Consistent Global States

Sape Mullender

Huygens Systems Research Laboratory
Universiteit Twente
Enschede

Introduction

Suppose we have an active *distributed computation* and we want to ask questions of the sort

- Is the system deadlocked?
- How many processes are reading a particular file?
- What is the balance of the bank?

To resolve these questions, we must send messages to all participating processes. The problem is that, while the question is being answered, these processes continue to exchange messages.

Asynchronous System

Let's investigate what we can say about the *global state* of a computation in an *asynchronous system*.

A collection of sequential *processes*, $p_1, p_2, ..., p_n$, and a network of *communication channels* between pairs of processes.

Communication is reliable but incurs potentially unbounded delays. One process may be arbitrarily faster or slower than another.

Distributed Computation

A distributed computation is a *single execution* of a *distributed program* by a collection of *processes*. Each sequential process generates a sequence of *events* that are either *internal events*, or *communication events*

The *local history* of process p_i during a computation is a (possibly infinite) sequence of events $h_i = e_i^1, e_i^2, ...$

A *partial local history* of a process is a prefix of the local history $h_i^n = e_i^1, e_i^2, ..., e_i^n$

The *global history* of a computation is the set $H = \bigcup_{i=1}^{n} h_i$

State of a Computation

Imagine stopping a distributed computation by stopping all of its processes *simultaneously*. The combined states of each of the processes, plus the contents of the messages in transit between processes will then tell us the exact *global state* of the computation.

With this global state, we can reconstruct the balance of the distributed banking system, or we can tell whether or not a system is deadlocked.

The problem with our asynchronous system is that there is no such thing as *simultaneity*.

Cause and Effect

Since there is no notion of *simultaneity* in an asynchronous system, we need something else.

We do have a notion of *cause and effect*, however. If one event e_i *caused* another event e_j to happen, then e_i and e_j could never have happened simultaneously. e_i *happened before* e_j.

When we cannot not look *inside* a distributed computation, but can only observe its communication, we cannot tell whether one event *causes* another, but only whether it *could have caused* another

Happened Before

We define a binary relation $\rightarrow$ over events, such that

1. If $e_i^k, e_i^l \in h_i$ and $k < l$, then $e_i^k \rightarrow e_i^l$
2. If $e_i = \text{send}(m)$ and $e_j = \text{receive}(m)$, then $e_i \rightarrow e_j$
3. If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

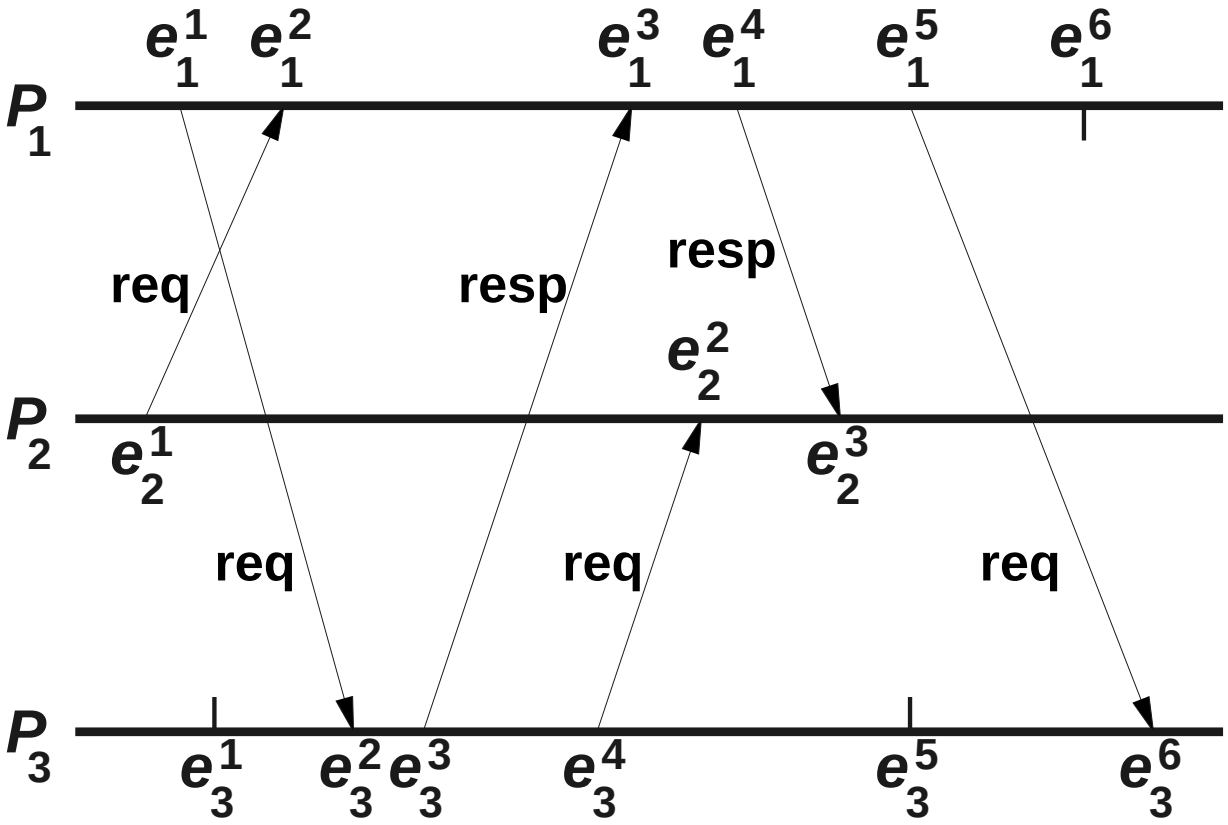
When $e \rightarrow e'$, we say e *causally precedes* e' or e *happened before* e'.

We define *concurrent* as $e \parallel e' \equiv \neg(e \rightarrow e' \vee e' \rightarrow e)$

Distributed Computation

A distributed computation is a *partially ordered set* (poset) $\mathcal{Y} = (H, \rightarrow)$.

