Distributed Systems
Time, clocks and the ordering of events

Alberto Montresor
(with contributions from Hein Meling (UiS))

University of Trento, Italy

2024/03/21

Contents

Contents

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 $h_i = e_i^0 e_i^1 e_i^2 \dots e_i^{m_i}$ (*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

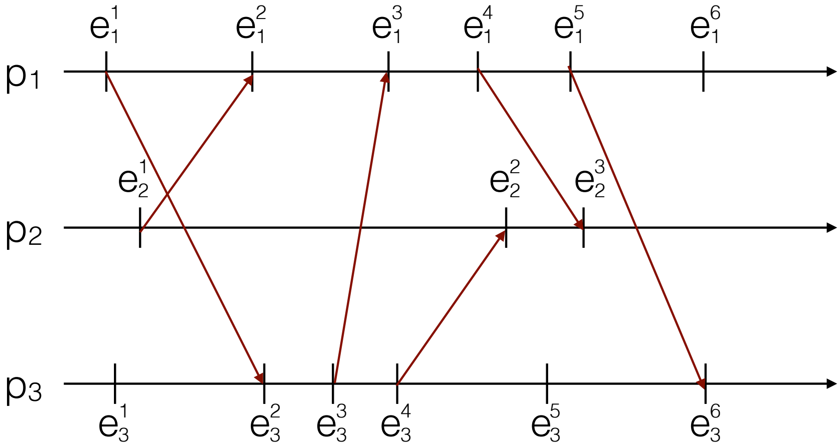
Happen-Before

Definition (Happen-before)

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

1. $\exists p_i \in \Pi : e = e_i^r, \quad e' = e_i^s, \quad r < s$
   (e and e' are executed by the same process, e before e')
2. $e = send(m, *) \wedge e' = receive(m)$
   (e is the send event of a message m and e' is the corresponding receive event)
3. $\exists e'' : e \to e'' \to e')$
   (in other words, $\to$ is transitive)

Space-Time Diagram of a Distributed Computation



Meaning of Happen-Before

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

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

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 ($e \not\rightarrow e' \wedge e' \not\rightarrow e$) are concurrent, and we write $e || e'$.

Homework

Prove or disprove that the $||$ relation is transitive.

Reality check

Forse non tutti sanno che...

The memory model of popular programming languages like Go, C++ and Java is based on the happen-before relation, which is used to define the semantics of concurrent programs. Communication between threads is based on the acquire and release of locks.

https://go.dev/ref/mem

https://docs.oracle.com/javase/specs/jls/se22/html/jls-17.html#jls-17.4

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 $\Sigma = (\sigma_1, \ldots, \sigma_n)$, one for each process.

Cut

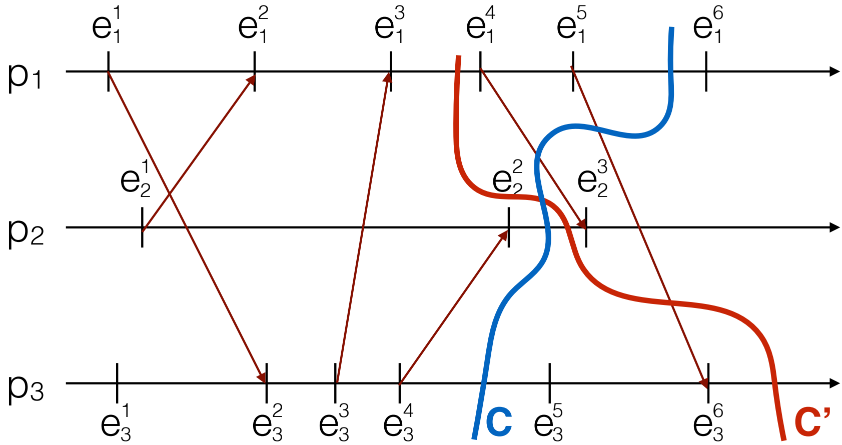
Definition (Cut)

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

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

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

Cuts


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

Definition (Consistent cut)

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

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

Definition (Consistent global state)

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

In other words:

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

Consistent cut

- 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.

Contents

Introduction

Definition (Global Predicate Evaluation)

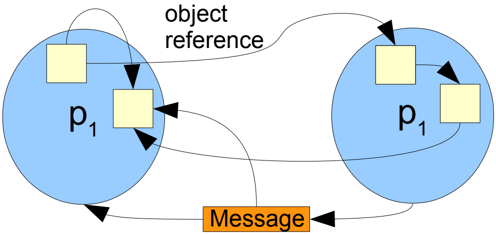
The problem of detecting whether the global state of a distributed system satisfies some predicate Φ.

Motivation

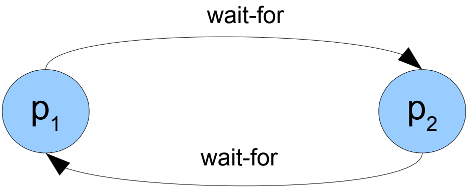
- Many problems in distributed systems require a reaction when the global state of the system satisfies some condition.
  - Monitoring: Notify an administrator in case of failures
  - Debugging: Verify whether an invariant is respected or not
  - Deadlock detection: can the computation continue?
  - 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

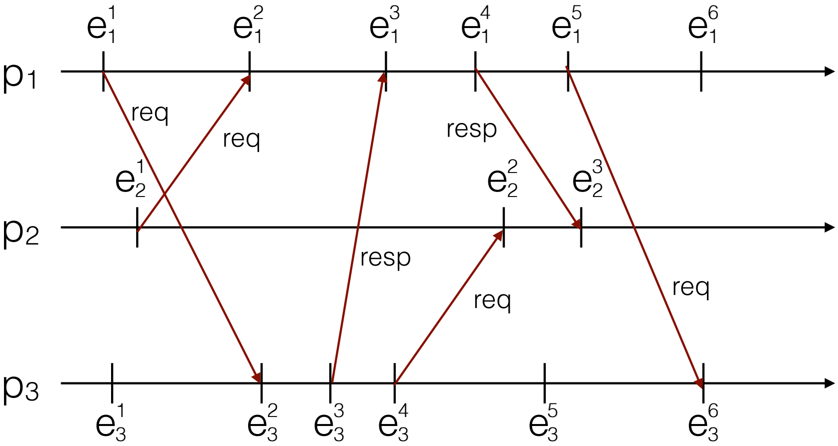


Why GPE is difficult

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 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



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 methods on 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

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.

