MAD

Models & **A**lgorithms for **D**istributed systems

-- 2/5 --

download slides at

http://people.rennes.inria.fr/Eric.Fabre/

Today…

- Runs/executions of a distributed system are partial orders of events

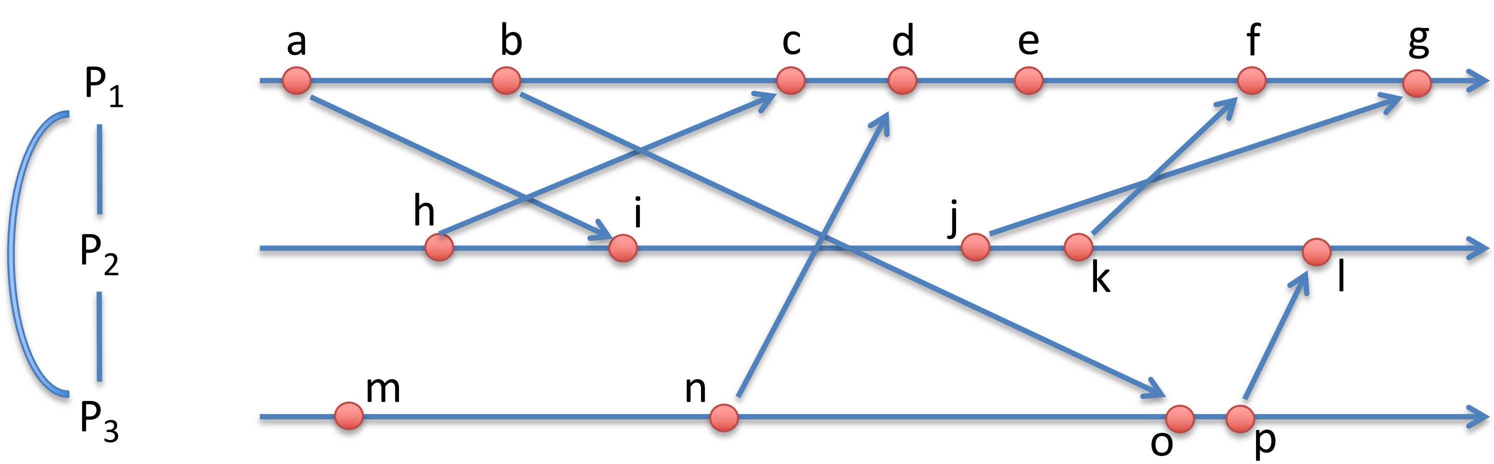
- We introduce
  - logical clocks (Lamport, Fidge-Mattern)
  - event structures
  - distributed algorithms to build them

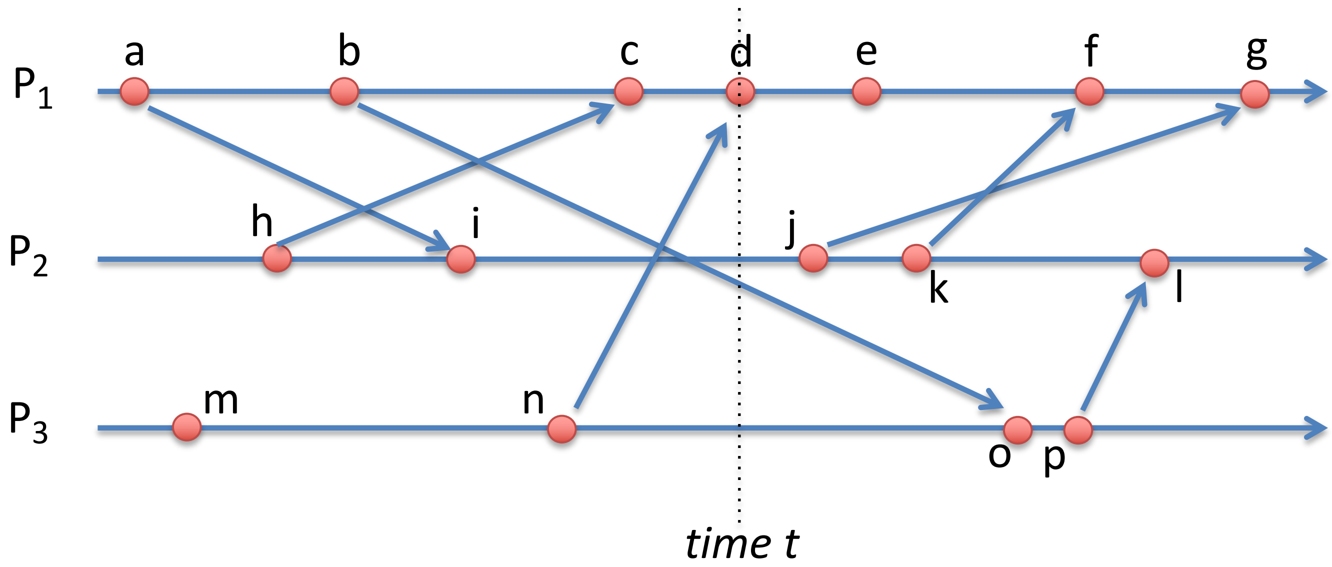
- Then explore applications to
  - money counting in a distributed transactional system
  - the construction of snapshots

Runs of distributed systems

Context

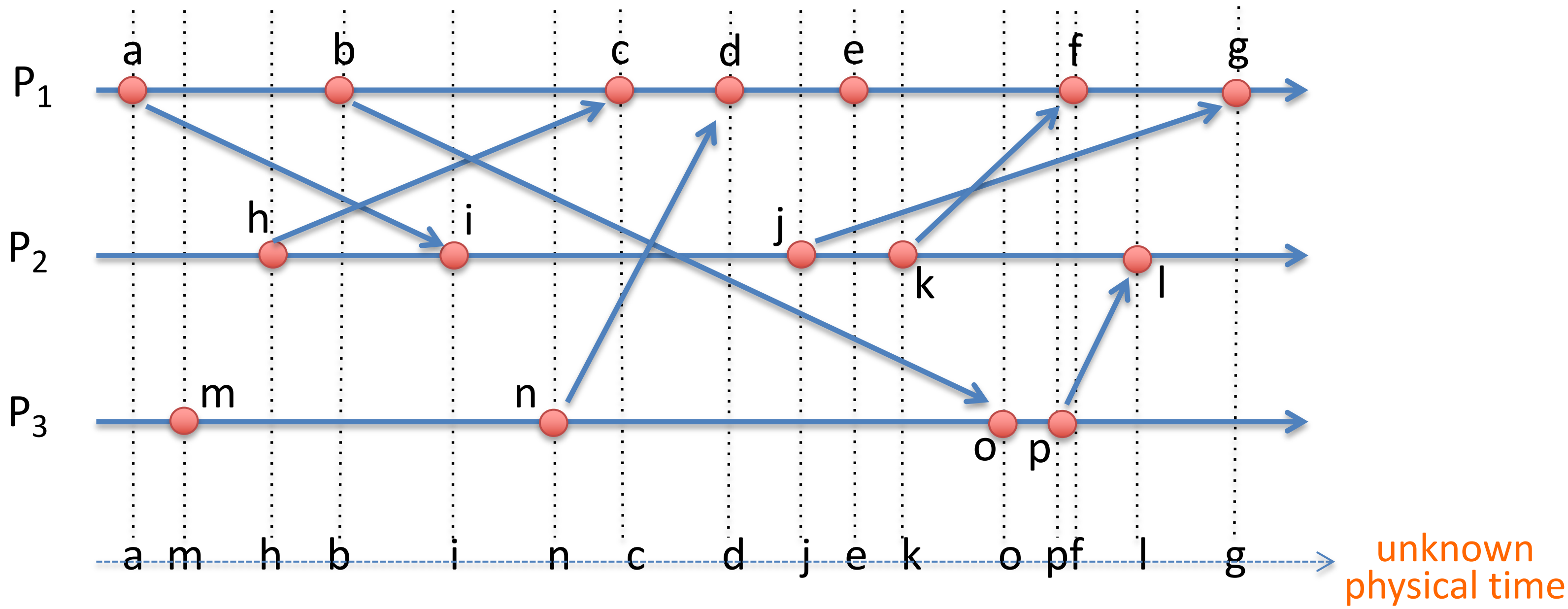
- We assume processes have UIDs {1,2,…,n}.
- So far, we had an undirected interaction graph of processes G=(V,E), where V={1,2,…,n}.
- Processes are asynchronous (no global clock), don't fail, messages eventually reach their destination.
- We now examine a run of such a distributed system, with local events in each process P_i, and message exchanges from P_i to P_j (where allowed).





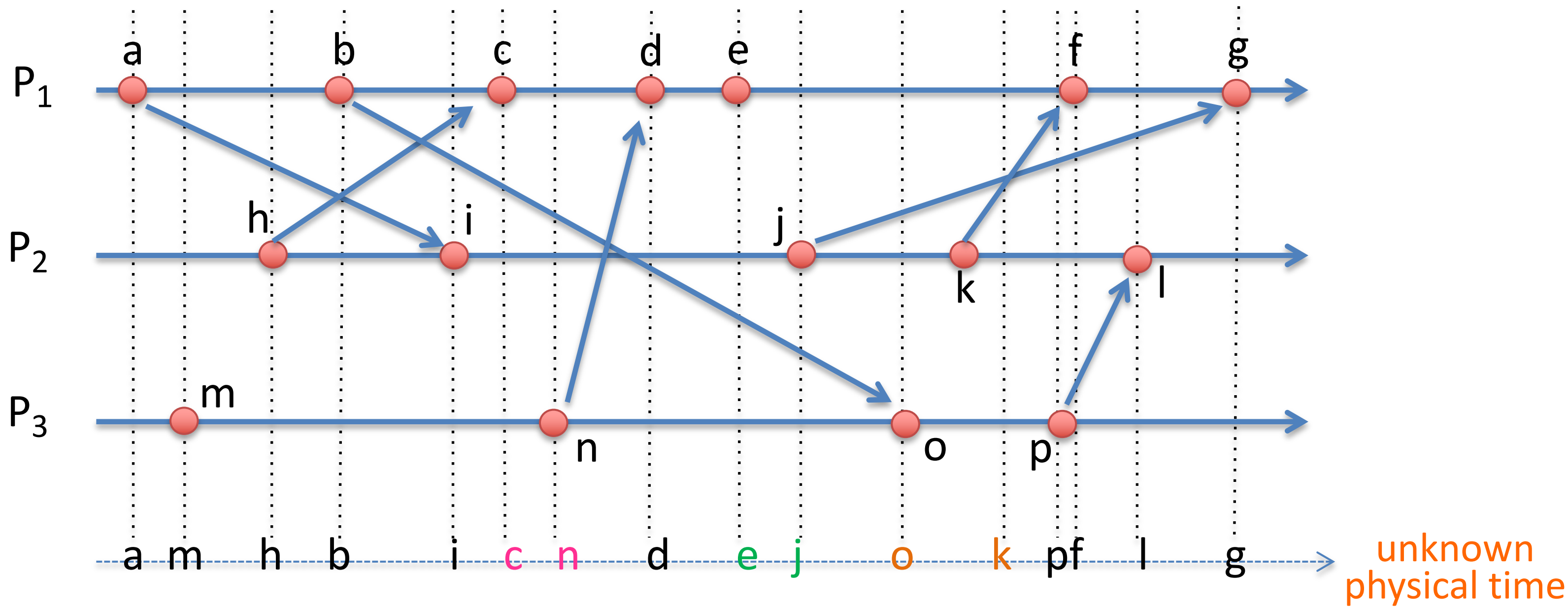
“a chronogram view”

