African University of Science and Technology

Time, clocks and ordering of events

Pr. Ousmane THIARE

http://www.ousmanethiare.com

August 18, 2014

Outline

Distributed Execution

Definition (Distributed algorithm)

A **distributed algorithm** is a collection of distributed automata, one per process

Definition (Distributed execution)

The **execution** of a distributed algorithm is a sequence of **events** executed by the processes

- **Partial execution**: a finite sequence of events
- **Infinite execution**: a infinite sequence of events

Possible events

- *send(m,p)*: sends a message m to process p
- *receive(m)*: receives a message m
- *local events* that change the local state

Histories

Definition (Local History)

The local history of process p_i is a (possibly infinite) sequence of events $h_i = e_i^0 e_i^1 e_i^2 \cdots e_i^{m_i}$ (canonical enumeration)

Definition (Partial history)

The partial history up to event e_i^k is denoted h_i^k and is given by the prefix of the first k events of h_i

Histories

- Local histories do not specify any relative timing between events belonging to different processes.
- We need a notion of ordering between events, that could help us in deciding whether:
  - one event occurs before another
  - they are actually concurrent

Outline

Happen-Before

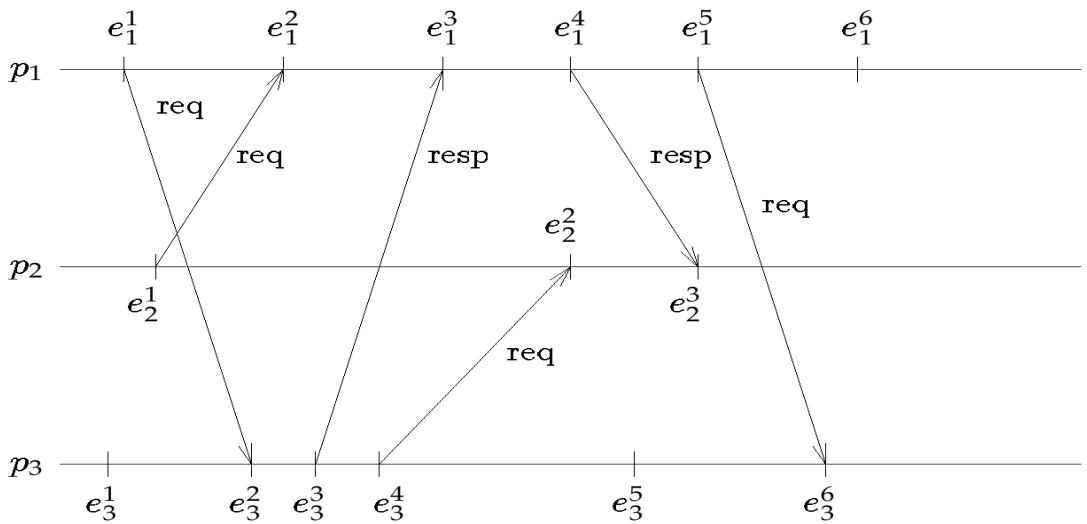
Definition (Happen-before)

We say that an event e happens-before an event e', and write $e \longrightarrow e'$, if one of the following three cases is true:

- $\exists p_i \in \prod : e = e_i^r, e' = e_i^s, r < s$
  (if e and e' are executed by the same process, e before e')
- e=send(m)$\wedge$ e'=receive(m)
  (if e is the send event of a message m and e' is the corresponding receive event)
- $\exists e'' : e \longrightarrow e'' \longrightarrow e'$
  (in other words, $\longrightarrow$ is transitive)

Happen-Before

Space-Time Diagram of a Distributed Computation



Happen-Before

Meaning of Happen-Before)

If $e \longrightarrow e'$, this means that we can find a series of events $e^1 e^2 e^3 \cdots e^n$, where $e^1 = e$ and $e^n = e'$, such that for each pair of consecutive events e^i and e^{i+1}:

- e^i and e^{i+1} are executed on the same process, in this order
- $e^i = send(m)$ and $e^{i+1} = receive(m)$

Notes:

- happen-before captures the concept of potential causal ordering
- happen-before captures a flow of data between two events.
- Two events e, e' that are not related by the happen-before relation ($e \not\longrightarrow e' \wedge e' \not\longrightarrow e$) are concurrent, and we write $e||e'$.

Outline

Global States

Definition (Local state)

- The local state of process p_i after the execution of event e_i^k is denoted σ_i^k
- The local state contains all data items accessible by that process
- Local state is completely private to the process
- σ_i^0 is the initial state of process p_i

Definition (Global state)

The global state of a distributed computation is an n-tuple of local states $\Sigma = (\sigma_1, \cdots, \sigma_n)$, one for each process.

Cut

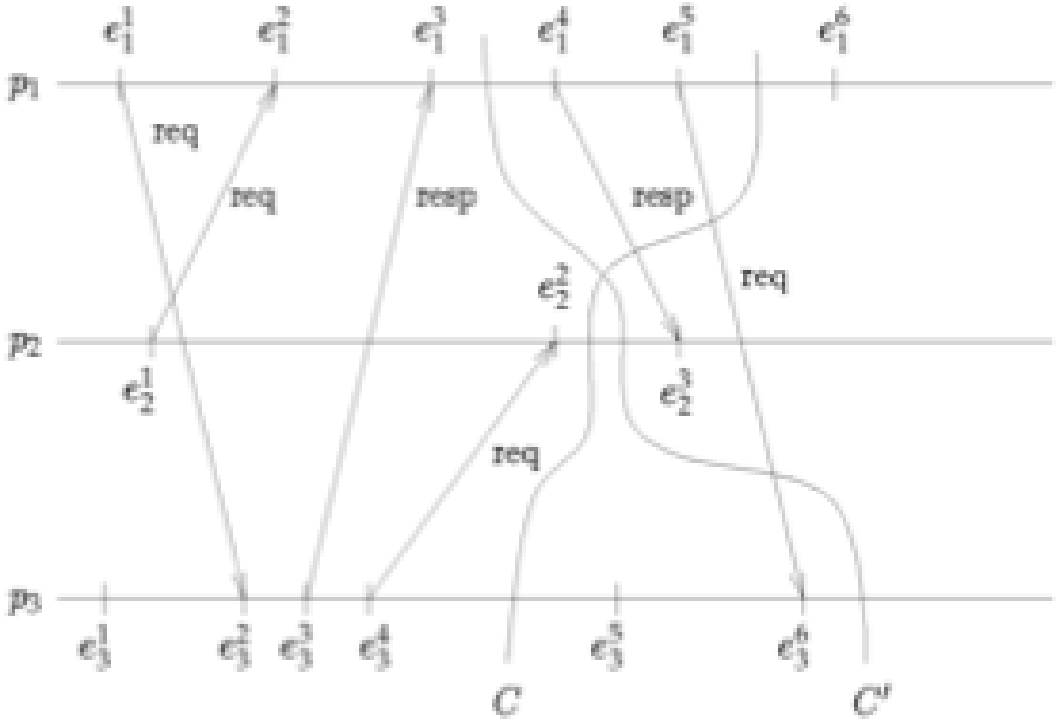
Definition (Cut)

