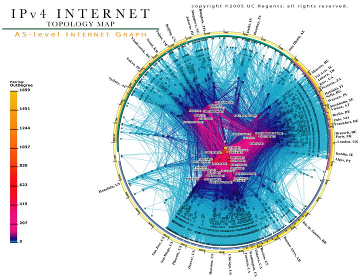



FIGURE 6.2 Drawing of 4 Node Network
(Courtesy of Alex McKenzie)

## The Web

- This is one of the application on the Internet
- Web pages browsing
- History: hypertext at CERN in 1990
- Whole load of applications on Internet:
  Mails, Voice-over-IP, Online Games, P2P




## Goal now: How does it work?

- What are the big ideas (models)?
- Classical way of solving problems

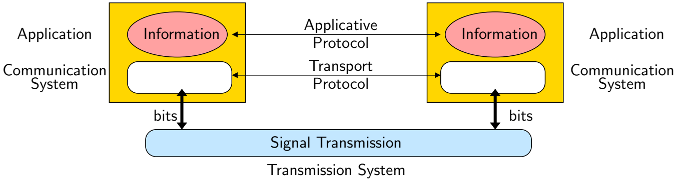
# Layered Protocol Stack

## Complexity of Existing Networks

- Lot of differing element categories (hosts, routers, links, applications)
- Lot of sort of elements in each category (huge amount of router models)

## How to deal with this complexity

- Several layers, each solves one given issue (problem separation)
- Each layer defines:
  - SAP (Service Access Point): service offered to higher layers
  - Protocol between peers (using services of lower layers)
  - PDU (Protocol Data Unit): format of exchanged data




- Advantages: Decomposing, Reusability, Separate interface/implementation
- Issues: Performance loss

# The OSI Model

## Organization in seven layers

- Applications: Common functions
- Presentation: Data representation and encryption (XDR)
- Session: Interhost communication (dialog setup)
- Transport: end-to-end connexion, reliability (fragmentation, multiplexing, streaming)
- Network: Path determination and logical addressing (routing, congestion, interconnection)
- Liaison: Physical addressing (Transmission between 2 sites, packet delimiting)
- Physical: Signal Transmission (converting between bits and signal)

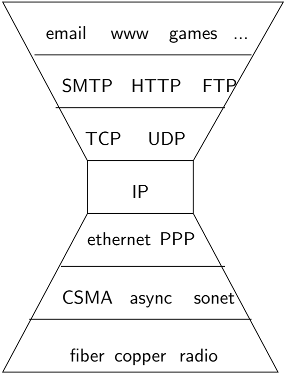
## Problems

- Standardization too slow, not (always) implemented
- Represents more than a 1m-high pile of paper

# The TCP/IP Model

That's what got implemented

- Applications
- Transport: Transport between processes
- Network: Routing
- Transmission: On local network

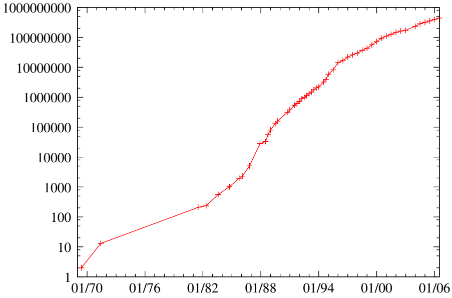



Look how it draws a hourglass centered on IP

# Internet Design

## Internet History

- 1969 : ARAPANET, Internet's ancestor
  Military System during cold war
  "Fault" tolerance ⇒ decentralized
- 1978 : first email
- 1978 : *Lamport's clock*
- 1991 : HTTP and WWW
- 1995 : Yahoo and altavista




http://www.isc.org/index.pl?/ops/ds/host-count-history.php

## Conception choices of the Internet

- The Distributed Algorithmic developed *in parallel*
- The designers of the Internet were pragmatics
- Theoretical sacrifices for quick usability

# Dealing with misconfiguration in IP

## Problem

- Each administrator configures its own machine
- Misconfigurations may lead to cycles in shortest path (for example)

⇒ "Mad" packets saturate the network

## Solution

- Each packet has a given Time To Live (TTL – that's a logical time)
- Each router decreases the TTL of packets it routes
- A packet which TTL reaches 0 is eliminated

## Issue induced by the solution

- The transport layer can lose packets
- Higher layer must deal with it...

# TCP: Adding Reliability

## Problem

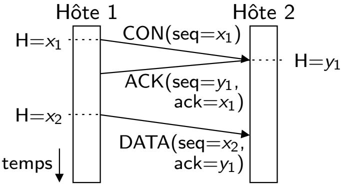
- Messages streams may arrive out of order
- Each message may get lost, late or duplicated

## Solution 1

- Packets are numbered, and delivered in order only

## Solution 2

- Expects an ACK for every message (re-emit after timeout)
- Duplicated are detected from seq number
- Olds ($>$ 120 sec) and dups are eliminated




## Services offered to higher layers

- FIFO channel without undetected loss
- (but also congestion handling)

# Résolution de noms sur Internet: DNS

## Motivation

- Les humains n'aiment pas les IP, les machines n'aiment pas les noms longs

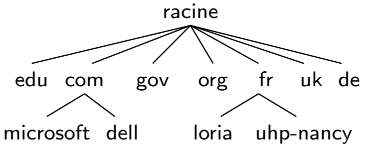
⇒ besoin d'un service d'annuaire

## Problème

- Un annuaire unique ne passe pas à l'échelle
  (volume données, point faible, distance=latence, maintenance)

## Solution

- Base de données distribuée et hiérarchique
- Autorité délégué dans chaque branche
- Caches locaux de données




## Gestion de la cohérence entre copies: soft-state

- Les enregistrements ont un âge maximal
- Ensuite, il les rafraîchir (redemander à une source d'autorité)

⇒ problèmes de cohérence existants, mais limités dans le temps

# Theory or practice?

## Bases of the misunderstanding

- Academics like clear abstractions and pure models
- Users like systems which work (the most often)
- Scalability is cost-effective (scale savings, increased market shares)
- Perfect consistency rarely mandatory in real life

**Brewer's Theorem** (PODC'00 – proof by Gilbert&Lynch, 2002)

*From the three following goals, you can have two at most!*

- *Consistent (broadly defined)*
- *Available*
- *Partitions don't stop system*

# Le choc des cultures

## Systèmes distribués classiques: sémantique ACID

- A: Atomicité (tous ou personne)
- C: Consistance
- I: Isolation
- D: Durable

## Systèmes utilisés sur Internet: sémantique BASE

- BA: *Basically Available* (souvent disponible)
- S: *Soft-state* (ou *scalable*)
- E: *Eventually consistant* consistance à terme

### ACID

- Consistance avant tout
- Disponibilité moins fondamental
- Pessimiste
- Analyse rigoureuse
- Mécanismes complexes

### BASE

- Disponibilité avant tout
- Consistance faible acceptée
- Optimiste
- *Best-effort*
- Simple et rapide

# Brewer's Theorem

## What can we expect from a distributed system?

- Strong Consistency: every node share the same view, even during updates
- High Availability: every node can find replica, even when some other nodes fail
- Partition Tolerance: properties kept when system partitioned (network failures)

## CAP Theorem (Conjectured by Brewer)

- From these three systemic requirements, you can get at most two
- The choice of the forgotten one has strong implications

E. Brewer. *Towards robust distributed systems.* (Invited Talk) PODC 2000.

Gilbert & Lynch *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*, ACM SIGACT 33:2, 2002

# Possible Design Choices

## Consistency and Availability

- If you want transactions, you must get (and keep) your node connected
- Approaches: (classical in distributed algorithmic)
  - Two-phase commit; Cache invalidation (cf. coherence corpus)
- Examples: systems for LANs (DB, FS, ...)

## Consistency and Partition-Tolerance

- System freeze allowed $\leadsto$ consistency even if transient partition
- Approaches: (also classical in distributed algorithmic)
  - Pessimistic locks; Quorums and Elections (detecting the partitioning)
- Examples: distributed DB, distributed locks

## Availability and Partition-Tolerance

- When you forget about consistency, everything becomes easier
- Approaches: (typical on the Internet)
  - TTL and soft-state; Optimistic updates with conflict resolution
- Examples: DNS, Cache Web

# Conclusion on historical distributed systems

## What we saw

- Algorithmic of distributed systems is complex
  - Lots of impossibility results
  - Easy problems quite rare
  - Hard to quantify the cost of a solution and its matching to the needs
- Some existing systems (Internet) are much more pragmatic
  - Exchange strong consistency for good availability and partition-tolerance
  - Mandatory for scalability

- These are two distinct origins of the modern research in distributed systems

$\Rightarrow$ *Very* active domain

## Ce que nous ne verrons pas ici

- Systèmes temps réel (industriels, militaire), applications de la théorie
- Solution de programmation distribuée
  - Distribution implicite: RPC, Java RMI, CORBA, J2EE, .NET.
  - Distribution explicite (pour schizo ;): Sockets BSD, Erlang, mOZart, *GRAS*.

# Quatrième chapitre

## Peer-to-Peer Systems

- Introduction
  - Overlays
  - Current P2P Applications
  - Worldwide Computer Vision
- Unstructured P2P File Sharing
  - Napster
  - Gnutella
  - KaZaA
- Structured P2P: DHT Approaches
  - DHT service, issues and seminal ideas
  - Chord
  - CAN
  - Pastry
- Applications
  - File sharing using DHT
  - Persistent file storage
  - Mobility Management
  - Content Distribution Networks
  - BitTorrent
  - Anonymous Activities
    - Storm Botnet
    - Tor System
- Quelques défis supplémentaires
  - Proximité réseau
  - Confiance entre participants
  - Dynamicité du système
- Conclusion

# Peer-to-Peer: What is it?

## *Peer* definition from Merriam-Webster:

- one that is of equal standing with another;
- one belonging to the same societal group (based on age, grade, or status)

## Definition of P2P

1. Significant autonomy from central servers
2. Exploits resources at the edges of the Internet
   (storage and content, CPU cycles, human presence)
3. Individual nodes have intermittent connectivity, being added & removed

- Not strict requirements, instead typical characteristics

## It's a broad definition:

- P2P file sharing: Napster, Gnutella, KaZaA, eDonkey, etc
- P2P communication: Instant messaging, Voice-over-IP (Skype)
- P2P computation: seti@home, volunteer computing
- DHTs (& apps): Chord, CAN, Pastry, Tapestry

# Motivations

## Promises

- Organic growth (lower deployment and operating costs)
- Independent from the infrastructures
- Scalable, Robust

## There is a strong need for such systems

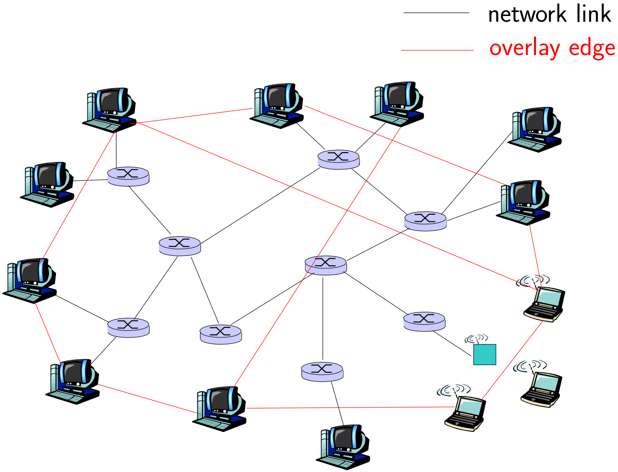
- Cooperative computations
- Robust services
- Ad-hoc networks
- It's hard to setup a large network otherwise

## Technology make these systems possible

- Computer always more powerful: every PC can be a server
- Wireless systems
- New algorithms for scalable systems
- Solutions to build safe systems from unsafe components

# P2P Systems Organization: Overlays

## Overlay Networks




**Layers**

| |
|---|
| Applications |
| Support for decentralized applications (overlay) |
| Network (TCP, UDP) |

# Overlay Graph

## Virtual edge

- TCP connection
- or simply a pointer to an IP address

## Overlay maintenance

- Periodically ping to make sure neighbor is still alive
- Or verify liveness while messaging
- If neighbor goes down, may want to establish new edge
- New node needs to bootstrap

## Kind of overlays

- Unstructured overlays: e.g., new node randomly chooses three existing nodes as neighbors
- Structured overlays: e.g., edges arranged in restrictive structure
- Network Proximity: Not necessarily taken into account

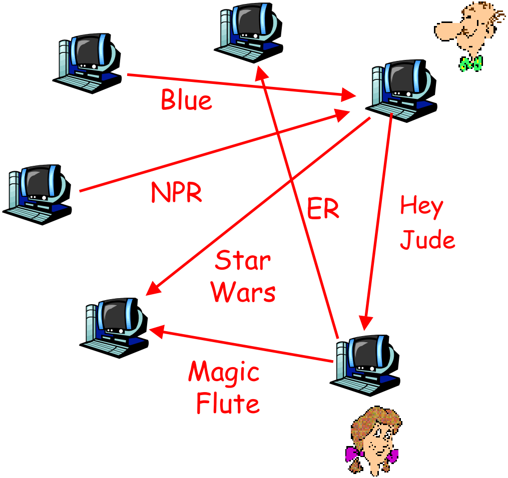
# 1. Overview of P2P

- overlay networks
- current P2P applications
  - P2P file sharing & copyright issues
  - Instant messaging / voice over IP
  - P2P distributed computing
- worldwide computer vision

# P2P file sharing

- Alice runs P2P client application on her notebook computer
- Intermittently connects to Internet; gets new IP address for each connection
- Registers her content in P2P system
- Asks for "Hey Jude"
- Application displays other peers that have copy of Hey Jude.
- Alice chooses one of the peers, Bob.
- File is copied from Bob's PC to Alice's notebook: P2P
- While Alice downloads, other users uploading from Alice.