- e = local event at P_1
- a = sending of a message at P_1, i = reception of this message at P_2
- channels need not be FIFO : see j→g and k→f
- in each process, events are totally ordered (local clock)
- the “physical time” can be seen as given by vertical slices
  no one knows this physical time (we only know it exists… up to relativity!)



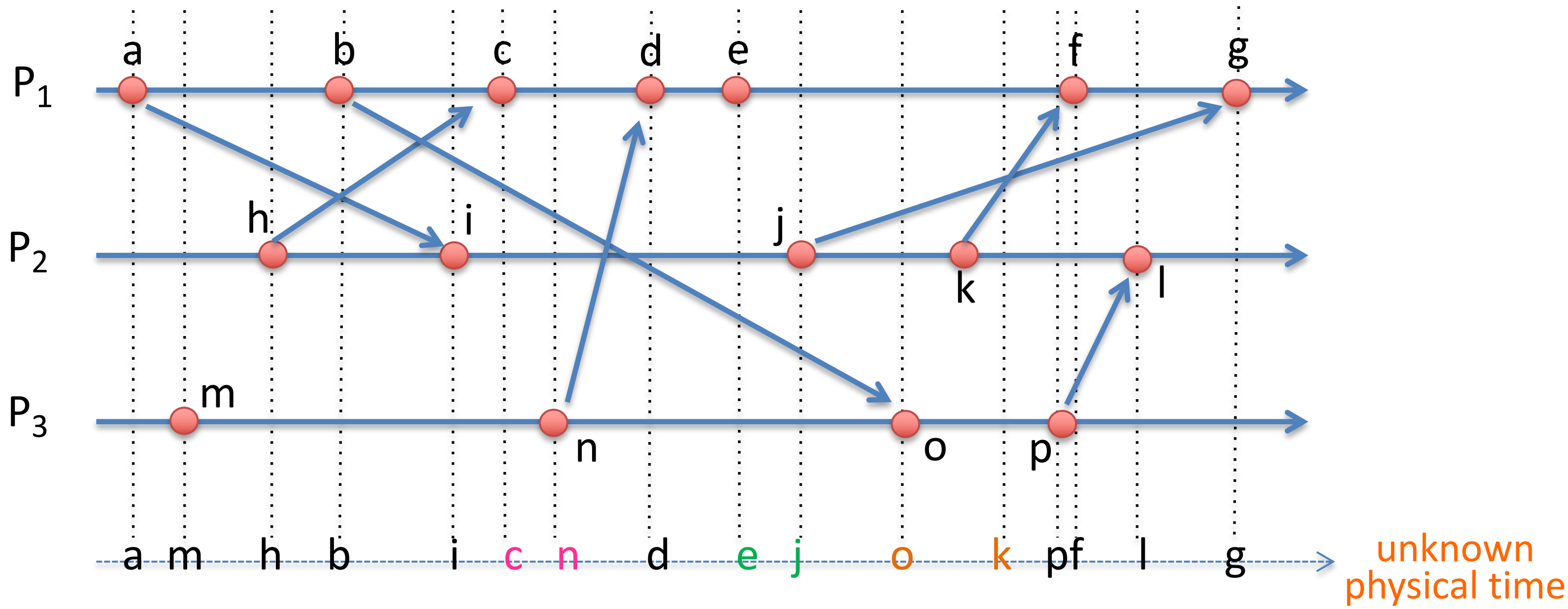
“a chronogram view”

- e = local event at P_1
- a = sending of a message at P_1, i = reception of this message at P_2
- channels need not be FIFO : see j→g and k→f
- in each process, events are totally ordered (local clock)
- the “physical time” can be seen as given by vertical slices
  no one knows this physical time (we only know it exists… up to relativity!)



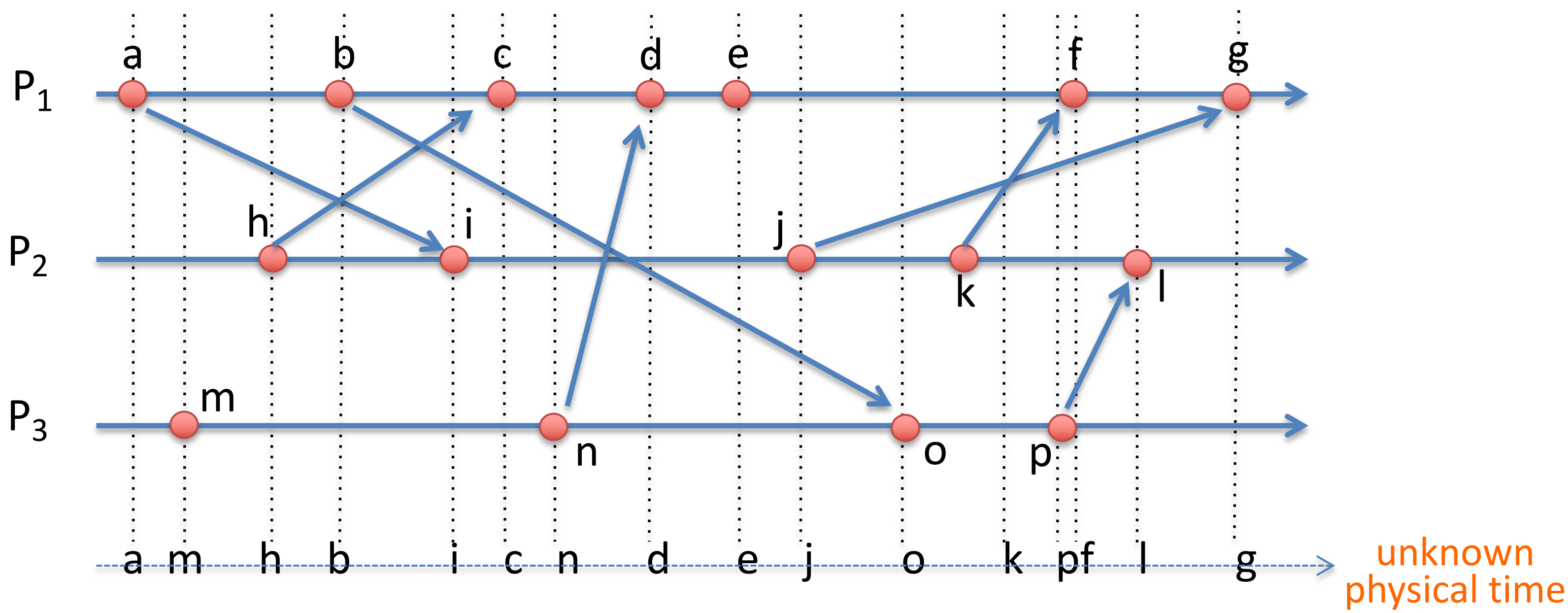
“a chronogram view”

- e = local event at P_1
- a = sending of a message at P_1, i = reception of this message at P_2
- channels need not be FIFO : see j→g and k→f
- in each process, events are totally ordered (local clock)
- the “physical time” can be seen as given by vertical slices
  no one knows this physical time (we only know it exists… up to relativity!)



“a chronogram view”

- events can slide on their axis, and preserve their ordering in processes, and the emission/reception ordering
- this yields another possible (total) ordering of events in physical time, resulting in the same final global state of the system, but going through different intermediate global states
- this advocates the modeling of a run as a **partial order of events**



Questions to address

- Q : how to (formally) define and handle a run as a partial order of events, rather than a sequence ?
- Q : the physical time is lost : can we instead track/compute this partial order ?
- Q : can we compute one (or all) possible total ordering(s) ?
- Q : what are the possible intermediate (global) states along a run ?

Event structures

Warning

Runs of distributed systems can be modeled in numerous (quite often uselessly complex) manners :

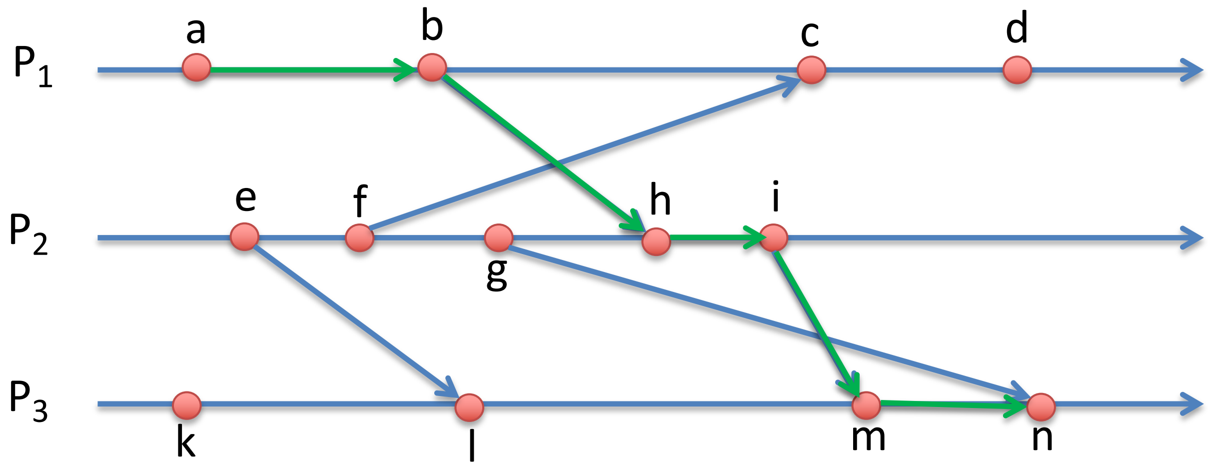
- one can start from communicating automata (Lynch)
- or more simply from processes with local actions, emissions and their matching receptions (Lamport, Fidge, Raynal)
- or even more simply from partially ordered events... (Mattern, Winskel, MacMillan, Nielsen, Engelfriet)
- … this goes with simple to more complex proofs for similar results !

