IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

FUNDAMENTALS OF SYNCHRONIZATION AND DISTRIBUTED CONSENSUS

YANN BUSNEL

HEAD OF SRCD DEPARTMENT
(*NETWORK SYSTEMS, CYBERSECURITY AND DIGITAL LAW*)

FT-BCT @ VJTI - 30/09/19

- **Synchronization problems are common in everyday life:**
  - When you credit your bank account, this credit must not be lost because at the same time the bank debits a cheque
  - A car park with several entrances, with a capacity of N spaces, must not allow $N + 1$ vehicles to enter
  - Romeo and Juliet can only take each other by the hand if they meet at their appointment
  - A couple do the dishes. One is washing. The other one wipes. The drainer synchronizes them.
  - Some train lines have single track sections. On these sections, only trains running in the same direction at any given time can be used.

- **Bank**
  - When you credit your bank account, this credit must not be lost because at the same time the bank debits a cheque
- **Problem of mutual exclusion**
  - A resource must only be accessible by an entity at a given time.
  - For example, a memory area containing the balance of an account.

- **Parking**
  - A car park with several entrances, with a capacity of N spaces, must not allow N + 1 vehicles to enter
- **Cohort problem**
  - A group of limited size is allowed to offer/use a service.
  - For example, an Internet connection server that must not allow more than *N* parallel connections.

CORRESPONDANCE WITH COMPUTER SCIENCES

- **Love date**
  - Romeo and Juliet can only take each other by the hand if they meet at their appointment
- **Problem of *handing over the baton***
  - Work is divided into processes.
  - Case, for example, of 2 processes that must exchange information at a given time during their execution before continuing

- **Housework**
  - A couple do the dishes. One is washing. The other one wipes. The drainer synchronizes them.
- **Problem of producers/consumers**
  - A consumer can only consume if all producers have done their job.
  - For example, a process need to send buffers that have been filled by other processes

- **Railway traffic**
  - Some train lines have single track sections. On these sections, only trains running in the same direction at any given time can be used.
- **Reader/Writer Problem**
  - Competition between different categories of entities must be managed in a consistent way.
  - For example, a periodic "cleanup" background task that can only be triggered when the main tasks are inactive.

- **Mutual exclusion**
  - Allows management of concurrent access to shared resources
  - Core tool: the semaphore
    - composed of a variable (its value) and a queue (blocked processes)
    - 2 primitives: obtaining and releasing
- **Deadlock**
  - Any situation such as two or more processes are each waiting for a shared resource already allocated to the other

ILLUSTRATION: DINNER OF THE PHILOSOPHER

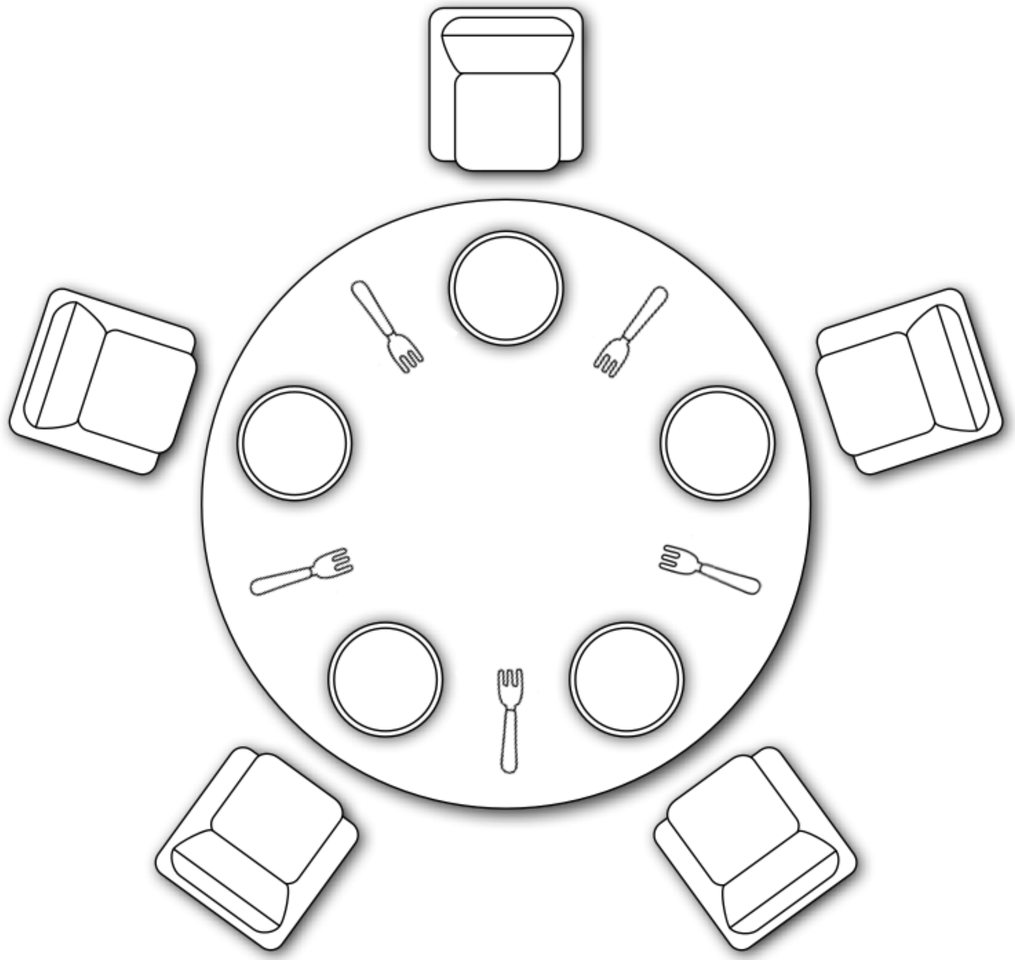


ILLUSTRATION: DINNER OF THE PHILOSOPHER — STATES

- **3 possible states**
  - Thinking
  - Hungry
  - Eating

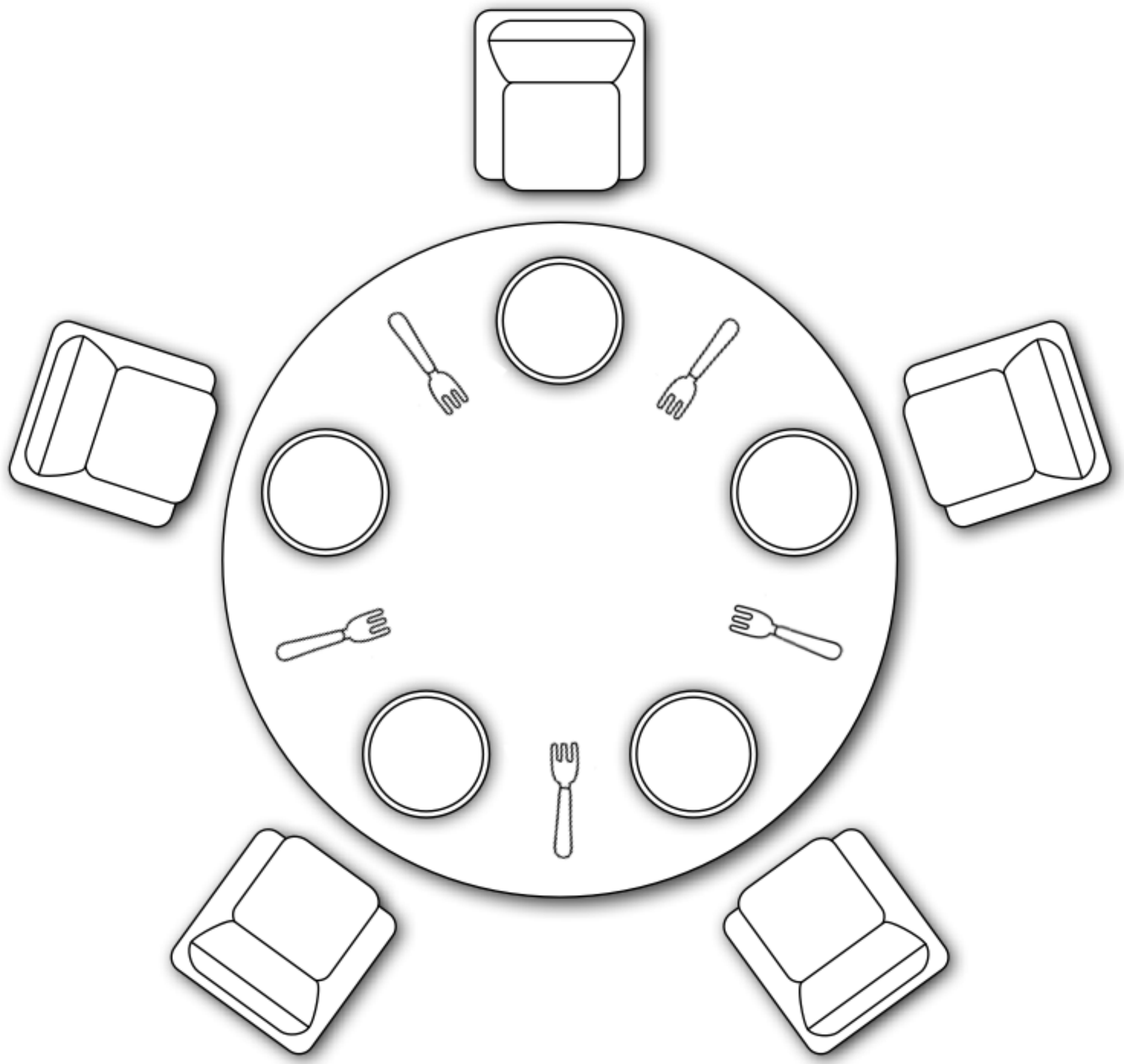


ILLUSTRATION: DINNER OF THE PHILOSOPHER — RULES

- A philosopher cannot eat and think at the same time;
- A thinking philosopher can go hungry at any time;
- A hungry philosopher is trying to start eating;
- To eat, a philosopher must have control of the two forks around him;
- After eating, a philosopher puts the forks back on and starts thinking again.