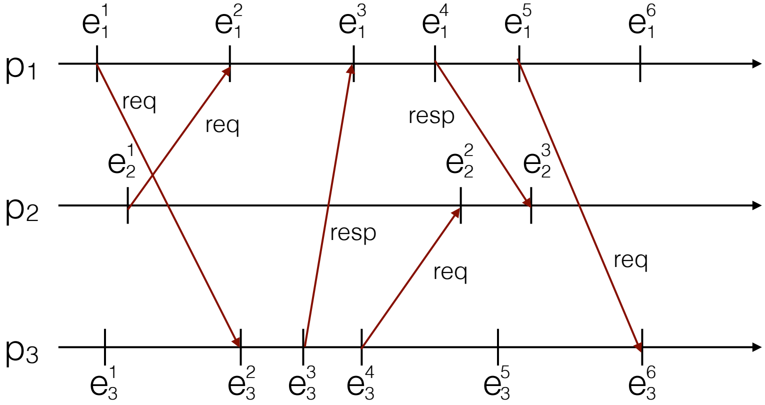
- In other words, the events of p_i appear in R in the same order in which they appear in h_i.
- A run corresponds to the notion that events in a distributed computation actually occur in a total order
- A distributed computation may correspond to many runs

Definition (Consistent run)

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

Runs and consistent runs

- $e_1^1\, e_1^2\, e_1^3\, e_1^4\, e_1^5\, e_1^6\, e_2^1\, e_2^2\, e_2^3\, e_3^1\, e_3^2\, e_3^3\, e_3^4\, e_3^5\, e_3^6$?
- $e_1^1\, e_2^1\, e_3^1\, e_1^2\, e_3^2\, e_3^3\, e_1^3\, e_1^4\, e_3^4\, e_2^2\, e_2^3\, e_1^5\, e_3^5\, e_1^6\, e_3^6$?



Monitoring Distributed Computations

- Assumptions:
  - A monitor process p_0 is responsible for evaluating Φ
  - We assume p_0 is distinct from the observed processes $p_1 \dots p_n$
  - Monitoring events do not alter canonical enumeration of "real" events
- Observed processes send notifications about local events to p_0
- p_0 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.

Contents

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/concurrency relations between events may not be accurately captured

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

Definition (Logical clock)

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

$$LC : H \to T$$

Definition (Clock Condition)

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

Definition (Strong Clock Condition)

$$e \to e' \Leftrightarrow LC(e) < LC(e')$$

Scalar logical clocks

Definition (Scalar logical clocks)

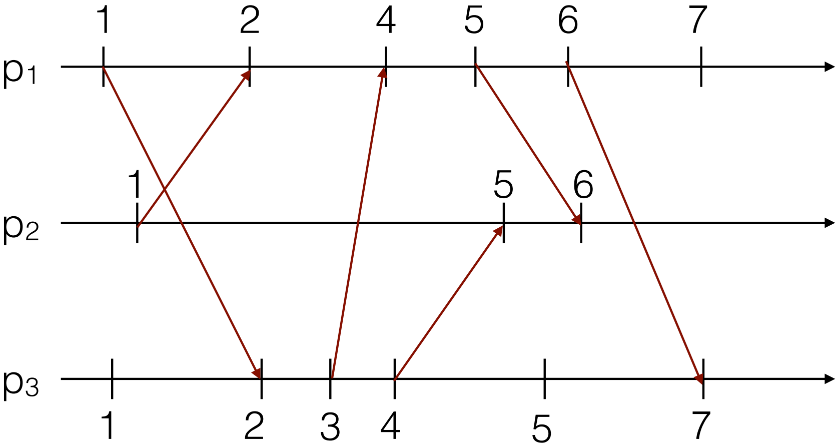
- Lamport's scalar logical clock is a monotonically increasing software counter
- Each process p_i keeps its own logical clock LC_i
- The timestamp of event e executed by process p_i is denoted $LC_i(e)$
- Messages carry the timestamp of their *send* event
- Logical clocks are initialized to 0

Update rule

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

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

Scalar logical clocks



Properties

Theorem

Scalar logical clocks satisfy Clock condition, i.e.

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

Properties

Theorem

Scalar logical clocks satisfy Clock condition, i.e.

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

Proof.