(simple notion of) event structure

- it is a finite DAG (directed acyclic graph) $\mathcal{E} = (E, \rightarrow)$
- events are partitioned into n subsets (processes)

$$E = E_1 \uplus .. \uplus E_n$$

- events in each E_i form a path : total ordering due to local clock
- an event $e \in E_i$ has at most one direct successor/predecessor $e' \notin E_i$: models emission/reception of a message



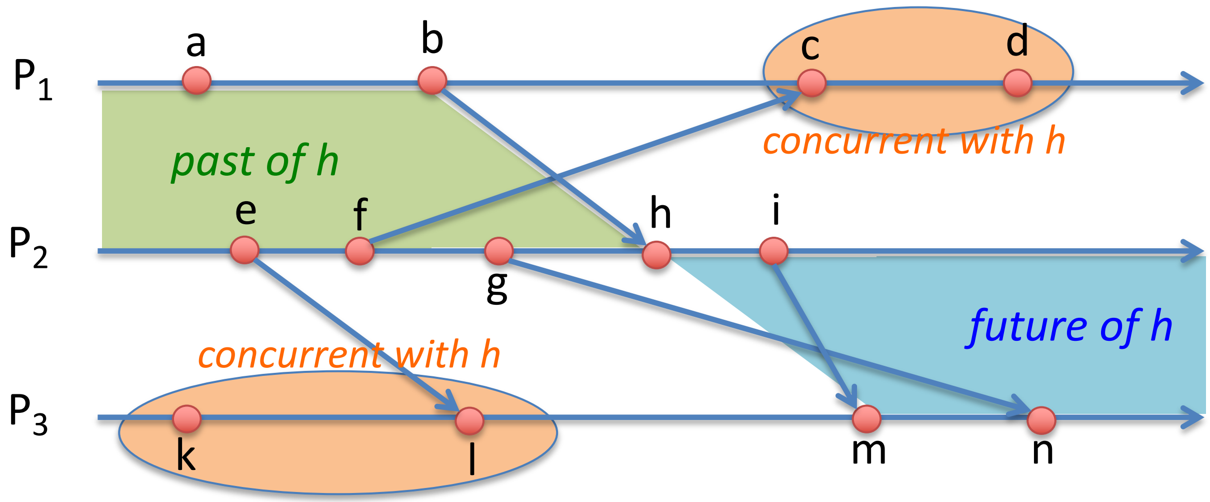
- partial order on events : $e \prec e'$ iff $e \rightarrow^* e'$ in the DAG,
  i.e. $\prec$ is the smallest partial order (= transitive+irreflexive) relation generated by $\rightarrow$

$$a \rightarrow b \rightarrow h \rightarrow i \rightarrow m \rightarrow n \quad \Rightarrow \quad a \prec n$$

- past of an event e = predecessors of e for $\prec$
- future of an event e = successors of e for $\prec$

- concurrency : $e \perp e'$ iff $e \not\prec e'$ and $e' \not\prec e$

$$a \perp k \qquad h \perp c \qquad b \perp l \qquad b \not\perp m \qquad c \perp m$$



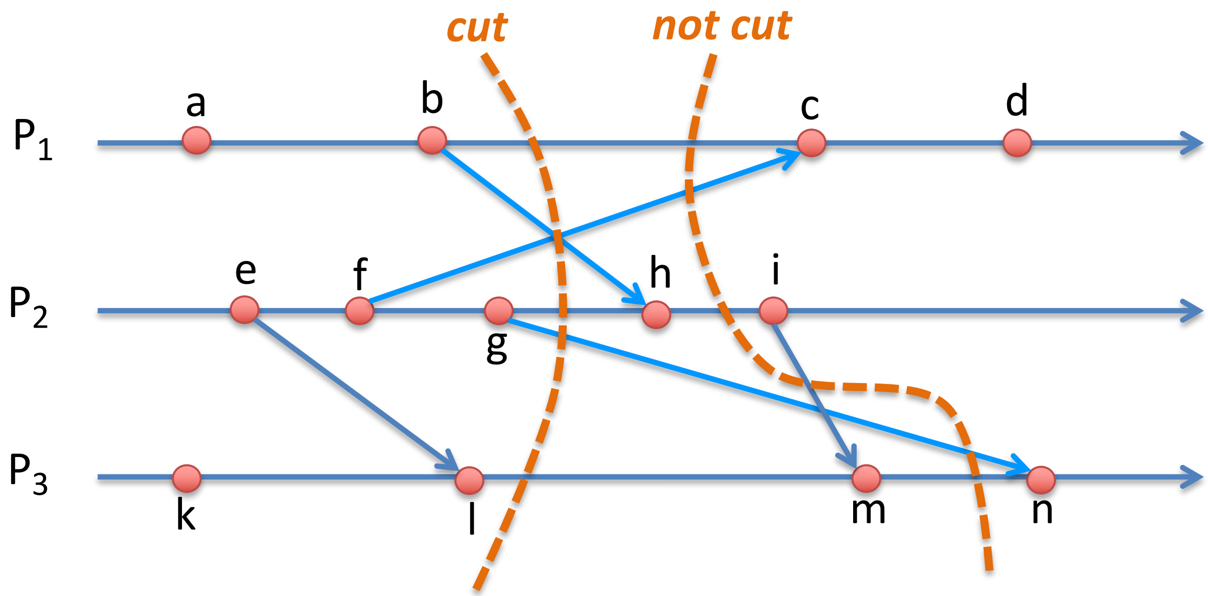
- partial order on events : $e \prec e'$ iff $e \rightarrow^* e'$ in the DAG,
  i.e. $\prec$ is the smallest partial order (= transitive+irreflexive) relation generated by $\rightarrow$

$$a \rightarrow b \rightarrow h \rightarrow i \rightarrow m \rightarrow n \quad \Rightarrow \quad a \prec n$$

- past of an event e = predecessors of e for $\prec$
- future of an event e = successors of e for $\prec$

- concurrency : $e \perp e' \quad \text{iff} \quad e \not\prec e' \text{ and } e' \not\prec e$

$$a \perp k \qquad h \perp c \qquad b \perp l \qquad b \not\perp m \qquad c \perp m$$



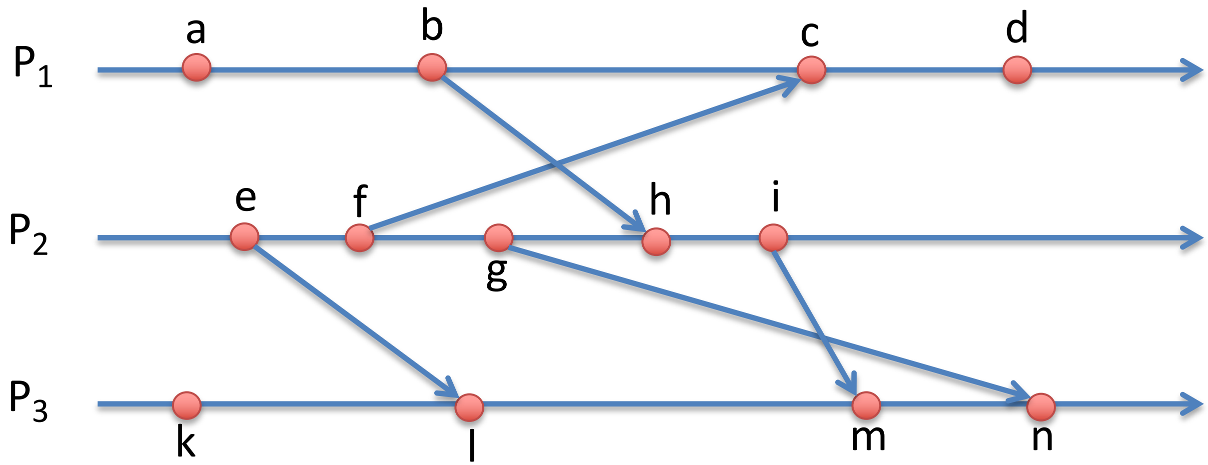
- A **cut** in $\mathcal{E} = (E, \to)$ is a subset $E' \subseteq E$ closed for the precedence relation $\prec$

$$\forall e, e' \in E, \quad e \in E' \wedge e' \prec e \quad \Rightarrow \quad e' \in E'$$

- Maximal events in a cut can be seen as a line/curve, cutting all threads, thus defining a past (E') and a future (E\E'). The line represents a possible "present."

- **Interpretation**: a cut identifies a possible global state of the distributed process, that could be characterized by the *current state of each process*, and the *messages already sent but not yet received* ("in flight" messages).

- Remark: it is generally not possible to have cuts with no pending messages, *i.e.* that do not separate emission from reception of a message.
Exercise: build an example.



- A linear extension of $\prec$ is a total order $<$ in E preserving $\prec$:

$$\forall e, e' \in E, \quad e \prec e' \quad \Rightarrow \quad e < e'$$

- Obtained by recursively adding arcs $e \rightarrow e'$ for some pair of concurrent events, $e \perp e'$, then completing $\prec$ by transitivity, until $\prec$ becomes a total order.

- "Thm": any linear extension < of $\prec$ is a possible execution order (in physical time) for the events present in the event structure $\mathcal{E} = (E, \rightarrow)$

  Proof: trivial, as messages transit times are unknown. [See also later.]

- Visually : how to build all such orderings ?
  - imagine events are pearls on a necklace, made of n threads/strings, one per process
  - pearls are free to move along each string, but cannot overpass one another…
  - … but edges $e \rightarrow e'$ (messages) must always point to the right (= to the future)

Remarks

- We will see later how to encode sets of partial orders in convenient data structures, in order to compute with them.