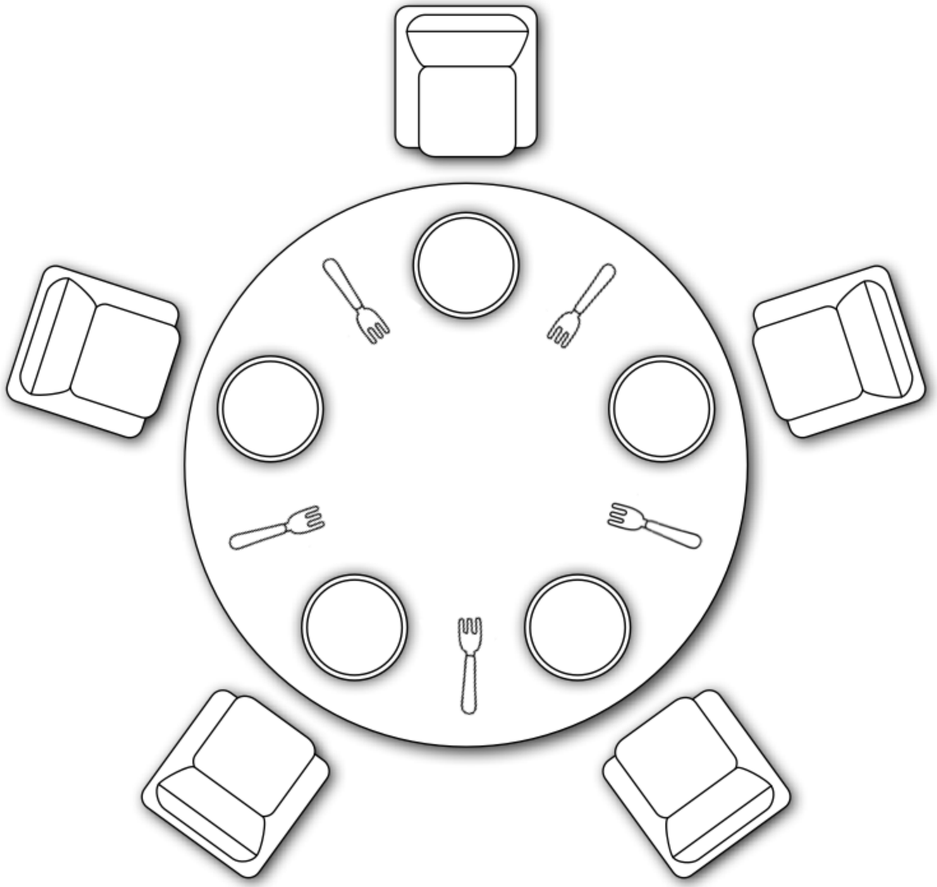
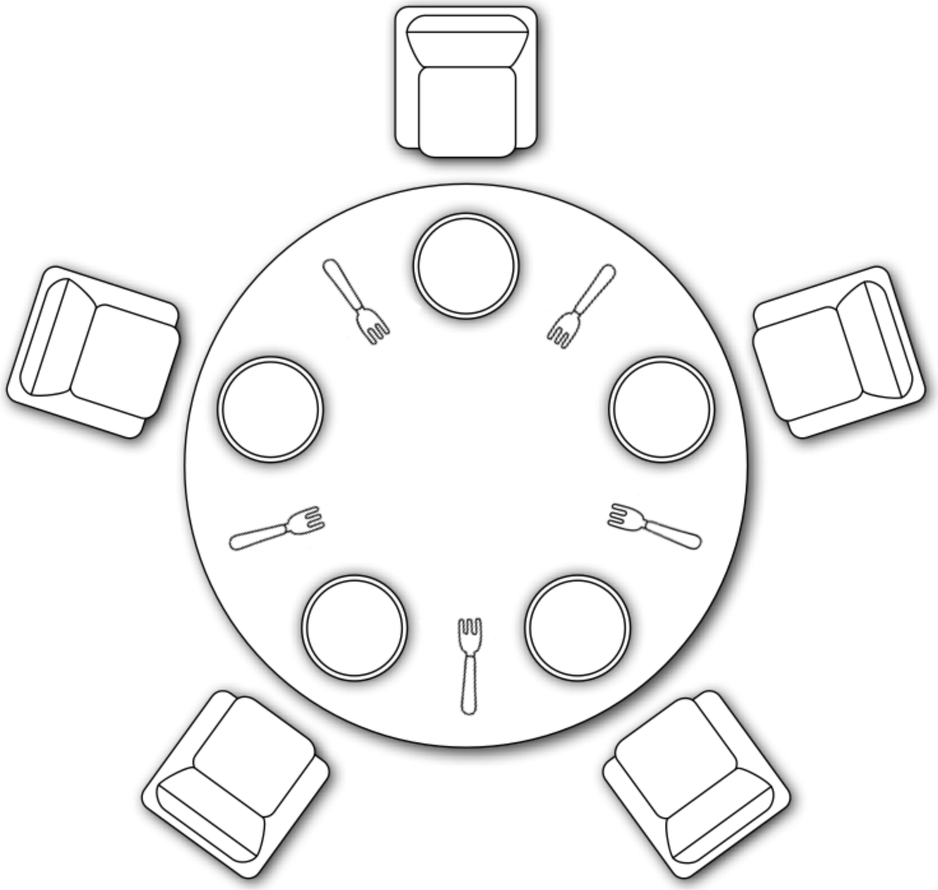


ILLUSTRATION: DINNER OF THE PHILOSOPHER — ALGORITHM

- **All philosophers are identical.**
- **Philosopher Algorithm**

```
1. Repeat
2.    Think()
3.    P(LeftFork)
4.    P(RightFork)
5.    Eating()
6.    V(RightFork)
7.    V(LeftFork)
8. endRepeat
```



Q: Find an execution history that leaves all philosophers in a hungry state.

ILLUSTRATION: DINNER OF THE PHILOSOPHER — ALGORITHM

Q: Find an execution history that leaves all philosophers in a hungry state.

- All philosophers become hungry at the same time
- Everyone take the fork to the left of their plate
- If everyone succeeds, each range will be used
- All try to take the fork to the right of their plate

  ⇨ **Deadlock**

  $\phi 1.1 \rightarrow \phi 2.1 \rightarrow \phi 2.1 \rightarrow \phi 4.1 \rightarrow \phi 5.1$
  $\rightarrow \phi 1.2 \rightarrow \phi 2.2 \rightarrow \phi 3.2 \rightarrow \phi 4.2 \rightarrow \phi 5.2$
  $\rightarrow \phi 1.3 \rightarrow \phi 2.3 \rightarrow \phi 3.3 \rightarrow \phi 4.3 \rightarrow \phi 5.3$
  $\rightarrow \phi 1.4 \rightarrow \phi 2.4 \rightarrow \phi 3.4 \rightarrow \phi 4.4 \rightarrow \phi 5.4 \rightarrow \perp$

- **All philosophers are identical.**
- **Philosopher Algorithm**

```
1. Repeat
2.    Think()
3.    P(LeftFork)
4.    P(RightFork)
5.    Eating()
6.    V(RightFork)
7.    V(LeftFork)
8. endRepeat
```

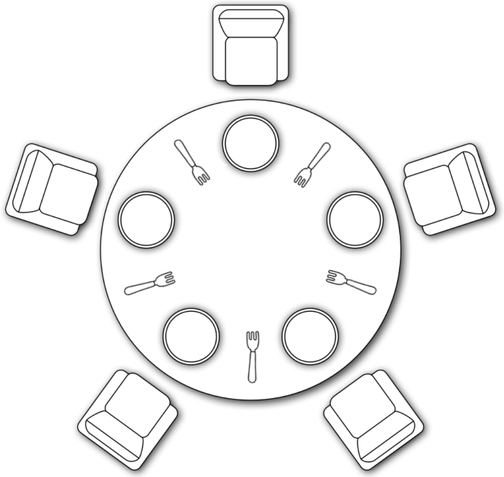
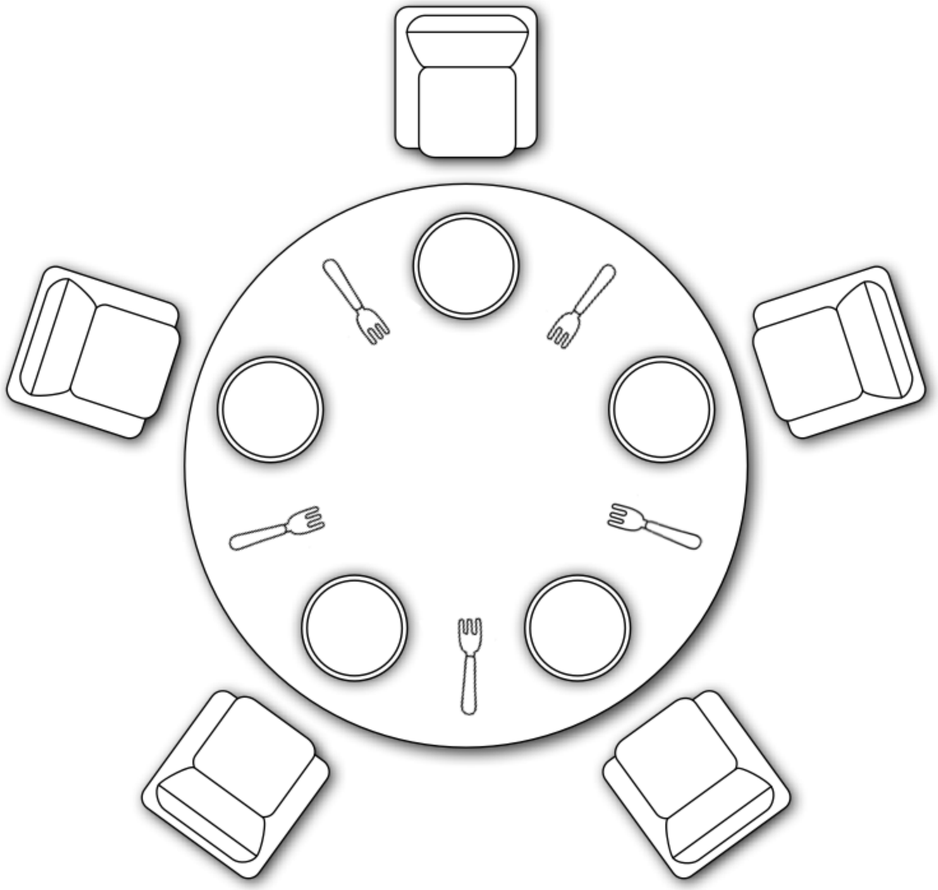


ILLUSTRATION: DINNER OF THE PHILOSOPHER — ALGORITHM

- **2 additional functions :**
  - isAvailable: tests the current use of a fork
  - isHandled: tests the current possession of a fork
- **Philosopher Algorithm**

```
1. Repeat
2.    Think()
3.    P(LeftFork)
4.    P(RightFork)
5.    Eating()
6.    V(RightFork)
7.    V(LeftFork)
8. endRepeat
```