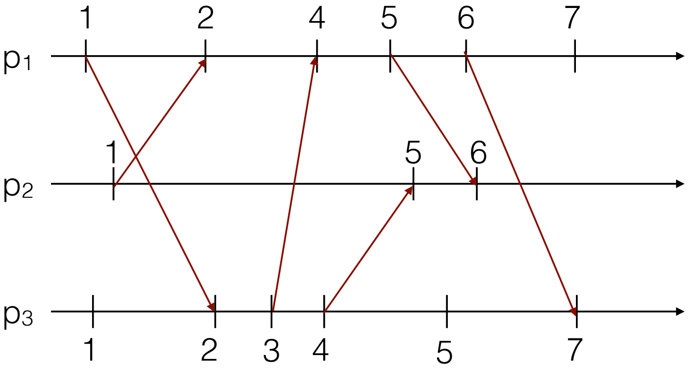
This immediately follows from the update rules of the clock. $\square$

Scalar logical clocks

Theorem

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

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



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.

$$\theta(e) = \{e' \in H \mid e' \to e\} \cup \{e\}$$

Theorem

Causal histories satisfy Strong clock condition

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.

$$\theta(e) = \{e' \in H \mid e' \rightarrow e\} \cup \{e\}$$

Theorem

Causal histories satisfy Strong clock condition

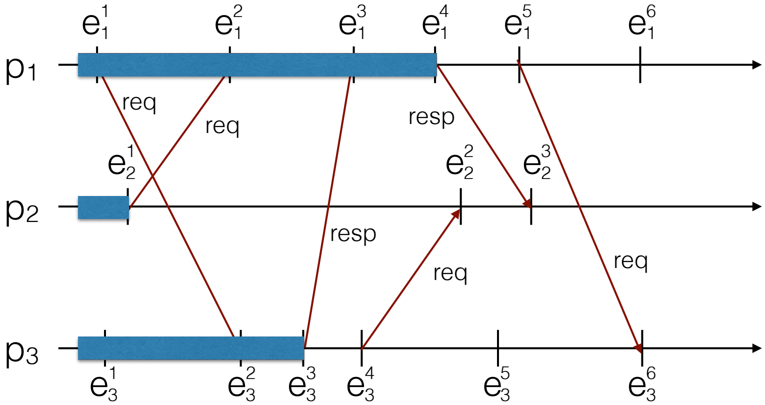
Proof.

$$\forall e \neq e' : LC(e) < LC(e') \stackrel{def}{\Leftrightarrow} \theta(e) \subset \theta(e') \Leftrightarrow e \in \theta(e') \Leftrightarrow e \rightarrow e'$$

□

Example

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



Vector clocks

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

Definition

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

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

Vector clocks: implementation

- Each process p_i maintains a vector clock VC_i, initially all zeroes;
- When event e_i is executed, its vector clock is updated and assumes the value of $VC(e_i)$;
- If $e_i = \mathit{send}(m, *)$, 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:

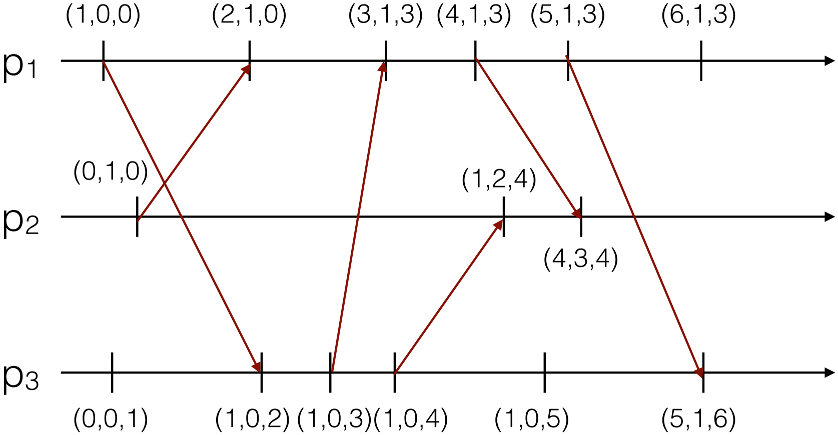
- If e_i is an internal or *send* event:

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

- If $e_i = \mathit{receive}(m)$:

$$\begin{aligned} VC_i[j] &= \max\{VC_i[j], TS(m)[j]\} \quad \forall j \neq i \\ VC_i[i] &= VC_i[i] + 1 \end{aligned}$$

Example


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

Properties of Vector clocks

"Less than" relation for Vector clocks

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

Strong Clock Condition

$$e \rightarrow e' \Leftrightarrow VC(e) < VC(e') \Leftrightarrow \theta(e) \subset \theta(e')$$

Simple Strong Clock Condition

$$e_i \rightarrow e_j \Leftrightarrow VC(e_i)[i] \leq VC(e_j)[i]$$

Properties of Vector clocks

Definition (Concurrent events)

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

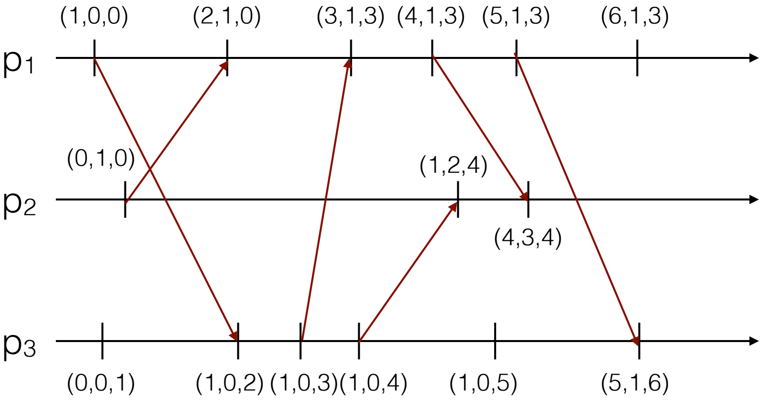
$$(VC(e_i)[i] > VC(e_j)[i]) \wedge (VC(e_j)[j] > VC(e_i)[j])$$

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

Example: ?