# Millions of content servers




14

# Killer deployments

- Napster
  - disruptive; proof of concept
- Gnutella
  - open source
- KaZaA/FastTrack
  - Today more KaZaA traffic then Web traffic!
- eDonkey / Overnet
  - Becoming popular in Europe
  - Appears to use a DHT

Is success due to massive number of servers, or simply because content is free?

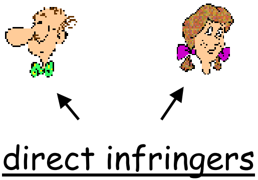
# P2P file sharing software

- Allows Alice to open up a directory in her file system
  - Anyone can retrieve a file from directory
  - Like a Web server
- Allows Alice to copy files from other users' open directories:
  - Like a Web client
- Allows users to search nodes for content based on keyword matches:
  - Like Google

# Copyright issues (1)

Direct infringement:

- end users who download or upload copyrighted works




Indirect infringement:

- Hold an individual accountable for actions of others
- Contributory
- Vicarious

# Copyright issues (2)

**Contributory infringer:**

- knew of underlying direct infringement, and
- caused, induced, or materially contributed to direct infringement

**Vicarious infringer:**

- able to control the direct infringers (e.g., terminate user accounts), and
- derived direct financial benefit from direct infringement (money, more users)

(knowledge not necessary)

# Copyright issues (3)

**Betamax VCR defense**

- Manufacturer not liable for contributory infringement
- "capable of substantial non-infringing use"
- But in Napster case, court found defense does not apply to all vicarious liability

**Guidelines for P2P developers**

- total control so that there's no direct infringement

or

- no control over users – no remote kill switch, automatic updates, actively promote non-infringing uses of product
- Disaggregate functions: indexing, search, transfer
- No customer support

# Instant Messaging

- Alice runs IM client on her PC
- Intermittently connects to Internet; gets new IP address for each connection
- Registers herself with "system"
- Learns from "system" that Bob in her buddy list is active
- Alice initiates direct TCP connection with Bob: P2P
- Alice and Bob chat.

- Can also be voice, video and text.

We'll see that Skype is a VoIP P2P system

# P2P Distributed Computing

seti@home

- Search for ET intelligence
- Central site collects radio telescope data
- Data is divided into work chunks of 300 Kbytes
- User obtains client, which runs in backgrd
- Peer sets up TCP connection to central computer, downloads chunk
- Peer does FFT on chunk, uploads results, gets new chunk

Not peer to peer, but exploits resources at network edge

# 1. Overview of P2P

- overlay networks
- P2P applications
- worldwide computer vision

# Worldwide Computer Vision

Alice's home computer:

- Working for biotech, matching gene sequences
- DSL connection downloading telescope data
- Contains encrypted fragments of thousands of non-Alice files
- Occasionally a fragment is read; it's part of a movie someone is watching in Paris
- Her laptop is off, but it's backing up others' files

- Alice's computer is moonlighting
- Payments come from biotech company, movie system and backup service

Your PC is only a component in the "big" computer

# Worldwide Computer (2)

**Anderson & Kubiatowicz: Internet-scale OS**

- Thin software layer running on each host & central coordinating system running on ISOS server complex
- allocating resources, coordinating currency transfer
- Supports data processing & online services

**Challenges**

- heterogeneous hosts
- security
- payments

**Central server complex**

- needed to ensure privacy of sensitive data
- ISOS server complex maintains databases of resource descriptions, usage policies, and task descriptions

# Quatrième chapitre

## Peer-to-Peer Systems

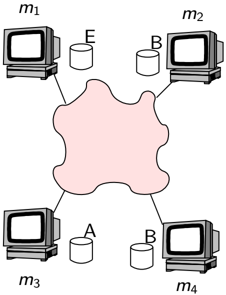

- Introduction
  - Overlays
  - Current P2P Applications
  - Worldwide Computer Vision
- Unstructured P2P File Sharing
  - Napster
  - Gnutella
  - KaZaA
- Structured P2P: DHT Approaches
  - DHT service, issues and seminal ideas
  - Chord
  - CAN
  - Pastry
- Applications
  - File sharing using DHT
  - Persistent file storage
  - Mobility Management
  - Content Distribution Networks
  - BitTorrent
  - Anonymous Activities
    - Storm Botnet
    - Tor System
- Quelques défis supplémentaires
  - Proximité réseau
  - Confiance entre participants
  - Dynamicité du système
- Conclusion

# Historique des systèmes pair-à-pair

## Motivations

- Permet d'obtenir de la musique (mp3) gratuitement de l'internet
- Principe: partager stockage et bande passante des participants (individus)
- Modèle: Tout le monde peut télécharger de ce que chacun stocke
- Difficultés principales:
  - Échelle: des milliers, des millions de machines
  - Dynamicité: les machines viennent et partent à tout moment (*churn*)

## Napster (popularized P2P even if Eternity [Ross Anderson] exists since 96)



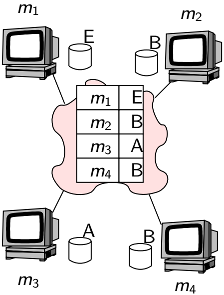
# Historique des systèmes pair-à-pair

## Motivations

- Permet d'obtenir de la musique (mp3) gratuitement de l'internet
- Principe: partager stockage et bande passante des participants (individus)
- Modèle: Tout le monde peut télécharger de ce que chacun stocke
- Difficultés principales:
  - Échelle: des milliers, des millions de machines
  - Dynamicité: les machines viennent et partent à tout moment (*churn*)

## Napster (popularized P2P even if Eternity [Ross Anderson] exists since 96)

- Index centralisé du contenu de toutes les machines
- Après la recherche, échange entre clients (P2P)
- Avantages:
  - Simple à implémenter
  - Possibilité de recherche avancée
- Défauts:
  - Extensibilité (?)
  - Point central (*single point of failure*)



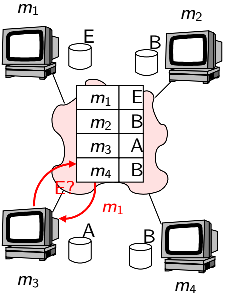
# Historique des systèmes pair-à-pair

## Motivations

- Permet d'obtenir de la musique (mp3) gratuitement de l'internet
- Principe: partager stockage et bande passante des participants (individus)
- Modèle: Tout le monde peut télécharger de ce que chacun stocke
- Difficultés principales:
  - Échelle: des milliers, des millions de machines
  - Dynamicité: les machines viennent et partent à tout moment (*churn*)

## Napster (popularized P2P even if Eternity [Ross Anderson] exists since 96)

- Index centralisé du contenu de toutes les machines
- Après la recherche, échange entre clients (P2P)
- Avantages:
  - Simple à implémenter
  - Possibilité de recherche avancée
- Défauts:
  - Extensibilité (?)
  - Point central (*single point of failure*)



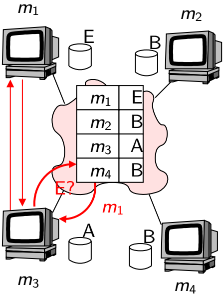
# Historique des systèmes pair-à-pair

## Motivations

- Permet d'obtenir de la musique (mp3) gratuitement de l'internet
- Principe: partager stockage et bande passante des participants (individus)
- Modèle: Tout le monde peut télécharger de ce que chacun stocke
- Difficultés principales:
  - Échelle: des milliers, des millions de machines
  - Dynamicité: les machines viennent et partent à tout moment (*churn*)

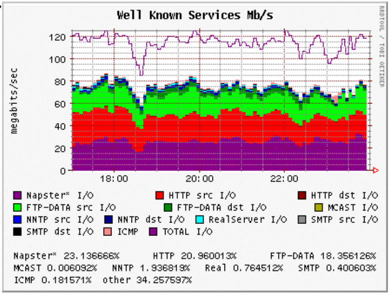
## Napster (popularized P2P even if Eternity [Ross Anderson] exists since 96)

- Index centralisé du contenu de toutes les machines
- Après la recherche, échange entre clients (P2P)
- Avantages:
  - Simple à implémenter
  - Possibilité de recherche avancée
- Défauts:
  - Extensibilité (?)
  - Point central (*single point of failure*)



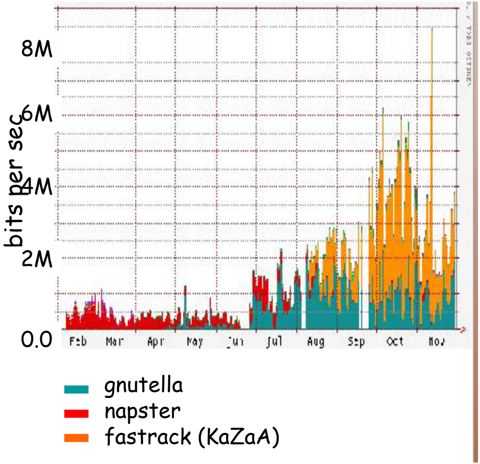
# Napster

- program for sharing files over the Internet
- a "disruptive" application/technology?
- history:
  - 5/99: Shawn Fanning (freshman, Northeasten U.) founds Napster Online music service
  - 12/99: first lawsuit
  - 3/00: 25% UWisc traffic Napster
  - 2/01: US Circuit Court of Appeals: Napster knew users violating copyright laws
  - 7/01: # simultaneous online users: Napster 160K, Gnutella: 40K, Morpheus (KaZaA): 300K



# Napster

- judge orders Napster to pull plug in July '01
- other file sharing apps take over!

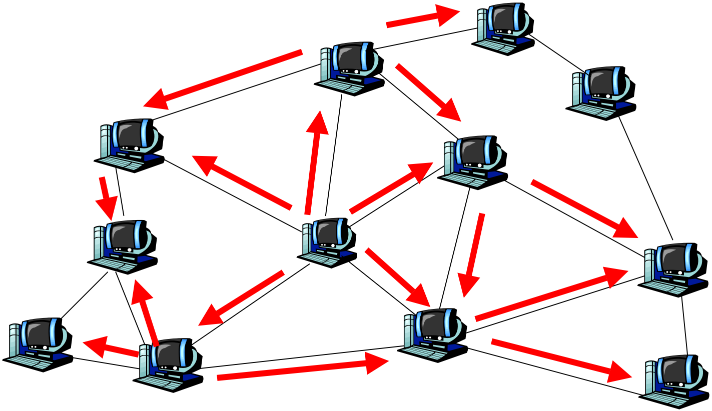


# Quatrième chapitre

## Peer-to-Peer Systems

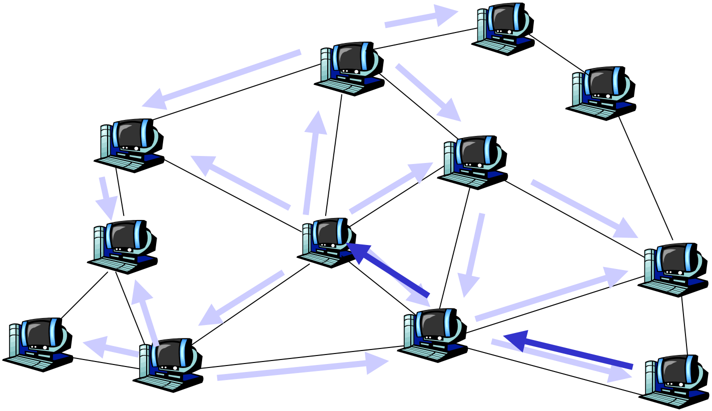

- Introduction
  - Overlays
  - Current P2P Applications
  - Worldwide Computer Vision
- Unstructured P2P File Sharing
  - Napster
  - Gnutella
  - KaZaA
- Structured P2P: DHT Approaches
  - DHT service, issues and seminal ideas
  - Chord
  - CAN
  - Pastry
- Applications
  - File sharing using DHT
  - Persistent file storage
  - Mobility Management
  - Content Distribution Networks
  - BitTorrent
  - Anonymous Activities
    - Storm Botnet
    - Tor System
- Quelques défis supplémentaires
  - Proximité réseau
  - Confiance entre participants
  - Dynamicité du système
- Conclusion

# Distributed Search/Flooding



35

# Distributed Search/Flooding



# Gnutella

- focus: decentralized method of searching for files
  - central directory server no longer the bottleneck
  - more difficult to "pull plug"
- each application instance serves to:
  - store selected files
  - route queries from and to its neighboring peers
  - respond to queries if file stored locally
  - serve files

# Gnutella

- Gnutella history:
  - 3/14/00: release by AOL, almost immediately withdrawn
  - became open source
  - many iterations to fix poor initial design (poor design turned many people off)
- issues:
  - how much traffic does one query generate?
  - how many hosts can it support at once?
  - what is the latency associated with querying?
  - is there a bottleneck?

# Gnutella: limited scope query

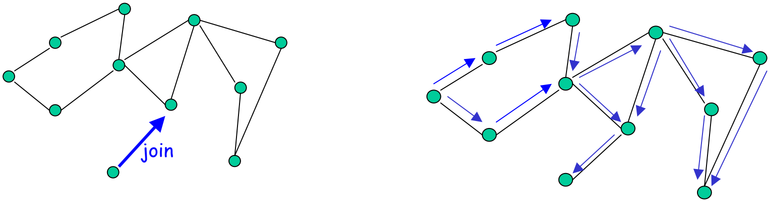
Searching by flooding:

- if you don't have the file you want, query 7 of your neighbors.
- if they don't have it, they contact 7 of their neighbors, for a maximum hop count of 10.
- reverse path forwarding for responses (not files)

Note: Play gnutella animation at:
http://www.limewire.com/index.jsp/p2p

# Gnutella overlay management

- New node uses bootstrap node to get IP addresses of existing Gnutella nodes
- New node establishes neighboring relations by sending join messages



# Gnutella in practice

- Gnutella traffic << KaZaA traffic
- 16-year-old daughter said "it stinks"
  - Couldn't find anything
  - Downloads wouldn't complete
- Fixes: do things KaZaA is doing: hierarchy, queue management, parallel download,...