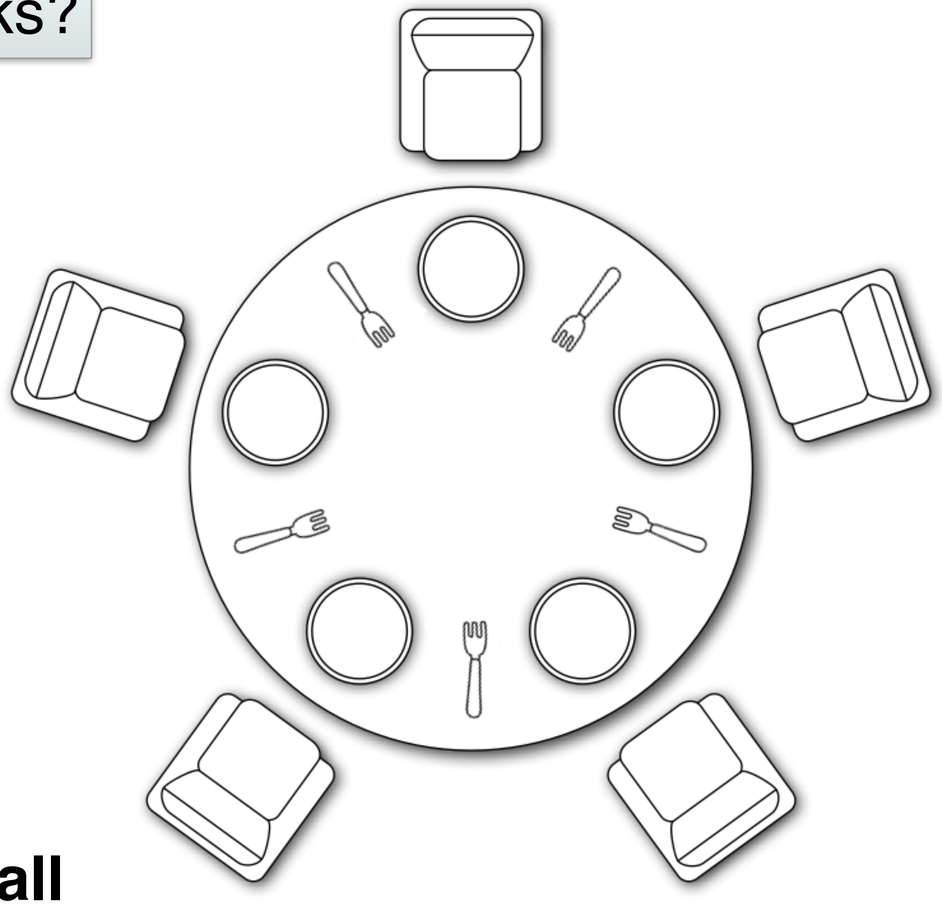
Q: How to modify the algorithm to avoid deadlocks?

ILLUSTRATION: DINNER OF THE PHILOSOPHER — GLOBAL ALLOCATION

Q: How to modify the algorithm to avoid deadlocks?

- **Philosopher Algorithm**

```
1. Repeat
2.    Think()
3.    while not isHandled(RightFork)
4.      if isAvailable(LeftFork)
5.        P(LeftFork)
6.        if isAvailable(RightFork)
7.          P(RightFork)
8.        else
9.          V(LeftFork)
10.       endif
11.     endif
12.   endwile
13.   Eating()
14.   V(RightFork)
15.   V(LeftFork)
16.endRepeat
```



- **2 possibilities only:**
  get both forks or none at all

Q: From the modified algorithm, find an execution that eternally starves a philosopher

ILLUSTRATION: DINNER OF THE PHILOSOPHER — GLOBAL ALLOCATION

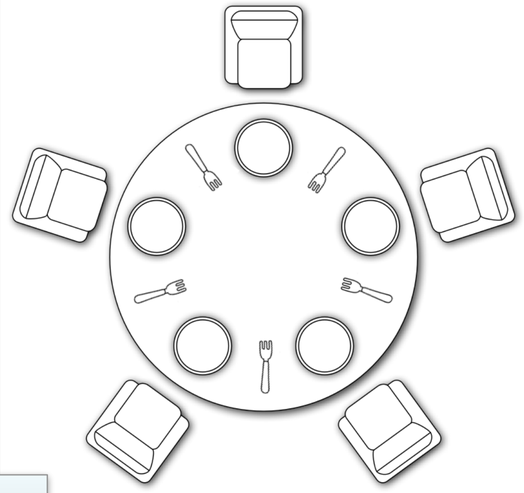
Q: From the modified algorithm, find an execution that eternally starves a philosopher

- The idea is for two philosophers surrounding a third to form a coalition.
- It is enough for the philosopher ϕ3 to think while ϕ1 eats and vice versa, while making sure that they will always eat for a short time together.

$$\phi1.1 \rightarrow \phi1.2 \rightarrow \ldots \phi1.13 \rightarrow$$
$$\phi3.1 \rightarrow \phi3.2 \rightarrow \ldots \phi3.13 \rightarrow$$
$$\phi1.14 \rightarrow \phi1.15 \rightarrow \phi1.16 \rightarrow \phi1.1 \rightarrow \phi1.2 \rightarrow \ldots \phi1.13 \rightarrow$$
$$\phi3.14 \rightarrow \phi3.15 \rightarrow \phi3.16 \rightarrow \phi3.1 \rightarrow \phi3.2 \rightarrow \ldots \phi3.13 \rightarrow$$
$$\phi1.14 \rightarrow \ldots$$

- **Philosopher Algorithm**

```
1.  Repeat
2.    Think()
3.    while not isHandled(RightFork)
4.      if isAvailable(LeftFork)
5.        P(LeftFork)
6.        if isAvailable(RightFork)
7.          P(RightFork)
8.        else
9.          V(LeftFork)
10.       endif
11.     endif
12.   endwile
13.   Eating()
14.   V(RightFork)
15.   V(LeftFork)
16.endRepeat
```



Q: From the modified algorithm, find an execution that eternally starves all philosophers

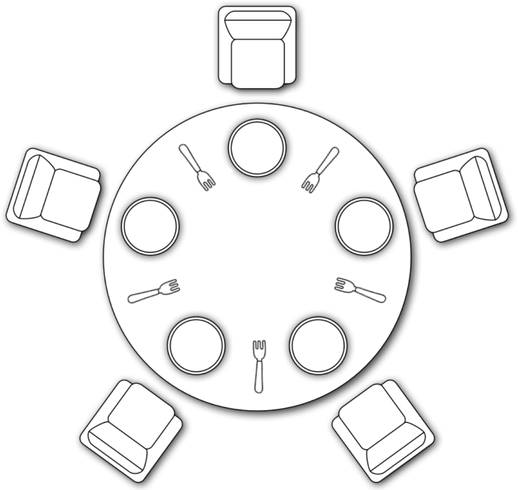
ILLUSTRATION: DINNER OF THE PHILOSOPHER — GENERALIZED FAMINE

Q: From the modified algorithm, find an execution that eternally starves all philosophers

- All philosophers perform lines 1 to 5 at the same time
- No "right fork" will therefore be free: all will put back their "left fork" (see line 9)
- Return to the starting point (line 3) and reproduce the same diagram:
  ⇨ **No deadlock but famine**

Philosopher Algorithm

```
1.  Repeat
2.    Think()
3.    while not isHandled(RightFork)
4.      if isAvailable(LeftFork)
5.        P(LeftFork)
6.        if isAvailable(RightFork)
7.          P(RightFork)
8.        else
9.          V(LeftFork)
10.       endif
11.     endif
12.   endwile
13.   Eating()
14.   V(RightFork)
15.   V(LeftFork)
16.endRepeat
```



DEADLOCK

- **Necessary and sufficient conditions (Coffman, 1971)**
  4 simultaneous NHAs:
  - Resources in mutual exclusion
    - concerned resources that cannot be shared
  - Pending processes
    - process retains allocated resources
  - No pre-emption of resources
    - non-preemptible concerned resources
  - Blocked process circular chain
    - generalization of reciprocal requests

DEADLOCK – COPING STRATEGY

- **Prevention (static)**
- **Detection with healing**
- **Avoidance (dynamic prevention)**

AGREEMENT

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

AGREEMENT PANORAMA

- **Family of problems in distributed algorithms**
- **Categories of problems in this family**
  - Agreement on access to a shared resource
    - Mutual exclusion
  - Agreement on a process playing a particular role
  - Election of a Master
    - Agreement on a common value
  - Consensus
    - Agreement on an action to be taken by all or person
  - Transaction
    - Agreement on the order to send messages to all
  - Atomic diffusion

CONSENSUS VS LEADER ELECTION

- **« Selecting a unique leader is equivalent to solving the consensus problem »**
  - *Consensus on Transaction Commit*
  - by Jim Gray and Leslie Lamport
    - Published on 1 January 2004, revised on 5 July 2017
    - Microsoft Research Research Report

ILLUSTRATION : WATERLOO

- **Several old guard marshals are scattered throughout the Belgian countryside**
- **Must decide "Attack" or "Retreat"**
- **Communicates with sergeants on horseback**
  - Indefinite travel time
- **Possibility of losing a marshal (capture, or worse...)**
  - All the sergeants sent to him will be lost

ILLUSTRATION : WATERLOO

- **Each marshal has an initial idea of the strategy**
- **Properties to be respected**
  - **Termination**
    All marshals who are not prisoners make a decision;
  - **Consensus**
    All Marshals who make a decision make the same decision;
  - **Validity**
    If all the marshals are of the same initial opinion, this must correspond to the final decision.

WATERLOO – SIMPLIFIED MODEL

- **Furtive sergeants and unattainable marshals**
- **Q:** How to solve the problem in this case?
- **A:** 2 possibilities:
  - Concentrate information on a single site
  - All marshals collect all the information

- **Validation algorithm**
  - The coordinator asks each Marshal to agree on *Attack* or *Retreat*, sending a message to each.
  - Each Marshal answers the coordinator on *Attack* or *Retreat*.
    - If his own choice is *Retreat*, the Marshal immediately cancels the battle, and retreats.
  - The coordinator collects the answers of the marshals.
    - If there is no retreat response, *Attack* is decided.
    - Otherwise, *Retreat* is decided.
  - The coordinator relays the decision to the marshals.