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

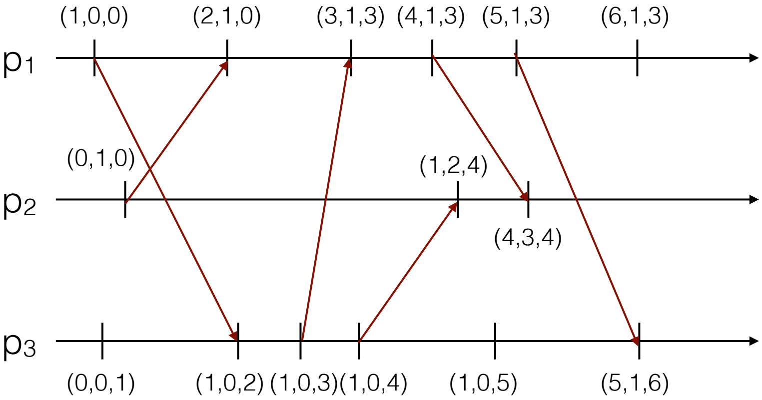
$$(VC(e_i)[i] < VC(e_j)[i]) \vee (VC(e_j)[j] < VC(e_i)[j])$$

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.

Example: ?

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 $(c_1, \ldots, c_n)$ is consistent if and only if:

$$\forall i, j \in [1 \ldots n] : VC(e_i^{c_i})[i] \geq VC(e_j^{c_j})[i]$$

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

Contents

Contents

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 the sequence of notifications, i.e. an observation of the distributed run.

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 the notifications sent by a single process to the monitor may be delayed arbitrarily and thus arrive in any possible order

Observations which are not runs

Problem

Observations may not correspond to a run because the notifications sent by a single 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 notifications are never re-ordered

Notification ordering

Definition (FIFO Order)

Two notifications sent by p_i to p_0 must be delivered in the same order in which they were sent:

$$\forall m, m' : send_i(m, p_0) \to send_i(m', p_0) \Rightarrow deliver_0(m) \to deliver_0(m')$$

Uh? What is "deliver"?

Delivery Rules

How to order notifications?

- To be ordered, each notification m carries a timestamp $TS(m)$ containing "ordering" information
- The act of providing the monitor with a notification in the desired order is called delivery; the event $deliver(m)$ is thus distinct from $receive(m)$.
- The rule describing which notifications 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 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" notification 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*.

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 notifications are delivered in an order that respects the happen-before relation.

Notification ordering

Definition (Causal Order)

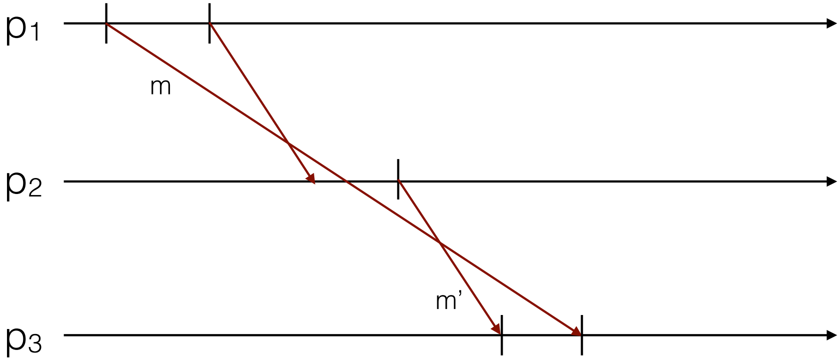
Two notifications sent by p_i and p_j to p_0 must be delivered following the happen-before relation:

$$\forall m, m' : send_i(m, p_0) \rightarrow send_j(m', p_0) \Rightarrow deliver_0(m) \rightarrow deliver_0(m')$$

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 □

Three implementations of the causal delivery rule:

- Real-time clocks
- Logical (scalar) clocks
- Vector clocks

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 notifications are delivered within a time δ
- The timestamp of notification m corresponding to an event e is $TS(m) = RC(e)$.

Definition (DR1: Real-time delivery rule)

At any time, deliver all received notifications in increasing timestamp order.

Theorem

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

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 notifications are delivered within a time δ
- The timestamp of notification m corresponding to an event e is $TS(m) = RC(e)$.

Definition (DR1: Real-time delivery rule)

At any time, deliver all received notifications 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 m received by p_0 is stable at p_0 if no notification m' with $TS(m') < TS(m)$ can be received in the future by p_0

Definition (DR1: Delivery rule for RC)

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

Theorem

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

Proof

Safety: Clock Condition for RC

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

Note that $RC(e) < RC(e') \not\Rightarrow e \rightarrow e'$, but this rule is sufficient to obtain consistent observations, as two notifications $e \rightarrow e'$ 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.

Passive monitoring with logical clocks

Initial assumptions

- All processes have access to a logical clock LC;
  let $LC(e)$ be the logical clock at which e is executed
- The timestamp of notification m corresponding to an event e is $TS(m) = LC(e)$

Definition (DR2: Deliver Rule for LC)

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

Passive monitoring with logical clocks

Safety: Clock Condition for LC

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

Note that $LC(e) < LC(e') \not\Rightarrow e \rightarrow e'$, but this rule is sufficient to obtain consistent observations, as two events $e \rightarrow e'$ are never ordered incorrectly

Passive monitoring with logical clocks

Liveness: Stability

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

Passive monitoring with logical clocks

Liveness: Stability

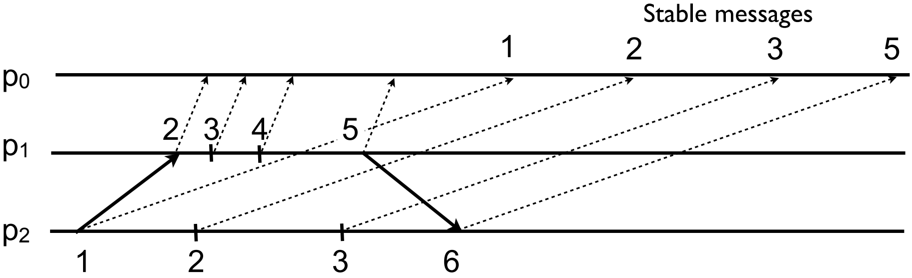
We need a way to reproduce the concept of δ in an asynchronous system, otherwise no notification will be ever delivered.