# Quatrième chapitre

## Peer-to-Peer Systems


- Introduction
  - Overlays
  - Current P2P Applications
  - Worldwide Computer Vision
- Unstructured P2P File Sharing
  - Napster
  - Gnutella
  - KaZaA
- Structured P2P: DHT Approaches
  - DHT service, issues and seminal ideas
  - Chord
  - CAN
  - Pastry
- Applications
  - File sharing using DHT
  - Persistent file storage
  - Mobility Management
  - Content Distribution Networks
  - BitTorrent
  - Anonymous Activities
    - Storm Botnet
    - Tor System
- Quelques défis supplémentaires
  - Proximité réseau
  - Confiance entre participants
  - Dynamicité du système
- Conclusion

# KaZaA: The service

- more than 3 million up peers sharing over 3,000 terabytes of content
- more popular than Napster ever was
- more than 50% of Internet traffic ?
- MP3s & entire albums, videos, games
- optional parallel downloading of files
- automatically switches to new download server when current server becomes unavailable
- provides estimated download times

# KaZaA: The service (2)

- User can configure max number of simultaneous uploads and max number of simultaneous downloads
- queue management at server and client
  - Frequent uploaders can get priority in server queue
- Keyword search
  - User can configure "up to x" responses to keywords
- Responses to keyword queries come in waves; stops when x responses are found
- From user's perspective, service resembles Google, but provides links to MP3s and videos rather than Web pages

# KaZaA: Technology

## Software

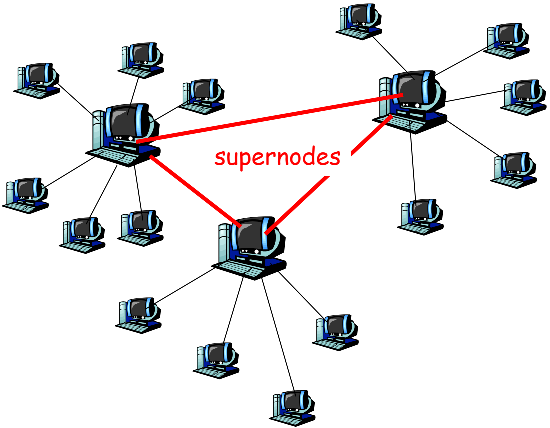
- Proprietary
- control data encrypted
- Everything in HTTP request and response messages

## Architecture

- hierarchical
- cross between Napster and Gnutella

# KaZaA: Architecture

- Each peer is either a supernode or is assigned to a supernode
  - 56 min avg connect
  - Each SN has about 100-150 children
  - Roughly 30,000 SNs
- Each supernode has TCP connections with 30-50 supernodes
  - 0.1% connectivity
  - 23 min avg connect



# KaZaA: Architecture (2)

- Nodes that have more connection bandwidth and are more available are designated as supernodes
- Each supernode acts as a mini-Napster hub, tracking the content and IP addresses of its descendants
- Does a KaZaA SN track only the content of its children, or does it also track the content under its neighboring SNs?
  - Testing indicates only children.

# KaZaA metadata

- When ON connects to SN, it uploads its metadata.
- For each file:
  - File name
  - File size
  - Content Hash
  - File descriptors: used for keyword matches during query
- Content Hash:
  - When peer A selects file at peer B, peer A sends ContentHash in HTTP request
  - If download for a specific file fails (partially completes), ContentHash is used to search for new copy of file.

# KaZaA: Overlay maintenance

- List of potential supernodes included within software download
- New peer goes through list until it finds operational supernode
  - Connects, obtains more up-to-date list, with 200 entries
  - Nodes in list are "close" to ON.
  - Node then pings 5 nodes on list and connects with the one
- If supernode goes down, node obtains updated list and chooses new supernode

# KaZaA Queries

- Node first sends query to supernode
  - Supernode responds with matches
  - If x matches found, done.
- Otherwise, supernode forwards query to subset of supernodes
  - If total of x matches found, done.
- Otherwise, query further forwarded
  - Probably by original supernode rather than recursively

# Parallel Downloading; Recovery

- If file is found in multiple nodes, user can select parallel downloading
  - Identical copies identified by ContentHash
- HTTP byte-range header used to request different portions of the file from different nodes
- Automatic recovery when server peer stops sending file
  - ContentHash

# KaZaA Corporate Structure

- Software developed by Estonians
- FastTrack originally incorporated in Amsterdam
- FastTrack also deploys KaZaA service
- FastTrack licenses software to Music City (Morpheus) and Grokster
- Later, FastTrack terminates license, leaves only KaZaA with killer service
- Summer 2001, Sharman networks, founded in Vanuatu (small island in Pacific), acquires FastTrack
  - Board of directors, investors: secret
- Employees spread around, hard to locate

# Lessons learned from KaZaA

KaZaA provides powerful file search and transfer service without server infrastructure

- Exploit heterogeneity
- Provide automatic recovery for interrupted downloads
- Powerful, intuitive user interface

Copyright infringement

- International cat-and-mouse game
- With distributed, serverless architecture, can the plug be pulled?
- Prosecute users?
- Launch DoS attack on supernodes?
- Pollute?

# Measurement studies by Gribble et al

- 2002 U. Wash campus study
- P2P: 43%; Web: 14%
- Kazaa objects fetched at most once per client
- Popularity distribution deviates substantially from Zipf distribution
  - Flat for 100 most popular objects
- Popularity of objects is short.

KaZaA users are patient

- Small objects (<10MB): 30% take more than hour to download
- Large objects (>100MB): 50% more than 1 day
- Kazaa is a batch-mode system, downloads done in background

Ross&Rubenstein, Infocom'02

# Pollution in P2P

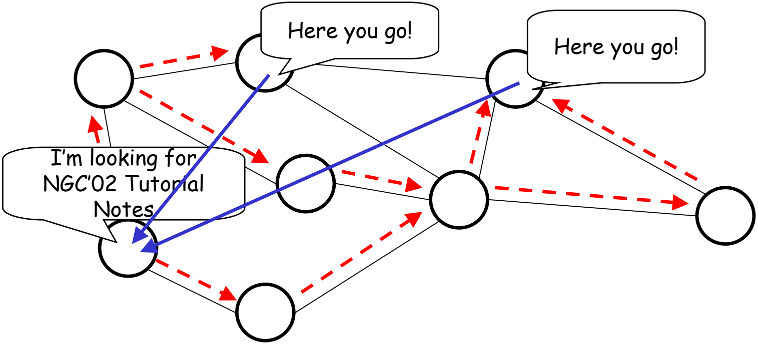
- Record labels hire "polluting companies" to put bogus versions of popular songs in file sharing systems
- Polluting company maintains hundreds of nodes with high bandwidth connections
- User A downloads polluted file
- User B may download polluted file before A removes it
- How extensive is pollution today?
- Anti-pollution mechanisms?

# Quatrième chapitre

## Peer-to-Peer Systems


- Introduction
  - Overlays
  - Current P2P Applications
  - Worldwide Computer Vision
- Unstructured P2P File Sharing
  - Napster
  - Gnutella
  - KaZaA
- Structured P2P: DHT Approaches
  - DHT service, issues and seminal ideas
  - Chord
  - CAN
  - Pastry
- Applications
  - File sharing using DHT
  - Persistent file storage
  - Mobility Management
  - Content Distribution Networks
  - BitTorrent
  - Anonymous Activities
    - Storm Botnet
    - Tor System
- Quelques défis supplémentaires
  - Proximité réseau
  - Confiance entre participants
  - Dynamicité du système
- Conclusion

# Challenge: Locating Content



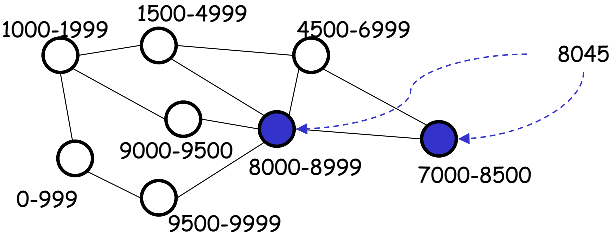

- Simplest strategy: expanding ring search
- If K of N nodes have copy, expected search cost *at least* N/K, i.e., O(N)
- Need many cached copies to keep search overhead small

# Directed Searches

- Idea:
  - assign particular nodes to hold particular content (or pointers to it, like an information booth)
  - when a node wants that content, go to the node that is supposed to have or know about it
- Challenges:
  - Distributed: want to distribute responsibilities among existing nodes in the overlay
  - Adaptive: nodes join and leave the P2P overlay
    - distribute knowledge responsibility to joining nodes
    - redistribute responsibility knowledge from leaving nodes

# DHT Step 1: The Hash

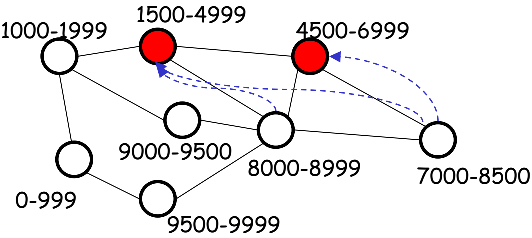
- Introduce a hash function to map the object being searched for to a unique identifier:
  - e.g., h("NGC'02 Tutorial Notes") → 8045
- Distribute the range of the hash function among all nodes in the network




- Each node must "know about" at least one copy of each object that hashes within its range (when one exists)

# "Knowing about objects"

- Two alternatives
  - Node can cache each (existing) object that hashes within its range
  - Pointer-based: level of indirection - node caches pointer to location(s) of object



# DHT Step 2: Routing

- For each object, node(s) whose range(s) cover that object must be reachable via a "short" path
- by the querier node (assumed can be chosen arbitrarily)
- by nodes that have copies of the object (when pointer-based approach is used)

- The different approaches (CAN,Chord,Pastry,Tapestry) differ fundamentally only in the routing approach
  - any "good" random hash function will suffice

# DHT Routing: Other Challenges

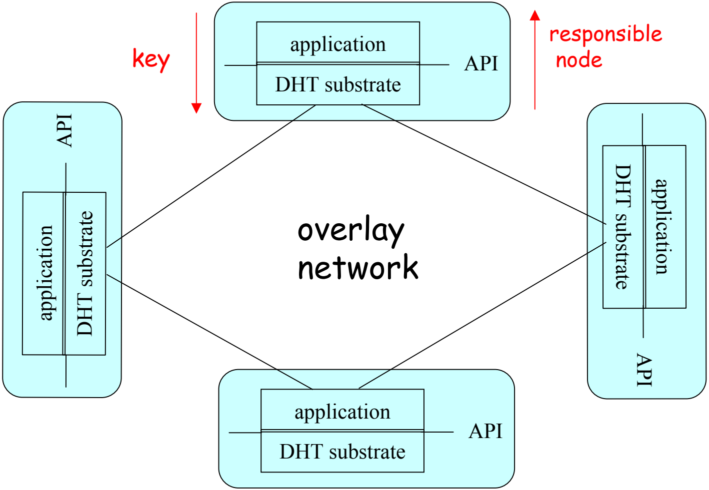
- # neighbors for each node should scale with growth in overlay participation (e.g., should not be O(N))
- DHT mechanism should be fully distributed (no centralized point that bottlenecks throughput or can act as single point of failure)
- DHT mechanism should gracefully handle nodes joining/leaving the overlay
  - need to repartition the range space over existing nodes
  - need to reorganize neighbor set
  - need bootstrap mechanism to connect new nodes into the existing DHT infrastructure

# DHT API

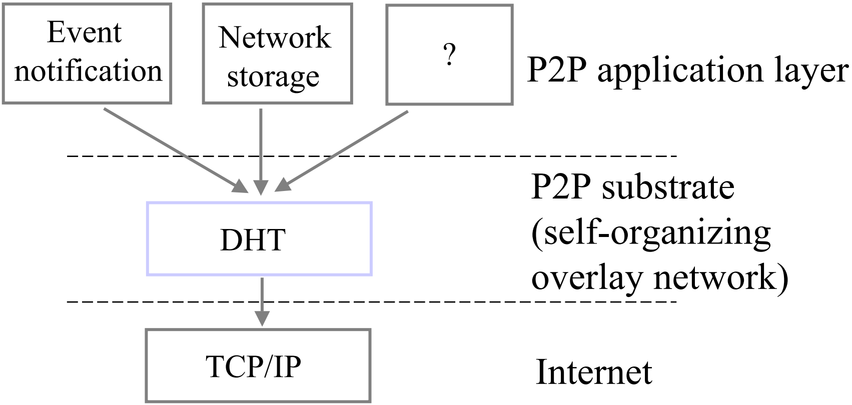
- each data item (e.g., file or metadata containing pointers) has a key in some ID space
- In each node, DHT software provides API:
  - Application gives API key k
  - API returns IP address of node that is responsible for k
- API is implemented with an underlying DHT overlay and distributed algorithms

# DHT API

each data item (e.g., file or metadata pointing to file copies) has a key



# DHT Layered Architecture



# CARP

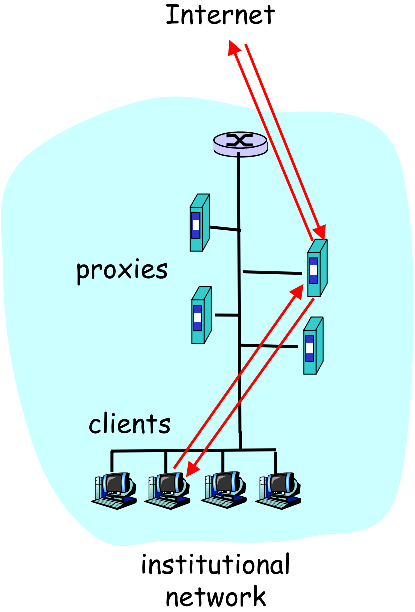
## DHT for cache clusters

- Each proxy has unique name

**key = URL = u**

- calc $h(proxy_n, u)$ for all proxies
- assign u to proxy with highest $h(proxy_n, u)$