A cut of a distributed computation is the union of n partial histories, one for each process:

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \cdots \cup h_n^{c_n}$$

- A cut may be described by a tuple $(c_1, c_2, \cdots, c_n)$, identifying the frontier of the cut, i.e. the set of last events, one per process.
- Each cut $(c_1, c_2, \cdots, c_n)$ has a corresponding global state $(\sigma_1^{c_1} \cup \sigma_2^{c_2} \cup \cdots \cup \sigma_n^{c_n})$.

Cut


e_1^1
e_1^2
e_1^3
e_1^4
e_1^5
e_1^6
p_1
req
req
resp
resp
req
e_2^2
p_2
e_2^1
e_2^3
req
p_3
e_3^1
e_3^2
e_3^3
e_3^4
e_3^5
e_3^6
C
C'

Consistent Cut

Consider cuts C' and C in the previous figure.

- Is it possible that cut C correspond to a "real" state in the execution of a distributed algorithm?
- Is it possible that cut C' correspond to a "real" state in the execution of a distributed algorithm?

Consistent Cut

Consider cuts C' and C in the previous figure.

Definition (Consistent cut)

A cut C is consistent is for all events e and e',

$$(e \in C) \wedge (e' \longrightarrow e) \Rightarrow e' \in C$$

Definition (Consistent global state)

A global state is consistent if the corresponding cut is consistent.

In other words

- A consistent cut is left-closed with regard to (w.r.t.) the happen-before relation
- All messages that have been received must have been sent before

Consistent Cut

Consider cuts C' and C in the previous figure.

- In the previous figures, C is consistent and C' is not.
- In the space-time diagram, a cut C is consistent if all the arrows start on the left of the cut and finish on the right of the cut.
- Consistent cuts represent the concept of scalar time in distributed computation: it is possible to distinguish between a "before" and an "after".
- Predicates can be evaluated in consistent cuts, because they correspond to potential global states that could have taken place during an execution.

Outline

Introduction

Consider cuts C' and C in the previous figure.

Definition (Global Predicate Evaluation)

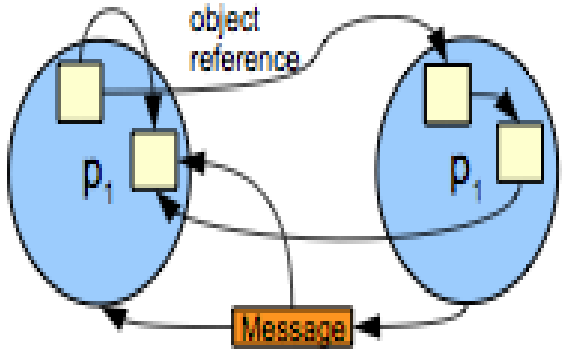
The problem of detecting whether the global state of a distributed system satisfies some predicate Φ.

Motivation

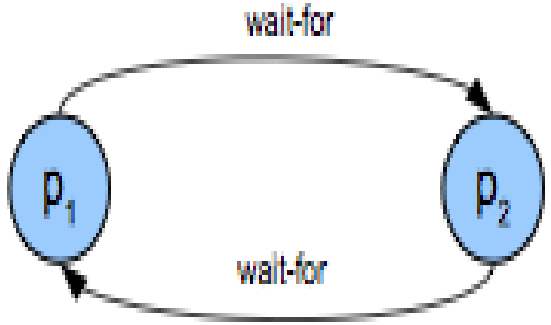
- Many important distributed problems require to react when when the global state of the system satisfies a given condition.
  - Monitoring: Notify an administrator in case of failures
  - Debugging: Verify whether an invariant is respected or not
  - Deadlock detection: can the computation continue?
  - Garbage collection: like Java, but distributed
- Thus, the ability to construct a global state and evaluate a predicate over it is a core problem in distributed computing.