A distributed computation can be depicted in a *space-time diagram*:

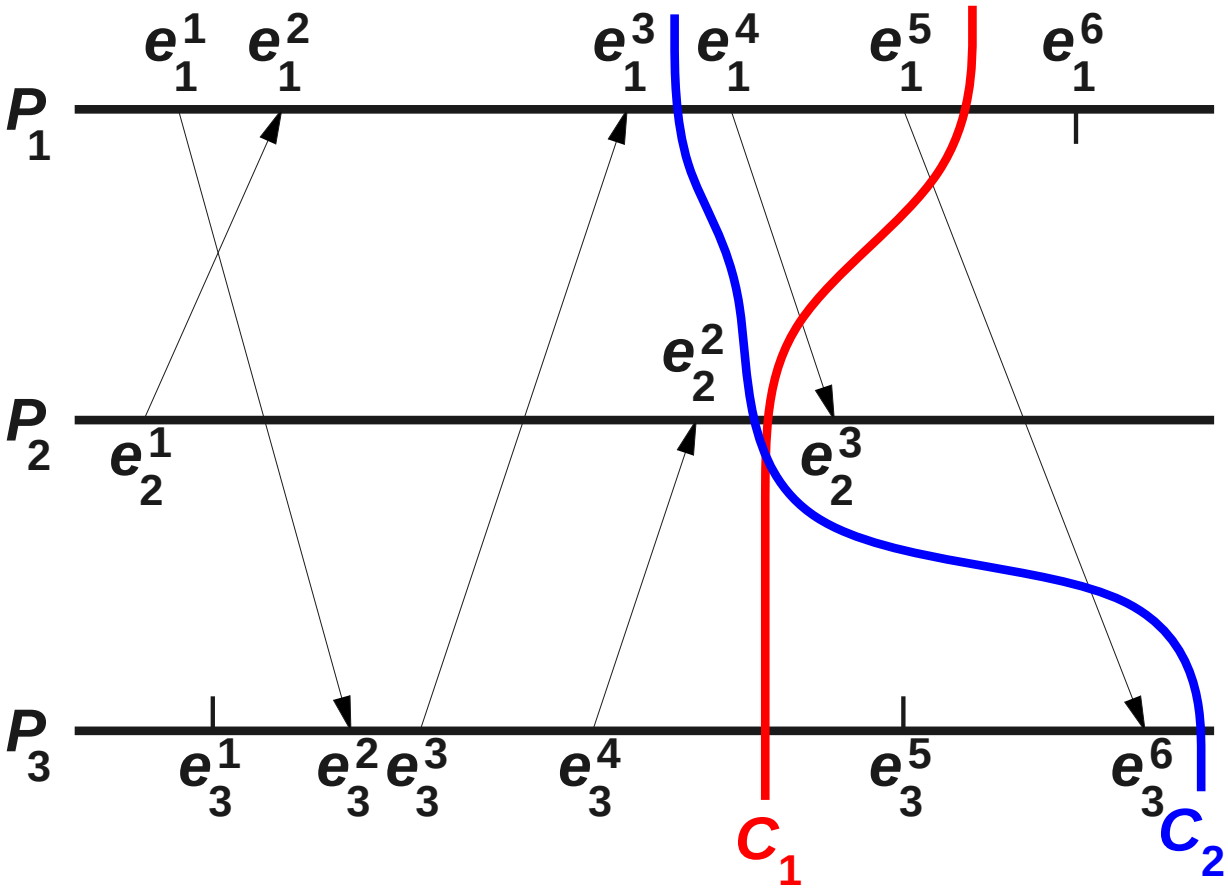

e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
P_1
req
resp
resp
e_2^2
P_2
e_2^1
e_2^3
req
req
req
P_3
e_3^1 e_3^2 e_3^3 e_3^4 e_3^5 e_3^6

Global States

The local state of process p_i after executing event e_i^k is denoted σ_i^k. The initial local state of p_i is σ_i^0.

A *global state* is a collection of local states $\Sigma = \bigcup_{i=1}^{n} \sigma_i$.

Cuts


e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
P_1
e_2^2
P_2
e_2^1 e_2^3
P_3
e_3^1 e_3^2 e_3^3 e_3^4 e_3^5 e_3^6
C_1
C_2

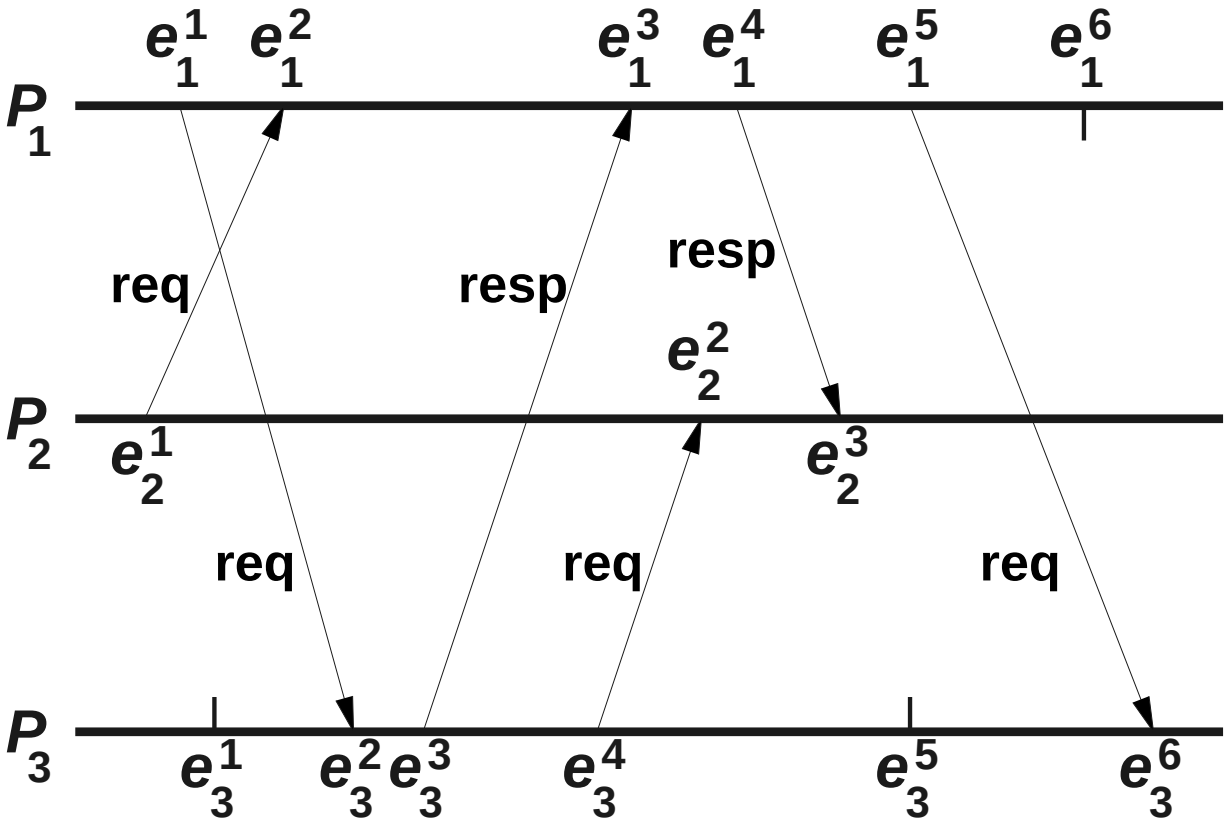
A *cut* C of a distributed computation is a collection of partial local histories: $C = \bigcup_{i=i}^{n} h_i^{c_i}$.