WATERLOO – SIMPLIFIED MODEL 2

- **Non-furtive sergeants and attainable marshals**
- **Known maximum travel time of sergeants**
- **Q:** How to solve the problem in this case?
- **A:** Only 1 possibility in this case:
  - ~~Concentrate information on a single site~~
  - All marshals collect all the information

- **Validation algorithm**
  - Each Marshal communicates to all the others his opinion on *Attack* or *Retreat*, sending a message to each;
  - Each non-prisoner marshal receiving a notice different from his own communicates it to all the others, sending a message to each one
    - Allows to take into account the case where a marshal is taken prisoner before having sent all his sergeants;
  - Any marshal whose opinion is not known after a given period of time shall be considered a prisoner.
    - Calculated on the maximum time a sergeant can deliver a message
  - Each non-prisoner marshal collects the marshals' answers.
    - If there is no retreat response, *Attack* is decided.
    - Otherwise, *Retreat* is decided.

WATERLOO – SIMPLIFIED MODEL 2

- **Non-furtive sergeants and attainable marshals**
- **Unknown maximum travel time of sergeants**
- **Q:** How to solve the problem in this case?
- **A:** Impossible to solve!
  - Proven since 1985 by Fischer, Lynch & Paterson (FLP):
    « In an **asynchronous system**, even with a **single faulty process**, it is **impossible to ensure that consensus** will be reached »

AGREEMENT

LET FORMALIZE IT!

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

ASYNCHRONOUS SYSTEM

- **Set of *n* processes, Π = {p_1, ..., p_n}**
  - Processes can fail
  - Failure by stop or Byzantine
- **Communication by messages**
  - No guarantee of delay
  - No guarantee on the order
  - No guarantee on the duplication or alteration of messages
- **Simple model**

UNIFORM GENERAL CONSENSUS

- Processes try to agree on a value
- Each process *proposes* an initial value $v \in V$
- At the end of the protocol, the processes agree on a value $\Diamond v$
- The value *decided* $\Diamond v$ no longer changes
- The value *decided* $\Diamond v$ was *proposed*

UNIFORM GENERAL CONSENSUS

- **Two operations**
  - `proposes`: sets the initial value of a process and shares it with others
  - `decide`: at the end of the protocol, returns the decided value
- **Expected properties**
  - **Termination** Any correct process must decide a value.
  - **Integrity** Any process decides at most once
  - **Uniform agreement** All processes that decide, decide the same value.
  - **Uniform validity** Any process that decides, decides a value v that has been proposed by one of the processes.

UNIFORM GENERAL CONSENSUS

- **Expected properties**
  - **Termination** Any correct process must decide a value.
  - **Integrity** Any process decides at most once
  - **Uniform agreement** All processes that decide, decide the same value.
  - **Uniform validity** Any process that decides, decides a value v that has been proposed by one of the processes.
- **General**
  - No restriction on initial values
- **Uniform**
  - No distinction on the nature of processes

FLP RESULT - MODEL

- **Set of *n* processes, Π = {p_1, ..., p_n}**
- **Deterministic algorithm (automaton)**
- **Each p_i process has a local value v_i**
- **$m \leq n$ faulty processes (crash)**
- **Communication**
  - Reliable broadcast (no loss or duplication)

FLP RESULT - MODEL

- **At each discrete moment a process executes an action**
- **Action ≡**
  - 1. receive a message (`receive`)
  - 2. change status
  - 3. possibly send a message (`send`)
- **Adversary controls the scheduling**
  - Choice of the active process
  - Process failure
  - Message delay

- **No deterministic algorithm in asynchronous environment to solve the consensus problem**

- **Reference :**
  - Fischer, Lynch & Paterson:
    Impossibility of Distributed Consensus with One Faulty Process
  - Initially in *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, March 21-23, 1983, Atlanta, Georgia, USA.
  - Complete version in *Journal of the Association for Computing Machinery (J-ACM)*, Vol. 32, No. 2, April 1985

PROBABILISTIC CONSENSUS [BEN-OR83]

- **Set of n processes, $\Pi = \{p_1, ..., p_n\}$**
- **Each p_i process has a local value v_i**
- **Communication**
  - Reliable distribution (no loss or duplication)
- **$m \leq n$ faulty processes (crash)**

PROBABILISTIC CONSENSUS [BEN-OR83]

- **At each discrete moment a process executes an action**
- **Action** ≡
  - 1. receive a message
  - 2. possibly draw a random value
  - 3. change state
  - 4. possibly send a message
- **Adversary controls the scheduling**
  - Choice of the active process
  - Process failure
  - Message delay

- **Safety**
  - **Uniform agreement:** All processes that decide, decide the same value.
  - **Uniform validity:** Any process that decides, decides a value *v* that has been proposed by one of the processes.
- **Liveliness**
  - **Termination**: Any correct process eventually decides a value.

PROBABILISTIC CONSENSUS [BEN-OR83] – EXPECTED PROPERTIES

- **Safety**
  - **Uniform agreement:** All processes that decide, decide the same value.
  - **Uniform validity:** Any process that decides, decides a value *v* that has been proposed by one of the processes.
- **Liveliness**
  - ~~**Termination**: Any correct process eventually decides a value.~~
  - **Termination**: Any correct process, almost certainly, decides a value.

⇒ Probabilistic algorithm in environment

asynchronous for $m < n/3$ [Lynch96]

REACHING AGREEMENT IN THE PRESENCE OF FAULTS [PSL80]

- **Set of n processes, $\Pi = \{p_1, ..., p_n\}$**
- **Each p_i process has a local value v_i**
- **Communication**
  - Reliable distribution (no loss or duplication)
  - "almost instantaneous" → $\varepsilon \approx 0$
  - Integrity and signature of messages
- **$m \leq n$ faulty processes**
  - Faulty process may refuse to transmit

$\Rightarrow$ Deterministe algorithm for $n \geq 3m + 1$

THE BYZANTINE GENERALS PROBLEM [LSP82]

- **Set of *n* processes, Π = {p_1, ..., p_n}**
- **One p_1 process has a local value v_1**
- **Communication**
  - Reliable distribution (no loss or duplication)
  - "almost instantaneous" → $\varepsilon \approx 0$
  - ~~Integrity and signature of messages~~
  - identified emitter
  - no detectable message
- ***m* ≤ *n* faulty processes**
  - Faulty process can lie about its value v_i
  - Faulty process can lie about a relayed value

⇒ Determinist algorithm for $n \geq 3m + 1$

THE BYZANTINE GENERALS PROBLEM [LSP82]

- **Set of *n* processes, Π = {p_1, ..., p_n}**
- **One p_1 process has a local value v_1**
- **Communication**
  - Reliable distribution (no loss or duplication)
  - "almost instantaneous" → $\varepsilon \approx 0$
  - Integrity and signature of messages
  - identified emitter
  - no detectable message
- ***m* ≤ *n* faulty processes**
  - Faulty process can lie about its value v_i
  - ~~Faulty process can lie about a relayed value~~

⇒ Determinist algorithm for $n \geq m$

- **Consensus :**
  - proposed `operation`(*v*)
  - operation `decide`()
- **Set of properties**
  - Agreement
  - Validity
  - Termination
  - Integrity

- **Consensus :**
  - proposed `operation`(*v*)
  - operation `decide`()
- **Set of properties**
  - Agreement
    - ≈ same decision of all (correct) participants
  - Validity
    - ≈ decision proposed by a (correct) participant
  - Termination
    - ≈ each (correct) participant decides
  - Integrity
    - ≈ each (correct) participant decides at most once

BITCOIN BACKBONE PROTOCOL: ANALYSIS AND APPLICATIONS [GKL17]

- **Set of n processes, $\Pi = \{p_1, ..., p_n\}$**
- **One p_1 process has a local value v_1**
- **Round-based execution**
- **Communication**
  - Reliable channels (no loss or duplication)
  - ~~Almost instantaneous~~
  - All messages are received within a round
  - ~~Integrity and signature of messages~~
- **Processes have "the same" computing power**
- **Each p_i process can perform q access to a crypto oracle**

Adversary

- **Control $m \leq n$ faulty processes**
  - $\Rightarrow mq$ access to the crypto oracle per round
- **Controls the distribution of messages**
- **Can spoof the origin of messages**
- **can control q correct processes during a round**

BITCOIN BACKBONE PROTOCOL: ANALYSIS AND APPLICATIONS [GKL17]

- **Most of the calculation held by the correct processes**
- **Network synchronization ≪ PoW resolution time**
  - ⇒ Common prefix shared by the correct ones
  - Otherwise, the proportion of blocks created by the correct ones tends towards $1/(1+\phi) \approx 38\%$
- **Proportion of blocks created by the adversary is limited (chain quality)**
- **Blockchain increases in the correct ones (chain increase)**

- **Agreement**
  - There is a round after which all the correct processes produce the same value
- **Validity**
  - The value produced by a correct process *p* is proposed by a correct process *q* in the same round
- **Agreement detectable by a correct process**
- **Identification of the processes that agree at this point**

BITCOIN BACKBONE PROTOCOL: ANALYSIS AND APPLICATIONS [GKL17]

- **Nakamoto Protocol ensures agreement**
- **...but not the validity**
- **A consensus protocol can be built**
  - Probabilistic guarantees

- **Lack of strong consistency on the blockchain**
- **Transient inconsistencies**
- **Potentially long:**
  - March 2013 ≈ 6 hours before stabilization
  - July 2019 ≈ 1 hour before stabilization
- **History can be manipulated by a malicious opponent**
  ⇒ Need to introduce strong consistency on the blockchain

- **Problem: How to introduce strong consistency on the blockchain in**
  - an open network,
  - on a large scale,
  - in a distributed manner,
  - secure,
  - and trustworthy?