Examples

Garbage collection



Deadlock

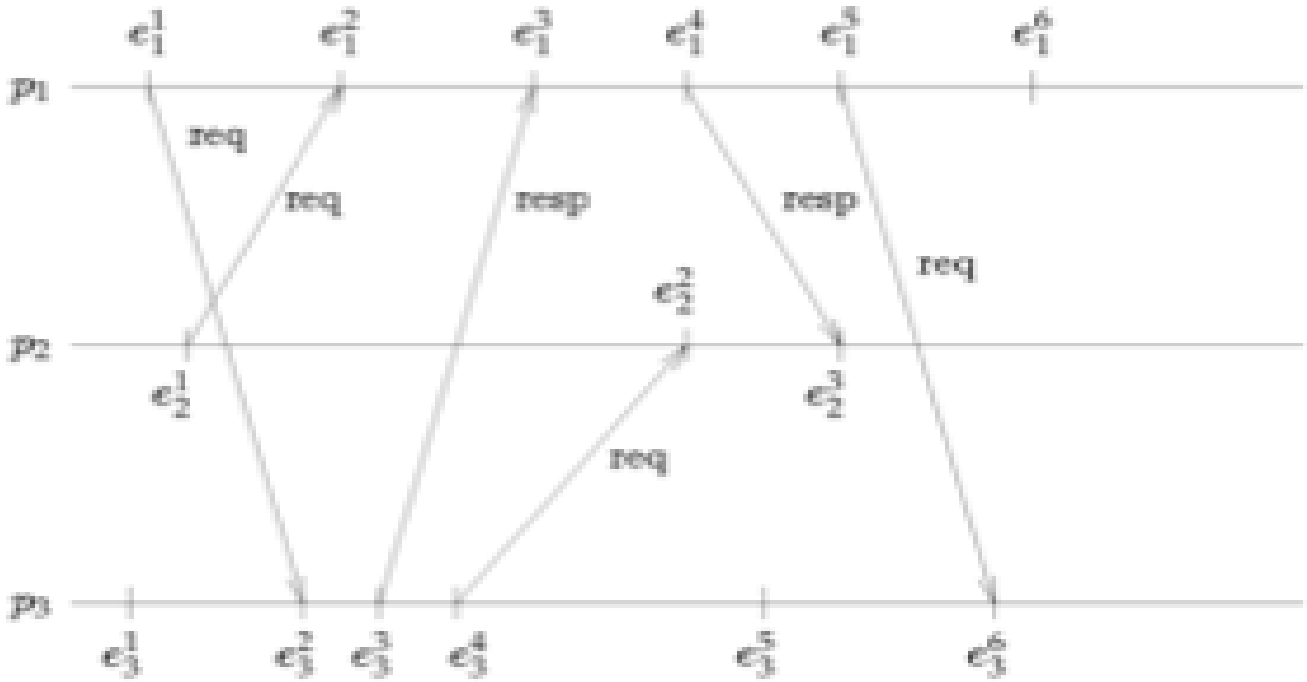


Why GPE is difficult

A global state obtained through remote observations could be

- obsolete: represent an old state of the system.
  Solution: build the global state more frequently
- inconsistent: capture a global state that could never have been occurred in reality
  Solution: build only consistent global states
- incomplete: not "capture" every moment of the system
  Solution: build all possible consistent global states

Space-Time Diagram of a Distributed Computation



Outline

Example – Deadlock detection on a multi-tier system

Processes in the previous figures use RPCs:

- Client sends a request for method execution; blocks.
- Server receives request.
- Server executes method; may invoke other methods in other servers, acting as a client.
- Server sends reply to client
- Clients receives reply; unblocks.

Such a system can deadlock, as RPCs are blocking. It is important to be able to detect when a deadlock occurs.

Runs and consistent runs

Processes in the previous figures use RPCs:

Definition (Run)

A run of global computation is a total ordering R that includes all the events in the local histories and that is consistent with each of them.

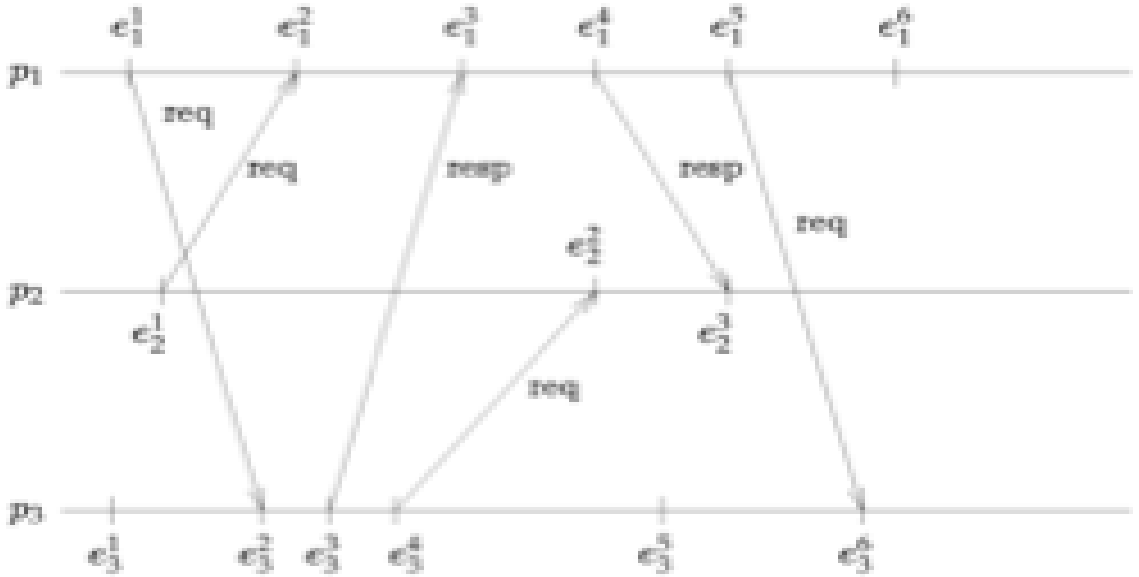
- In other words, the events of p_i appear in R in the same order in which they appear in h_i.
- A run corresponds to the notion that events in a distributed computation actually occur in a total order
- A distributed computation may correspond to many runs

Definition (Consistent run)

A run R is said to be consistent if for all events e and e', $e \longrightarrow e'$ implies that e appears before e' in R.

Runs and consistent runs

- $e_1^1 e_2^1 e_3^1 e_1^2 e_2^2 e_3^2 e_3^1 e_3^2 e_3^3 e_4^1 e_4^3 e_5^1 e_5^3 e_6^1 e_6^3$?
- $e_1^1 e_2^1 e_3^1 e_1^2 e_3^2 e_3^3 e_3^1 e_4^1 e_3^2 e_4^3 e_2^2 e_5^1 e_5^3 e_6^1 e_6^3$?



Monitoring Distributed Computations

- Assumptions:
  - There is a single process p_0 called monitor which is responsible for evaluating Φ
  - We assume that the monitor p_0 is distinct from the observed processes $p_1 \cdots p_n$
  - Events executed on behalf of monitoring do not alter canonical enumeration of "real" events
- In general, observed processes send notifications about local events to the monitor, which builds an observation.

Observations

Definition (Observation)

The sequence of events corresponding to the order in which notification messages arrive at the monitor is called an observation.

Given the asynchronous nature of our distributed system, any permutation of a run R is a possible observation of it.

Definition (Consistent observation)

An observation is consistent if it corresponds to a consistent run.