The set of last events $e_i^{c_i}$ $(i = 1, ..., n)$ included in the cut is called the *frontier*

Runs

A *run* of a distributed computation is a *total ordering* R of its events that corresponds to an *actual execution*

$$R = e_3^1 e_1^1 e_3^2 e_2^1 e_3^3 e_3^4 e_2^2 e_1^2 e_3^5 e_1^3 e_1^4 e_1^5 e_3^6 e_2^3 e_1^6$$


e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
P_1
req
resp
resp
e_2^2
P_2
e_2^1
e_2^3
req
req
req
P_3
e_3^1 e_3^2 e_3^3 e_3^4 e_3^5 e_3^6

Observations

An observation is a total ordering Ω of events constructed from within the system. A single run may have many observations:

$$R = e_3^1\, e_1^1\, e_3^2\, e_2^1\, e_3^3\, e_3^4\, e_2^2\, e_1^2\, e_3^5\, e_1^3\, e_1^4\, e_1^5\, e_3^6\, e_2^3\, e_1^6$$

$$O_1 = e_2^1\, e_1^1\, e_3^1\, e_3^2\, e_3^4\, e_1^2\, e_2^2\, e_3^3\, e_1^3\, e_1^4\, e_3^5 \ldots$$

$$O_2 = e_1^1\, e_3^1\, e_2^1\, e_3^2\, e_1^2\, e_3^3\, e_3^4\, e_1^3\, e_2^2\, e_3^5\, e_3^6 \ldots$$

$$O_3 = e_3^1\, e_2^1\, e_1^1\, e_1^2\, e_3^2\, e_3^3\, e_1^3\, e_3^4\, e_1^4\, e_2^2\, e_1^5 \ldots$$

Monitoring Distributed Computations

Observing a distributed computation from within is called *monitoring*.

We introduce an extra process P_0, the *monitor*. It 'interrogates' the other processes by sending them messages. From the replies received, the monitor constructs a global state of the computation.

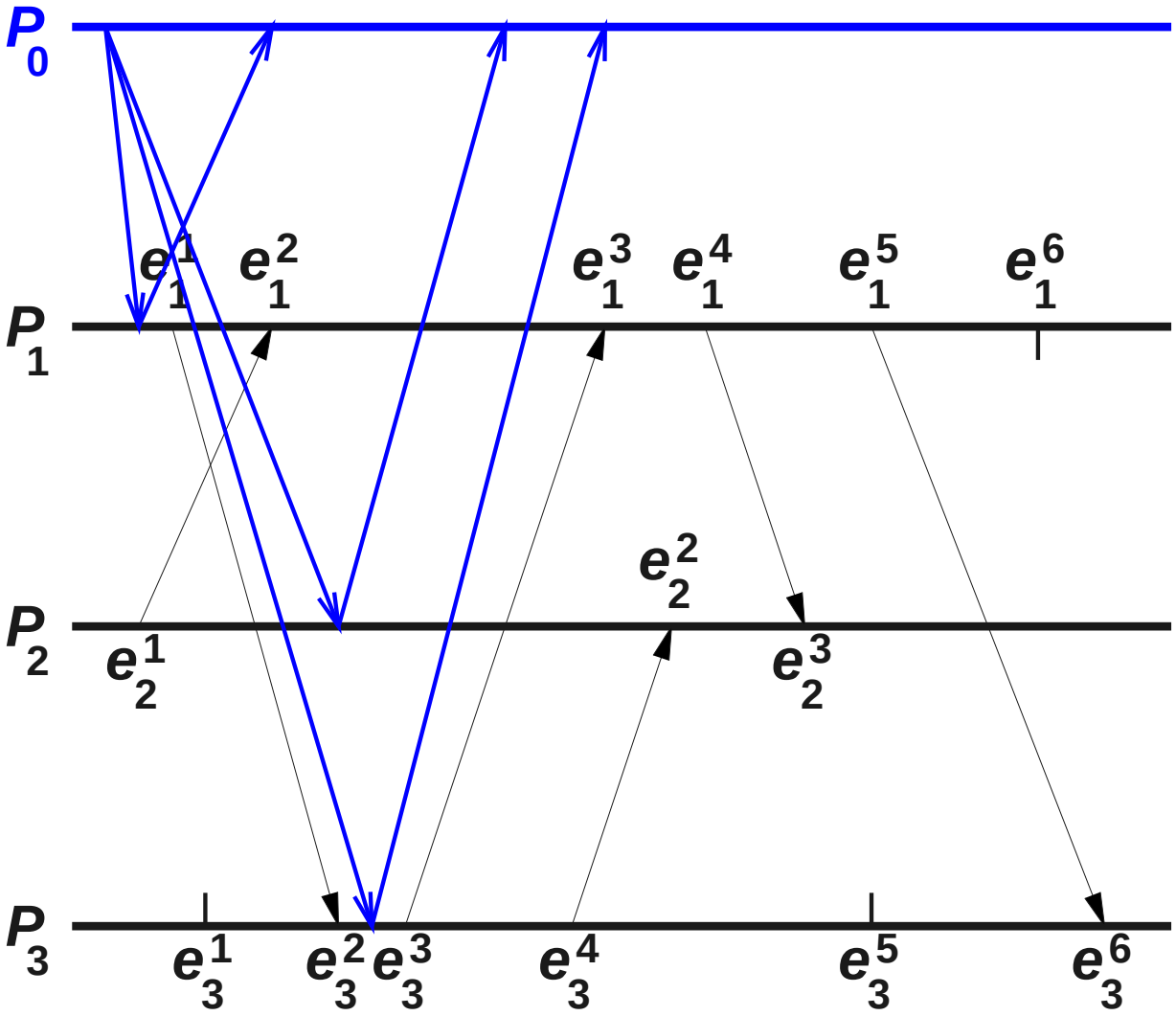
The thousand-dollar question is whether the state constructed by the monitor is a state that actually occurred in the distributed computation.

Remember, we're monitoring an asynchronous system from within!

Naive Algorithm #1 ...

- p_0 sends a message to each process asking a state report.
- When p_i receives such a message, it replies with its current state σ_i.
- When all n processes have replied, p_0 constructs the global state (σ_1, σ_2, ... σ_n).

… and Why it Doesn't Work



Consistency

From within an asynchronous system, in general, there is no way to tell whether a particular global state ever occurs.

Instead, we can define a *consistent state* as one that *could* have occurred.