No safe operating **algorithm** based only on miners who have created a block and **ensure these properties**

E. Anceaume, T. Lajoie-Mazenc, R. Ludinard and B. Sericola: *Safety Analysis of Bitcoin Improvement Proposals*. 15th IEEE Symposium on Network Computing and Applications, 2016

COORDINATION

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

BLOCKCHAIN REQUIRES ORDER

- **Any ledger absolutely needs order**
  - One cannot spend money that has not been received
  - One cannot spend money that is already spend
- **Blockchain transactions (and blocks) must be**
  - Ordered
  - Unambiguously
  - Without the need for a trusted third party
- **A blockchain in a different order is a different blockchain**

LOGICAL TIME

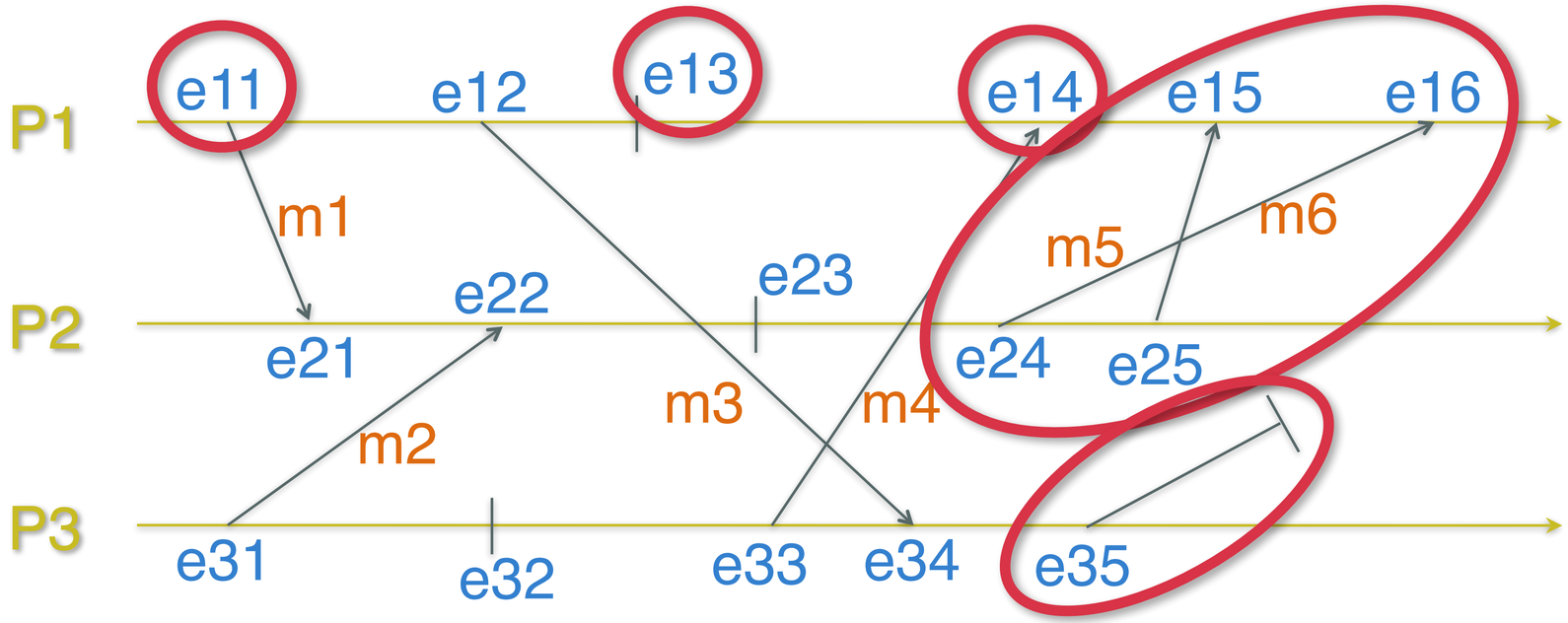
- **Time that is not related to physical time**
- **Objective: to be able to specify the scheduling of process execution and communication**
  - Based on local process events, messages sent and received
  - Creating a logical scheduling
- **Requirement: Creation of a logical clock**

CHRONOGRAM

- **Describes the temporal scheduling of process events and message exchanges**
- **3 types of events:**
  - Transmission
  - Reception
  - Internal event
- **The messages exchanged must respect the topology of the process connection via the channels**

CHRONOGRAM – EXAMPLE

- 3 processes, linked 2 to 2
- Any message propagation times
- Possibility of message loss

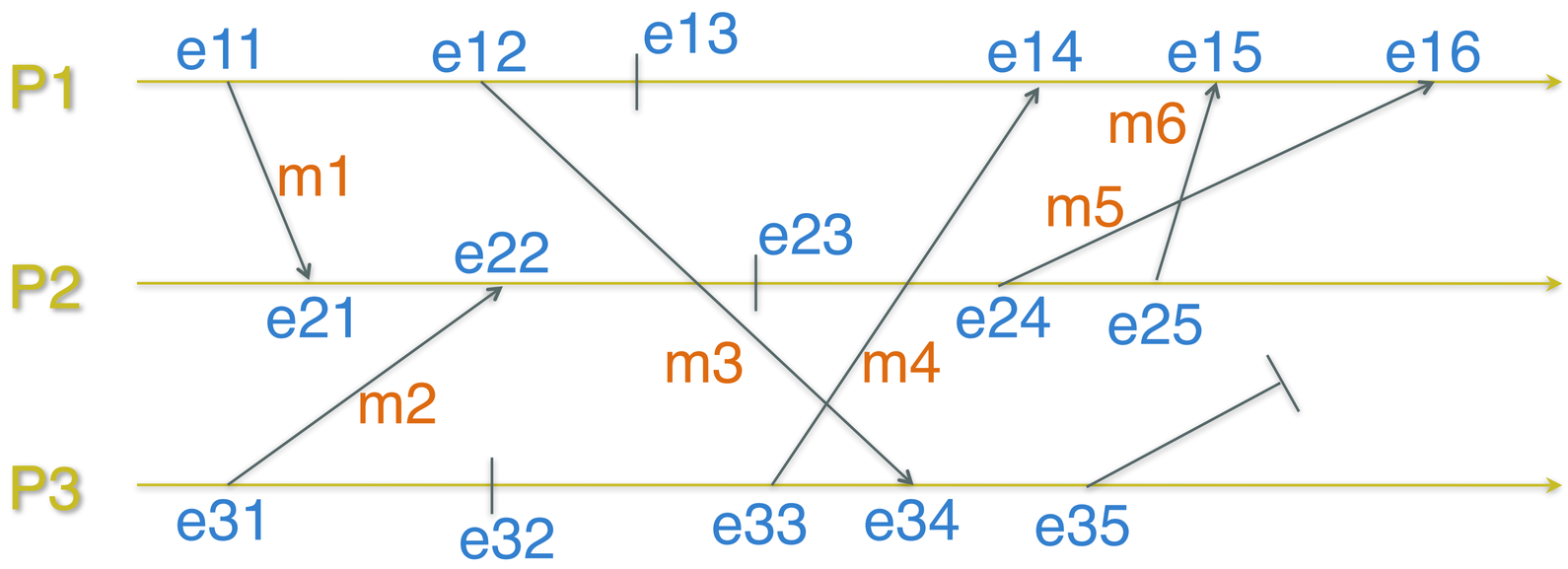


CAUSAL DEPENDENCY

- **Between 2 events if one is to take place before the other**
- **Notation: e → e' (e must take place before e')**
- **Dependency properties**
  - If e and e' are events of the same process, e precedes locally e'
  - If e is the transmission of a message, e' is the reception of this message
  - There is an event f such as e → f and f → e'