- In modern computer science, event structures are studied *per se.*
  They are simply event sets E (possibly infinite…) enriched with several relations like
  - precedence, or causality
  - conflict : different possible outcomes/futures
  - alternative causes/predecessors of events
  - asymmetric causality (*e* can appear concurrently or after *e'*, but not before)
  - etc.

Logical clock

Operating Systems — R. Stockton Gaines, Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

Historically

- introduced by Lamport in '78
- was one of the contributions motivating the Turing award
- easy & pleasant to read, applications described, but a little frustrating on formalization and proofs.
  Read it !

Objective

- build one possible total ordering of events, by attaching a logical time to them
- do this with a distributed asynchronous algorithm

Objective:

- tag every event *e* with a logical clock value *C(e)*, taken in some totally ordered set
- these ticks should reflect one linear extension of $\prec$ in run $\mathcal{E} = (E, \rightarrow)$

$$\forall e, e' \in E, \quad e \prec e' \quad \Rightarrow \quad C(e) < C(e')$$

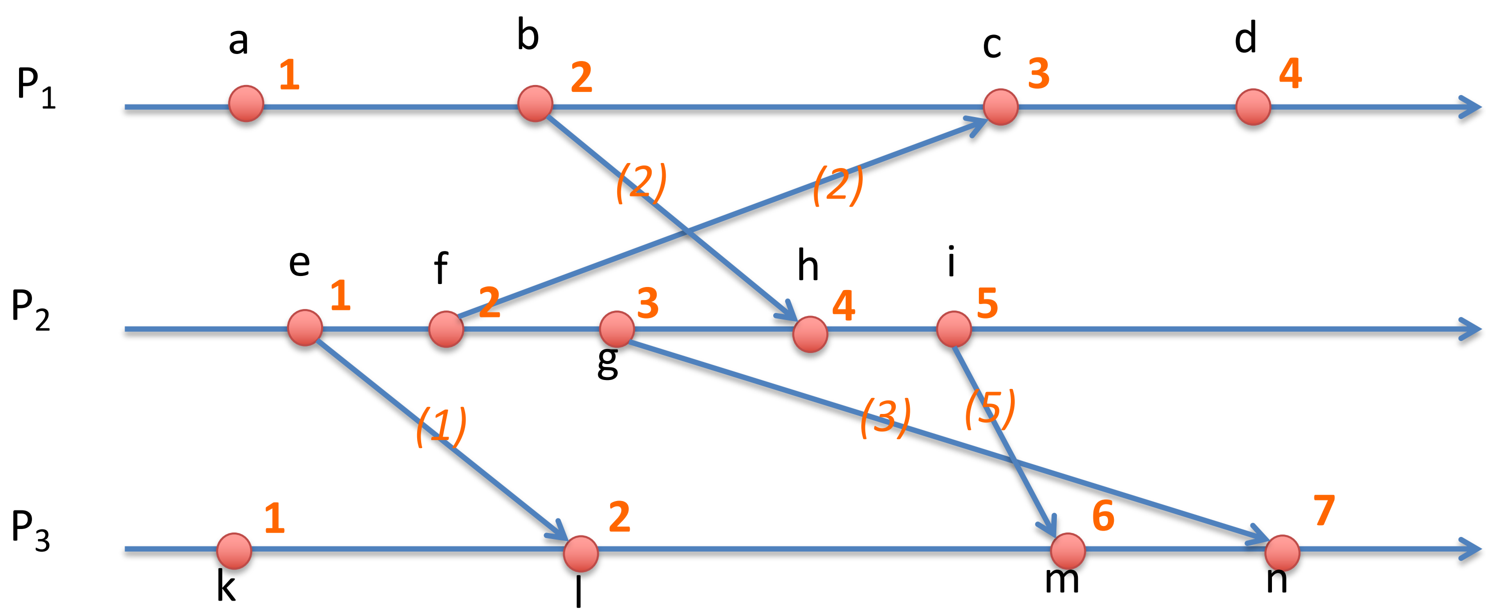
- notice that it is sufficient to guarantee only

$$\forall e, e' \in E, \quad e \rightarrow e' \quad \Rightarrow \quad C(e) < C(e')$$

  and to make sure that C defines a total order.
- we want compute these tags with a distributed algorithm

Algorithm:

- if $e \in E_i$ is a <u>new event</u> in process P_i
  - if $\exists e' \in E_i,\ e' \rightarrow e$ then $C(e) = C(e') + 1$
  - otherwise $C(e) = 1$
- if $e \in E_i$ is the <u>sending</u> of some message *m* from P_i to P_j
  - send $C(m) = C(e)$ with message *m* *(piggybacking)*
- if $e \in E_i$ is the <u>reception</u> of a message m tagged by $C(m)$
  - make correction $C(e) := max(C(e), C(m) + 1)$



Algorithm:

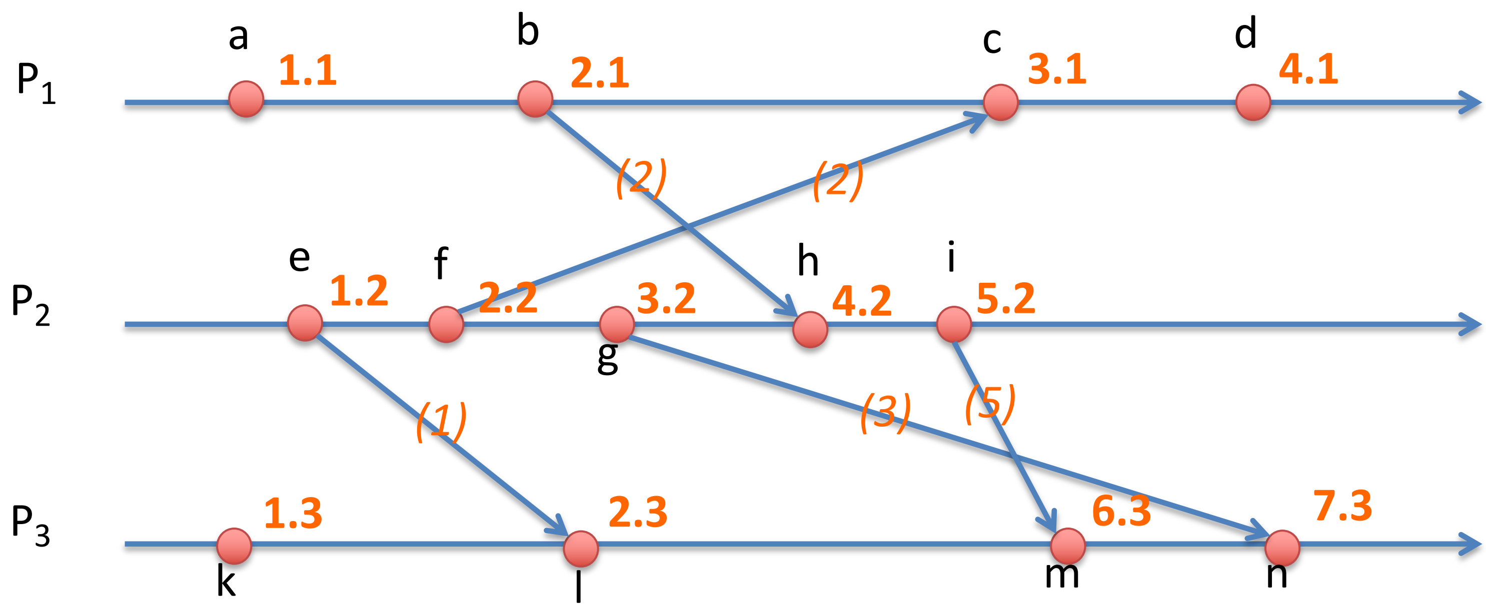
- if $e \in E_i$ is a <u>new event</u> in process P_i
  - if $\exists e' \in E_i,\ e' \to e$ then $C(e) = C(e') + 1$
  - otherwise $C(e) = 1$
- if $e \in E_i$ is the <u>sending</u> of some message *m* from P_i to P_j
  - send $C(m) = C(e)$ with message *m* *(piggybacking)*
- if $e \in E_i$ is the <u>reception</u> of a message m tagged by $C(m)$
  - make correction $C(e) := max(C(e), C(m) + 1)$

Properties

- clearly ensures $\forall e, e' \in E, \quad e \to e' \quad \Rightarrow \quad C(e) < C(e')$
- but events may not be totally ordered : concurrent events could have the same tag
- a total order is obtained by appending index i to $C(e)$ for $e \in E_i$
  the total order is the lexicographic order on pairs $(C(e), i)$
- each process can order its received messages in a unique manner
  - consistent with what all other processes do
  - and consistent with the causality of events in the run
  - however, this might not be the true order of message production in physical time… …which anyway is lost forever !