A cut C is consistent if $\forall e, e' : \ (e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$

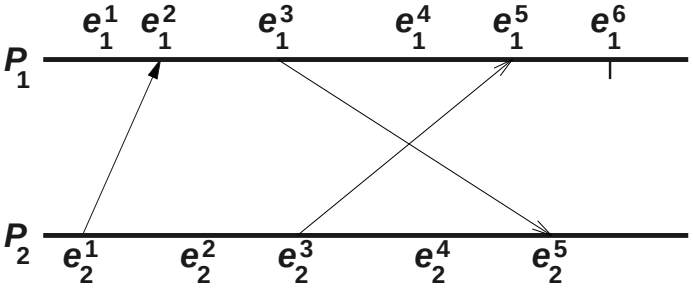
More consistency

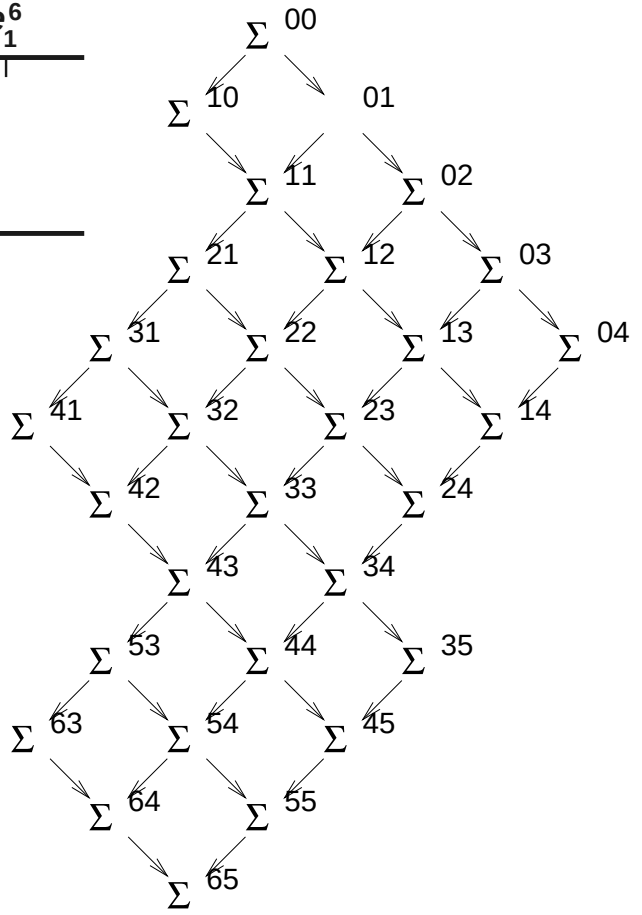
A *consistent global state* is one that corresponds to a consistent cut.

A *consistent observation* is a total ordering of the events in H that is consistent with the partial order defined by causal precedence.

A consistent observation $\Omega = e^1, e^2, \ldots$ corresponds to a sequence of global states $\Sigma^0, \Sigma^1, \ldots$ where Σ^i is derived from Σ^{i-1} by executing the single event e^i.

Lattice of Global States





Constructing Observations

- Suppose that each process $p_1, p_2, ...$, when it executes an event, also sends a notification (as an extra event, as it were) to the monitor, p_0.
- When the monitor receives a notification, it adds it to the observation being constructed:
- For each event of p_i, the monitor updates its local state σ_i.
- At any time, the global state is constructed as the n-tuple of latest local states.

Why doesn't that work?

Message delays are arbitrary, so p_0 can receive an arbitrary permutation of the messages produced by the p_i.

So what can we do to construct *consistent* observations?

Reception and Delivery

When the monitor receives a notification of an event, it should not immediately *act* on it. It should first check if there aren't any other messages still under way that should be acted on first.

We distinguish between *receiving* a message and *delivering* one: When the message arrives it is received. Then a *delivery rule* decides when a message can be delivered (acted upon).

FIFO

Communication from process p_i to p_j is *First-in-First-Out* (FIFO) if for all message m and m':

$$\text{send}_i(m) \rightarrow \text{send}_i(m') \Rightarrow \text{deliver}_j(m) \rightarrow \text{deliver}_j(m')$$

FIFO is trivial to implement (sequence numbers) but FIFO alone is not sufficient to guarantee the consistency of observations.

Synchronous System: Clocks

Assume that all processes have access to a global *real-time clock* RC and that all message delays are bounded by δ. Every notification message m for an event e contains a *timestamp* $TS(m) = RC(e)$

DR1: At time t, deliver all received messages with timestamps up to $t - \delta$ in increasing timestamp order.

Clock Condition: $e \rightarrow e' \;\Rightarrow\; RC(e) < RC(e')$

The observations constructed this way are consistent because the *clock condition* is satisfied. Real-time clocks satisfy the clock condition.

Back to Asynchronous Systems

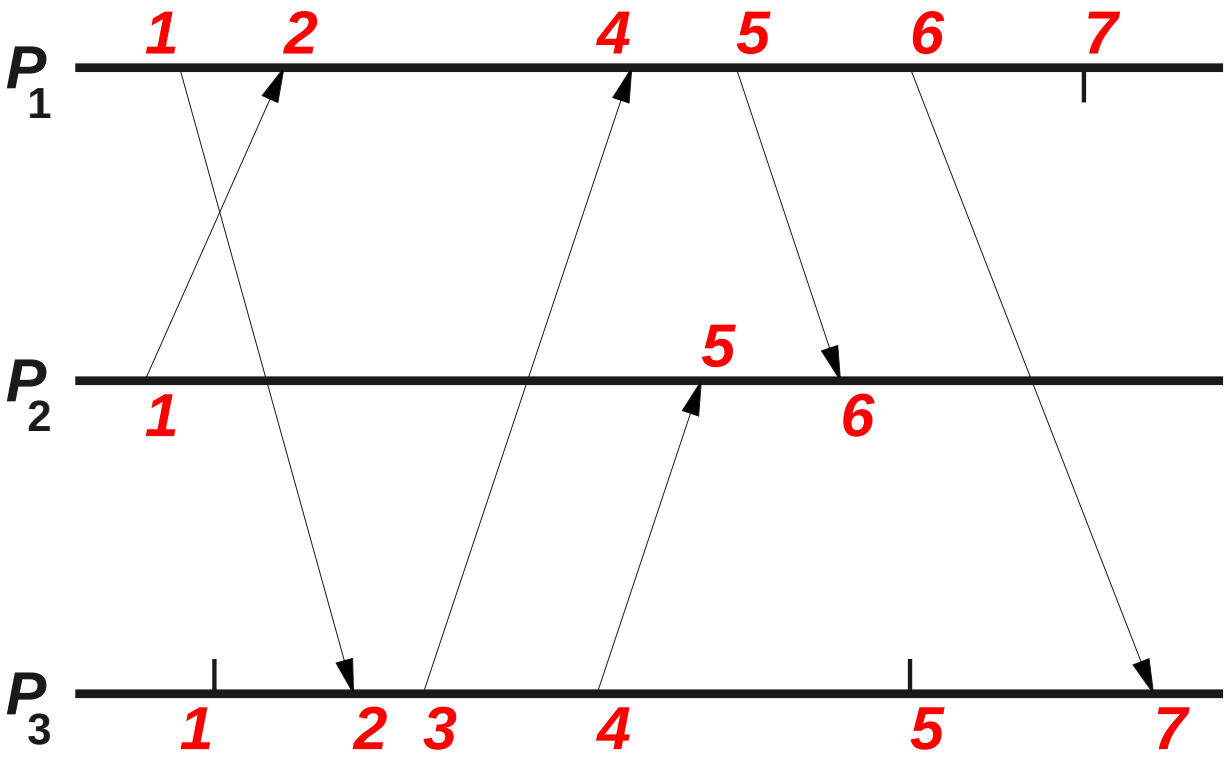
We invent a clock in an asynchronous system that satisfies the clock condition.