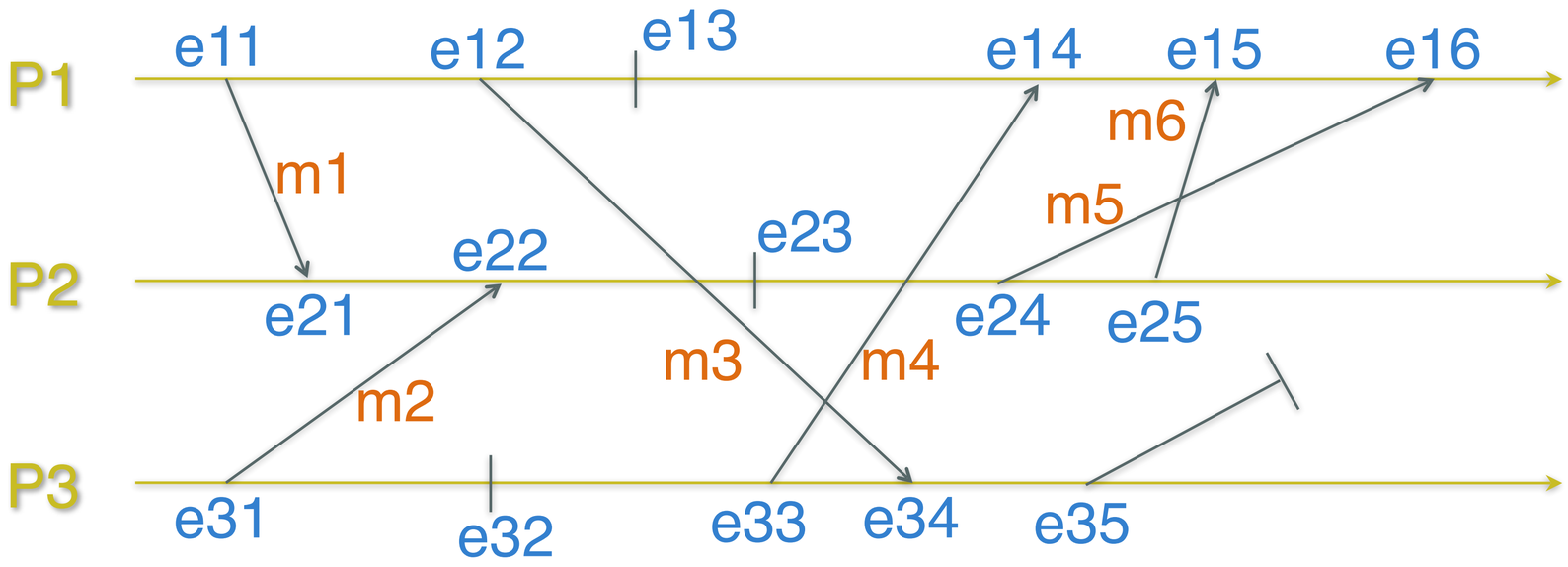
Scheduling events

- Causal dependencies define partial orders for sets of system events

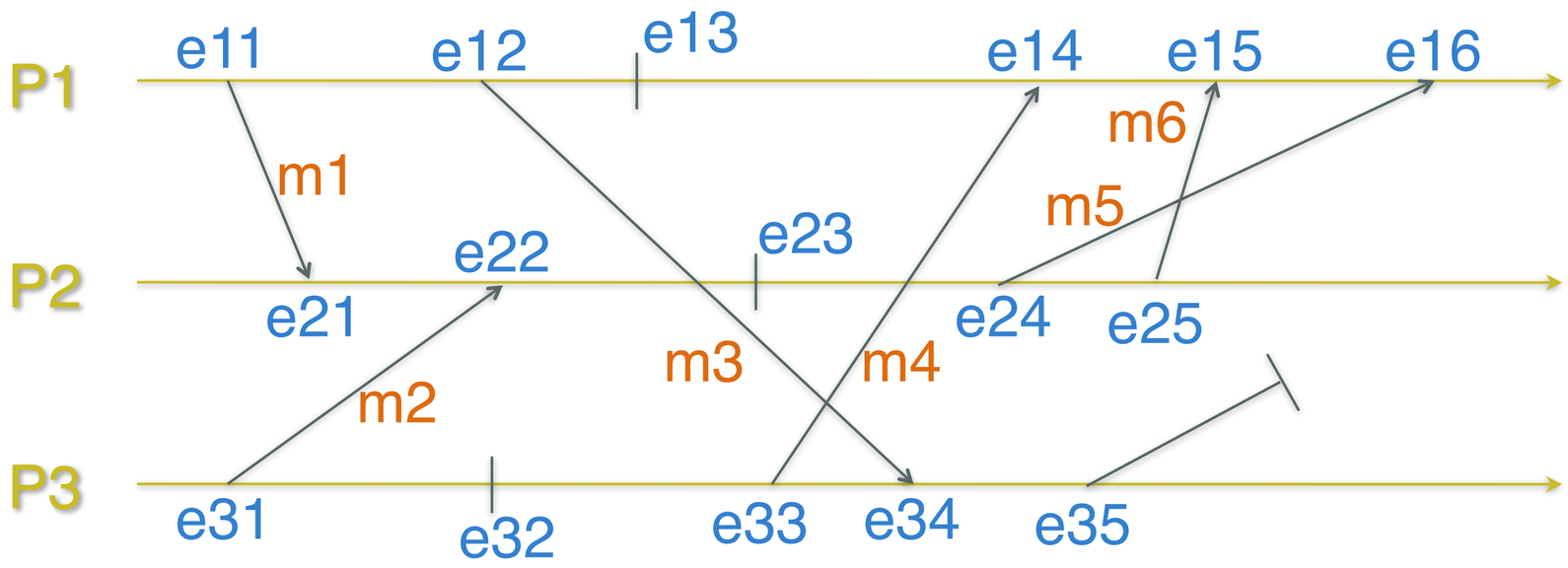


CAUSAL DEPENDENCY

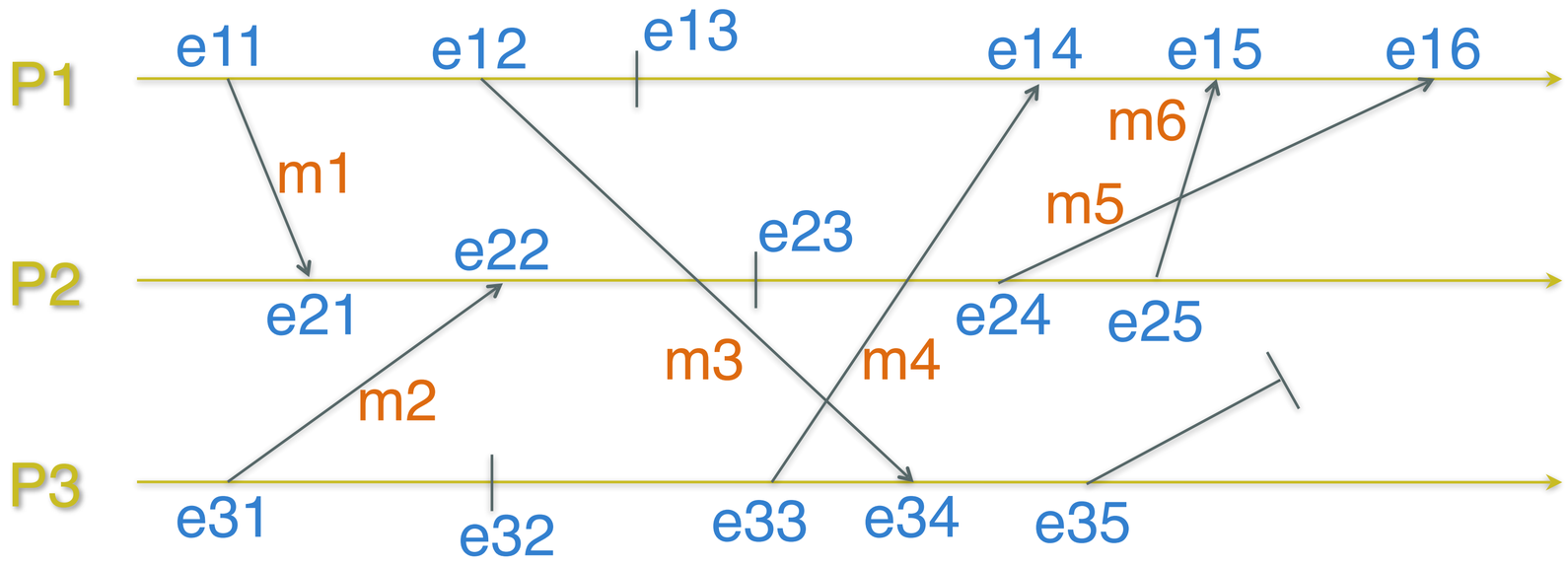
- **Locally: e11 → e12, e12 → e13**
- **On message: e12 → e34**
- **By transitivity: e12 → e35 (as e34 → e35), e11 → e13**



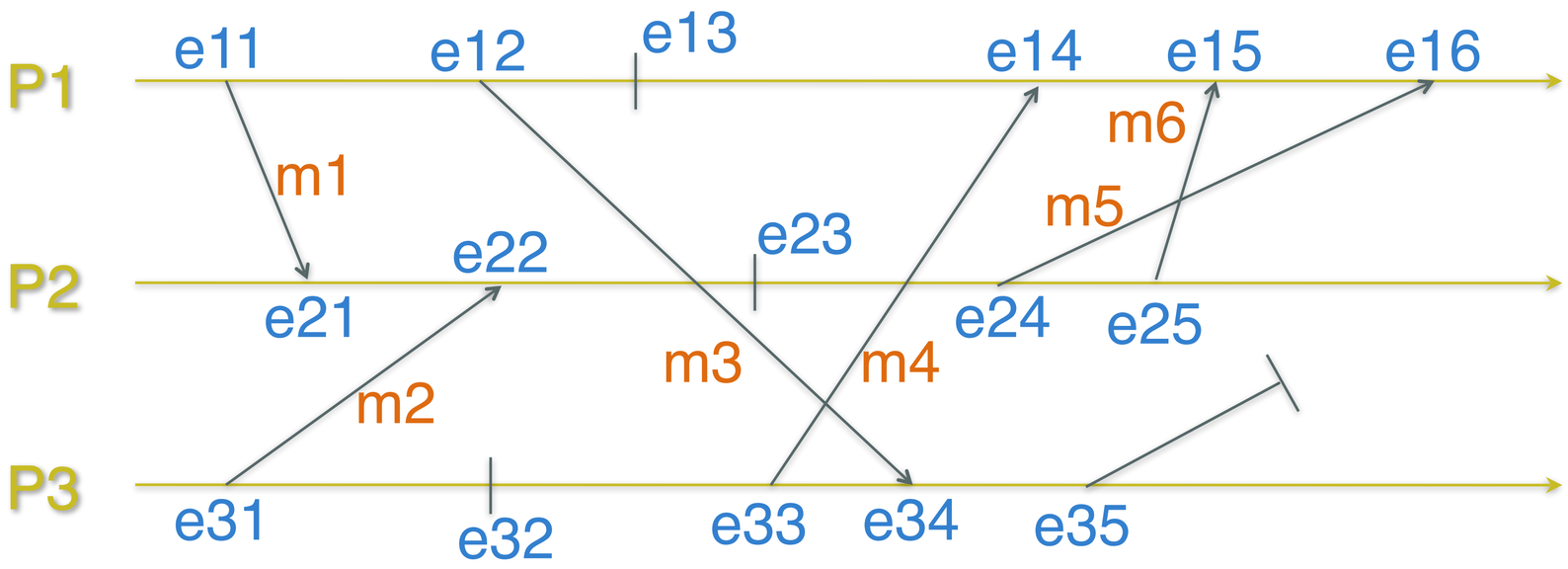
- **Causal dependency between e12 and e32?**
  - No, a priori: absence of causal dependency
  - Causally unrelated events occur in *parallel*



- **Parallelism relationship : ||**
  - $e \,||\, e' \Leftrightarrow \neg((e \rightarrow e') \vee (e' \rightarrow e))$
  - Does not mean that the 2 events take place simultaneously, but that they can take place in any order



- Dating of each of the system events
- With respect to causal dependencies between events



- **Timestamp (Lamport clock)**
  - One data per event
  - Information at the global level
- **Vectorial (Mattern clock)**
  - One vector per event
  - Information on each of the other processes
- **Matrix**
  - One matrix per event
  - Information on the information that each of the processes in the other processes knows

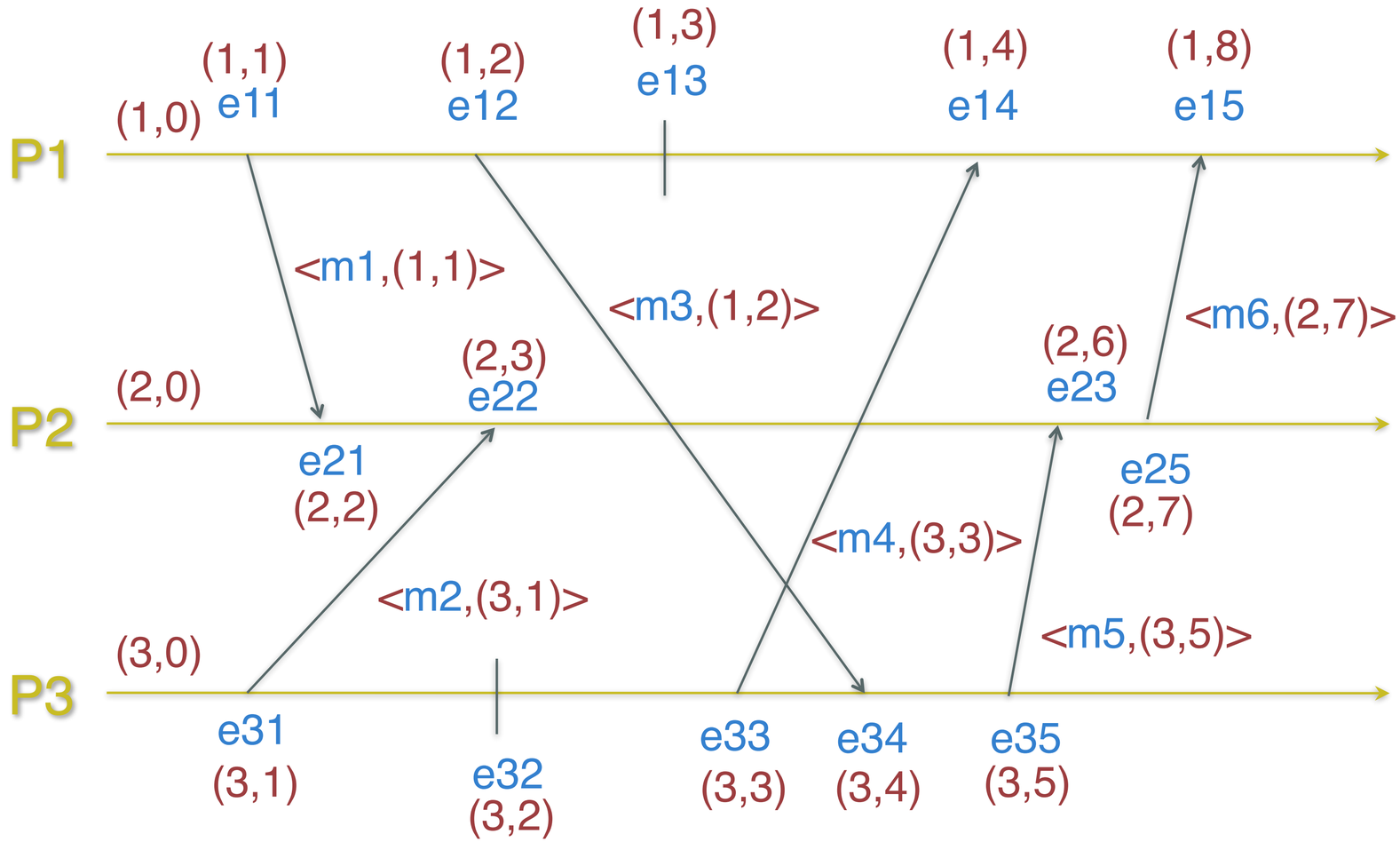
LAMPORT CLOCK (1978)