Solution

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



Passive Monitoring with Vector Clocks

Variables maintained at p_0

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

When a notification is deliverable by p_0?

A notification $m \in M$ from process p_j is deliverable as soon as p_0 can verify that there is no other notification m' (neither in M nor in the channels) such that $send(m', p_0) \rightarrow send(m, p_0)$.

Implementing Causal Delivery with Vector Clocks

- $m \in M$: a notification sent by p_j to p_0
- m': the last notification 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

$$send_k(m', p_0) \rightarrow send_k(m'', p_0) \rightarrow send_j(m, p_0)$$

Implementing Causal Delivery with Vector Clocks

Two conditions to be verified to check if m can be delivered:

- There is no earlier message from p_j that has not been delivered yet.

  How can we express this condition?

- $\forall k \neq j$, 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: $send_k(m', p_0) \rightarrow send_k(m'', p_0) \rightarrow send_j(m, p_0)$

  How can we express this condition?

Implementing Causal Delivery with Vector Clocks

Two conditions to be verified to check if m can be delivered:

- 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: $send_k(m', p_0) \rightarrow send_k(m'', p_0) \rightarrow send_j(m, p_0)$

  How can we express this condition?

Implementing Causal Delivery with Vector Clocks

Two conditions to be verified to check if m can be delivered:

- There is no earlier message from p_j that has not been delivered yet.

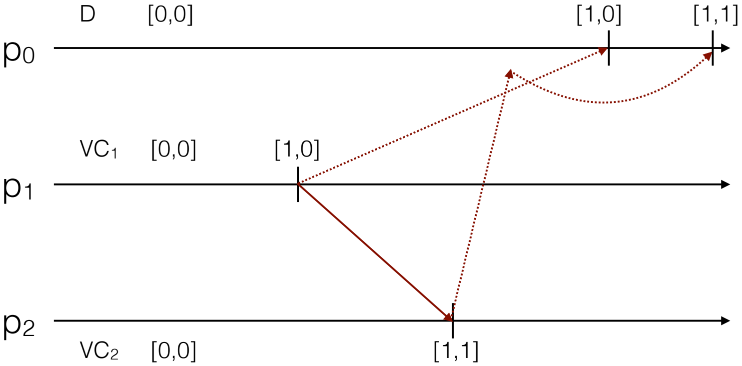
  Causal Delivery Rule, Part 1 $D[j] == TS(m)[j] - 1$

- $\forall k \neq j$, 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: $send_k(m', p_0) \rightarrow send_k(m'', p_0) \rightarrow send_j(m, p_0)$

  Causal Delivery Rule, Part 2: $\forall k \neq j : D[k] \geq TS(m)[k]$
  It follows from Weak Gap Detection

Example

$$\Big(D[j] == TS(m)[j] - 1\Big) \wedge \Big(\forall k \neq j : D[k] \geq TS(m)[k]\Big)$$



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)

Contents

Snapshot Protocol

Problem

- Goal: To build the global state "on demand" of the monitor.
- How: By taking "pictures" (snapshots) of the local state when instructed
- Challenge: To build a consistent global state.

Applications

- Failure recovery: a global state (checkpoint) is periodically saved; to recover from a failure, the system is restored to the last saved checkpoint.
- Distributed garbage collection
- Monitoring distributed events (e.g., industrial process control)

Chandy-Lamport Snapshot Algorithm

- This particular protocol enables to reason about “channel states”
- GPE can be delayed with respect to passive monitoring
- Assumption: processes communicate through FIFO channels

Snapshot Protocol

Channel State

- For each channel between p_i and p_j
  - $x_{i,j}$ contains the messages sent by p_i but not received yet by p_j
  - $x_{j,i}$ contains the messages sent by p_j but not received yet by p_i
- Note: channel state can be obtained by storing appropriate information in the local state, but it is complicated.

Recorded information

Each process p_i will record its local state σ_i and the content of its incoming channels $x_{j,i}$.

Snapshot Protocol

Assumptions:

- Communication channels satisfy FIFO order

Assumptions to be relaxed later:

- Access to a real time clock RC;
- Message delays are bounded by some known value δ;
- Relative process speeds are bounded;
- Message m is tagged with a timestamp $TS(m) = RC(e)$, where $e = send(m)$.

Snapshot Protocol, v.1

1. p_0 chooses a time t_s far enough in the future that a message containing t_s sent by p_0 will be received by all processes by t_s:

$$t_s = now() + \delta$$

2. p_0 sends a message *"take a snapshot at t_s"* to all processes in Π
3. When $RC_i = t_s$ do:
    1. p_i records its local state
    2. p_i sends an empty message to all processes in Π
    3. p_i starts to record all messages received over each incoming channel
4. When p_i receives a message m from j such that $TS(m) \geq t_s$, it stops recording messages for incoming channel j
5. Each p_i sends its recorded local state and channel states to p_0

Snapshot Protocol, v.1

Liveness The empty messages at 3.2 guarantee liveness.

Safety This protocol constructs a consistent global state; actually, this global state did in fact occur. Formally:

Let C_s be the cut associated to the constructed global state;

$$\begin{array}{rcl} (e \in C_s) \wedge (e' \rightarrow e) & \Rightarrow & \\ (e \in C_s) \wedge (RC(e') < RC(e)) & \Rightarrow & \text{C.C.} : e' \rightarrow e \Rightarrow RC(e') < RC(e) \\ (e' \in C_s) & & C_s \text{ Def.} : e \in C_s \Leftrightarrow RC(e) < t_s \end{array}$$

Can we use logical clocks instead of a real time clock?