Each process maintains a local positive-integer variable LC, the process' *logical clock*. $LC(e_i)$ is the value of p_i's logical clock when executing event e_i.

Update rules:

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

Logical Clock



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

Gap Detection

Trying to deliver messages in increasing logical-time-stamp order presents a problem:

We cannot deliver a message with $TS = t$ unless we are certain that no message with $TS < t$ can be received.

We need to be able to do *gap detection*:

Gap Detection: Given two events e and e' and their clock values $LC(e)$ and $LC(e')$, and $LC(e) < LC(e')$, determine whether an event e'' exists such that $LC(e) < LC(e'') < LC(e')$.

Stable Messages

A message m received by process p is *stable* at p if no future messages with timestamps smaller than $TS(m)$ can be received by p.

DR2: Deliver all received messages that are stable at p_0 in increasing time-stamp order.

Stability

When communication between processes p_0 and p_i is FIFO, and p_0 receives m from p_i with $TS(m)$, p_0 cannot later receive a message m' with $TS(m') < TS(m)$

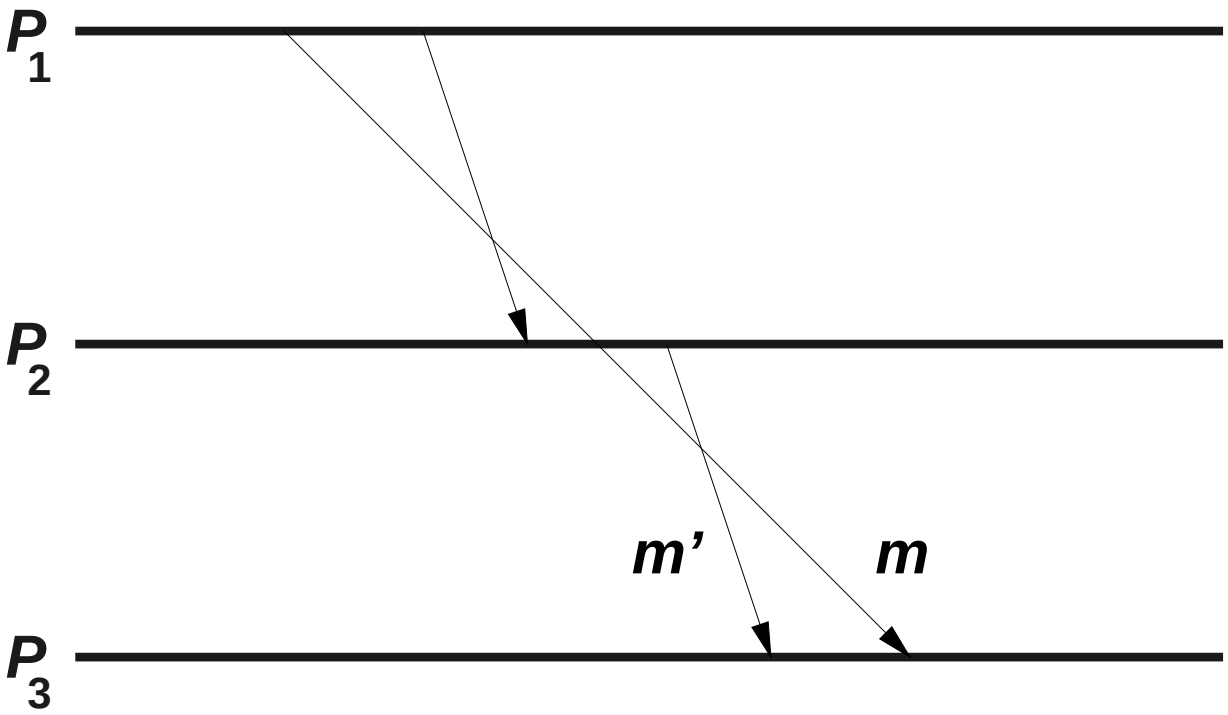
Stability of m at p_0 is guaranteed when p_0 has received a message with timestamp greater than $TS(m)$ from *all* other processes.

Causal Order

Generalization of FIFO gives *Causal Order*:

$$\text{send}_i(m) \rightarrow \text{send}_j(m') \Rightarrow \text{deliver}_k(m) \rightarrow \text{deliver}_k(m')$$