> if proxy added or removed, u is likely still in correct proxy



# CARP (2)

- circa 1997
  - Internet draft: Valloppillil and Ross
- Implemented in Microsoft & Netscape products
- Browsers obtain script for hashing from proxy automatic configuration file (loads automatically)

Not good for P2P:

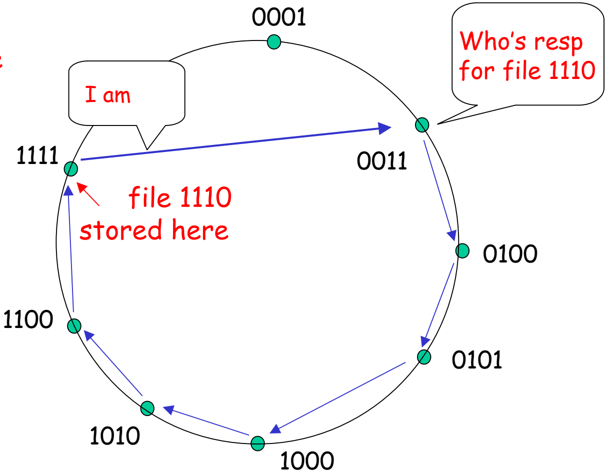
- Each node needs to know name of all other up nodes
- i.e., need to know O(N) neighbors
- But only O(1) hops in lookup

# Consistent hashing (1)

- Overlay network is a circle
- Each node has randomly chosen id
  - Keys in same id space
- Node's successor in circle is node with next largest id
  - Each node knows IP address of its successor
- Key is stored in closest successor

# Consistent hashing (2)

O(N) messages on avg to resolve query




Note: no locality among neighbors

100

# Consistent hashing (3)

## Node departures

- Each node must track $s \geq 2$ successors
- If your successor leaves, take next one
- Ask your new successor for list of its successors; update your s successors

## Node joins

- You're new, node id k
- ask any node n to find the node n' that is the successor for id k
- Get successor list from n'
- Tell your predecessors to update their successor lists
- Thus, each node must track its predecessor

# Consistent hashing (4)

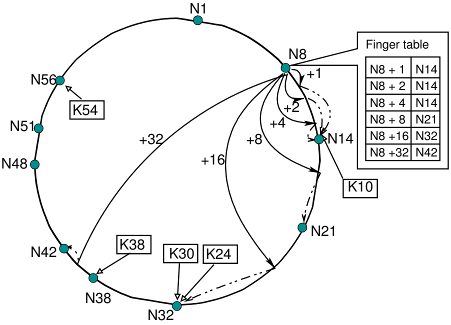
- Overlay is actually a circle with small chords for tracking predecessor and k successors
- # of neighbors = s+1: O(1)
  - The ids of your neighbors along with their IP addresses is your "routing table"
- average # of messages to find key is O(N)

Can we do better?

# Chord, MIT

## Principe de base

- Espace d'adressage circulaire; données sur noeud suivant; voisins: $n + 2^i, \forall i$

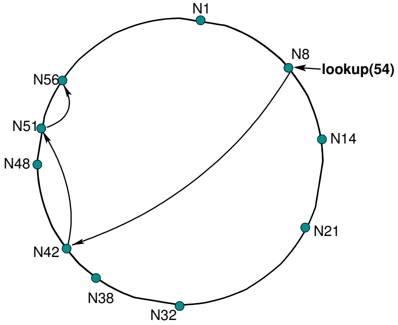



Stoica et Al, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, ACM SIGCOMM 2001.

# Chord, MIT

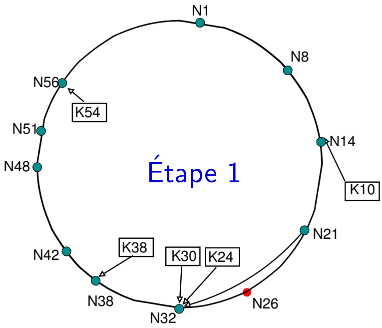
## Principe de base

- Espace d'adressage circulaire; données sur noeud suivant; voisins: $n + 2^i, \forall i$
- Recherche en $O(\log(n))$



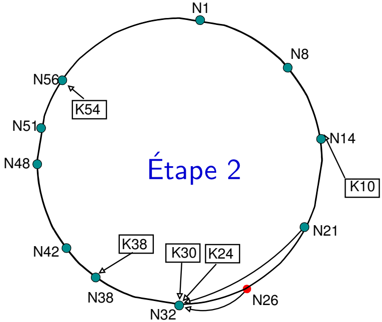

Stoica et Al, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, ACM SIGCOMM 2001.

# Insertion d'un nœud dans Chord



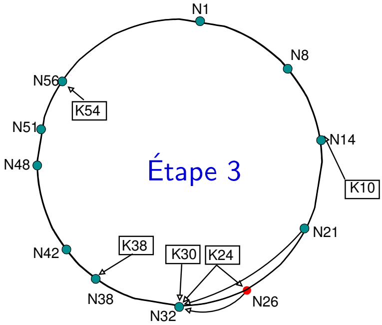

- Service maintenu durant insertion
- Insertions concurrentes possibles

# Insertion d'un nœud dans Chord



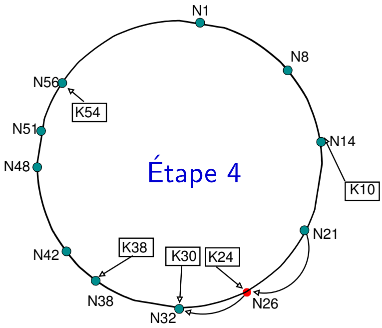

- Service maintenu durant insertion
- Insertions concurrentes possibles

# Insertion d'un nœud dans Chord




- Service maintenu durant insertion
- Insertions concurrentes possibles

# Insertion d'un nœud dans Chord

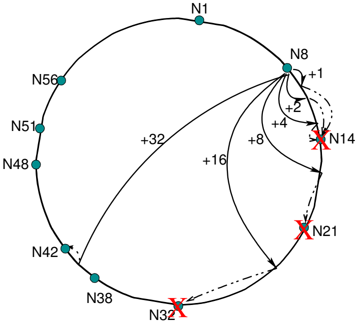



- Service maintenu durant insertion
- Insertions concurrentes possibles

# Autres propriétés de Chord

## Retrait d'un nœud Chord

- Table = liste de $O(\log(N))$ successeurs
  ⇒ Probablement correct même si hécatombe de nœuds (proba mort = 1/2 )



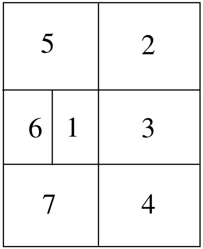

## Autres propriétés (démontrées)

- Résistance probable aux morts simultanées
- Possibilité d'ajouts simultanés
- Résistance à la mort de noeud lors de l'ajout d'autres
- Équilibrage de charge entre les noeuds

# Content-Addressable Network (CAN), Berkley

## Principe de base

- Idée: Chaque nœud a un morceau de l'espace d'adressage (d dimensions, torique)

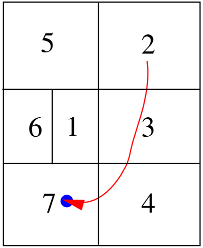



RFHKS, *A scalable content-addressable network*, ATAPCC'01

# Content-Addressable Network (CAN), Berkley

## Principe de base

- Idée: Chaque nœud a un morceau de l'espace d'adressage (d dimensions, torique)
- Routage: proche en proche ($\Rightarrow 0(dn^{1/d})$ sauts; $|\text{table}| = O(d)$)

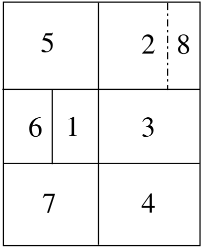



RFHKS, *A scalable content-addressable network*, ATAPCC'01

# Content-Addressable Network (CAN), Berkley

## Principe de base

- Idée: Chaque nœud a un morceau de l'espace d'adressage (d dimensions, torique)
- Routage: proche en proche ($\Rightarrow 0(dn^{1/d})$ sauts; $|\text{table}| = O(d)$)
- Ajout d'un nœud: il s'approprie un morceau

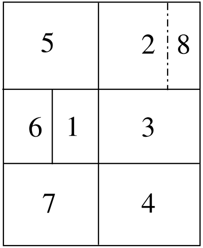



RFHKS, *A scalable content-addressable network*, ATAPCC'01

# Content-Addressable Network (CAN), Berkley

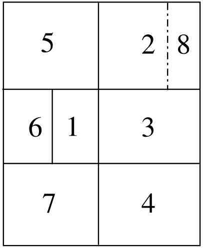
## Principe de base

- Idée: Chaque nœud a un morceau de l'espace d'adressage (d dimensions, torique)
- Routage: proche en proche ($\Rightarrow 0(dn^{1/d})$ sauts; $|\text{table}|=O(d)$)
- Ajout d'un nœud: il s'approprie un morceau
- Mort d'un nœud: un voisin récupère sa zone




RFHKS, *A scalable content-addressable network*, ATAPCC'01

# Content-Addressable Network (CAN), Berkley




## Principe de base

- Idée: Chaque nœud a un morceau de l'espace d'adressage (d dimensions, torique)
- Routage: proche en proche ($\Rightarrow 0(dn^{1/d})$ sauts; $|\text{table}|=O(d)$)
- Ajout d'un nœud: il s'approprie un morceau
- Mort d'un nœud: un voisin récupère sa zone

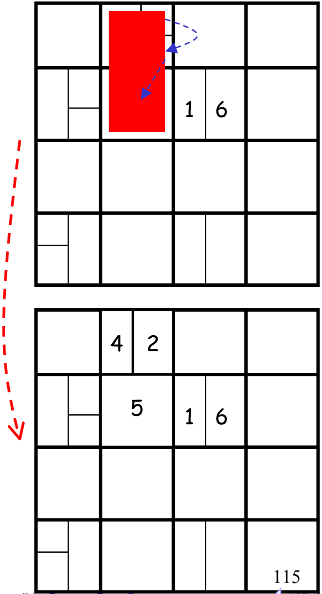
## Raffinements

- Réalités: Plusieurs espaces d'adressage
  $\leadsto$ meilleure résistance (réplication) ; latence moins bonne
- Meilleur routage: Choix de voisin selon distance réseau (pour diagonales)
- Zones recouvrantes:
  $\leadsto$ moins de sauts, latence par saut moindre; meilleur résistance
- Place dans espace d'addressage en fonction localisation physique:
  $\leadsto$ meilleure localité, distribution moins bonne

RFHKS, *A scalable content-addressable network*, ATAPCC'01

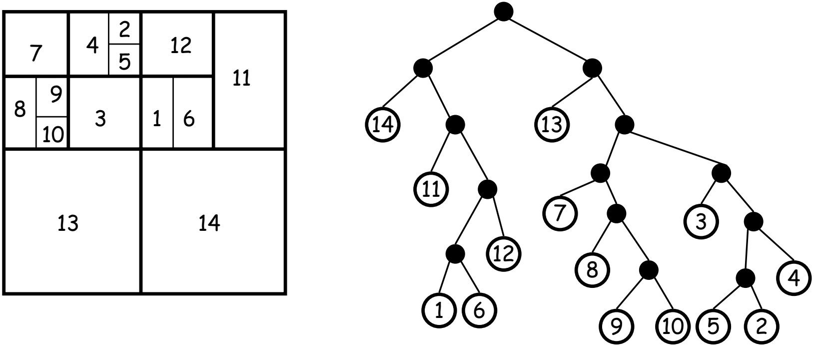
# CAN node removal

- Underlying cube structure should remain intact
  - i.e., if the spaces covered by s & t were not formed by splitting a cube, then they should not be merged together
- Sometimes, can simply collapse removed node's portion to form bigger rectangle
  - e.g., if 6 leaves, its portion goes back to 1
- Other times, requires juxtaposition of nodes' areas of coverage
  - e.g., if 3 leaves, should merge back into square formed by 2,4,5
  - cannot simply collapse 3's space into 4 and/or 5
  - one solution: 5's old space collapses into 2's space, 5 takes over 3's space



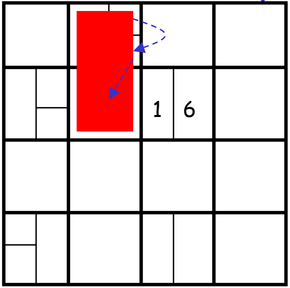

115

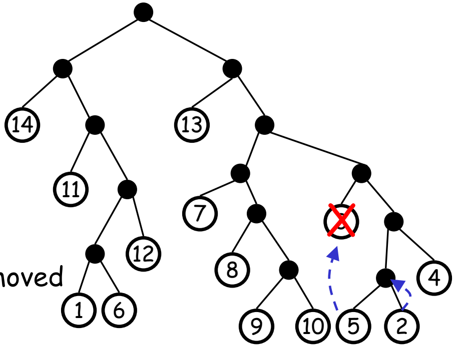
# CAN (recovery from) removal process




- View partitioning as a binary tree of
  - leaves represent regions covered by overlay nodes (labeled by node that covers the region)
  - intermediate nodes represent "split" regions that could be "reformed", i.e., a leaf can appear at that position
  - siblings are regions that can be merged together (forming the region that is covered by their parent)

# CAN (recovery from) removal process




- Repair algorithm when leaf s is removed
  - Find a leaf node t that is either
    - s's sibling
    - descendant of s's sibling where t's sibling is also a leaf node
  - t takes over s's region (moves to s's position on the tree)
  - t's sibling takes over t's previous region
- Distributed process in CAN to find appropriate t w/ sibling:
  - current (inappropriate) t sends msg into area that would be covered by a sibling
  - if sibling (same size region) is there, then done. Else receiving node becomes t & repeat

Ross&Rubenstein, Infocom'02

# Quatrième chapitre

## Peer-to-Peer Systems

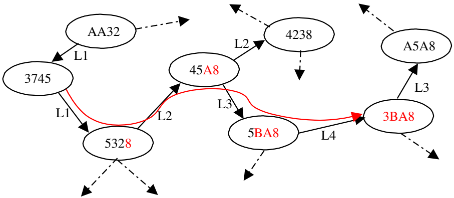

- Introduction
  - Overlays
  - Current P2P Applications
  - Worldwide Computer Vision
- Unstructured P2P File Sharing
  - Napster
  - Gnutella
  - KaZaA
- Structured P2P: DHT Approaches
  - DHT service, issues and seminal ideas
  - Chord
  - CAN
  - Pastry
- Applications
  - File sharing using DHT
  - Persistent file storage
  - Mobility Management
  - Content Distribution Networks
  - BitTorrent
  - Anonymous Activities
    - Storm Botnet
    - Tor System
- Quelques défis supplémentaires
  - Proximité réseau
  - Confiance entre participants
  - Dynamicité du système
- Conclusion

# Algorithme de Plaxton

Structure de données distribuée servant de table de routage.