How to obtain consistent observations

- The happen-before relation captures the concept of potential causality
- In the "day-to-day" life, causality/concurrency are tracked using physical time
  - We use loosely synchronized watches;
  - Example: I have withdrawn money from an ATM in Trento at 13.00 on 17th May 2006, so I can prove that I've not withdrawn money on the same day at 13.20 in Paris
- In distributed computing systems:
  - the rate of occurrence of events is several magnitudes higher
  - event execution time is several magnitudes smaller
- If physical clocks are not precisely synchronized, the causality/concurrence relations between events may not be accurately captured

Outline

Logical clocks

Instead of using physical clocks, which are impossible to synchronize, we use logical clocks.

- Every process has a logical clock that is advanced using a set of rules
- Its value is not required to have any particular relationship to any physical clock.
- Every event is assigned a timestamp, taken from the logical clock
- The causality relation between events can be generally inferred from their timestamps

Logical clocks

Instead of using physical clocks, which are impossible to synchronize, we use logical clocks.

Definition (Logical clock)

A logical clock LC is a function that maps an event e from a distributed system execution to an element of a time domain T :

$$LC : H \longrightarrow T$$

Definition (Clock Consistency)

$$e \longrightarrow e' \Rightarrow LC(e) < LC(e')$$

Definition (Strong Clock Consistency)

$$e \longrightarrow e' \Longleftrightarrow LC(e) < LC(e')$$

Outline

Scalar logical clocks (I)

Instead of using physical clocks, which are impossible to synchronize, we use logical clocks.

Definition (Scalar logical clocks)

- A Lamport's scalar logical clock is a monotonically increasing software counter
- Each process p_i keeps its own logical clock LC_i
- The timestamp of event e executed by process pi is denoted $LC_i(e)$
- Messages carry the timestamp of their send event
- Logical clocks are initialized to 0

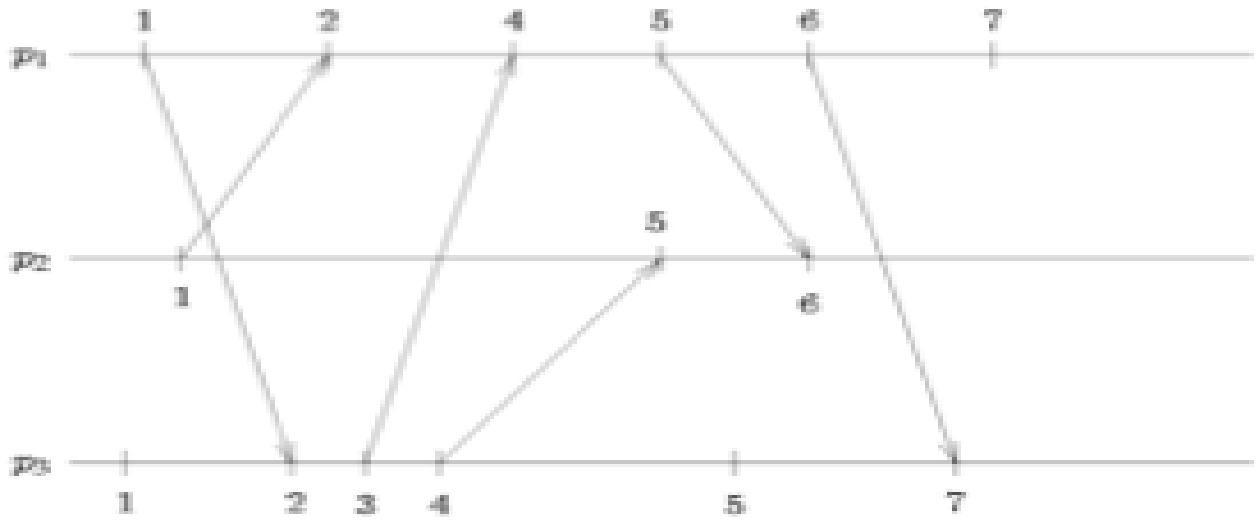
Scalar logical clocks (II)

Update rule

Whenever an event e is executed by process p_i, its local logical clock is updated as follows:

$$LC_i = \begin{cases} LC_i + 1 & \text{If } e_i \text{ is an internal or send event} \\ max\{LCi, TS(m)\} + 1 & \text{If } e_i\text{=receive(m)} \end{cases}$$

Scalar logical clocks (III)



Properties

Theorem

Scalar logical clocks satisfy Clock consistency, i.e.

$$e \longrightarrow e' \Rightarrow LC(e) < LC(e')$$

Proof

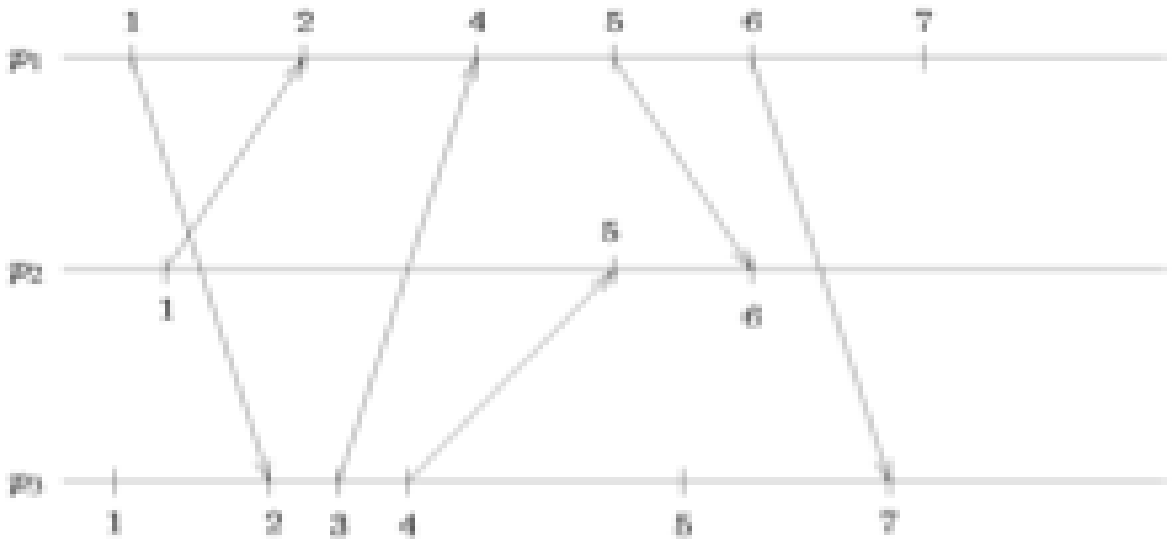
This immediately follows from the update rules of the clock.