FIFO order between all processes is not enough to guarantee causal order


P1
P2
P3
m'
m

Strong Clock Condition

We know that, with a logical clock or a real-time clock, we have

Clock Condition: $e \rightarrow e' \;\Rightarrow\; RC(e) < RC(e')$

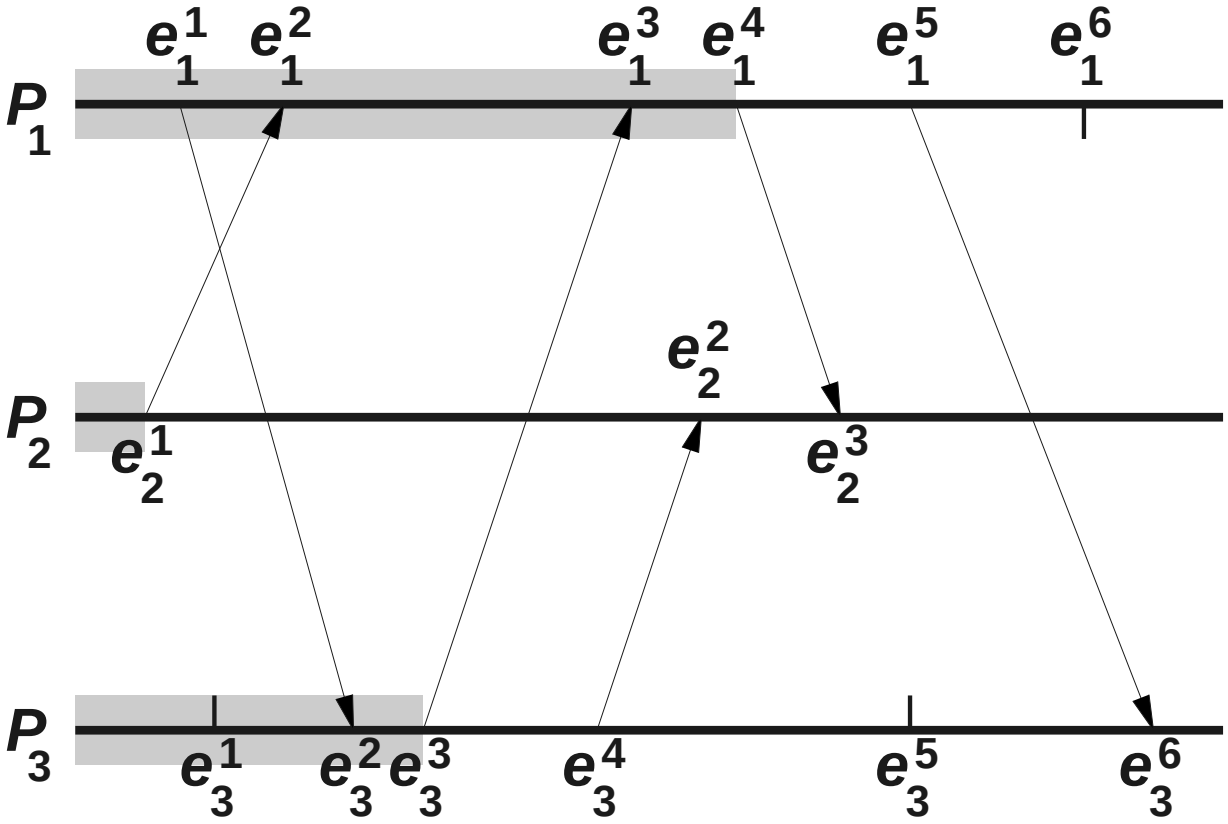
Delivering messages in time-stamp order may be overly restrictive, because it is possible that $RC(e) < RC(e')$, but $e \not\rightarrow e'$. It would be nicer to have a clock that satisfies the

Strong Clock Condition: $e \rightarrow e' \;\equiv\; RC(e) < RC(e')$

Causal History

The *causal history* of event e in a distributed computation $(H, \rightarrow)$ is the set

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



Causal Histories

Causal histories can be used to implement the strong clock condition:

$$e \rightarrow e' \equiv \theta(e) \subset \theta(e')$$

Or, if $e \neq e'$:

$$e \rightarrow e' \equiv e \in \theta(e')$$

The *projection* of $\theta(e)$ on process p is the set $\theta_i(e) = \theta(e) \cap h_i$.

$$e_i^k \in \theta_i(e) \Rightarrow \forall\, j \leq k : e_i^j \in \theta_i(e)$$

Vector Clocks

$\exists k_i : \theta_i(e) = h_i^{k_i}$. k_i represents θ_i.

$\theta(e) = \bigcup_{i=1}^{n} \theta_i(e)$, so the entire causal history of event e can be represented by an n-dimensional vector of $k_{1,\ldots,n}$, the *vector clock*:

$$VC(e)[i] = k \Leftrightarrow \theta_i(e) = h_i^k$$

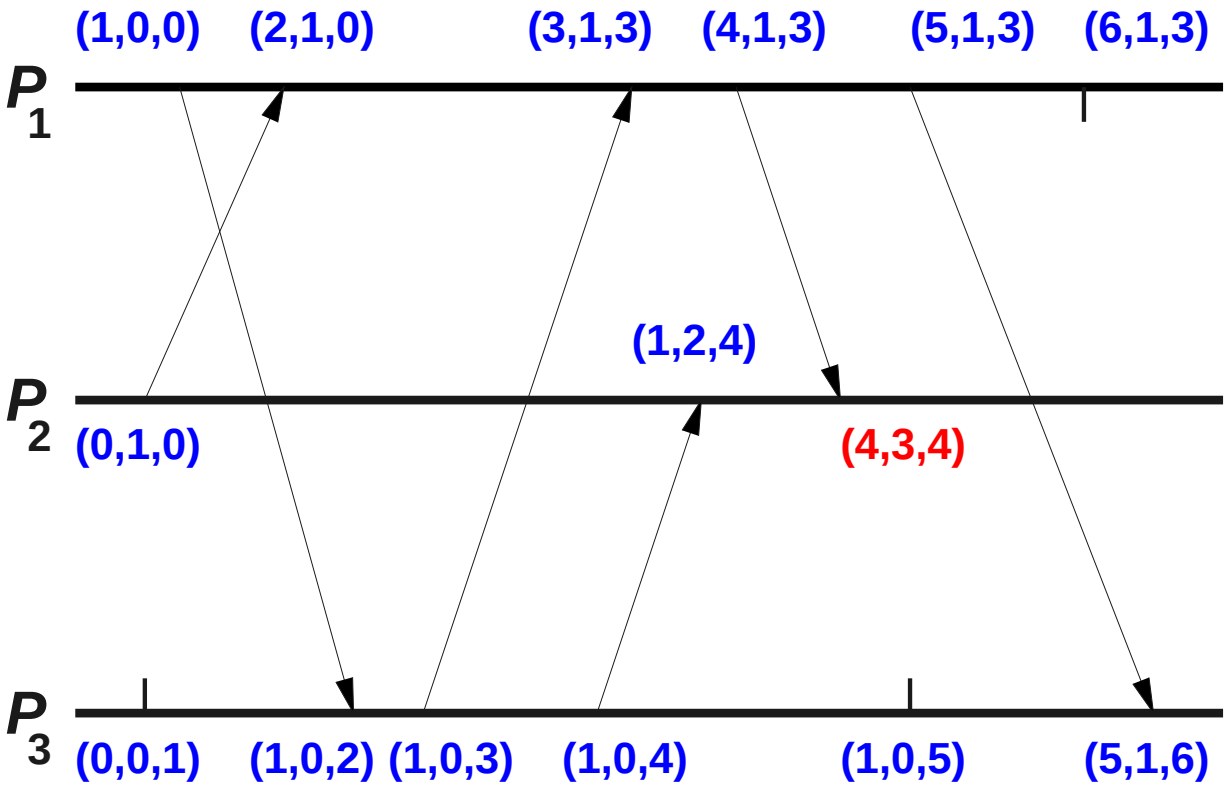
Update Rules

$$VC(e_i)[i] := VC[i] + 1 \qquad e_i \text{ internal or send event}$$

$$VC(e_i) := \max\{VC, TS(m)\};$$

$$VC(e_i)[i] := VC[i] + 1 \qquad e_i \text{ receive}(m)$$

Vector Clocks



Operational Interpretation

$VC(e_i)[i]$ $\equiv$ number of events p_i has executed up to and including e_i.