## Idée de base

- Chaque noeud a une clé d'identification unique (répartition uniforme)
- Routage de proche en proche dans l'espace des clés par suffixe commun
  Exemple : (3745→3BA8) = (???8→??A8→?BA8→3BA8)
- Table: Ligne $i \leadsto$ préfix commun taille $i$; Colonne $j$ : caractère '$j$' ensuite.




| | | | |
|---|---|---|---|
| – | 1201 | 3202 | 2123 |
| 3200 | – | 3220 | 2130 |
| 3010 | 3110 | 2210 | – |
| 0310 | 1310 | – | 3310 |

Table du nœud 2310 en base 4.

☺ Petite table ($b(\log_b(N))$), peu de saut ($\lceil \log_b(N) \rceil$)

☹ Pas d'algo pour construire la table

Plaxton et Al, *Accessing nearby copies of replicated objects in a distributed environment*, SPAA'97.

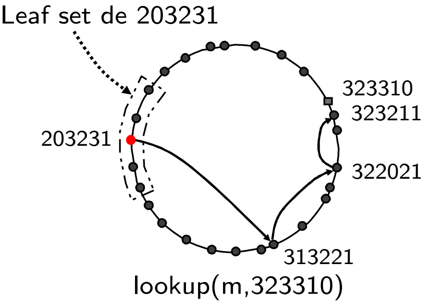
# Systèmes P2P basés sur Plaxton

## Tapestry et Pastry

- En 2001, deux algos sont proposés pour créer les tables et les tenir à jour
- Tapestry: Thésard de U. Berkley; Pastry: U. Rice et Microsoft Research
- Idée de base: mélange de Chord et Plaxton
- Différence:
  - Optimisations diverses et variées, principalement
  - L'histoire retiendra surtout Pastry

Table de 203231

| | | | |
|---|---|---|---|
| 0* | 1* | 2* | 3* |
| 20* | 21* | 22* | 23* |
| 200* | 201* | 202* | 203* |
| 2030* | 2031* | 2032* | 2033* |



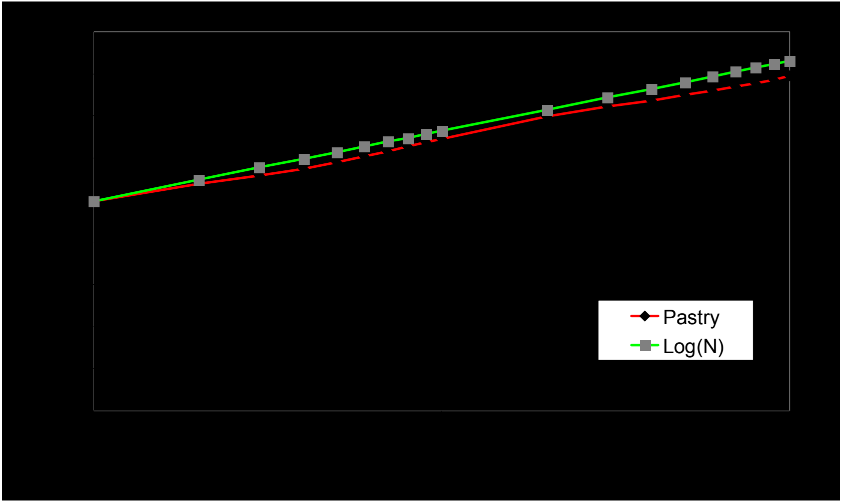
# Pastry: Experimental results

## Prototype

- implemented in Java
  - deployed testbed (currently ~25 sites worldwide)

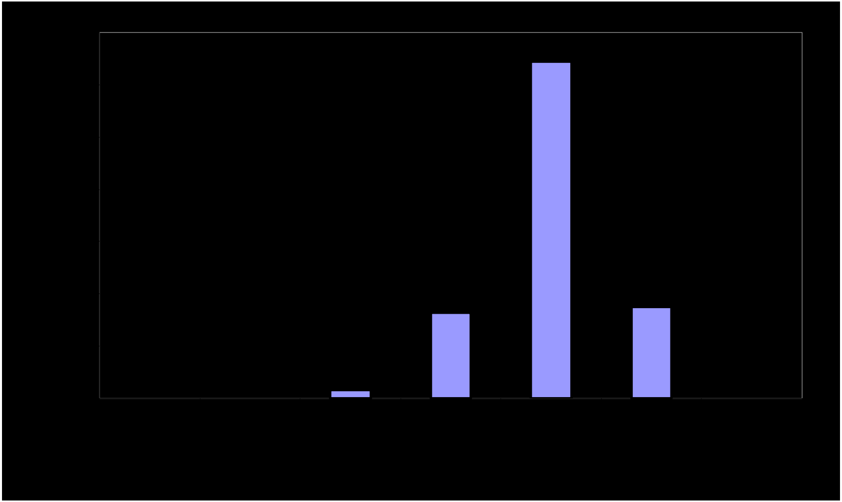
## Simulations for large networks

# Pastry: Average # of hops



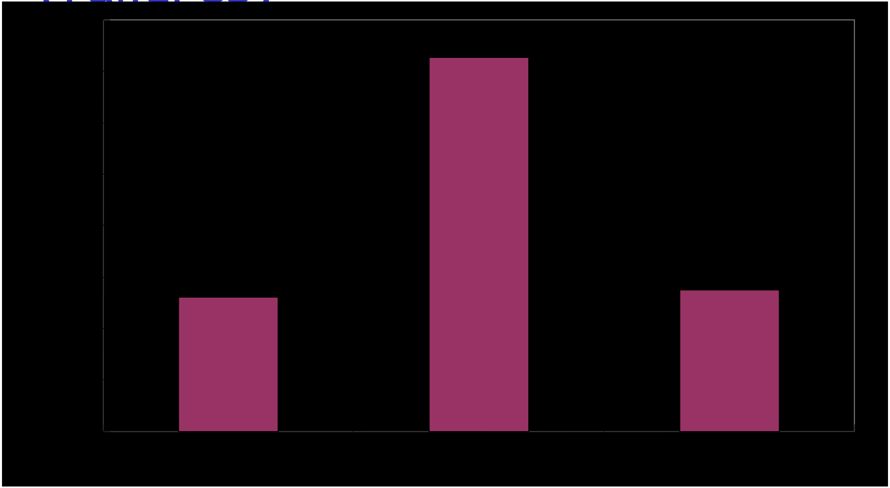

L=16, 100k random queries

# Pastry: # of hops (100k nodes)



L=16, 100k random queries

# Pastry: # routing hops (failures)



L=16, 100k random queries, 5k nodes, 500 failures

# Pastry, Tapestry et les autres

## Ajout de nœuds

- Nouveau venu choisi un ID aléatoirement
- Envoi d'un message à cet ID
- Le nœud le plus proche de cet ID répond, avec ses tables de routage

## Départ de nœuds

- Messages fréquents pour vérifier la validité de la table
- Échanges d'éléments de tables entre voisins vivants

## Autres overlay P2P proposés dans la littérature

- Kademlia: Un peu plaxton, mais routage par XOR binaire au lieu de préfixe
- Bamboo: accent mis sur la tolérance au *churn*
- SkipNet: accent mis sur la localité réseau
- Kelips: accent mis sur efficacité des recherches
- Accordeon: balance entre temps de recherche et maintenance des tables
- openDHT: tentative d'unification

# Comparaisons entre systèmes pair-à-pair

## Comparaison entre P2P non-structurés et DHT

- DHT préférables pour: recherche exacte d'éléments rares
- Non-structurés préférables pour: recherche approchée, *churn* extrême

Castro, Costa, Rowstron, *Debunking some myths about structured and unstructured overlays*, NSDI'05.

## Comparaison entre DHT

| | CAN<br>Dim $d$ | Chord | Pastry<br>base $b$ | Tapestry<br>base $b$ |
|---|---|---|---|---|
| Taille table | $O(d)$ | $\log_2(N)$ | $b\log_b(N) + O(b)$ | $b\log_b(N)$ |
| # saut | $O(d \times N^{1/d})$ | $\log_2(N)$ | $\log_b(N)$ | $\log_b(N)$ |
| # msg ajout | $O(d \times N^{1/d})$ | $O(\log_2^2(N))$ | $O(\log_b(N))$ | $O(\log_b(N)^2)$ |
| Retrait | ?? | $O\left(\log^2 N\right)$ | ?? | ?? |
| Localité<br>(mobilité) | non | non | oui<br>non | oui<br>oui |
| Sécurité | non | non | à l'étude | non |

# Quatrième chapitre

## Peer-to-Peer Systems


- Introduction
  - Overlays
  - Current P2P Applications
  - Worldwide Computer Vision
- Unstructured P2P File Sharing
  - Napster
  - Gnutella
  - KaZaA
- Structured P2P: DHT Approaches
  - DHT service, issues and seminal ideas
  - Chord
  - CAN
  - Pastry
- Applications
  - File sharing using DHT
  - Persistent file storage
  - Mobility Management
  - Content Distribution Networks
  - BitTorrent
  - Anonymous Activities
    - Storm Botnet
    - Tor System
- Quelques défis supplémentaires
  - Proximité réseau
  - Confiance entre participants
  - Dynamicité du système
- Conclusion

# File sharing using DHT

**Advantages**

- Always find file
- Quickly find file
- Potentially better management of resources

**Challenges**

- File replication for availability
- File replication for load balancing
- Keyword searches

There is at least one file sharing system using DHTs: Overnet, using Kademlia

# File sharing: what's under key?

## Data item is file itself

- Replicas needed for availability
- How to load balance?

## Data item under key is list of pointers to file

- Must replicate pointer file
- Must maintain pointer files: consistency

# File sharing: keywords

- Recall that unstructured file sharing provides keyword search
  - Each stored file has associated metadata, matched with queries
- DHT: Suppose key = h(artist, song)
  - If you know artist/song exactly, DHT can find node responsible for key
  - Have to get spelling/syntax right!
- Suppose you only know song title, or only artist name?

# Keywords: how might it be done?

Each file has XML descriptor

```
<song>
<artist>David
  Bowie</artist>
<title>Changes</title>
<album>Hunky Dory</album>
<size>3156354</size>
</song>
```

Key is hash of descriptor: k = h(d)

Store file at node responsible for k

Plausible queries

$q_1$ = /song[artist/David Bowie][title/Changes] [album/Hunky Dory] [size/3156354]

$q_2$ = /song[artist/David Bowie][title/Changes]

$q_3$ = /song/artist/David Bowie

$q_4$ = /song/title/Changes

Create keys for each plausible query: $k_n = h(q_n)$

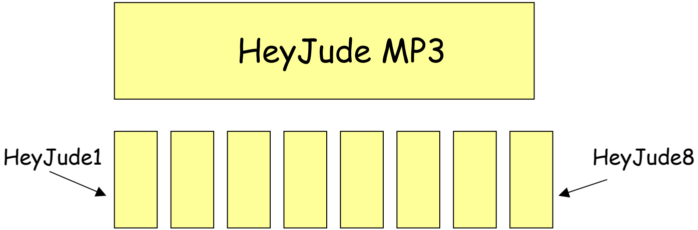
For each query key $k_n$, store descriptors d at node responsible for $k_n$

157

# Keywords: continued

- Suppose you input `q4 = /song/title/Changes`
- Locally obtain key for $q_4$, submit key to DHT
- DHT returns node n responsible for $q_4$
- Obtain from n the descriptors of all songs called Changes
- You choose your song with descriptor d, locally obtain key for d, submit key to DHT
- DHT returns node n' responsible for desired song

# Blocks




Each block is assigned to a different node

# Blocks (2)

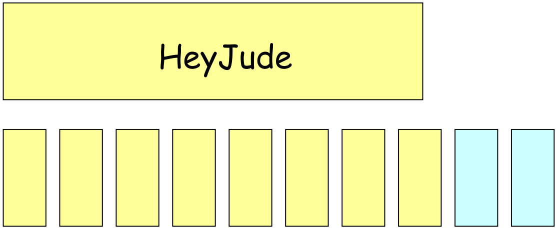
**Benefits**

- Parallel downloading
  - Without wasting global storage
- Load balancing
  - Transfer load for popular files distributed over multiple nodes

**Drawbacks**

- Must locate all blocks
- Must reassemble blocks
- More TCP connections
- If one block is unavailable, file is unavailable

# Erasures (1)




- Reconstruct file with any m of r pieces
- Increases storage overhead by factor r/m