Scalar logical clocks

Theorem

Scalar logical clocks do not satisfy Strong clock consistency, i.e.

$$LC(e) < LC(e') \not\Rightarrow e \longrightarrow e'$$



Outline

Causal histories clocks

Definition (Causal History)

The causal history of an event e is the set of events that happen-before e, plus e itself.

$$\theta(e) = \{e' \in H | e' \longrightarrow e\} \cup \{e\}$$

Theorem

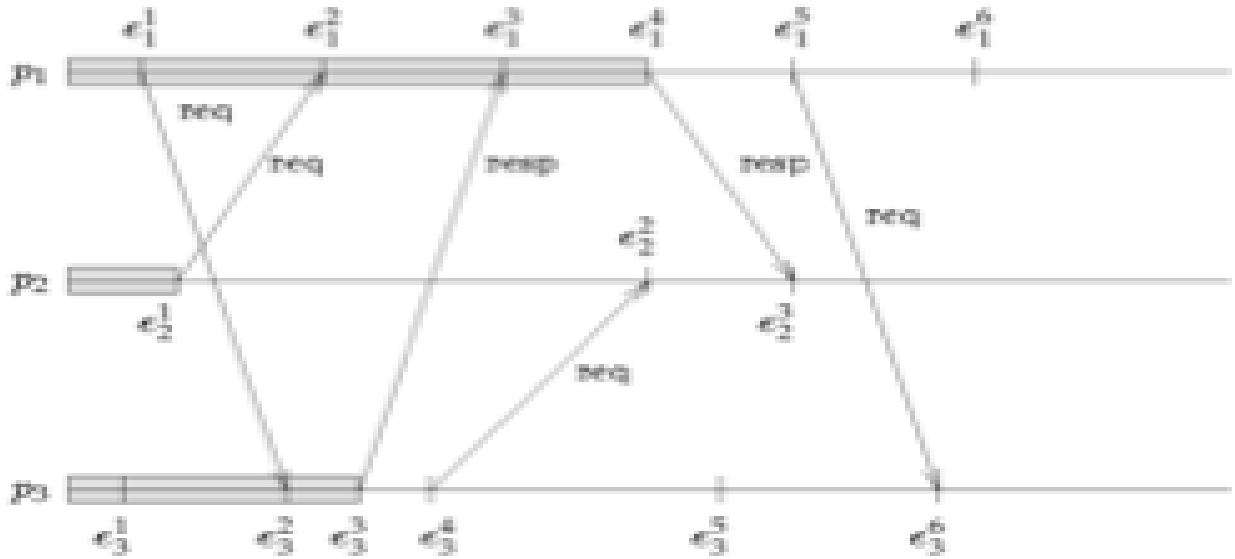
Causal histories satisfy Strong clock consistency

Proof

$$\forall e \neq e' : LC(e) < LC(e') \iff \theta(e) \subset \theta(e') \iff e \in \theta(e') \iff e \longrightarrow e'$$

Example

Problem:
Causal histories tend to grow too much; they cannot be used as "timestamps" for messages.



Vector clocks

- Causal history projection: $\theta_i(e) = \theta(e) \cap h_i = h_i^{c_i}$
- $\theta(e) = \theta_1(e) \cup \theta_2(e) \cup \cdots \cup \theta(e) = h_1^{c_1} \cup h_2^{c_2} \cup \cdots \cup h_n^{c_n}$
- In other words, $\theta(e)$ is a cut, which happens to be consistent.
- Cuts can be represented by their frontiers: $\theta(e) = (c_1, c_2, \cdots, c_n)$

Definition

The vector clock associated to event e is a n-dimensional vector VC(e) such that

$$VC(e)[i] = c_i \; where \; \theta_i(e) = h_i^{c_i}$$

Vector clocks: Implementation

- Each process p_i maintains a vector clock VC_i, initially all zeroes;
- When event e_i is executed, VC_i assumes the value of $VC(e_i)$;
- If $e_i = send(m)$, the timestamp of m is $TS(m) = VC(e_i)$;

Update rule

When event e_i is executed by process p_i, VC_i is updated as follows:

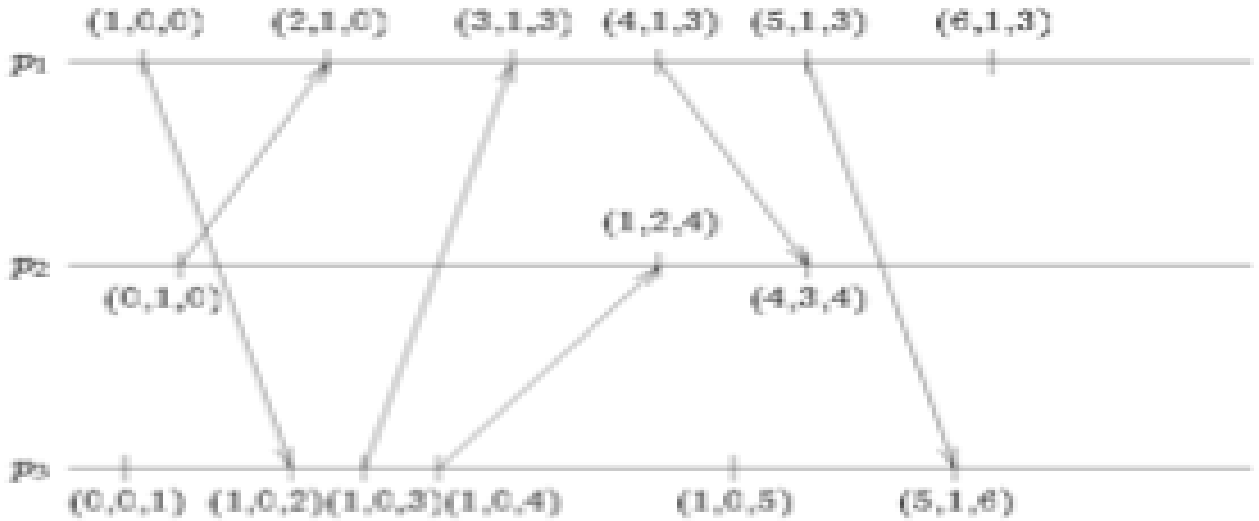
- If e_i is an internal or send event:

$$VC_i[i] = VC_i[i] + 1$$

- If $e_i = receive(m)$:

$$VC_i[j] = max\{VC_i[j], TS(m)[j]\}\ \forall j \neq i$$
$$VC_i[i] = VC[i] + 1$$

Example


(1,0,0)
(2,1,0)
(3,1,3)
(4,1,3)
(5,1,3)
(6,1,3)
p1
(1,2,4)
p2
(0,1,0)
(4,3,4)
p3
(0,0,1)
(1,0,2)
(1,0,3)
(1,0,4)
(1,0,5)
(5,1,6)