- **A date is associated with each event using a pair (s, nb)**
  - s : process number
  - nb : event number
- **Respect for causal dependency**
  - $e \rightarrow e' \Rightarrow H(e) < H(e')$
    - $H(s, nb) < H(s', nb')$ if $(nb < nb')$ or $(nb = nb' \text{ and } s < s')$
  - But not the other way around: $H(e) < H(e') \Rightarrow \neg (e' \rightarrow e)$
    - That is to say: either $e \rightarrow e'$, or $e \parallel e'$

LAMPORT CLOCK (1978) – CREATION

- **Locally, each Pi process has a local logic clock Hi, initialized to 0**
  - Used to date events
- **For each local event in Pi**
  - Hi = Hi + 1: the local clock is incremented
  - The event is locally dated by Hi
- **Transmission of a message by Pi**
  - Hi is incremented by 1 and after, the message is sent with (i, Hi) as the timestamp
- **Receiving a message m with timestamp (s, nb)**
  - Hi = max(Hi, nb) +1 and marks the reception event with this update Hi
    - Hi is possibly shifted to the clock of the other process before being incremented

LAMPORT CLOCK (1978) – EXAMPLE



LAMPORT CLOCK (1978)

- **Overall scheduling**
- **Using Hi, we order all system events among themselves**
- **Total order obtained is arbitrary**
  - If causal dependency between 2 events: order respects dependency
  - If causal independency between: choice arbitrary of an order between the 2
    - No problem in practice since they are independent

MATTERN CLOCK (1989)

- **Mattern & Fidge Clock, 1989-91**
  - Clock that ensures the reciprocal of causal dependency
    - $H(e) < H(e') \Rightarrow e \rightarrow e''$
  - Also allows to know if 2 events are parallel (not causally dependent)
  - Does not, however, define a global total order

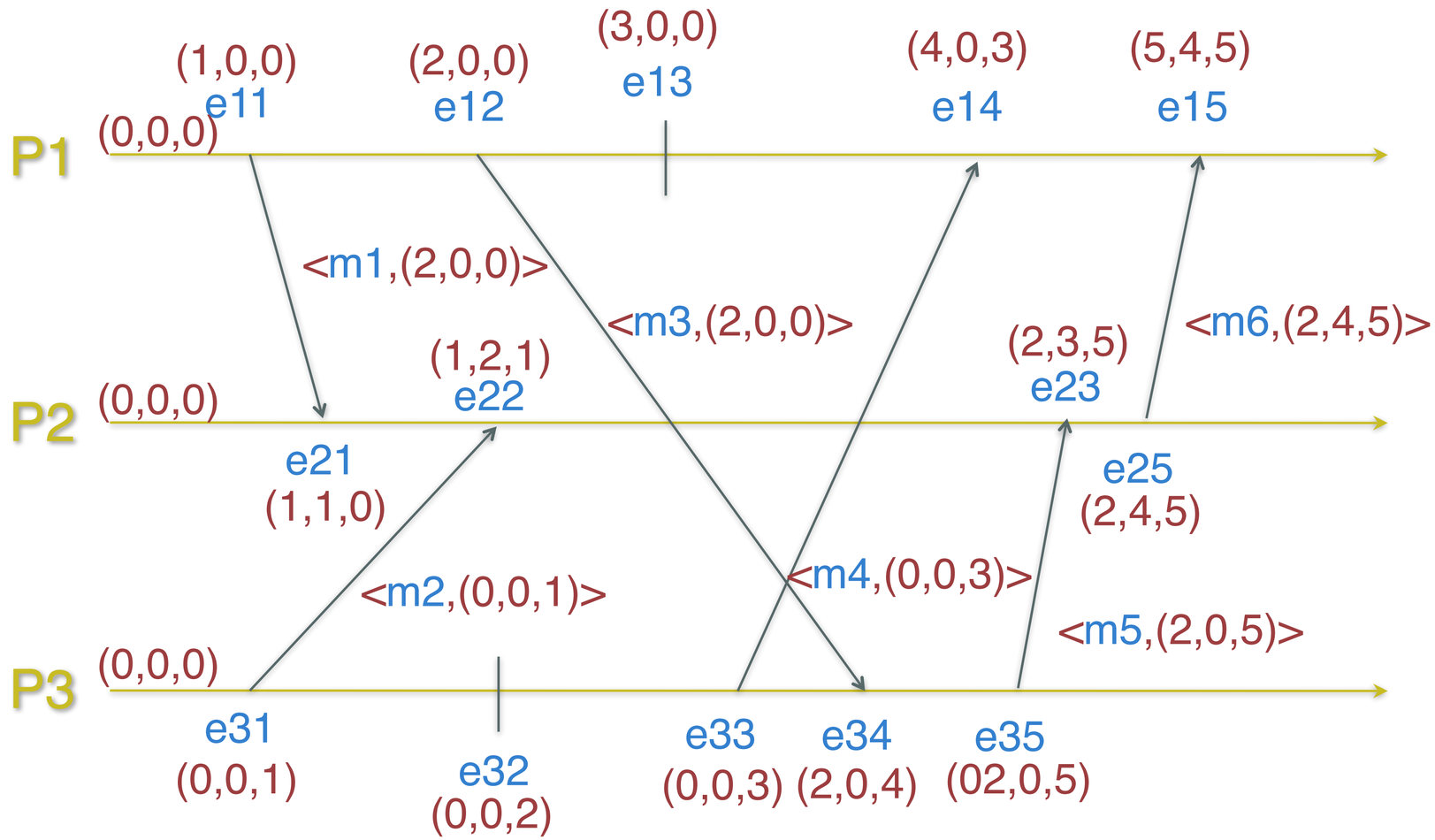
MATTERN CLOCK (1989)

- **Principle**
  - Use of vector V of size equal to the number of processes
    - Locally, each Pi process has a vector Vi
    - A message is sent with a date vector
  - For each Pi process, each Vi [j] cell of the vector will contain values of the clock of the Pj process

MATTERN CLOCK (1989)

- **Initialization: for each process Pi, Vi = (0, ..., 0)**
- **For a Pi process, at each of its events (local, emission, reception):**
  - Vi [i] = Vi [i] + 1
    - Incrementing the local event counter
  - If a message is sent, then Vi is sent with the message
- **For a Pi process, when receiving a message m containing a vector Vm, we update the boxes j ≠ i of its local vector Vi**
  - ∀j : Vi [j] = max (Vm [j], Vi [j])
    - Memorizes the number of events on Pj that are causally dependent on Pj with respect to the transmission of the message
    - The reception of the message is therefore also causally dependent on these events on Pj

MATTERN CLOCK (1989) – EXAMPLE



MATTERN CLOCK (1989)

- **Partial order relationship on dates**
  - $V \leq V'$ defined by $\forall i : V[i] \leq V'[i]$
  - $V < V'$ defined by $V \leq V'$ and $\exists j$ as $V[j] < V'[j]$
  - $V \parallel V'$ defined by $\neg(V < V') \wedge \neg(V' < V)$

MATTERN CLOCK (1989)

- **Causal dependency and independency**

- **Mattern Clock provides the following properties, with e and e' two events and V(e) and V(e') their dates**
  - $V(e) < V(e') \Rightarrow e \rightarrow e''$
    - If two dates are ordered, there is necessarily a causal dependence between the dated events
  - $V(e) \parallel V(e') \Rightarrow e \parallel e'$
    - If there is no order between the 2 dates, the 2 events are causally independent

- **n process : matrix M of (n x n) to date each event**
- **On Pi process**
  - Line i: Information on Pi events
    - Mi [i, i] : number of events performed by Pi
    - Mi [i, j] : number of messages sent by Pi to Pj (with j ≠ i)
  - Line j (with j ≠ i)
    - Mi [j, k] : number of messages known to have been sent by Pj to Pk
    - Mi [j, j] : number of events known on Pj (with j ≠ k)

MATRIX CLOCK

- **A Pi process has knowledge about the number of messages that a Pj process has sent to Pk**
  - When we receive a message from another process, we compare the received clock with the local clock
    - Can determine if a message should not be received before the current one