# Erasures (2)

## Benefits

- Parallel downloading
  - Can stop when you get the first m pieces
- Load balancing
- More efficient copies of blocks
  - Improved availability for same amount of global storage

## Drawbacks

- Must reassemble blocks
- More TCP connections

# Persistent file storage

- PAST layered on Pastry
- CFS layered on Chord

## P2P Filesystems

- Oceanstore
- FarSite

# PAST: persistence file storage

## Goals

- Strong persistence
- High availability
- Scalability
  - nodes, files, queries, users
- Efficient use of pooled resources

## Benefits

- Provides powerful backup and archiving service
- Obviates need for explicit mirroring

191

# Mobility management

- Alice wants to contact bob smith
  - Instant messaging
  - IP telephony
- But what is bob's current IP address?
  - DHCP
  - Switching devices
  - Moving to new domains

# Mobility Management (2)

- Bob has a unique identifier:
  - bob.smith@foo.com
  - k =h(bob.smith@foo.com)
- Closest DHT nodes are responsible for k
- Bob periodically updates those nodes with his current IP address
- When Alice wants Bob's IP address, she sends query with k =h(bob.smith@foo.com)

# Mobility management (3)

- Obviates need for SIP servers/registrars
- Can apply the same idea to DNS
- Can apply the same idea to any directory service
  - e.g., P2P search engines

# Quelques applications P2P

## Rendez-vous (système d'annuaire)

- Motivation: Utilisateurs mobiles (changements d'IP)
- Application: Chat, Téléphonie, (voire DNS)
- Principe: Insertion régulière IP dans le système

## Stockage de fichier

- (fonction originelle avec Napster)
- Avantages: grande capacité disque, gros lien, réplication, ...
- Exemple: Usenet
  - Le système, lancé en 1981, a une croissance exponentielle
  - Seuls 50 sites ont tout car stockage + bande passante = 30000$

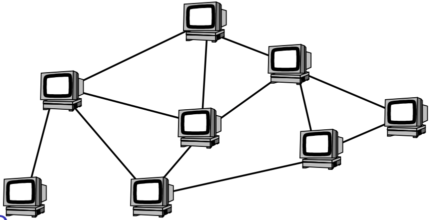
  $\Rightarrow$ bon candidat aux DHT

# Content Distribution Networks

## Applications

- Multicast (multimédia)
- Systèmes de notification d'événements

## Principe: Construction de l'arbre de diffusion d'après l'overlay



## Défis du routage P2P

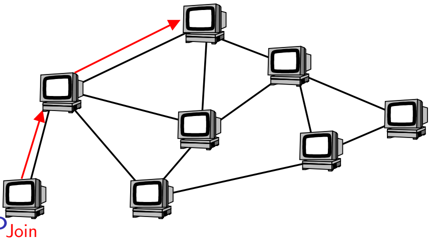
- Dynamisme de l'arbre
- Répartition de charge
- Proximité réseau

# Content Distribution Networks

## Applications

- Multicast (multimédia)
- Systèmes de notification d'événements

**Principe:** Construction de l'arbre de diffusion d'après l'overlay




## Défis du routage P2P

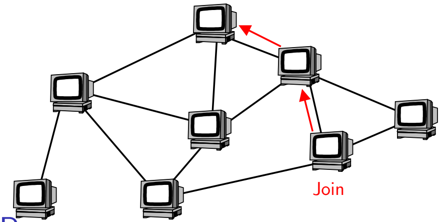
- Dynamisme de l'arbre
- Répartition de charge
- Proximité réseau

# Content Distribution Networks

## Applications

- Multicast (multimédia)
- Systèmes de notification d'événements

## Principe: Construction de l'arbre de diffusion d'après l'overlay




## Défis du routage P2P

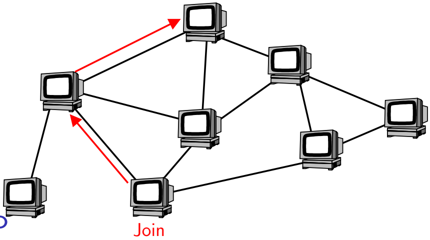
- Dynamisme de l'arbre
- Répartition de charge
- Proximité réseau

# Content Distribution Networks

## Applications

- Multicast (multimédia)
- Systèmes de notification d'événements

**Principe:** Construction de l'arbre de diffusion d'après l'overlay




## Défis du routage P2P

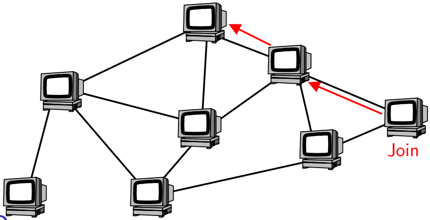
- Dynamisme de l'arbre
- Répartition de charge
- Proximité réseau

# Content Distribution Networks

## Applications

- Multicast (multimédia)
- Systèmes de notification d'événements

**Principe:** Construction de l'arbre de diffusion d'après l'overlay




## Défis du routage P2P

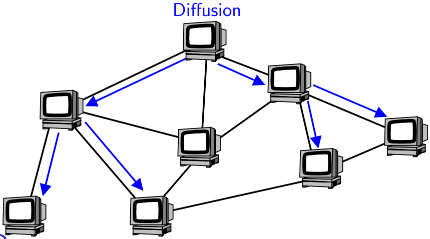
- Dynamisme de l'arbre
- Répartition de charge
- Proximité réseau

# Content Distribution Networks

## Applications

- Multicast (multimédia)
- Systèmes de notification d'événements

## Principe: Construction de l'arbre de diffusion d'après l'overlay




## Défis du routage P2P

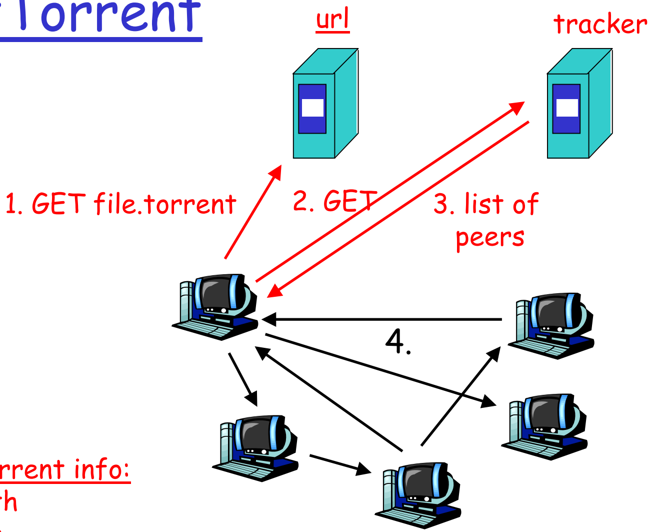
- Dynamisme de l'arbre
- Répartition de charge
- Proximité réseau

# Quatrième chapitre

## Peer-to-Peer Systems


- Introduction
  - Overlays
  - Current P2P Applications
  - Worldwide Computer Vision
- Unstructured P2P File Sharing
  - Napster
  - Gnutella
  - KaZaA
- Structured P2P: DHT Approaches
  - DHT service, issues and seminal ideas
  - Chord
  - CAN
  - Pastry
- Applications
  - File sharing using DHT
  - Persistent file storage
  - Mobility Management
  - Content Distribution Networks
  - BitTorrent
  - Anonymous Activities
    - Storm Botnet
    - Tor System
- Quelques défis supplémentaires
  - Proximité réseau
  - Confiance entre participants
  - Dynamicité du système
- Conclusion

# BitTorrent




file.torrent info:

- length
- name
- hash
- url of tracker

# BitTorrent: Pieces

- File is broken into pieces
  - Typically piece is 256 KBytes
  - Upload pieces while downloading pieces
- Piece selection
  - Select rarest piece
  - Except at beginning, select random pieces
- Tit-for-tat
  - Bit-torrent uploads to at most four peers
  - Among the uploaders, upload to the four that are downloading to you at the highest rates
  - A little randomness too, for probing

# NATs

- nemesis for P2P
- Peer behind NAT can't be a TCP server
- Partial solution: reverse call
  - Suppose A wants to download from B, B behind NAT
  - Suppose A and B have each maintain TCP connection to server C (not behind NAT)
  - A can then ask B, through C, to set up a TCP connection from B to A.
  - A can then send query over this TCP connection, and B can return the file
- What if both A and B are behind NATs?

Quatrième chapitre

# Peer-to-Peer Systems

- Introduction
  - Overlays
  - Current P2P Applications
  - Worldwide Computer Vision
- Unstructured P2P File Sharing
  - Napster
  - Gnutella
  - KaZaA
- Structured P2P: DHT Approaches
  - DHT service, issues and seminal ideas
  - Chord
  - CAN
  - Pastry
- Applications
  - File sharing using DHT
  - Persistent file storage
  - Mobility Management
  - Content Distribution Networks
  - BitTorrent
  - Anonymous Activities
    - Storm Botnet
    - Tor System
- Quelques défis supplémentaires
  - Proximité réseau
  - Confiance entre participants
  - Dynamicité du système
- Conclusion

# Anonymous Activities

## Suppose clients want to perform anonymous communication

- Requestor wishes to keep its identity secret
- Deliverer wishes to also keep identity secret

## Whitehat Motivations

- Protect privacy
- Fight censorship

## Blackhat Motivations

- Avoid the detection of criminal activity
- Hide crucial infrastructure: "mothership" servers, monitoring and control servers, etc

# Example systems

- BotNets
  - networks of compromised PCs
  - initially IRC-based; now increasingly P2P
  - main servers and operator wants to stay anonym
- Anonym networks
  - Dedicated (closed or open) networks
  - some variation of “mixing” communication so that participants cannot be traced back
  - remailer networks, low latency networks, friends-networks

3

# Storm Botnet

- appeared in 2007 January
- primarily for sending spam
- advanced P2P technology
- size estimated between 500,000 and 50 million
- aggressive measures for protection
  - regular download of updates to prevent reverse engineering
  - DDoS attack against external hosts that attempt to probe its operations

5

# Storm Botnet Technology

- uses overnet protocol, based on the kademlia DHT
  - key space is 128 bit binary (usual DHT design)
  - routing is based on XOR distance
    - **eg d(001,110)=001⊕110=111**
  - for 0<=i<=128 there is a "bucket" of k(=20) addresses that are at distance from $[2^i, 2^{i+1})$
  - these buckets are kept fresh from observing traffic (preferring oldest, but live nodes), and proactive lookup if needed
  - lookup uses the 3 closest nodes in parallel

# Storm Botnet Technology

- Storm bots periodically search for a given key
  - key is generated using the current date and a random number from [0,31]
  - value of that key contains an encrypted URL
  - which in turn contains new binary updates and other files to download
- for some reason
  - if this lookup fails, bots rejoin the network with new ID and repeat the search
- file sharing networks such as eDonkey can be used to store these keys! (same protocol)

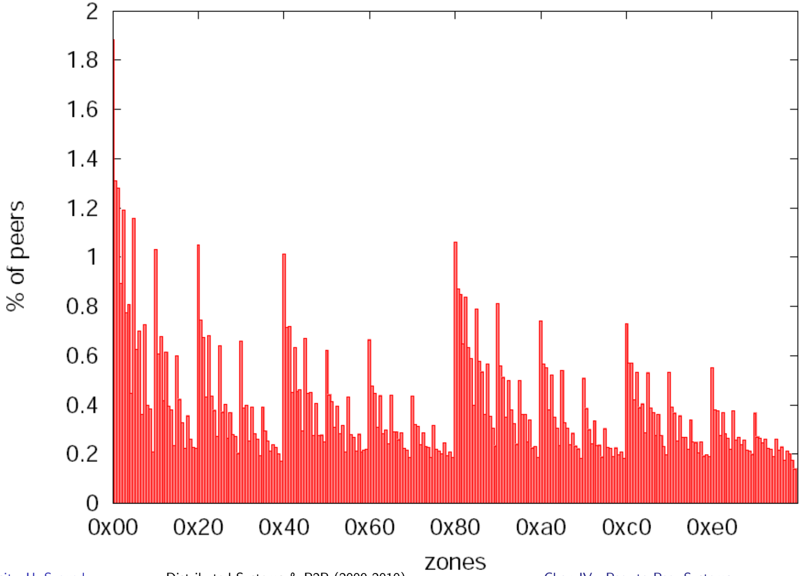
7

# Measurements

- Crawler: kademlia client that
  - performs queries for random keys
  - records node ID, IP and port that is returned
- seed list
  - 400 hard-wired IP-s in the Storm bot binary
  - storm bot run in a honeypot for 5 hours: 4000 peers
- full crawls (entire 128 bit space)
- zone crawl (space with a fixed prefix)
- estimated size: around 500,000

8

# Uneven distribution of storm bot IDs




9

# Explanation of uneven distribution: war against the Storm?

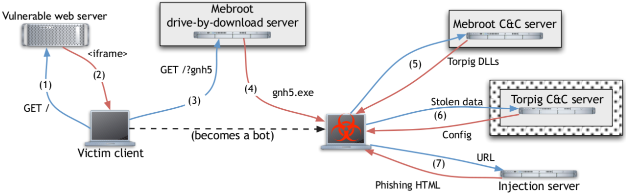
- Around 1% of returned IP addresses bogous
- But 45% of unique Ids have one of these addresses
- These IDs are responsible for the non-uniformity of the ID distribution as well
- possible explanation
  - index poisoning
  - we are witnessing efforts to fight the Storm Botnet

10

# Whitehats vs Blackhats

## May 2009: Torpig hijack

- Classical BotNet, specialized in data stealing (through pishing)
- Researchers managed to get the control of the Torpig botnet for 10 days
- The botnet get commands from C&C servers, changing domain name regularly
- Researchers registered future names before criminals
- New binary uploaded after 10 days; 70Gb of personal data retrieved; Measurements: ≈180k nodes