Algorithm:

- if $e \in E_i$ is a <u>new event</u> in process P_i
  - if $\exists e' \in E_i,\ e' \rightarrow e$ then $C(e) = C(e') + 1$
  - otherwise $C(e) = 1$
- if $e \in E_i$ is the <u>sending</u> of some message *m* from P_i to P_j
  - send $C(m) = C(e)$ with message *m* *(piggybacking)*
- if $e \in E_i$ is the <u>reception</u> of a message m tagged by $C(m)$
  - make correction $C(e) := max(C(e), C(m) + 1)$



a e k b f l c g d h i m n

Applications

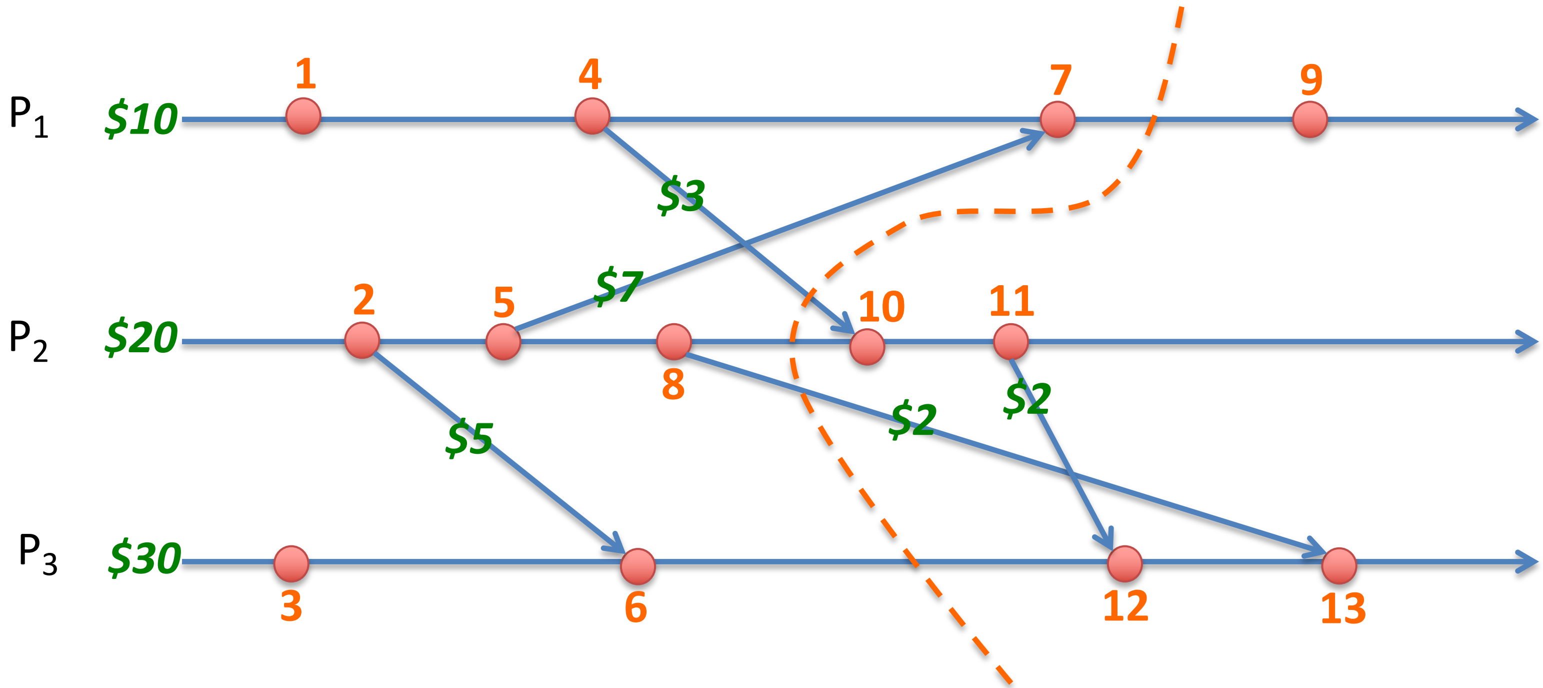
- Shared objects/states
- Mutual exclusion (by broadcasting resource requests : read details in Lamport's paper)
- **Banking problem**
  - determine the total amount of money circulating among a set of actors (banks)
  - local state = their current balance
  - messages = transactions (money sent)

Principle

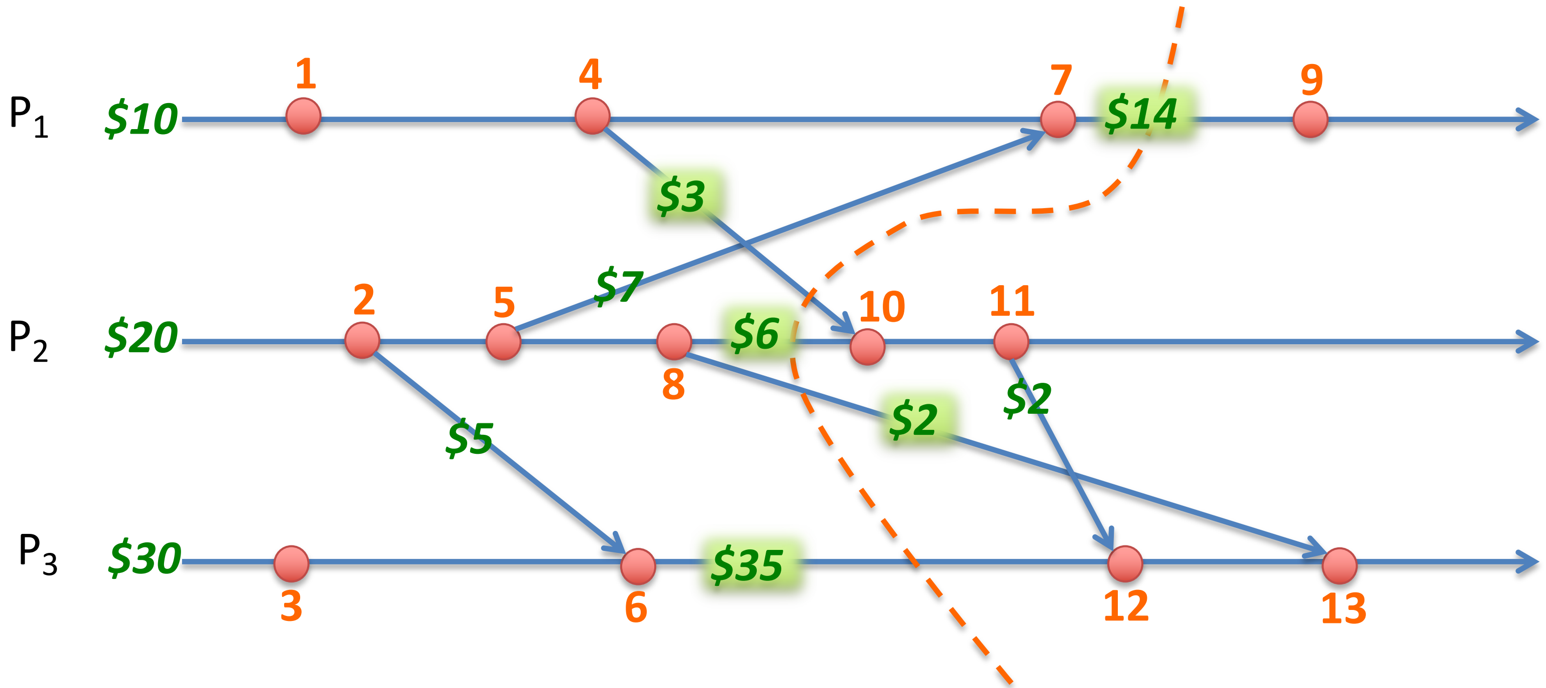
- tag events and messages with a logical time
- assume all messages arrive, and message flows never stop
- decide some logical time slice t at which counting takes place

then

- all processes wait until they have an event greater than time t
- collect the balance of each bank after the last event preceding time t
- determine the amount of money "in flight" at time t between all pairs P_i and P_j (i.e. sent by P_i to P_j, but not yet received by P_j)
- easy :
  - P_i knows how much it sent to P_j before time t
  - P_j knows how much it received from P_i before time t



P1
$10
P2
$20
P3
$30
1
4
7
9
2
5
10
11
8
3
6
12
13
$3
$7
$5
$2
$2



Before time 9

- P_2 sent \$5+\$2=\$7 to P_3
- P_3 received \$5 from P_2
- \$2 are in flight

Applications

- Definition of a snapshot (checkpoint), i.e. capture of a consistent global state from where a (failing) distributed computation could restart
- **General idea** : at some logical time t, all processes store
  - their current state, and
  - the content of messages that have been sent and are not yet received
- similar to the banking problem, where "in flight" messages must also be identified and stored.
- Specific case of FIFO channels : see the Chandy-Lamport algorithm ('85), that uses a marker to separate past messages from new ones in a channel.

Distributed Snapshots: Determining Global States of Distributed Systems

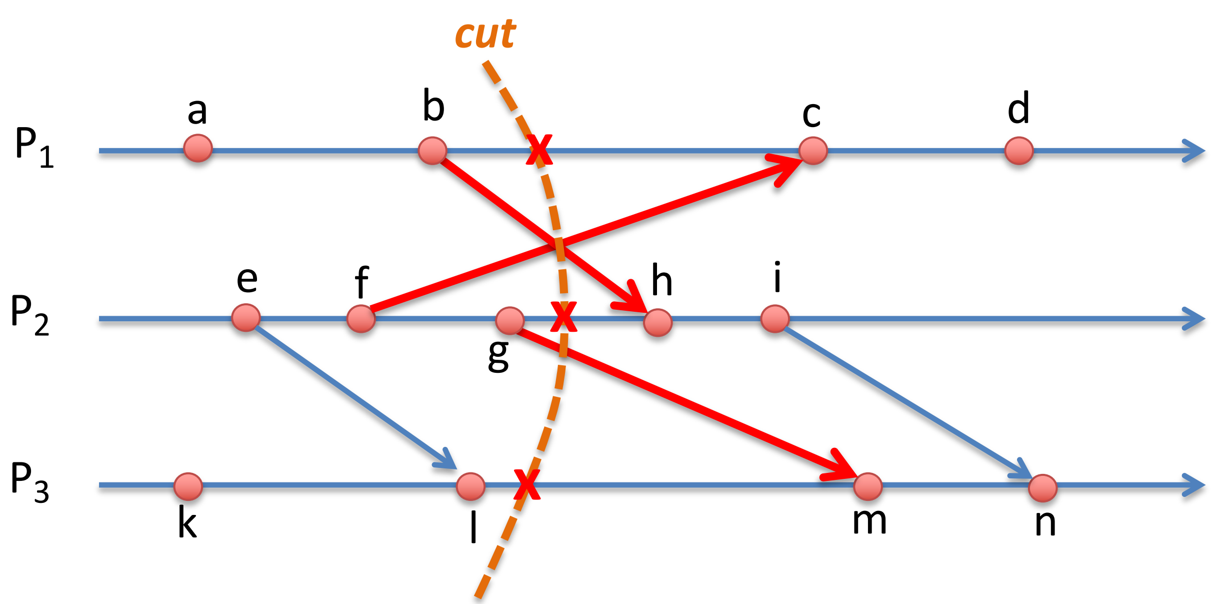
K. MANI CHANDY
University of Texas at Austin
and
LESLIE LAMPORT
Stanford Research Institute

This paper presents an algorithm by which a process in a distributed system determines a global state of the system during a computation. Many problems in distributed systems can be cast in terms of the problem of detecting global states. For instance, the global state detection algorithm helps to solve an important class of problems: stable property detection. A stable property is one that persists: once a stable property becomes true it remains true thereafter. Examples of stable properties are "computation has terminated," "the system is deadlocked" and "all tokens in a token ring have disappeared." The stable property detection problem is that of devising algorithms to detect a given stable property. Global state detection can also be used for checkpointing.