$VC(e_i)[j]$ $\equiv$ number of events of p_j that causally precede event e_i of p_j $(j \neq i)$

Properties of Vector Clocks

Less than relationship:

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

Property 1: (Strong Clock Condition)

$$e \rightarrow e' \equiv VC(e) < VC(e').$$

Property 2: (Simple Strong Clock Condition.) Given event e_i in p_i and e_j in p_j, and $i \neq j$:

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

Properties of Vector Clocks

Events e_i and e_j are *pairwise inconsistent* if they cannot belong to the frontier of the same consistent cut.

Property 3: (Concurrency.) Given event e_i in p_i and e_j in p_j:

$$e_i \parallel e_j \equiv \quad (VC(e_i)[i] > VC(e_j)[i]) \wedge$$
$$(VC(e_j)[j] > VC(e_i)[j]).$$

Property 4: (Pairwise Inconsistent.) Event e_i of p_i is pairwise inconsistent with event e_j of p_j $(i \neq j)$, if and only if

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

Properties of Vector Clocks

Property 5: (Consistent Cut.) A cut defined by $(c_1, ..., c_n)$ is consistent if and only if:

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

Property 6: (Weak Gap Detection) Given event e_i in p_i and e_j in p_j,

$$k \neq j \wedge VC(e_i)[k] < VC(e_j)[k] \Rightarrow \neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j)$$

Implementing Causal Delivery

- Processes increment the local component of their vector clocks only for events that are notified to the monitor.
- Each message m carries a timestamp $TS(m)$ which is the vector clock value of the event being notified by m.
- All messages received but not delivered by p_0 are maintained in a set $\mathcal{M}$, initially empty.
- Process p_0 maintains an array $D[1...n]$ of counters, initially all zeroes, such that $D[i] = TS(M_i)[i]$, where m is the last message delivered from p_i.

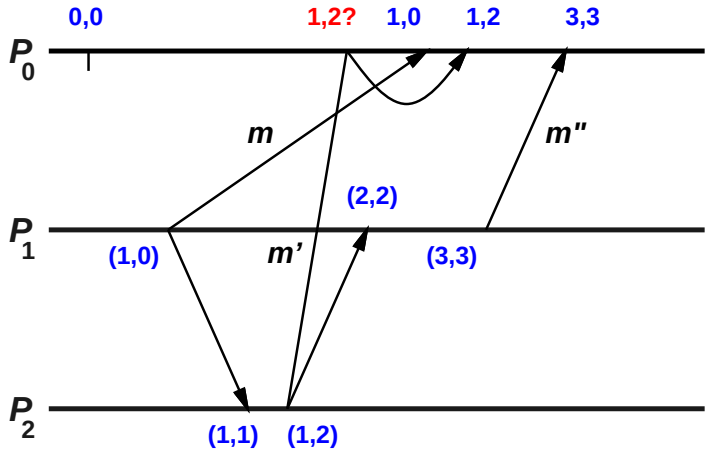
Implementing Causal Delivery

DR3: (Causal Delivery.) Deliver message m from p_j when both

$$D[j] = TS(m)[j] - 1$$

$$D[k] \geq TS(m)[k], \forall k \neq j$$

are satisfied. When p_0 delivers m, it sets $D[j]$ to $TS(m)[j]$.



Distributed Snapshots

When the monitor takes snapshots of the local states in an uncoordinated manner, the resulting global state is not guaranteed to be consistent.

We shall develop a snapshot protocol that only constructs consistent global states.

Assumptions

The channels preserve FIFO order.

The *state of a channel*, $\chi_{i,j}$, from p_i to p_j are those messages that p_i has sent to p_j but p_j has not yet received.

The set of channels from any process to p_i is designated IN_i (*incoming* with respect to p_i). The set of channels from p_i to other processes is OUT_i (*outgoing* with respect to p_i).

Each process p_i will record its local state σ_i and the state of its incoming channels $\chi_{j,i}, \forall p_j \in IN_i$.

Snapshots Protocol Nr. 1

Assume that all processes have access to a global real-time clock RC, that all message delays are bounded and that relative processing speeds are bounded.

1. Process p_0 sends a message *'take snapshot at* t_{ss}*'* to all processes, where t_{ss} is far enough in the future.
2. When RC reads t_{ss}, each p_i records its local state σ_i, sends an empty message over all of its outgoing channels and starts recording messages on its incoming channels. While this is going on, p_i does not execute any events.
3. When p_i receives a message with timestamp $\geq t_{ss}$, p_i stops recording and sends the $\chi_{j,i}$ to p_0.

Snapshots Protocol Nr. 2

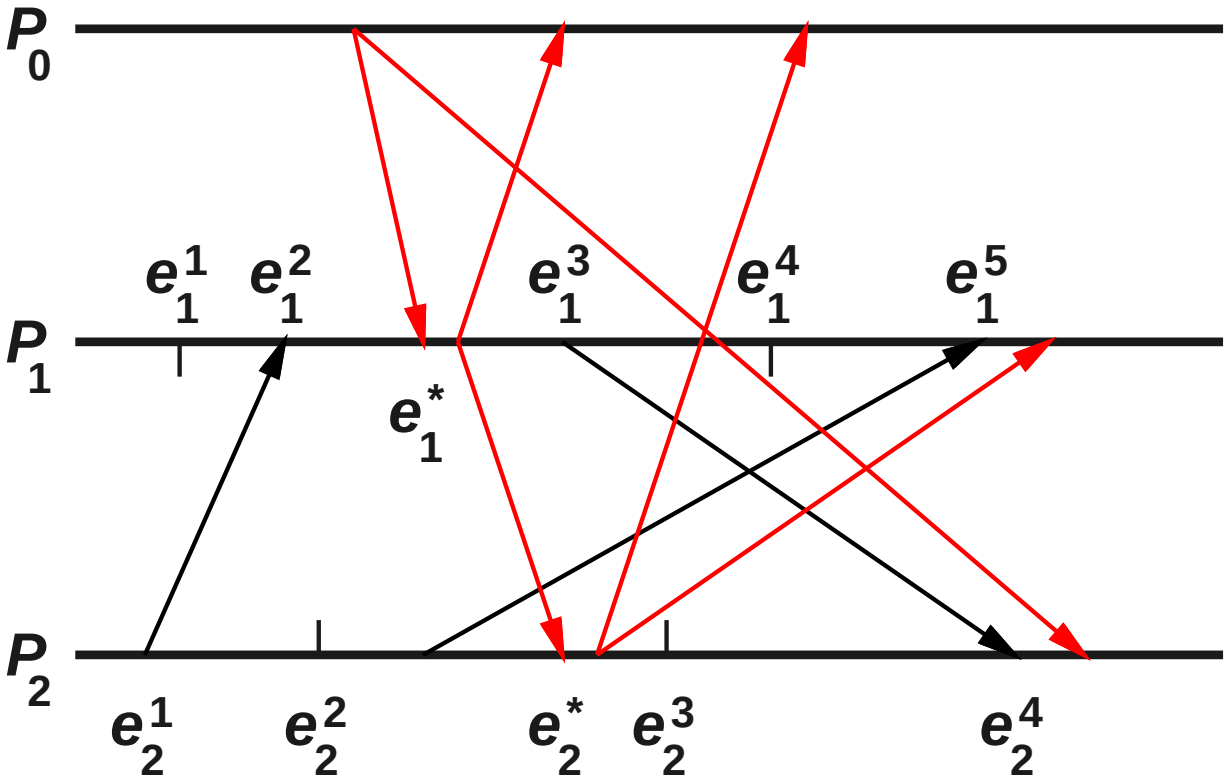
Substitute a logical clock for the real-time clock.