## 27 december 2009: Mega-D shut down

- Botnet responsible for about 10% of whole spam for months
- Got the ISP hosting them to shut 11 of 13 C&C servers
- Hijacked DNS registery of the other ones

# Tor

- Can provide anonymity for both clients and servers (the latter using the “.onion” domain)
- So called “onion” routing
- Originally funded by US Naval Research Lab
  - To provide protection for negotiators, agents, etc
  - but if only the Navy uses it, everyone knows it's the Navy: so it went public...
- Later taken over by Electronic Frontier Foundation (EFF)
- Currently a few thousand nodes

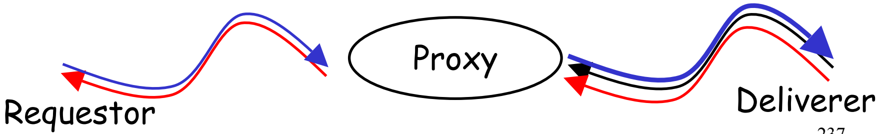
16

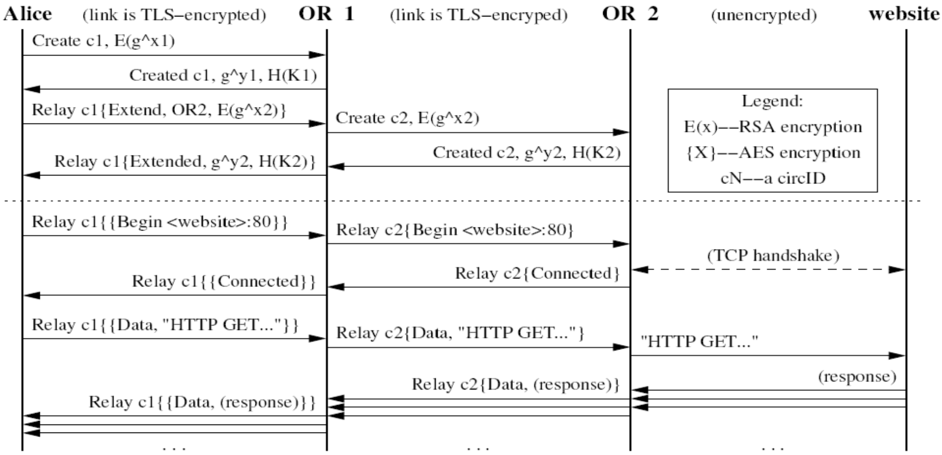
# Onion Routing

- A Node N that wishes to send a message to a node M selects a path (N, $V_1$, $V_2$, …, $V_k$, M)
  - Each node forwards message received from previous node
  - N can encrypt both the message and the next hop information recursively using public keys: a node only knows who sent it the message and who it should send to
- N's identity as originator is not revealed

# Anonymnity on both sides

- A requestor of an object receives the object from the deliverer without these two entities exhanging identities
- Utilizes a proxy
  - Using onion routing, deliverer reports to proxy (via onion routing) the info it can deliver, but does not reveal its identity
  - Nodes along this onion-routed path, A, memorize their previous hop
  - Requestor places request to proxy via onion-routing, each node on this path, B, memorize previous hop
  - Proxy→Deliverer follows "memorized" path A
  - Deliverer sends article back to proxy via onion routing
  - Proxy→Requestor via "memorized" path B






- the client never uses its public key
- onion: layers of AES encryption (a symmetric key encryption) based on secret key negotiated with Diffie Hellman during the circuit building

20

# Problems: last step

- link between Tor exit and service is unencrypted
  - people hosting Tor exits can see all traffic (but not the origin)
- Dan Egerstad: collected high value corporate and government email addresses
  - arrested in October 2007!
  - Egerstad says
    - **traffic to these email accounts probably originated from spies and not original owners**
    - **web traffic is mostly porn...**

21

# Other problems

- DNS leak
  - resolving DNS requests is still direct
  - latest version includes DNS resolver (understands .onion domain as well)
- traffic analysis
  - techniques exist that capture correlated traffic without global knowledge
- misuse
  - bittorrent clients often support Tor: huge traffic
  - criminals wanting to avoid detection

# Les systèmes P2P aujourd'hui

## Infrastructure choisie

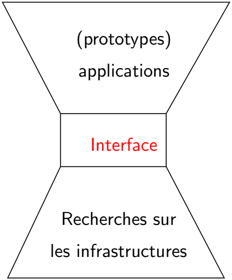
- Décentralisée, tirant profit des clients puissants

## Interface choisie

- Put(key, data)/Get(key): hachage classique
- lookup(key): recherche responsable de clé

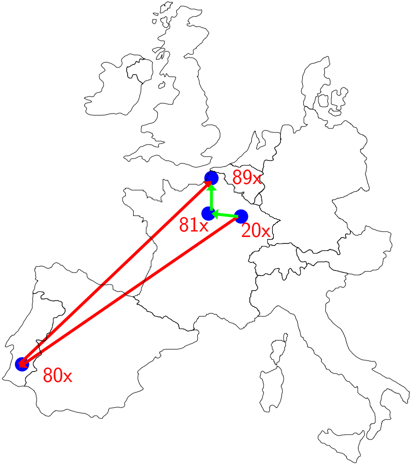
## La recherche en P2P

- Améliorations des infrastructures P2P
  - Exploration de nouvelles fonctions (cf. plus haut)
  - Conditions extrêmes (taille, churn)
- Standardisation de l'interface (⇒ openDHT)
- Prototypage et développement d'applications




Sablier P2P

# Défi: efficacité du routage vis-à-vis du réseau




## Défi:

- Adéquation overlay et réseau physique
- Réduire nombre sauts **et** latence

## Solution: Proximity Neighbor Selection

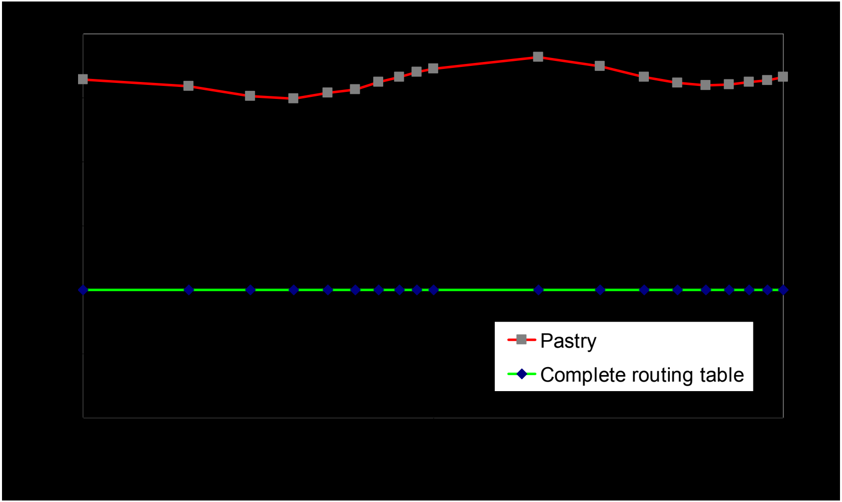
- Pour chaque case de la table, il y a plusieurs candidats
- Choisir le noœud possible le plus proche selon une métrique réseau (RTT)

## Résultat:

- Les routes dans l'overlay convergent physiquement
- Surcoût latence par rapport à IP: rapport constant ($< 3$)

Castro, Druschel, Hu, Rowstron *Proximity neighbor selection in tree-based structured peer-to-peer overlays*, Technical Report MSR-TR-2003-52, Microsoft Research, 2003.

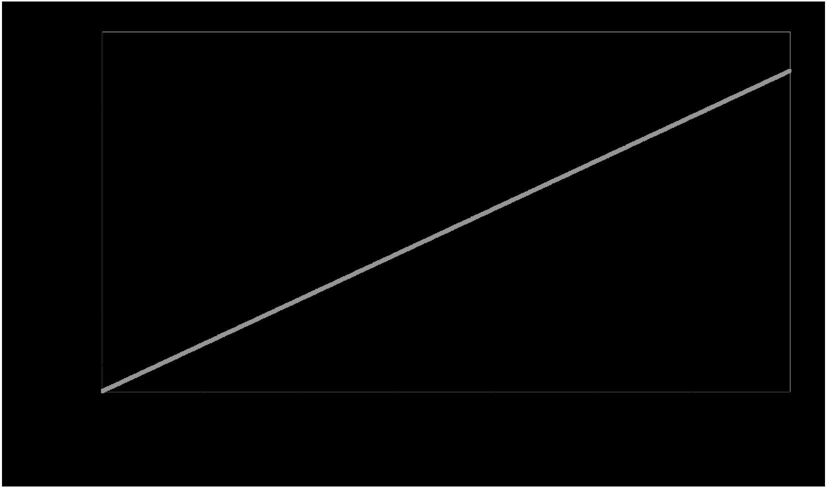
# Pastry: Distance traveled




L=16, 100k random queries, Euclidean proximity space

133

# Pastry delay vs IP delay



GATech top., .5M hosts, 60K nodes, 20K random messages

135

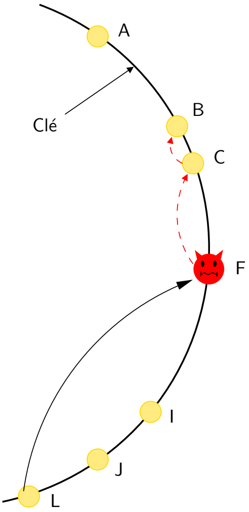
# Défi: participants mal intentionnés

## Gênent transmission des messages

1. Détruisent ou modifient messages
2. Faussent tables de routage

## Captent la gestion des objets

3. Choisissent leur ID
4. Utilisent de multiples ID (Attaque de Sybile)
5. Mentent lors des mises à jour des tables
6. Cherchent à partitionner le système au bootstrap




Castro, Druschel, Ganesh, Rowstron, Wallach, *Security for structured peer-to-peer overlay networks,* ODSI'02.

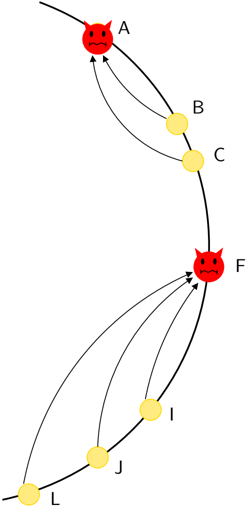
# Défi: participants mal intentionnés

## Gênent transmission des messages

1. Détruisent ou modifient messages
2. Faussent tables de routage

## Captent la gestion des objets

3. Choisissent leur ID
4. Utilisent de multiples ID (Attaque de Sybile)
5. Mentent lors des mises à jour des tables
6. Cherchent à partitionner le système au bootstrap




Castro, Druschel, Ganesh, Rowstron, Wallach, *Security for structured peer-to-peer overlay networks*, ODSI'02.

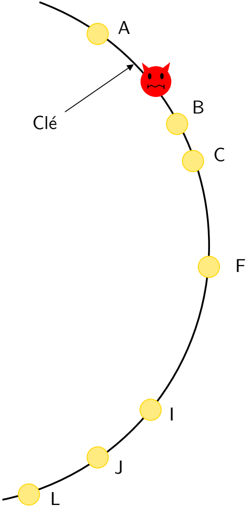
# Défi: participants mal intentionnés

## Gênent transmission des messages

1. Détruisent ou modifient messages
2. Faussent tables de routage

## Captent la gestion des objets

3. Choisissent leur ID
4. Utilisent de multiples ID (Attaque de Sybile)
5. Mentent lors des mises à jour des tables
6. Cherchent à partitionner le système au bootstrap




Castro, Druschel, Ganesh, Rowstron, Wallach, *Security for structured peer-to-peer overlay networks,* ODSI'02.

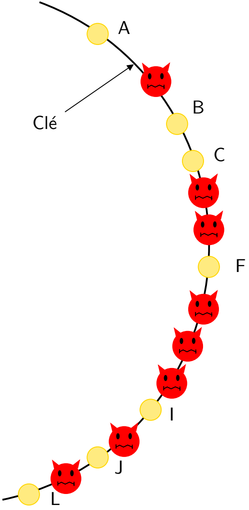
# Défi: participants mal intentionnés

## Gênent transmission des messages

1. Détruisent ou modifient messages
2. Faussent tables de routage

## Captent la gestion des objets

3. Choisissent leur ID
4. Utilisent de multiples ID (Attaque de Sybile)
5. Mentent lors des mises à jour des tables
6. Cherchent à partitionner le système au bootstrap




Castro, Druschel, Ganesh, Rowstron, Wallach, *Security for structured peer-to-peer overlay networks,* ODSI'02.

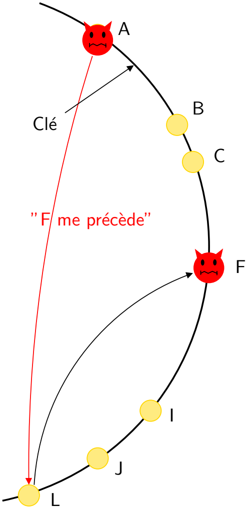
# Défi: participants mal intentionnés

## Gênent transmission des messages

1. Détruisent ou modifient messages
2. Faussent tables de routage

## Captent la gestion des objets

3. Choisissent leur ID
4. Utilisent de multiples ID (Attaque de Sybile)
5. Mentent lors des mises à jour des tables
6. Cherchent à partitionner le système au bootstrap




Castro, Druschel, Ganesh, Rowstron, Wallach, *Security for structured peer-to-peer overlay networks*, ODSI'02.