SUMMARY ON LOGICAL CLOCK

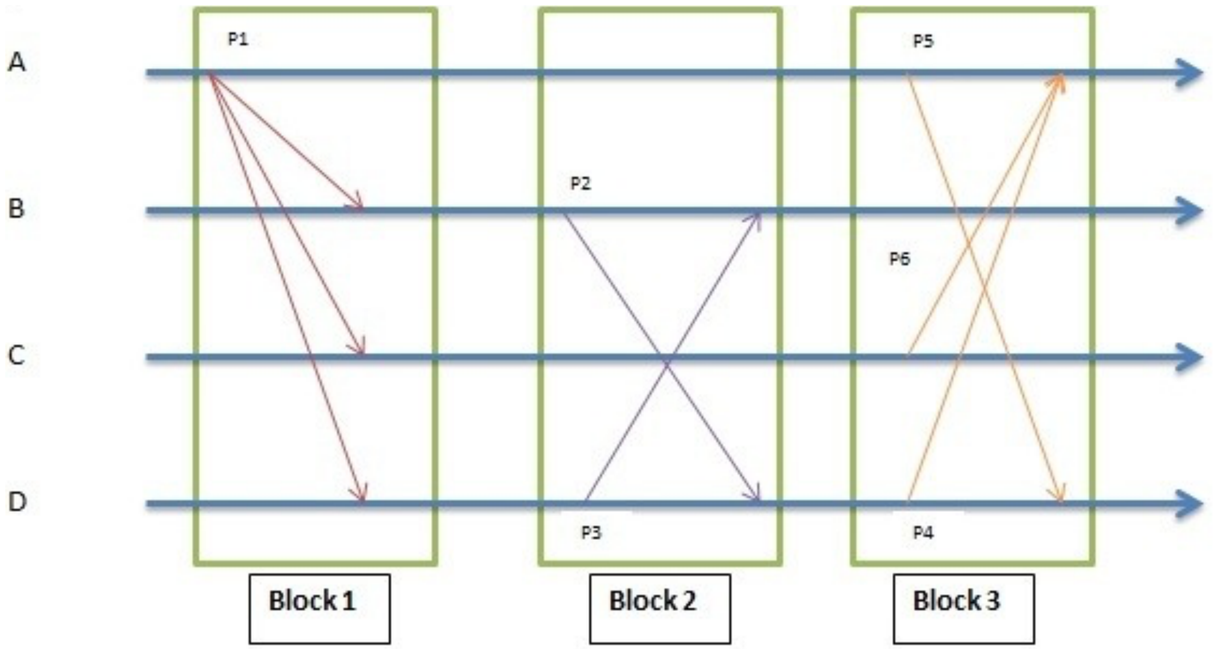
- **Information given by the clocks**
  - **Timestamp**: global knowledge of the number of system events
  - **Vector**: knowledge of events on each of the other processes
  - **Matrix**: knowledge of the knowledge of events that a process knows about others
- **Application, usage**
  - **Timestamp**: global scheduling, priority management
  - **Vector**: validation of the consistency of a global state, properties on diffusion
  - **Matrix**: ensure causal delivery of messages between several processes

SEEMS THAT LAMPORT CLOCK IS ENOUGH FOR BLOCKCHAIN

- **Lamport's clock provides total order, in a decentralized way**
  - Time in Lamport clock is relative to processes
- **In Blockchain, the concept is different**

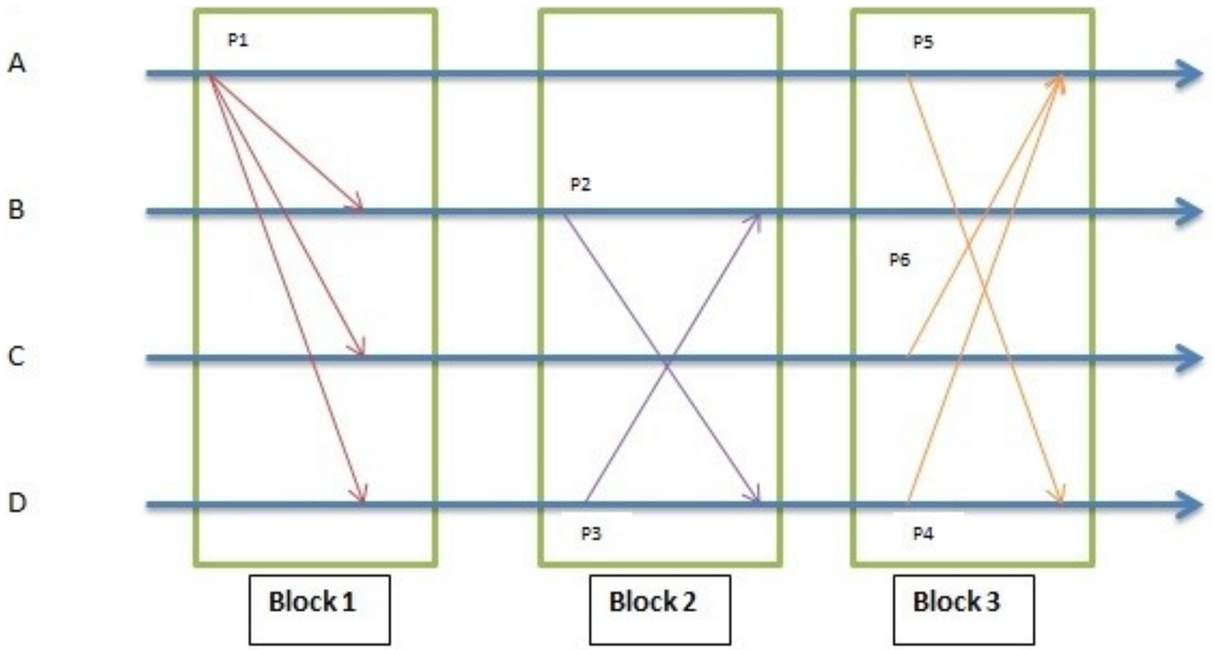
LET'S FOCUS ON A SIMPLE EXAMPLE

- **A has 5 Bitcoin and rest all had 0 Bitcoin**
- **A sends 1 Bitcoin to each B, C and D**
  - after some all has, A has 2 Bitcoin and rest all has 1 bitcoin
- **In Process P1, A sends bitcoin to B, C and D.**
  So total 3 processes, but all are irrelevant to each other
- **Here, no Timestamp needed**



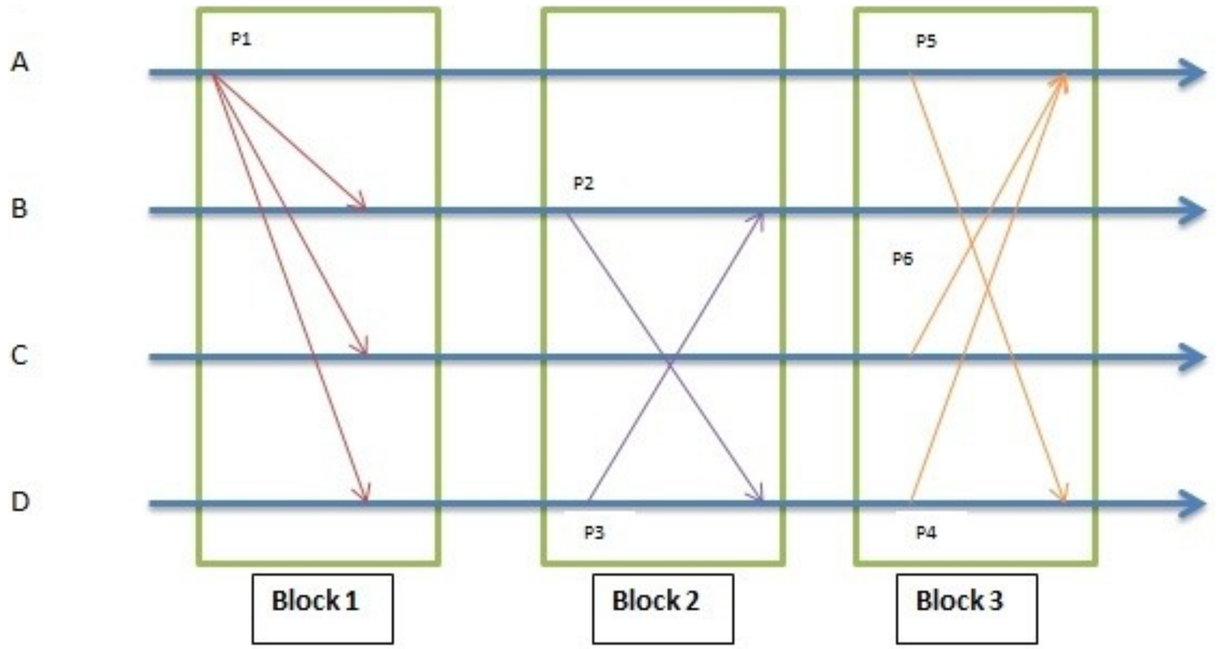
LET’S FOCUS ON A SIMPLE EXAMPLE

- Block generation is a mechanism to validate transaction and broadcast to every other node.
- So block generates here, name it Block 1 and time is 1.
- As per concept of Blockchain, all transaction occur at a time of block generations



LET'S FOCUS ON A SIMPLE EXAMPLE

- **Now, in P2, B sends bitcoin to D and in P3, D sends bitcoin to B**
- **P2 and P3 has same users BUT dealing with separate resource**
  - Bitcoin is a digital resource here each one is unique
  - Both B and D has separate ledger
- **They don't need to manage concurrency by their own**
- **Include in Block2**
  - every transaction needs to be validated



WHY NOT LAMPORT ’S CLOCK?

- **Lamport’s should be used where processes need same resources at a certain time**
  - So, Lamport’s clock manage concurrency
- **In the case of Blockchain**
  - Each node has a separate ledger and separate processes
  - They have block generation algorithm

PROOF-OF-WORK IS KIND OF LOGICAL CLOCK FOR BLOCKCHAIN

- **The Bitcoin blockchain PoW is simply a distributed, decentralized clock**
  - Nothing happens between Blocks
  - The difficulty in finding a conforming hash acts as a clock
- **It doesn’t matter that this clock is imprecise**
  - What matters is that it is the same clock for everyone
  - the state of the chain can be tied unambiguously to the ticks of this clock
- **This clock is operated by the multi-exahash rate of an unknown number of collective participants spread across the planet, completely independent of one another**

PROOF-OF-WORK COMPARES TO PROOF-OF-STAKE

- **The two are not comparable!**
- **Proof-of-Stake is about (randomly distributed) authority**
- **Proof-of-Work is a clock**

REFERENCES FOR THIS LECTURE

- Sacha Krakowiak : *Algorithmique et techniques de base des systèmes répartis* (Dunod)
- Michel Raynal : *Synchronisation et état global dans les systèmes répartis* (Eyrolles)
- George Couriolis, Jean Dollimore & Tim Kindberg : *Distributed Systems: Concept and Design*, 3ème édition, (Addison Wesley)
- Eric Cariou : *Système distribués* (Université de Pau)
- Nicolas Baudru - Traïan Muntean : *Programmation distribuée* (ESIL)
- Daniele Varacca : *Concurence* (Univsersité Paris 7)
- Romaric Ludinard : *Blockchain* (IMT Atlantique)
- Gregory Trubetskoy : *Blockchain PoW Is a Decentralized Clock* (grisha.org)
- Priyank Jani : *Why can't we use a Lamport clock to order blocks in a BC?* (quora.com)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THANKS FOR YOUR ATTENTION

yann.busnel@imt-atlantique.fr