Properties of Vector clocks (I)

"Less than" relation for Vector clocks

$$V < V' \iff (V \neq V') \wedge (\forall k : 1 \leq k \leq n : V[k] \leq V'[k])$$

Strong Clock Condition

$$e \longrightarrow e' \iff VC(e) < VC(e') \iff \theta(e) \subset \theta(e')$$

Simple Strong Clock Condition

$$e_i \longrightarrow e_j \iff VC(e_i)[i] \leq VC(e_j)[i]$$

Properties of Vector clocks (II)

Definition (Concurrent events)

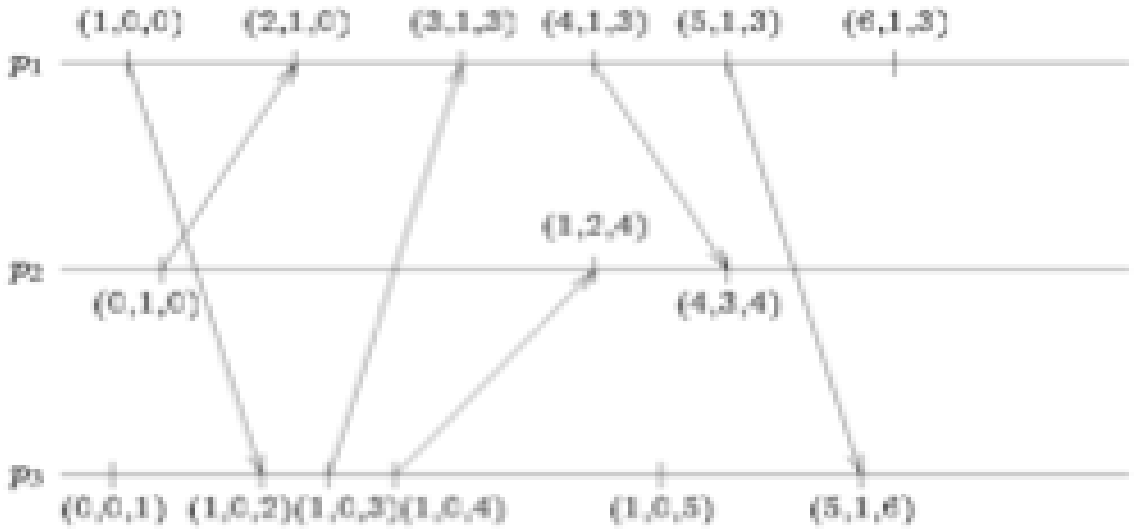
Events e_i and e_j are concurrent (i.e. $e_i||e_j$) if and only if:

$$(VC(e_i)[i] > VC(e_j)[i]) \wedge (VC(e_j)[j] > VC(e_i)[j])$$

In other words, event e_i does not happen-before e_j, and e_j does not happen before e_i.

Concurrent events

$$(VC(e_i)[i] > VC(e_j)[i]) \wedge (VC(e_j)[j] > VC(e_i)[j])$$



Properties of vector clocks

Definition (Pairwise Inconsistent)

Events e_i and e_j with $i \neq j$ are pairwise inconsistent if and only if

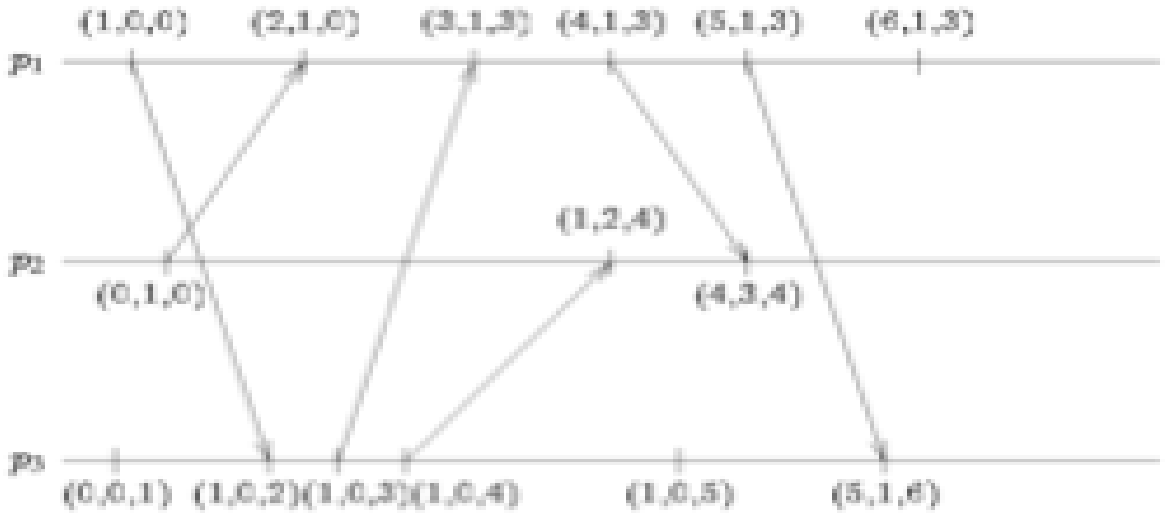
$$(VC(e_i)[i] < VC(e_j)[i]) \vee (VC(e_j)[j] < VC(e_i)[j])$$

In other words, two events are pairwise inconsistent if they cannot belong to the frontier of the same consistent cut. The formula characterize the fact that the cut include a receive event without including a send event.