Snapshot Protocol, v.2

From real time clocks to logical clocks:

- The construct " **when** $LC = t_s$ **do** *statement*" makes no sense
  - LCs are not continuous (e.g., they can jump from $t_s - 1$ to $t_s + 1$)
  - When $LC = t_s$, the event that caused the clock update is already done.
- Solution: before p_i executes an event e:
  - If e is an internal or send event, and $LC = t_s - 2$:
    - p_i executes e
    - p_i executes *statement*
  - If $e = receive(m) \wedge TS(m) \geq t_s$ and $LC < t_s - 1$:
    - p_i puts $LC = t_s - 1$
    - p_i executes *statement*
    - p_i executes e

Snapshot Protocol, v.2

From real time clocks to logical clocks:

- We assume that we can found a logical time t_s large enough that a message containing it sent by p_0 will be received by all other processes before t_s
- Impossible in an asynchronous distributed systems
- We will relax this assumption later

Snapshot Protocol, v.2

1. p_0 chooses a logical time t_s large enough
2. p_0 sends a message *"take a snapshot at t_s"* to all processes in Π
3. p_0 sets its logical clock to t_s;
4. When $LC_i = t_s$ do:
    1. p_i records its local state;
    2. p_i sends an empty message to all processes in Π;
    3. p_i starts to record all messages received over each incoming channel.
5. When p_i receives a message m from j such that $TS(m) \geq t_s$, it stops recording messages for incoming channel j
6. Each p_i sends its recorded local state and channel states to p_0.

Snapshot Protocol, v.3

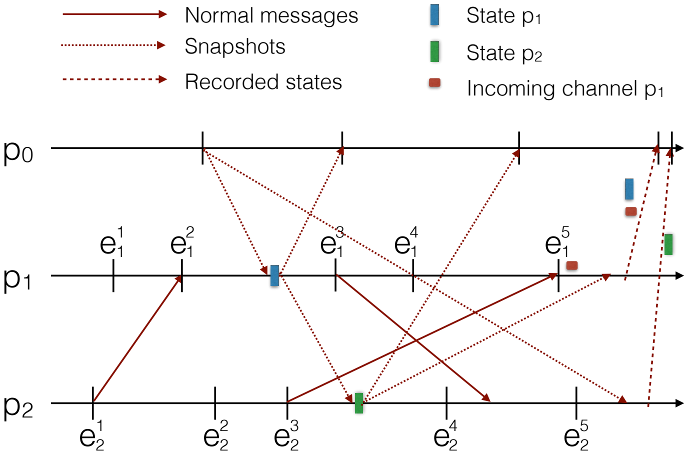
We now remove the need for t_s:

- A process may receive an "empty message" from a node before the "take snapshot at t_s" is actually received
- In other words, it may be already aware of the snapshot protocol
- We remove the "at t_s" and we use SNAPSHOT messages instead of empty ones
- We can now remove logical clocks completely, as messages are not timestamped any more

Snapshot Protocol, v.3

1. p_0 sends a message SNAPSHOT to itself ;
2. when p_i receives SNAPSHOT for the first time:
   - let p_j be the sender of this message
   - p_i records its local state σ_i;
   - sends SNAPSHOT to all processes in Π;
   - $x_{k,i} \leftarrow \emptyset \qquad \forall k \neq i$;
3. when p_i receives message $m \neq$ SNAPSHOT from $p_k, k \neq j$
   - $x_{k,i} \leftarrow x_{k,i} \cup \{m\}$
4. when p_i receives a SNAPSHOT from $p_k, k \neq j$ beyond the first time:
   - p_i stops recording messages in $x_{k,i}$;
5. when p_i has received a SNAPSHOT from all processes
   - p_i then sends its recorded local state and the channel states to p_0.

Example


Normal messages
Snapshots
Recorded states
State p1
State p2
Incoming channel p1
p0
p1
p2
e_1^1
e_1^2
e_1^3
e_1^4
e_1^5
e_2^1
e_2^2
e_2^3
e_2^4
e_2^5

Proof of correctness

Theorem

A global state built using the Chandy-Lamport snapshot algorithm is consistent.

Contents

Stable Predicates

Problem

- Let Σ be a global state built by one of the presented methods
- It represents a state of the past, potentially with no bearing to the present
- Does it make sense to evaluate predicate Φ on it?

A special case: stable predicates

Many systems properties have the characteristic that once they become true, they remain true.

- Deadlock
- Garbage collection
- Termination

A distributed computation may have many runs

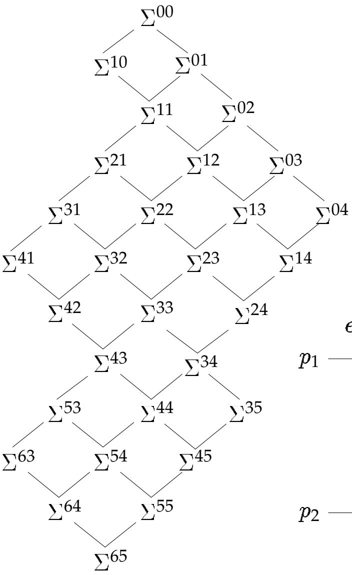
Definition (Leads-to relation)

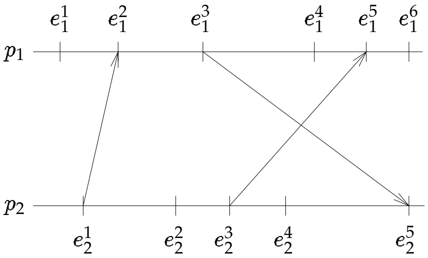
- A consistent run $R = e^1 e^2 \ldots$ results in a sequence of consistent global states $\Sigma^0 \Sigma^1 \Sigma^2 \ldots$, where Σ^0 denotes the initial global state.
- We say that a global state Σ leads to to a global state Σ', denoted $\Sigma \leadsto_R \Sigma'$ in a consistent run R if:
    - R results in a sequence of global states $\Sigma^0 \Sigma^1 \Sigma^2 \ldots$;
    - $\Sigma = \Sigma^i, \Sigma' = \Sigma^j, i < j$.
- We write $\Sigma \leadsto \Sigma'$ if there is a run R such that $\Sigma \leadsto_R \Sigma'$.