Snapshots Protocol Nr. 3

Due to Chandy and Lamport (1985).

1. p_0 sends itself a *take snapshot* message.
2. When p_i receives the first marker (from p_f, say), it records its local state σ_i and forwards the marker on all its outgoing channels. $\chi_{f,i}$ is set to empty and p_i starts recording messages received on the other incoming channels.
3. When p_i receives another marker (from p_s, say), it declares $\chi_{s,i}$ to be the set of recorded messages from p_s.

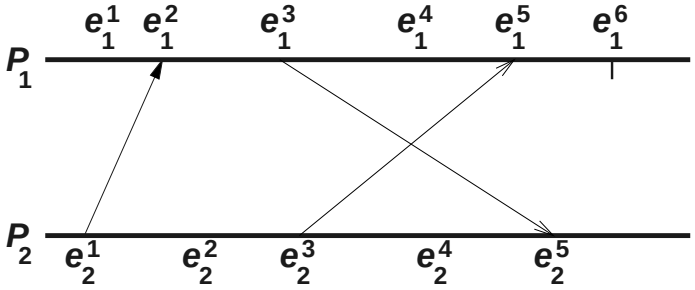
Chandy and Lamport Demo

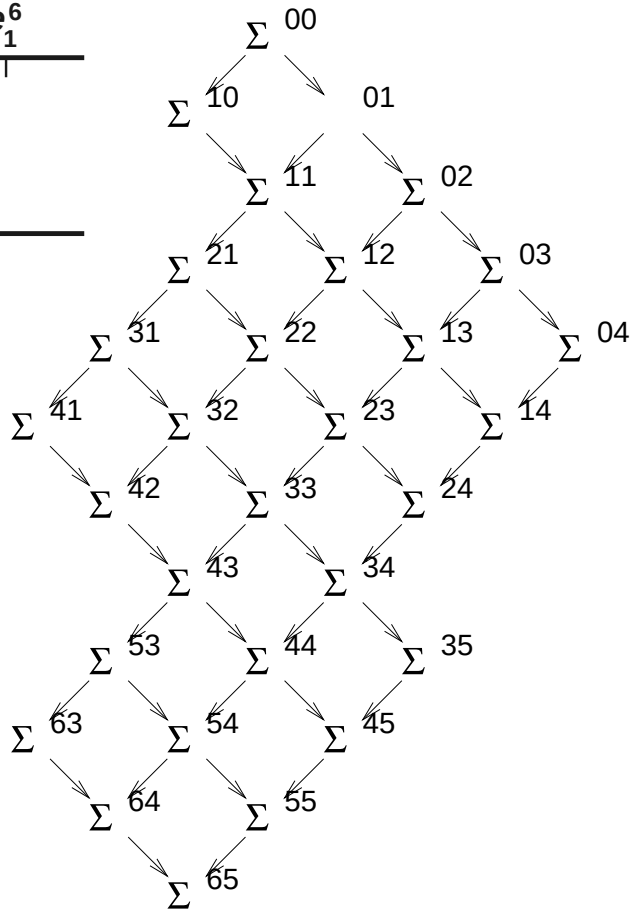


The constructed global state is Σ^{23} with $\chi_{1,2} = \emptyset$ and $\chi_{1,2} = \{m\}$.

Properties of Snapshots

Protocol starts in state Σ^{21} and terminates in Σ^{55} of the actual run $R = \Sigma^{00}\,\Sigma^{01}\,\Sigma^{11}\,\Sigma^{21}\,\Sigma^{31}\,\Sigma^{32}\,\Sigma^{42}\,\Sigma^{43}\,\Sigma^{44}\,\Sigma^{54}\,\Sigma^{55}\,\Sigma^{65}$





Distributed Snapshots

In the example, the run does not pass through the constructed global state Σ^{23}, but $\Sigma^{21} \rightsquigarrow \Sigma^{23} \rightsquigarrow \Sigma^{55}$

This result holds in general. If

$$\Sigma^i \equiv \text{global state in which the protocol is initiated}$$

$$\Sigma^f \equiv \text{global state in which the protocol terminates}$$

$$\Sigma^s \equiv \text{global state constructed}$$

Then $\Sigma^i \rightsquigarrow \Sigma^s \rightsquigarrow \Sigma^f$

Global Properties

If a property ϕ holds in the constructed state Σ^s, by the time this state has been constructed, ϕ may no longer hold.

However, some properties exist that remain true from the moment they first hold. *Deadlock* is a fine example. By definition, once a system is deadlocked, it remains deadlocked.

Stable Predicates

Definition. Predicate ϕ is *stable* in computation γ if and only if

$$\forall \Sigma, \Sigma' \in \gamma : \phi(\Sigma) \wedge (\Sigma \sim\!\!\rightarrow \Sigma') \Rightarrow \phi(\Sigma')$$

Let Σ^i, Σ^s, and Σ^f be as before. For stable predicate ϕ,

$$\begin{aligned} \phi(\Sigma^s) &\Rightarrow \phi(\Sigma^f) \\ \neg\phi(\Sigma^s) &\Rightarrow \neg\phi(\Sigma^i) \end{aligned}$$

Making non-stable predicates stable

Definition.

1. Distributed computation γ satisfies **Pos** ϕ, denoted $\gamma \vdash$ **Pos**ϕ, if and only if there exists an observation Ω of γ such that $\Omega \vdash \phi$. ∎
2. Distributed computation γ satisfies **Def** ϕ, denoted $\gamma \vdash$ **Def**ϕ, if and only if for all observations Ω of γ it is the case that $\Omega \vdash \phi$.