Properties of vector clocks

Pairwise Inconsistent

$$(VC(e_i)[i] < VC(e_j)[i]) \vee (VC(e_j)[j] < VC(e_i)[j])$$



Properties of vector clocks

Definition (Consistent Cut)

A cut defined by $(c_1, \cdots, c_n)$ is consistent if and only if: $\forall i, j \in [1 \cdots n]$:

$$VC(e_i^{c_i})[i] \geq VC(e_j^{c_j})[i])$$

In other words, a cut is consistent if its frontier does not contain any pairwise inconsistent pair of events.

A Passive Approach to GPE

How it works

- At each (relevant) event, each process sends a notification to the monitor describing it local state
- The monitor collects notifications to reconstruct an observation of the global state.

An observation taken in this way can correspond to:

- A consistent run
- A run which is not consistent
- No run at all

Can you find example of the three cases?
Can you explain why this happen?

Observations which are not runs

Problem

Observations may not correspond to a run because each notification sent by the process to the monitor may be delayed arbitrarily and thus arrive in any possible order

Solution

To adopt communication channels between the processes and the monitor that guarantee that messages are never re-ordered

Message ordering

Definition (FIFO Order)

Two messages sent by p_i to p_j must be delivered in the same order in which they were sent:

$$\forall m, m' : send_i(m) \longrightarrow send_i(m') \Rightarrow deliver_j(m) \longrightarrow deliver_j(m')$$

What is "deliver"?

Delivery Rules

How to order messages?

- To be ordered, each message m carries a timestamp TS(m) containing "ordering" information
- The act of providing the process with a message in the desired order is called delivery; the event deliver(m) is thus distinct from receive (m).
- The rule describing which messages can be delivered among those received is called delivery rule

FIFO Order - Implementation

- Each process maintains a local sequence number incremented at each notification sent
- The timestamp of a notification message corresponds to the local sequence number of the sender at the time of sending

Definition (FIFO Delivery Rule)

If the last notification delivered by p_0 from p_j has timestamp s, p_0 may deliver "any" message m received from p_j with TS(m)=s+1.

Observations which are not consistent runs

Problem

If we use FIFO order between all processes and p_0, all the observations taken by p_0 will be runs; but there is no guarantee that they are consistent runs.

Solution

To adopt communication channels that guarantee that notification arrives in an order that respects the happen-before relation..

Message ordering

Definition (Causal Order)

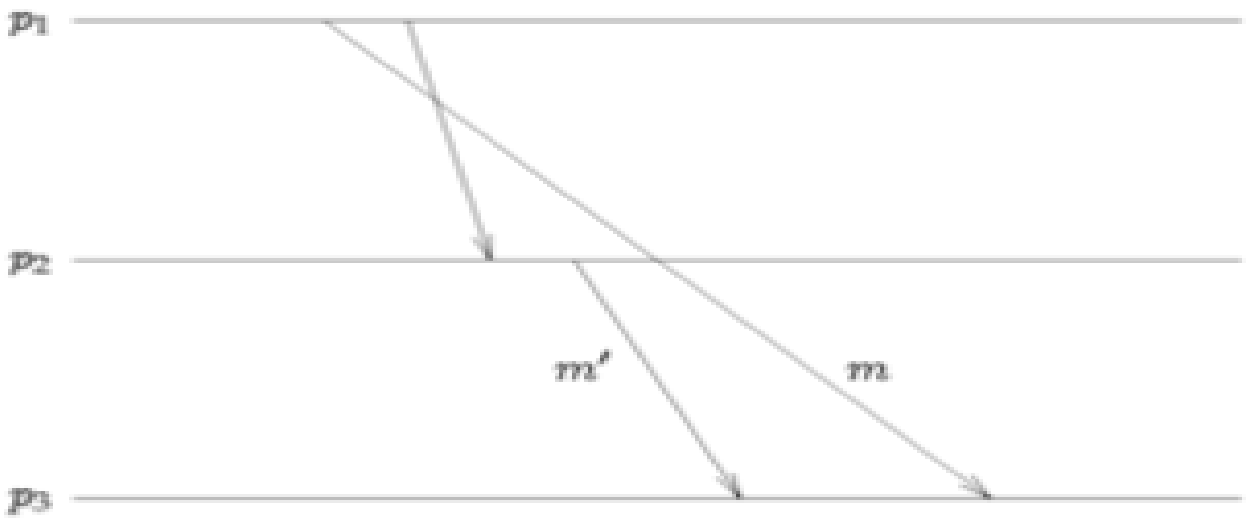
Two messages sent by pi and pj to pk must be delivered following the happen-before relation:

$$\forall m, m' : send_i(m) \longrightarrow send_j(m') \Rightarrow deliver_k(m) \longrightarrow deliver_k(m')$$

Question

FIFO order among all channels...
Is it sufficient to obtain Causal delivery?

Example



Causal delivery and consistent observations

Theorem

If p_0 uses a delivery rule satisfying Causal Order, then all of its observations will be consistent.

Proof

Definition of Causal Order ? definition of a consistent observation

Next, we will show three methods to implement a causal delivery rule

Outline

Passive monitoring with real-time

Initial assumptions

- All processes have access to a real-time clock RC
- Let RC(e) be the real-time at which e is executed
- All messages are delivered within a time δ
- The timestamp of message m sent by an event e=send(m) is TS(m)=RC(e).

Definition (DR1: Real-time delivery rule)

At time t, delivery all received notification messages m in increasing timestamp order.

Theorem

Observation O constructed by p_0 using DR1 is guaranteed to be consistent

Stability of messages

Definition (Stability)

A notification message m received by p_0 is stable at p_0 if no message m' with TS(m')¡TS(m) can be received in the future by p_0

Definition (DR1: Real-time delivery rule)

At time t, delivery all received notification messages m such that $TS(m) \leq t - \delta$ in increasing timestamp order.

Proof

?Safety: Clock Condition for RC

$$e \longrightarrow e' \Rightarrow RC(e) < RC(e')$$

Note that $RC(e) < RC(e') \not\Rightarrow e \longrightarrow e'$, but this rule is sufficient to obtain consistent observations, as two notifications $e \longrightarrow e'$ are never delivered in the incorrect order.

Liveness: Stability

At time t, any message sent by time $t - \delta$ is stable.

Note that real-time clocks do not support stability; it is the maximum delay of messages that enables it.