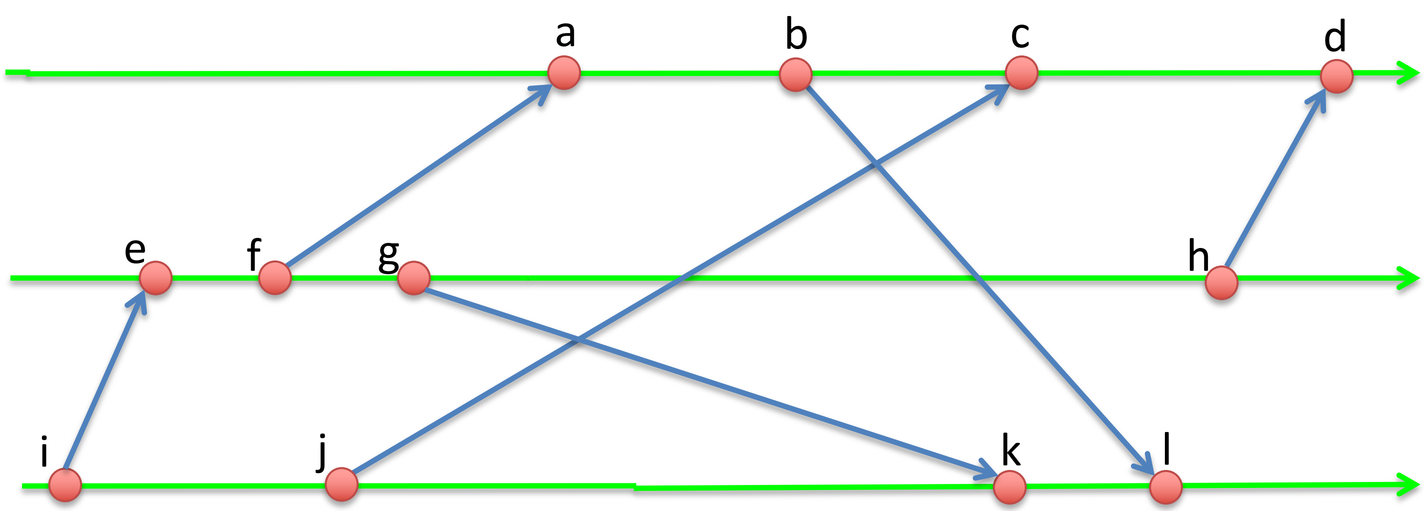
Worth reading :
important algorithm
+ historical interest.
Paper driven by examples,
not a formal presentation.

Chandy-Lamport snapshot

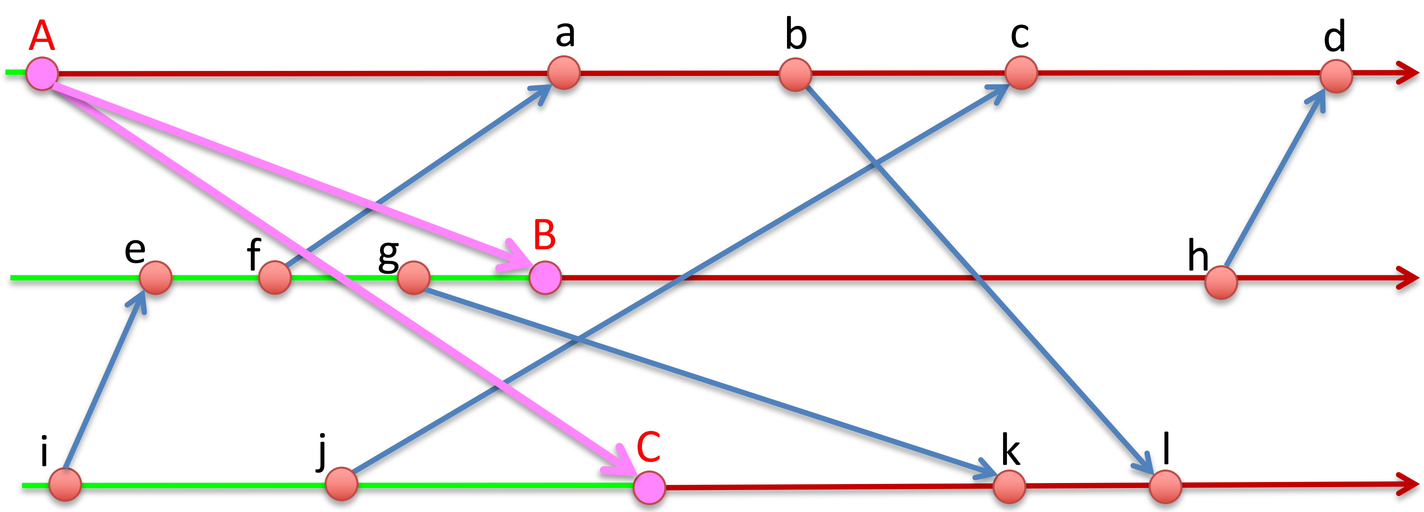
- **Objective:** determine a *consistent global state*, that is
  - the current state (x) of each process at a *consistent cut*
  - sequence of in-flight messages (→) in each channel (sent before cut, not yet received)
- Defines a state from which computations could restart in case of crash
- Could be a state that was never crossed by the current execution



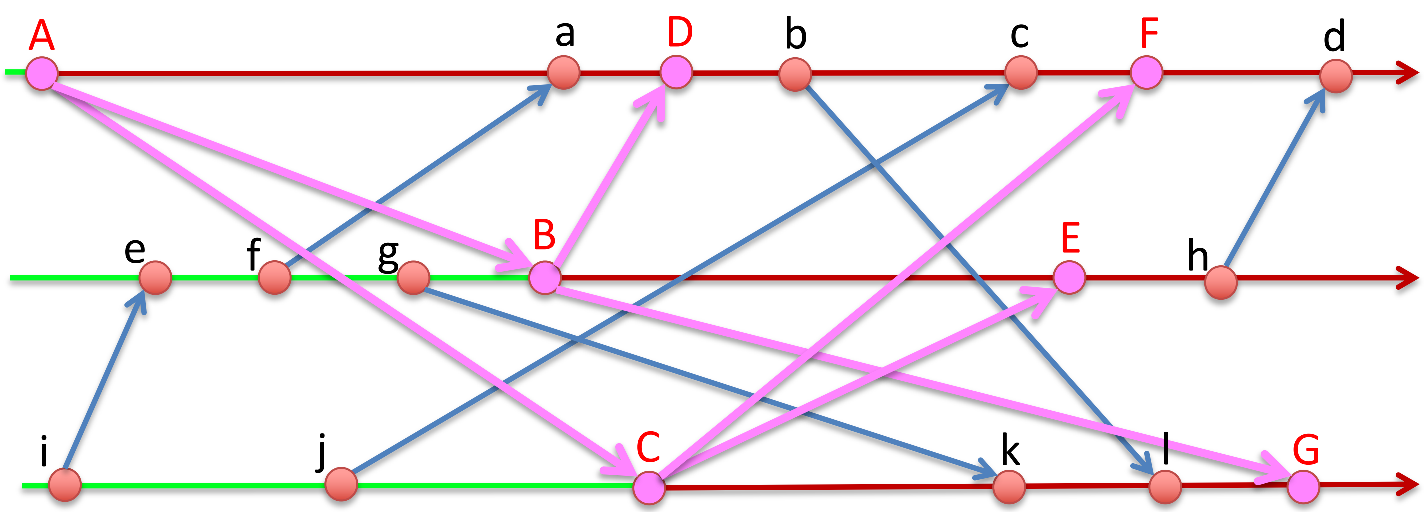
- **Assumptions**
  - Unidirectional FIFO lossless channels
  - A communication path (possibly multi-hops) exists between any pair of processes
  - One process initiates the snapshot
  - Snapshot is stored in a distributed manner
- Principle:
  - Flooding of a “cut” message from the initiator; this defines past and future
  - Flushing of channel messages, using the FIFO assumption



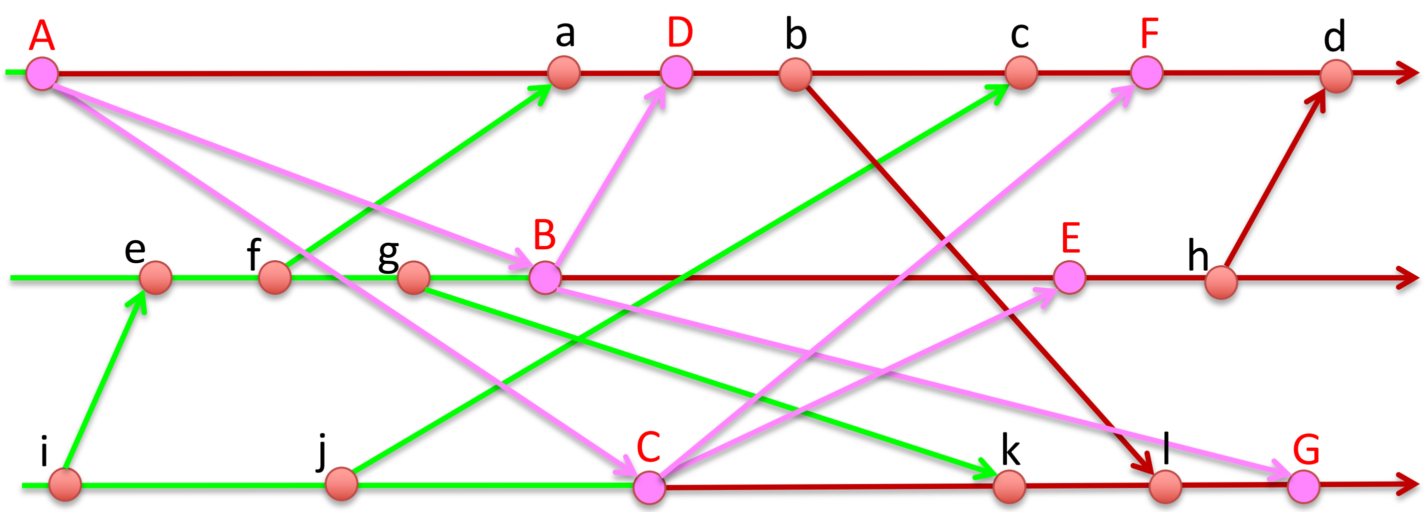
- **Assumptions**
  - Unidirectional FIFO lossless channels
  - A communication path (possibly multi-hops) exists between any pair of processes
  - One process initiates the snapshot
  - Snapshot is stored in a distributed manner
- Principle:
  - Flooding of a "cut" message from the initiator; this defines past and future
  - Flushing of channel messages, using the FIFO assumption