# Défi: participants mal intentionnés

## Gênent transmission des messages

1. Détruisent ou modifient messages
2. Faussent tables de routage

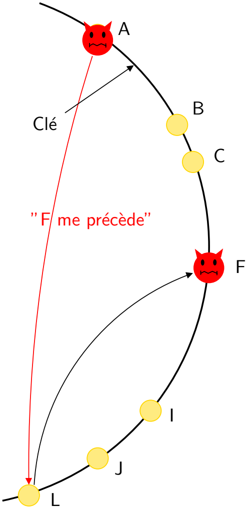
## Captent la gestion des objets

3. Choisissent leur ID
4. Utilisent de multiples ID (Attaque de Sybile)
5. Mentent lors des mises à jour des tables
6. Cherchent à partitionner le système au bootstrap

## Solution Pastry

- Multiple paths (contre 1)
- Protocoles sécurisés d'appartenance (contre 2)
- Choix des ID sécurisé (contre 3 et 4)
- Protocoles sécurisés pour routage (contre 5)

⇒ Fonctionnent malgré 25% de nœuds mal intentionnés




Castro, Druschel, Ganesh, Rowstron, Wallach, *Security for structured peer-to-peer overlay networks*, ODSI'02.

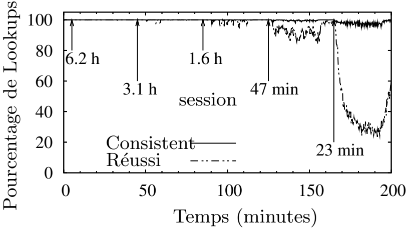
# Défi: churn

## Churn dans systèmes réels

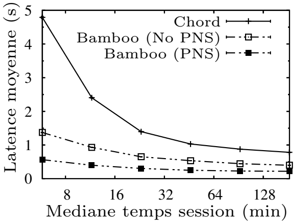
| Article | Système | Durée mesurée |
|---|---|---|
| SGG02 | Gnutella, Napster | 50% $< 60min$ |
| CLL02 | Gnutella, Napster | 31% $< 10min$ |
| SW02 | FastTrack | 50% $< 1min$ |
| BSV03 | Overnet | 50% $< 60min$ |
| GDS03 | Kazaa | 50% $< 2.4min$ |

MTTF $\approx$ 1 heure $\leadsto$ c'est énorme

## Comportement de DHT existants face au churn




Pastry




Chord

Rhea, Geels, Roscoe, Kubiatowicz, *Handling Churn in a DHT*, USENIX'04.

# Défi: churn

## Churn dans systèmes réels

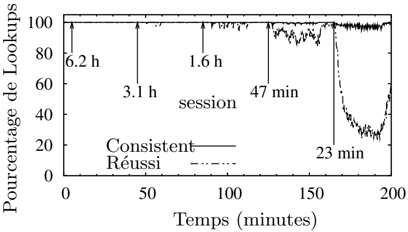
| Article | Système | Durée mesurée |
|---|---|---|
| SGG02 | Gnutella, Napster | $50\% < 60min$ |
| CLL02 | Gnutella, Napster | $31\% < 10min$ |
| SW02 | FastTrack | $50\% < 1min$ |
| BSV03 | Overnet | $50\% < 60min$ |
| GDS03 | Kazaa | $50\% < 2.4min$ |

MTTF $\approx$ 1 heure $\leadsto$ c'est énorme

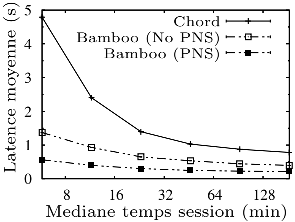
Ce problème reste entier

(même si bamboo l'aborde)

## Comportement de DHT existants face au churn




Pastry




Chord

Rhea, Geels, Roscoe, Kubiatowicz, *Handling Churn in a DHT*, USENIX'04.

# Quelques problématiques actuelles en P2P

- Recherche sous *churn* extrême, mobilité IP (meilleurs algorithmes de routage)
- Gestion des données sous *churn* extrême (création et recherches de réplicas)
- Tirer profit de la localité réseau (sans en dépendre)
- Outils analytiques adaptés (formalisation de systèmes en changement continu)
- Pannes byzantines (fonctionnement malgré participants malveillants)
- Intégrité des données (cryptographie, consistance)
- Généralisation (recherche approchée)
- Répartition de la charge et hétérogénéité
- Gestion des pare-feux, NAT et intranets
- Anonymicité, mesures anti-censure

- Certains de ces problèmes sont résolus dans certains travaux
- Jamais tous en même temps
- Bibliographie du domaine **très** fournie, difficile d'avoir un point de vue général

Risson, Moors, *Survey of Research towards Robust Peer-to-Peer Networks: Search Methods*, Computer Networks, 50(17):3485-521, 2006

# Chapter 5

## Réseaux de capteurs sans fil

# Réseaux de capteurs sans fil (Wireless Sensor Networks)

Principe: composants répartis pour faire des mesures

- Taille: une pièce → une boite d'allumettes
- Processeur: 8-bit → x86
- Mémoire: ko → Mo
- Radio: 20Kbps → 100 Kbps
- Sur batterie

Applications:

- Étude sismologique des bâtiments
- Transport des polluants: Même cause, même effet
- Écosystème des micro-organismes marins

- À chaque fois, maillage des mesures trop grossier
- ⇒ pas de modèle convenable
- Objectif: très nombreux petits senseurs pour affiner le maillage

# Défis des SensorNets

## Énergie

- Les composants sont sur batterie
- La durée de vie de l'ensemble devient une métrique de qualité

## Difficultés de communications

- Puissance (électrique) du réseau: varie avec $\frac{1}{distance^4}$
- 10m $\rightsquigarrow$ 5000 ops/bit transmit ; 100m $\rightsquigarrow$ 50 000 000 ops/bit transmit

$\Rightarrow$ Système fortement décentralisé

$\Rightarrow$ Éviter les communications longue distance autant que possible

## Pas de configuration

- Dissémination des capteurs "aléatoire"

$\Rightarrow$ Besoin d'auto-organisation

## Généralité contre spécificité

- Internet: une seule infrastructure pour toutes les applications
- Sensornet: chaque application a ses propres capteurs, sa propre infrastructure

# Comment obtenir les données

## Motivation et problème

- Clairement un objectif fondamental de ces infrastructures
- Impossible pour chaque composant de joindre un point central (énergie et bande passante limitée)

$\Rightarrow$ Diffusion

## Principe

- On ne sait pas quel nœud a quelle donnée

$\Rightarrow$ on demande une donnée, et la requête est propagée

- Les nœuds ayant l'information répondent

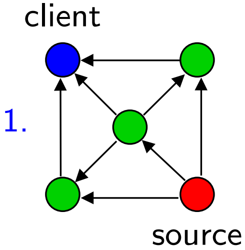
# Schéma de communication: routage data-centric

## Messages

- Paires {attribut, valeur}
- Trois types:
  - Intérêt (des clients)
  - Données (des sources)
  - Renforcement (pour le contrôle)

## Diffusion: deux phases

1. Inonde l'intérêt
2. Inonde les réponses (avec gradients)
3. Les clients renforcent (selon les gradients)
4. Passe les données sur les chemins renforcés



# Schéma de communication: routage data-centric

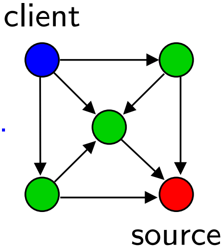
## Messages

- Paires {attribut, valeur}
- Trois types:
  - Intérêt (des clients)
  - Données (des sources)
  - Renforcement (pour le contrôle)

## Diffusion: deux phases

1. Inonde l'intérêt
2. Inonde les réponses (avec gradients)
3. Les clients renforcent (selon les gradients)
4. Passe les données sur les chemins renforcés

2.



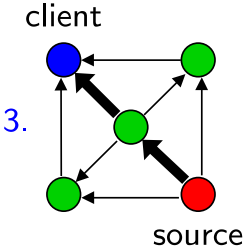
# Schéma de communication: routage data-centric

## Messages

- Paires {attribut, valeur}
- Trois types:
  - Intérêt (des clients)
  - Données (des sources)
  - Renforcement (pour le contrôle)

## Diffusion: deux phases

1. Inonde l'intérêt
2. Inonde les réponses (avec gradients)
3. Les clients renforcent (selon les gradients)
4. Passe les données sur les chemins renforcés



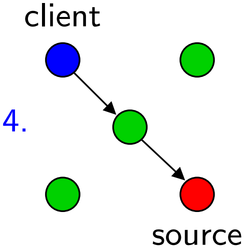
# Schéma de communication: routage data-centric

## Messages

- Paires {attribut, valeur}
- Trois types:
  - Intérêt (des clients)
  - Données (des sources)
  - Renforcement (pour le contrôle)

## Diffusion: deux phases

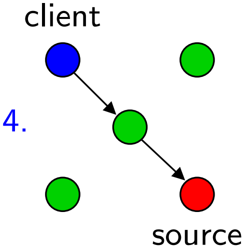
1. Inonde l'intérêt
2. Inonde les réponses (avec gradients)
3. Les clients renforcent (selon les gradients)
4. Passe les données sur les chemins renforcés



# Schéma de communication: routage data-centric

## Messages

- Paires {attribut, valeur}
- Trois types:
  - Intérêt (des clients)
  - Données (des sources)
  - Renforcement (pour le contrôle)



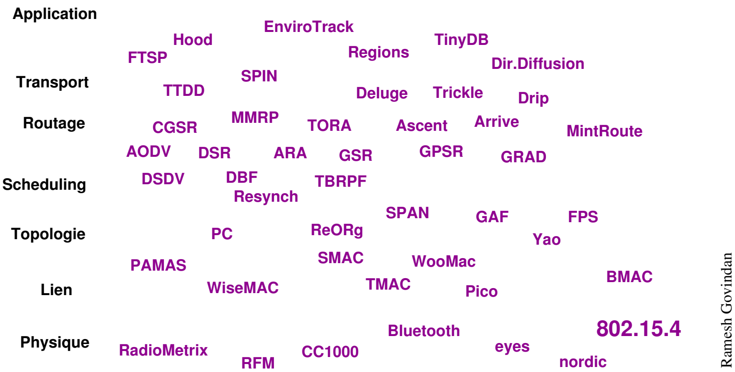

## Diffusion: deux phases

1. Inonde l'intérêt
2. Inonde les réponses (avec gradients)
3. Les clients renforcent (selon les gradients)
4. Passe les données sur les chemins renforcés

## Extension: mise place d'un arbre de diffusion

- Donne la possibilité de combiner les données au passage (min, max, etc)
- C'est encore plus dur...

# Les SensorNet aujourd'hui




## Ce n'est pas franchement un sablier...

- Composants développés séparément (+ suppositions différentes sur l'ensemble)
- Certains offrent une intégration verticale, mais rien en horizontal
- L'objectif semble être de se ramener à un sablier comme IP
- Oui, mais lequel?

# Vers une infrastructure SensorNet unifiée

## L'infrastructure de l'internet

- Objectif 1: connectivité universelle
  - Problème: diversité des technologies; Solution: protocole IP universel
- Objectif 2: flexibilité des applications
  - Problème: réseau adapté aux applications $\leadsto$ peu flexible (car réseau statique)
  - Solution *(end-to-end)*: services pas dans réseau, mais dans hôtes (modifiables)
- Résultat:
  - Protège applications de diversité matérielle, et réseau de diversité applicative
  - Accélère le développement et déploiement de chaque partie

## Les SensorNets

- Applications data-centric $\leadsto$ abstraction *end-to-end* inapplicable
  Traitement au sein du réseau souvent plus efficace
- Objectif: portabilité et reutilisabilité du code (dans la mesure du possible)
  - Pas connectivité universelle, ni flexibilité d'application pour réseaux statiques
- Internet: couches opaques $\Rightarrow$ abstraction simplifiée, mais efficacité décrue
- SensorNet: contraintes (énergétiques, etc) interdisent une telle perte

$\Rightarrow$ couches translucides (masquent les détails matériels, autorisent contrôle)

$\leadsto$ Échange légère perte d'efficacité contre réutilisabilité bien meilleure

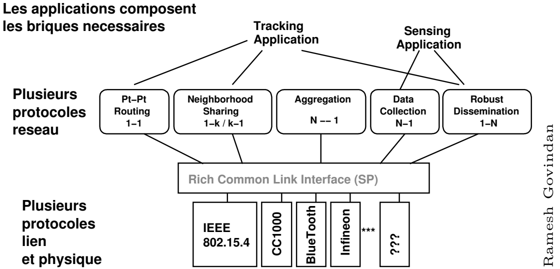
# Possible sablier pour les SensorNets

## Où est le goulot du sablier?

- Dans l'internet: routage *end-to-end* en *best-effort* (IP)
- Sensornets: saut unique par *broadcast best-effort* (SP – *single-hop*)?

- Abstraction assez expressive pour optimisations applicatives
- Abstraction assez pauvre pour capturer réalités matérielles sous-jacentes

## Vision d'ensemble possible



# Conclusion sur les SensorNets

## Pourquoi étudier ces systèmes

- Comme pour TCP/IP à l'époque, le besoin précède la théorie
- Ces systèmes sont déployés, on ne sait pas (vraiment) les utiliser
- C'est donc un thème porteur

## Intérêts théoriques de ces systèmes

- La brique de base est le broadcast, ça change tout
  on va pouvoir revisiter tous les algorithmes de base ;)
- C'est comme un réseau ad-hoc, mais sans mobilité
  c'est plus simple pour commencer

Sixième chapitre

# Conclusion

# Conclusion

## Ce que nous avons vu

- Le domaine des P2P, et des DHT à très large échelle
  - La consistance moins importante que l'échelle?
  - Maturation rapide, le champ scientifique se structure
- Le domaine des SensorNets
  - Encore une fois, les applications ont précédé la théorie
  - Tout est à refaire (broadcast vs IP)
  - Champ restant à défricher (d'un point de vue algorithmique, au moins)

## Ce que nous ne verrons pas (manque de temps)

- Des systèmes plus "classiques"
  - Systèmes de fichiers distribués
  - Bases de données distribuées
  - PKI