Outline

Proof

Passive monitoring with logical clocks

Initial assumptions

- All processes have access to a logical clock LC; let LC(e) be the real-time at which e is executed
- The timestamp of message m sent through an event e=send(m) is TS(m)=LC(e)

Definition (DR2: Deliver Rule for LC)

Deliver all received messages that are stable at p_0 in increasing timestamp order

Passive monitoring with logical clocks

?Safety: Clock Condition for LC

$$e \longrightarrow e' \Rightarrow LC(e) < LC(e')$$

Note that $LC(e) < LC(e') \not\Rightarrow e \longrightarrow e'$, but this rule is sufficient to obtain consistent observations, as two notifications $e \longrightarrow e'$ are never delivered in the incorrect order.

Passive monitoring with logical clocks

Liveness: Stability

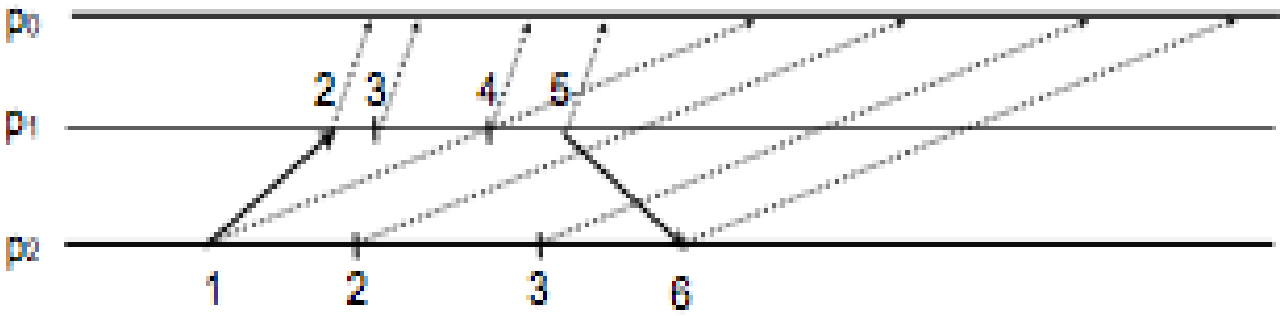
We need a way to reproduce the concept of δ in an asynchronous system, otherwise no notification message will be ever delivered.

Solution

- Each process communicates with p_0 using FIFO delivery
- When p_0 receives a message from pi describing an event e with timestamp TS(e), it is sure that it will never receive a message from pi describing an event e' with $TS(e') \leq TS(e)$
- Stability of message m at p_0 can be guaranteed when p_0 has received at least one message from all other processes with a timestamp greater or equal than TS(m)

Problems of Logical Clocks

- They add unnecessary delays to observations
- They require a constant flux of messages/events from all processes



Passive Monitoring with Vector Clocks

Variables maintained at p_0

- $\mathcal{M}$ the set of notification messages received but not yet delivered by p_0
- an array D, initialized to 0's, such that D[k] contains TS(m)[k] where m is the last notification message delivered by p_0 from process p_k.

When a notification message is deliverable by p_0?
A notification message $m \in \mathcal{M}$ from process p_j is deliverable as soon as p_0 can verify that there is no other notification message m' (neither in $\mathcal{M}$ nor in the channels) such that $send(m') \longrightarrow send(m)$.

Outline

Implementing Causal Delivery with Vector Clocks

- $m \in \mathcal{M}$: a notification message sent by p_j to p_0
- m': the last notification message delivered from process $p_k, k \neq j$

Definition (Weak Gap Detection)

If $TS(m')[k] < TS(m)[k]$ for some $k \neq j$, then there exists event $send_k(m'')$ such that

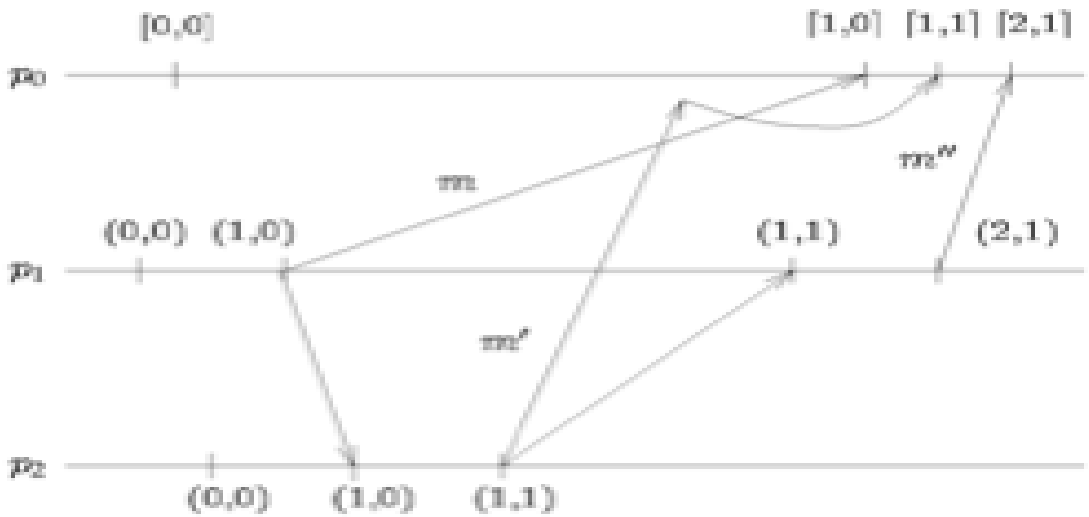
$$send_k(m') \longrightarrow send_k(m'') \longrightarrow send_j(m)$$

Implementing Causal Delivery with Vector Clocks

Two conditions to be verified:

- There is no earlier message from p_j that has not been delivered yet. Causal Delivery Rule, Part 1: D[j]==TS(m)[j]-1
- $\forall k \neq j$, let m' be the last message from p_k delivered by p_0 (D[k]=TS(m')[k]); we must be sure that no message m'' from p_k exists such that: $send_k(m') \longrightarrow send_k(m'') \longrightarrow send_j(m)$ Causal Delivery Rule, Part 2: $\forall k \neq j : D[k] \geq TS(m)[k]$ It follows from Weak Gap Detection

Example



Final Comments

- We have seen how to implement Causal Delivery "many-to-one"
- The same rules apply if we implement a mechanism for implementing "one-to-many" (reliable broadcast)