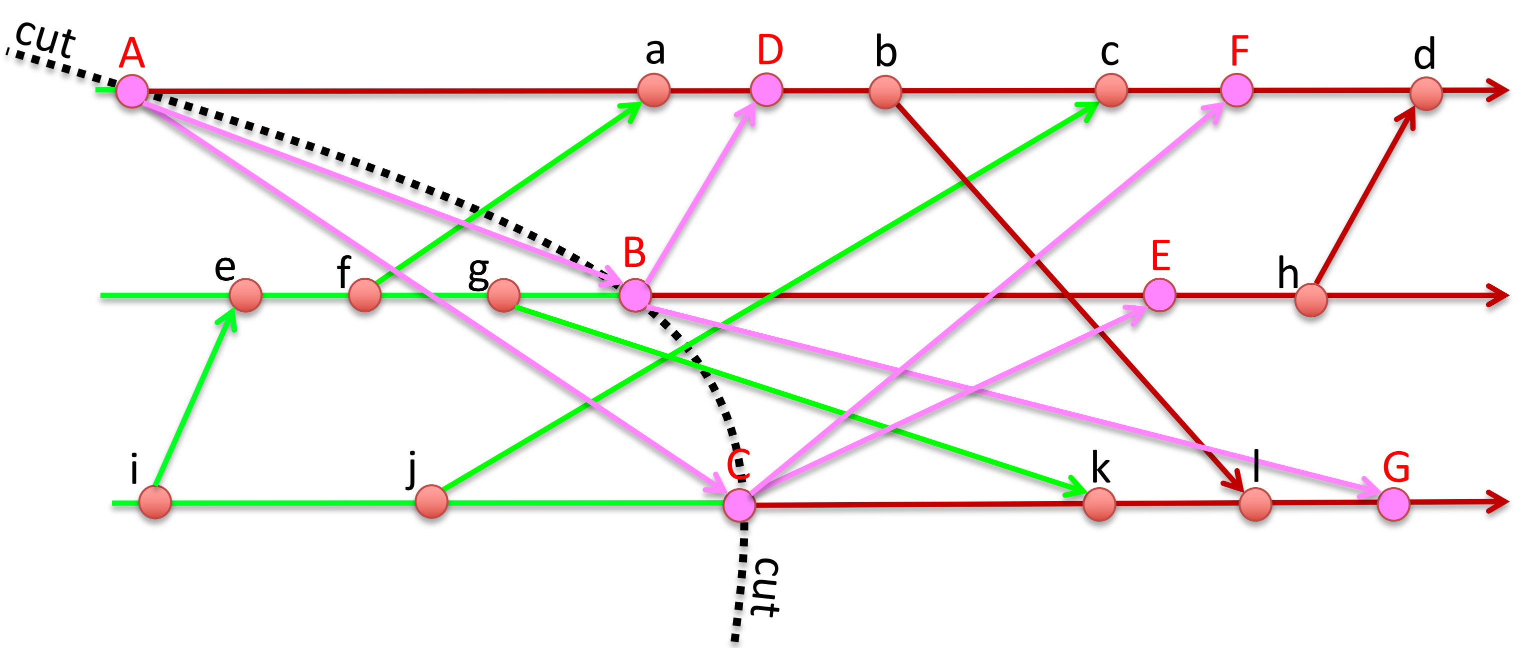
- **Assumptions**
  - Unidirectional FIFO lossless channels
  - A communication path (possibly multi-hops) exists between any pair of processes
  - One process initiates the snapshot
  - Snapshot is stored in a distributed manner
- Principle:
  - Flooding of a “cut” message from the initiator; this defines past and future
  - Flushing of channel messages, using the FIFO assumption



- **Assumptions**
  - Unidirectional FIFO lossless channels
  - A communication path (possibly multi-hops) exists between any pair of processes
  - One process initiates the snapshot
  - Snapshot is stored in a distributed manner
- Principle:
  - Flooding of a “cut” message from the initiator; this defines past and future
  - Flushing of channel messages, using the FIFO assumption



- **Assumptions**
  - Unidirectional FIFO lossless channels
  - A communication path (possibly multi-hops) exists between any pair of processes
  - One process initiates the snapshot
  - Snapshot is stored in a distributed manner
- Principle:
  - Flooding of a “cut” message from the initiator; this defines past and future
  - Flushing of channel messages, using the FIFO assumption



Chandi-Lamport algorithm

- Initiator P
  - P turns from green to red, stores its current state
    all subsequent messages from P are red
  - P sends a "cut" message to each neighbor Q (first red message in channel P→Q)
- FIFO assumption: in each channel
  - messages preceding "cut" are called green
  - messages following "cut" are called red
  - and similarly for processes: they change color when receiving "cut"

- Green process Q receives "cut" message from P
  - This is the first "cut" message received by Q
  - Q turns from green to red, stores its current state,
    all subsequent messages from Q are red
  - Q sends a "cut" message to each neighbor R (first red message in channel Q→R)
  - Q *starts recording* green messages on *each* incoming channel S→Q,
    preserving their ordering in each channel

- Red process Q receives a "cut" message from P
  - This is ***not*** the first "cut" message received by Q
  - Q *stops recording* green messages arriving on channel P→Q

Invariants + monotony (for proof of convergence)

- Messages in channels are green then red (when the first "cut" is sent) *[FIFO]*
- All "cut " messages are causally related to the one of the initiator
- Each process ultimately receives a "cut" from each other process
- In-flight messages in channel P→Q are exactly those that
  - follow the event *"Q turns red"*
  - precede the event *"Q receives "cut" from P"*

Questions/homework

1. Make the convergence + correctness proof rigorous.
2. Prove that the FIFO assumption is necessary.
3. Why is it a distributed storage of a global state ?
4. Can one gather the global state at the initiator of the snapshot ?
5. Prove that the snapshot builds a global state that could possibly have not been crossed by the actual (physical time) execution.
6. How can one have several possible initiators ?
7. How to restart computations from a snapshot ?
8. How to release the FIFO assumption ?

Vector clock

Timestamps in Message-Passing Systems That Preserve the Partial Ordering

Colin J. Fidge
Department of Computer Science, Australian National University, Canberra, ACT.

ABSTRACT

Timestamping is a common method of totally ordering events in concurrent programs. However, for applications requiring access to the global state, a total ordering is inappropriate. This paper presents algorithms for timestamping events in both synchronous and asynchronous message-passing programs that allow for access to the partial ordering inherent in a parallel system. The algorithms do not change the communications graph or require a central timestamp issuing authority.

Historically

- introduced independently by Fidge (Aust.) and Mattern (Germ.) in '88
- Fidge uses a slightly different construction, and is less formalized
- Mattern is a bit more formalized, and uses the notion of event structure.
- *Read Mattern !*

Vector clock

Virtual Time and Global States of Distributed Systems *

Friedemann Mattern †

Department of Computer Science, University of Kaiserslautem
D 6750 Kaiserslautern, Germany

Abstract

A distributed system can be characterized by the fact that the global state is distributed and that a common time base does not exist. However, the notion of time is an important concept in every day life of our decentralized "real world" and helps to solve problems like getting a consistent population census or determining the potential causality between events. We argue that a linearly ordered structure of time is not (always) adequate for distributed systems and propose a generalized non-standard model of time which consists of vectors of clocks. These clock-vectors are partially ordered and form a lattice. By using timestamps and a simple clock update mechanism the structure of causality is represented in an isomorphic way. The new model of time has a close analogy to Minkowski's relativistic space-time and leads among others to an interesting characterization of the global state problem. Finally, we present a new algorithm to compute a consistent global snapshot of a distributed system where messages may be received out of order.

Objective

- recover all possible consistent total orderings of events in a distributed run
- track the causality relations among events of a distributed system, with a distributed algorithm

A drawback of Lamport's logical time

- not all total orderings of events are accessible
- logical time is totally ordered : how to capture only causality ?

$$\forall e, e' \in E, \quad e \prec e' \quad \Rightarrow \quad C(e) < C(e')$$

- one would like to have :

$$\forall e, e' \in E, \quad e \prec e' \quad \Longleftrightarrow \quad VC(e) \prec VC(e')$$

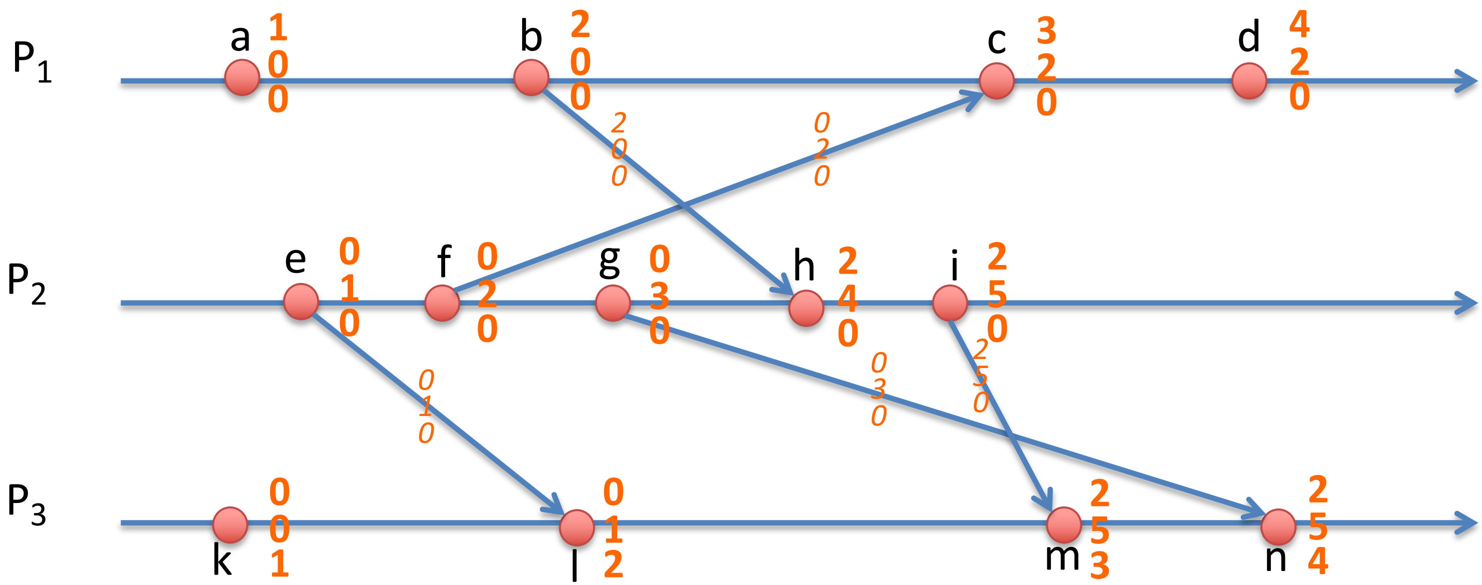
[see later for a definition of $\prec$]