A distributed computation may have many runs

Definition (Lattice)

- The set of all consistent global states of a computation along with the leads-to relation defines a lattice;
- n orthogonal axis, one per process;
- $\Sigma^{k_1 \ldots k_n}$ shorthand for the global state $(\sigma_1^{k_1}, \ldots, \sigma_n^{k_n})$;
- The level of $\Sigma^{k_1 \ldots k_n}$ is equal to $k_1 + \cdots + k_n$.
- A path in the lattice is a sequence of global states of increasing levels that corresponds to a consistent run.


Σ^{00}
Σ^{10}
Σ^{01}
Σ^{11}
Σ^{02}
Σ^{21}
Σ^{12}
Σ^{03}
Σ^{31}
Σ^{22}
Σ^{13}
Σ^{04}
Σ^{41}
Σ^{32}
Σ^{23}
Σ^{14}
Σ^{42}
Σ^{33}
Σ^{24}
Σ^{43}
Σ^{34}
Σ^{53}
Σ^{44}
Σ^{35}
Σ^{63}
Σ^{54}
Σ^{45}
Σ^{64}
Σ^{55}
Σ^{65}


e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
p_1
p_2
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5

Stable Predicates

Consider a global state construction protocol:

- Let Σ^a be the global state in which the protocol is initiated;
- Let Σ^f be the global state in which the protocol terminates;
- Let Σ^s be the global state constructed by the protocol

Since $\Sigma^a \rightsquigarrow \Sigma^s \rightsquigarrow \Sigma^f$, if Φ is stable, then:

$$\begin{aligned} \Phi(\Sigma^s) = \textbf{true} \quad &\Rightarrow \quad \Phi(\Sigma^f) = \textbf{true} \\ \Phi(\Sigma^s) = \textbf{false} \quad &\Rightarrow \quad \Phi(\Sigma^a) = \textbf{false} \end{aligned}$$

Deadlock Detection

Code

- Server code
- Server code, modified for snapshot protocol
- Monitor code for snapshot protocol
- Server code, modified for passive protocol
- Monitor code for passive protocol

Notes

- No need to store channel state in this case

Server code

Process p_i

$\textsc{Queue}$ *pending* $\leftarrow$ **new** $\textsc{Queue}()$
boolean *working* $\leftarrow$ **false**
while true do
  while *working* **or** *pending*.size() $= 0$ **do**
    receive $\langle m, p_j \rangle$
    if $m.type = \text{REQUEST}$ **then**
      pending.enqueue($\langle m, p_j \rangle$)
    else if $m.type = \text{RESPONSE}$ **then**
      $\langle m', p_k \rangle \leftarrow \mathsf{nextState}(m, p_j)$
      $working \leftarrow (m'.type = \text{REQUEST})$
      send m' **to** p_k
  while not *working* **and** *pending*.size() > 0 **do**
    $\langle m, p_j \rangle \leftarrow$ *pending*.dequeue()
    $\langle m', p_k \rangle \leftarrow \mathsf{nextState}(m, p_j)$
    $working \leftarrow (m'.type = \text{REQUEST})$
    send m' **to** p_k

Deadlock Detection through Snapshot

Approach

- All channels are based on FIFO delivery
- Add code to deal with Snapshot messages

Pros and Cons

- Generates overhead only when deadlock is suspected
- Introduces a delay between deadlock and detection

Server code, modified for active monitoring (1)

Process p_i

```
QUEUE pending ← new QUEUE()
boolean working ← false
boolean[] blocking ← {false, ..., false}
while true do
    while working or pending.size() = 0 do
        receive ⟨m, p_j⟩
        if m.type = REQUEST then
            blocking[j] ← true
            pending.enqueue(⟨m, p_j⟩)
        else if m.type = RESPONSE then
            ⟨m′, p_k⟩ ← nextState(m, p_j)
            working ← (m′.type = REQUEST)
            send m′ to p_k
            if m′.type = RESPONSE then
                blocking[k] ← false
```

Server code, modified for active monitoring (2)

Process p_i

```
    else if m.type = SNAPSHOT then
        if s = 0 then
            send ⟨SNAPSHOT, blocking⟩ to p_0
            send ⟨SNAPSHOT⟩ to Π − {p_i}
        s ← (s + 1) mod n
while not working and pending.size() > 0 do
    ⟨m, p_j⟩ ← pending.dequeue()
    ⟨m′, p_k⟩ ← nextState(m, p_j)
    working ← (m′.type = REQUEST)
    send m′ to p_k
    if m′.type = RESPONSE then
        blocking[k] ← false
```

Monitor code, for active monitoring

Process p_0

```
boolean[][] wfg ← new boolean[1...n][1...n]
while true do
    { Wait until deadlock is suspected }
    send ⟨SNAPSHOT⟩ to Π
    for k ← 1 to n do
        receive ⟨m, j⟩
        wfg[j] ← m.data
    if there is a cycle in wfg then
        { the system is deadlocked }
```

Deadlock detection through passive monitoring

Approach

- Sends a message to p_0 for each relevant event
- Communication with p_0 based on causal delivery