Fidge-Mattern's idea

- one local clock C_i per process P_i
- clock ticks are tuples/vectors $VC(e) = (C_1(e), ..., C_n(e)) \in \mathbb{N}^n$

Algorithm:

- if $e \in E_i$ is a new event in process P_i
  - if $\exists e' \in E_i, \; e' \rightarrow e$ then $VC(e) = (C_1(e'), ..., C_i(e') + 1, ..., C_n(e'))$
  - otherwise $VC(e) = (0, ..., 0, 1, 0, ..., 0)$ with 1 on the i^{th} coordinate
- if $e \in E_i$ is the sending of some message *m* from P_i to P_j
  - send $VC(m) = VC(e)$ with message *m (piggybacking)*
- if $e \in E_i$ is the reception of a message *m* tagged by $VC(m)$
  - make correction $\forall k, \; C_k(e) = max(C_k(e), C_k(m))$ in $VC(e)$



P1
P2
P3
a 1 0 0
b 2 0 0
c 3 2 0
d 4 2 0
e 0 1 0
f 0 2 0
g 0 3 0
h 2 4 0
i 2 5 0
k 0 0 1
l 0 1 2
m 2 5 3
n 2 5 4
2 0 0
0 2 0
0 1 0
0 3 0
2 5 0

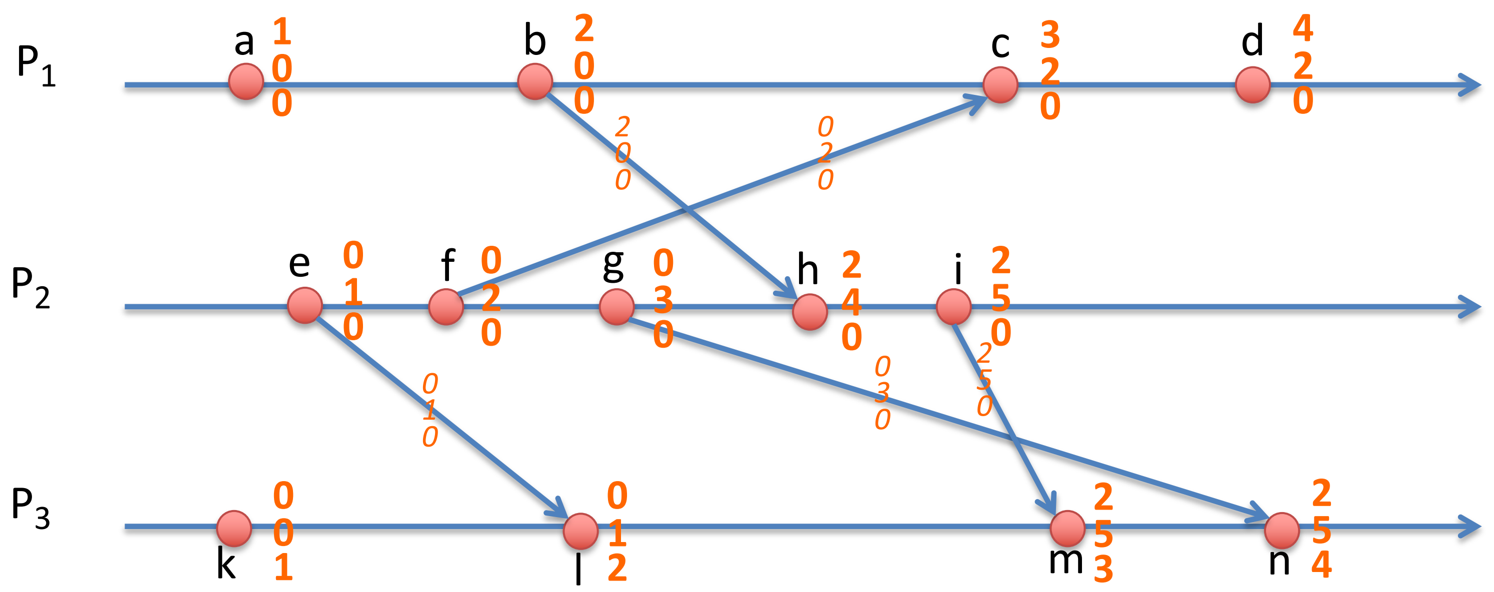
Thm: "The event structure $\mathcal{E} = (E, \rightarrow)$ and Mattern's vector clock values on it generate isomorphic trellises of event subsets." [Mattern'88]

$$\forall e, e' \in E, \quad e \prec e' \quad \Longleftrightarrow \quad VC(e) \prec VC(e')$$

Proof

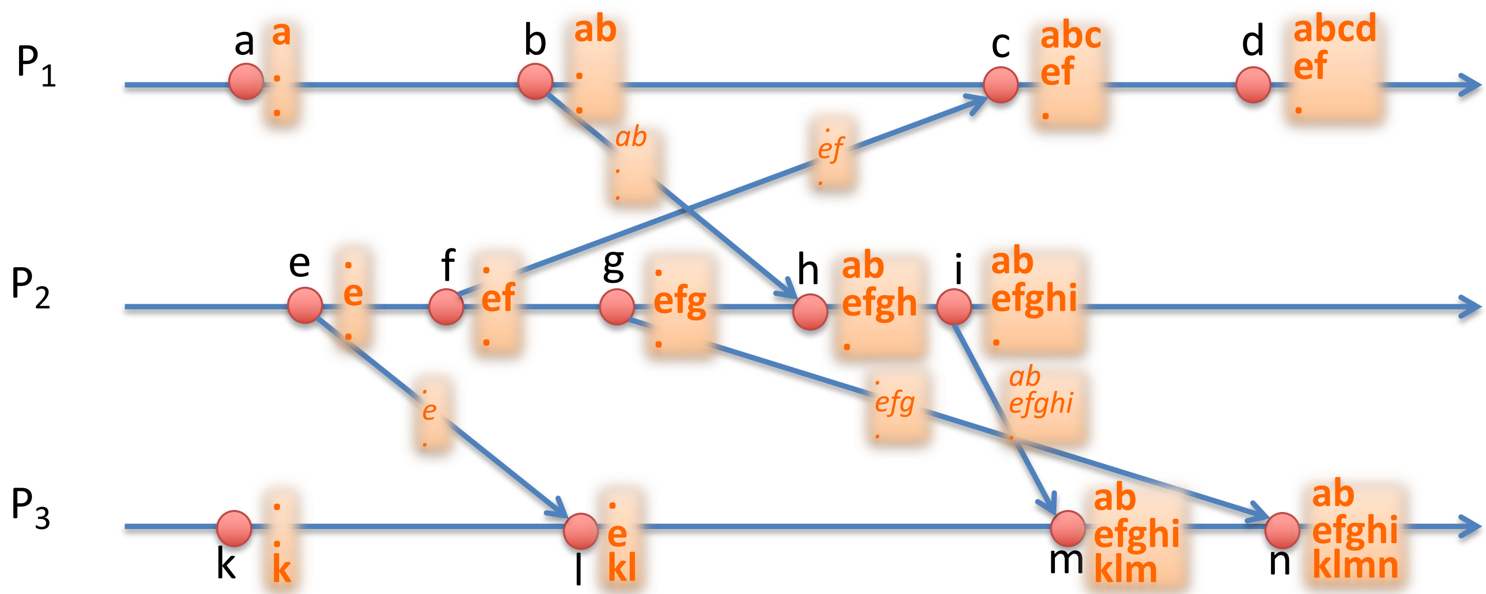
- one defines $VC(e) \preceq VC(e')$ by $\forall i,\ C_i(e) \leq C_i(e')$
- and $VC(e) \prec VC(e')$ by $VC(e) \preceq VC(e') \ \wedge\ VC(e) \neq VC(e')$

- the theorem expresses that the <u>partial order</u> due to the DAG $\mathcal{E} = (E, \rightarrow)$ and the one derived from vector clock values are identical
- proof of $\Longrightarrow$ is the same as for Lamport's logical clock (by construction)
- proof of $\Longleftarrow$ is more involved *(see hint below)*



P1
P2
P3
a
1 0 0
b
2 0 0
c
3 2 0
d
4 2 0
2 0 0
0 2 0
e
0 1 0
f
0 2 0
g
0 3 0
h
2 4 0
i
2 5 0
0 1 0
0 3 0
2 5 0
k
0 0 1
l
0 1 2
m
2 5 3
n
2 5 4

An equivalent version of Mattern's vector clock
(one to one correspondence)



The text recoding of the Vector Clock captures exactly all events that are causal predecessors of some given event in the DAG.
[Lamport's clock was placing more predecessors in the past of some event.]

Applications

- distributed debugging : to keep track of the causality of events
- snapshots (storage of consistent global states), when channels are not FIFO (Chandy-Lamport not applicable)

Take home messages

runs of distributed systems

- are partial orders (causal relations) of events
- better encoded as event structures
- this partial order can be tracked by distributed algorithms
- this is the starting point of more elaborate functions (snapshots, mutual exclusion, detection of stable properties,…)

next time

- processes as automata
- models for distributed systems
- true concurrency semantics to capture causality/parallelism