Pros and Cons

- Simpler approach, but complexity is hidden by the causal delivery mechanism
- Latency limited to message delays
- Higher overhead

Server code, modified for passive monitoring (1)

Process p_i

```
QUEUE pending ← new QUEUE()
boolean working ← false
while true do
    while working or pending.size() = 0 do
        receive ⟨m, p_j⟩
        if m.type = REQUEST then
            send ⟨REQUESTED, j, i⟩ to p_0
            pending.enqueue(⟨m, p_j⟩)
        else if m.type = RESPONSE then
            ⟨m′, p_k⟩ ← nextState(m, p_j)
            working ← (m′.type = REQUEST)
            send m′ to p_k
            if m′.type = RESPONSE then
                send ⟨RESPONDED, i, k⟩ to p_0
```

Server code, modified for passive monitoring (2)

Process p_i

```
while not working and pending.size() > 0 do
    ⟨m, p_j⟩ ← pending.dequeue()
    ⟨m′, p_k⟩ ← nextState(m, p_j)
    working ← (m′.type = REQUEST)
    send m′ to p_k
    if m′.type = RESPONSE then
        send ⟨RESPONDED, i, k⟩ to p_0
```

Monitor code, for passive monitoring

Process p_0

boolean[ ][ ] *wfg* ← **new boolean**$[1 \ldots n][1 \ldots n]$
while true do
  receive $\langle m, p_j \rangle$
  if $m.type =$ RESPONDED **then**
    $wfg[m.from, m.to] \leftarrow$ **false**
  else
    $wfg[m.from, m.to] \leftarrow$ **true**
  if there is a cycle in *wfg* **then**
    { the system is deadlocked }

Contents

Non-stable predicates

Problems of non-stable predicates

- The condition encoded by the predicate may not persist long enough for it to be true when the predicate is evaluated
- If a predicate Φ is found to be true by the monitor, we do not know whether Φ *ever* held during the *actual* run.

Conclusions

- Evaluating a non-stable predicate over a single computation makes no sense
- The evaluation must be extended to the entire lattice of the computation

Predicates over entire computations

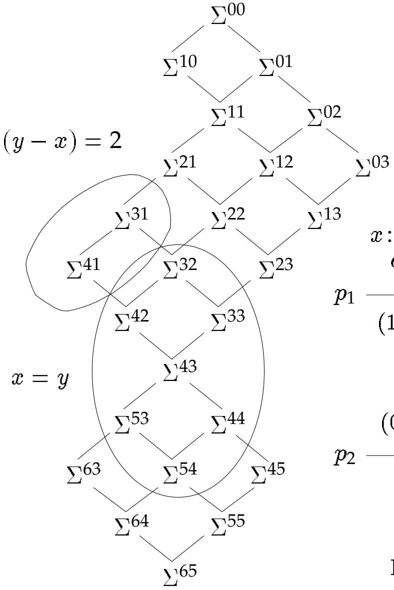
It is possible to evaluate a predicate over an entire computation using an observation obtained by a passive monitor.

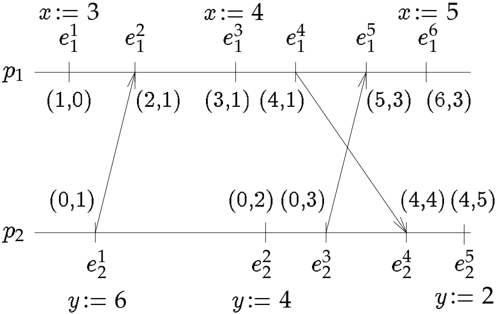
- **Possibly**(Φ): There exists a consistent observation O of the computation such that Φ holds in a global state of O.
- **Definitely**(Φ): For every consistent observation O of the computation, there exists a global state of O in which Φ holds.

Example (Debugging)

If **Possibly**(Φ) is true, and Φ identifies some erroneous state of the computation, than there is a bug, even if it is not observed during an actual run.

Example: **Possibly**$((y - x) = 2)$, **Definitely**$(x = y)$





Initially $x = 0$ and $y = 10$

Predicates over entire computations

Theorem

Possibly *and* **Definitely** *are not duals:*

$$\neg\textbf{Possibly}(\Phi) \not\Leftrightarrow \textbf{Definitely}(\neg\Phi)$$
$$\neg\textbf{Definitely}(\Phi) \not\Leftrightarrow \textbf{Possibly}(\neg\Phi)$$

Example

Possibly$(x \neq y)$, **Definitely**$(x = y)$

Algorithms for detecting **Possibly** and **Definitely**

- We use the passive approach in which processes send notifications of events relevant to Φ to the monitor p_0;
- Events are tagged with vector clocks;
- The monitor collects all the events and builds the lattice of global states.
  How?
- To detect **Possibly**(Φ): if there exists one global state in which Φ is true, then return **true**, otherwise **false**.
- To detect **Definitely**(Φ): mark nodes where Φ is true with a value 1, the other nodes with value 0. If the cost of the shortest path between the initial state and the final state is larger than 0, return **true**, otherwise **false**.

Algorithms for detecting **Possibly** and **Definitely**

Problems

- The number of states grows exponentially with the number of total events.
- Techniques can be used to reduce the number of events
  - Only those relevant to Φ
  - Forcing periodic synchronization
  - Reducing the complexity of predicates (conjunction of local predicates)

Reading Material

- O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms.

  In S. Mullender, editor, *Distributed Systems (2^{nd} ed.)*. Addison-Wesley, 1993.

  http://www.disi.unitn.it/~montreso/ds/papers/ConsistentGlobalStates.pdf

Reality Check: Interesting links

- Clocks are bad, or welcome to distributed systems
- Why vector clocks are easy
- Why vector clocks are hard
- Why Cassandra doesn't need vector clocks