Distributed Systems (5DV147)

Time and Global States

Fall 2014

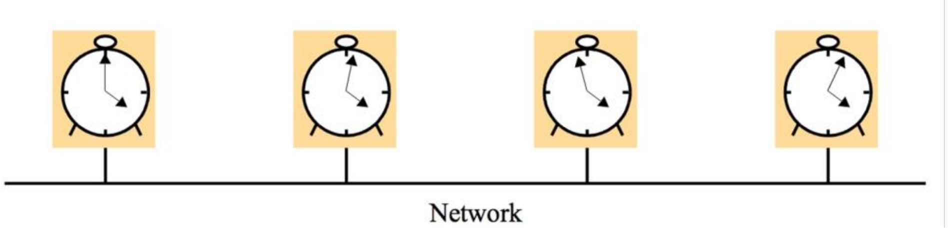
Time and the lack thereof

Motivation examples

- ❑ Replication
  - ➢ Updates applied in the same order at all sites
- ❑ Monitoring
  - ➢ all processes receive notification events in the same order
- ❑ Allocation of share resources
  - ➢ Fairness in processing requests

Why do we not have global time?

- Clocks drift, are inaccurate, may fail arbitrarily, etc.



A global notion of a correct time would be tremendously useful

Why is this a problem?

- ❑What does it mean that one event occurs after another one?
- ❑How can we know if events are concurrent if we can't compare when they happened?

… but, perhaps, all we need is that all nodes agree on a form of time

- ❑…or, at least, agree on the order in which events occur
- ❑Not a global time but a global clock

Logical time and logical clocks

Motivation

- Difficult to have a single global time
- What can we do? Let’s consider one processes:

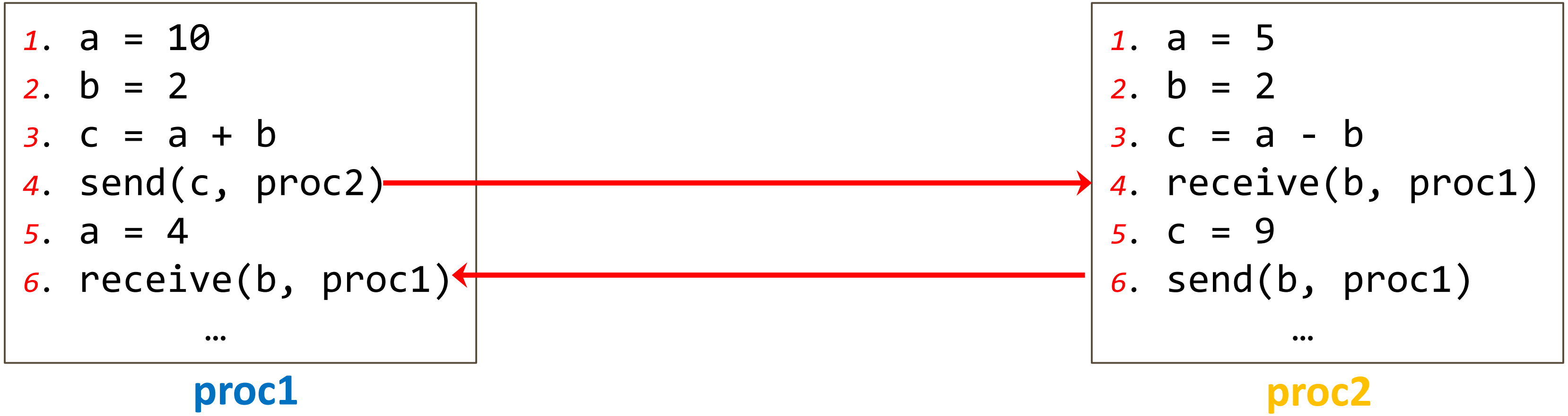
```
1. a = 10
2. b = 2
3. c = a + b
4. send(c, proc2)
5. a = 4
      …
i. receive(b, proc1)
      …
```

proc1

- What can we say about the order in which these operations are executed?

(1, 2, 3, 4, 5, …, i, …)

Now for two processes …



proc1

```
1. a = 10
2. b = 2
3. c = a + b
4. send(c, proc2)
5. a = 4
6. receive(b, proc1)
         …
```

proc2

```
1. a = 5
2. b = 2
3. c = a - b
4. receive(b, proc1)
5. c = 9
6. send(b, proc1)
         …
```

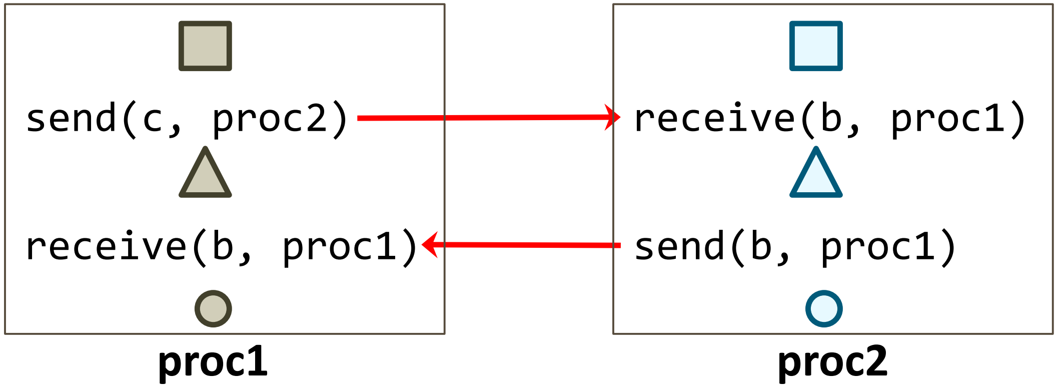
What can we say about the combined order of execution?

What can we say about **proc1**.3 and **proc2**.2?

What can we say about **proc1**.4 and **proc2**.4?

What can we say about **proc1**.6 and **proc2**.6?

Now for two processes …



send(c, proc2)

receive(b, proc1)

proc1

receive(b, proc1)

send(b, proc1)

proc2

proc1.send proc2.receive proc2.send proc1.receive

… we can say something about the order of some operations

What do we know now?

- ☐ We know the order of events occurring at the same process
- ☐ We know something about *send* and *receive* events
  - ➢ *send* causes a *receive*
  - ➢ *receive* is the effect of *send*
- ☐ Cause and effect may not be violated
  - ➢ An effect cannot be observed before the cause
  - ➢ *send* operations must always come before *receive* operations

Let’s be more formal

Let’s consider a distributed system P, of N processes:

p_i, $i = 1, 2, \ldots, N$

Each process has state s_i

Three type of events e can occur at each p_i :

Internal events, send events, receive events

Events are ordered within a process by the relation $\rightarrow_i$

$$e^0 \rightarrow_i e^1 \rightarrow_i e^2$$

Events define a history of p_i as described by $\rightarrow_i$

$$history(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$$

Happened-before relation “→”

HB1: If there exists a process p_i: $e \rightarrow_i e'$, then $e \rightarrow e'$

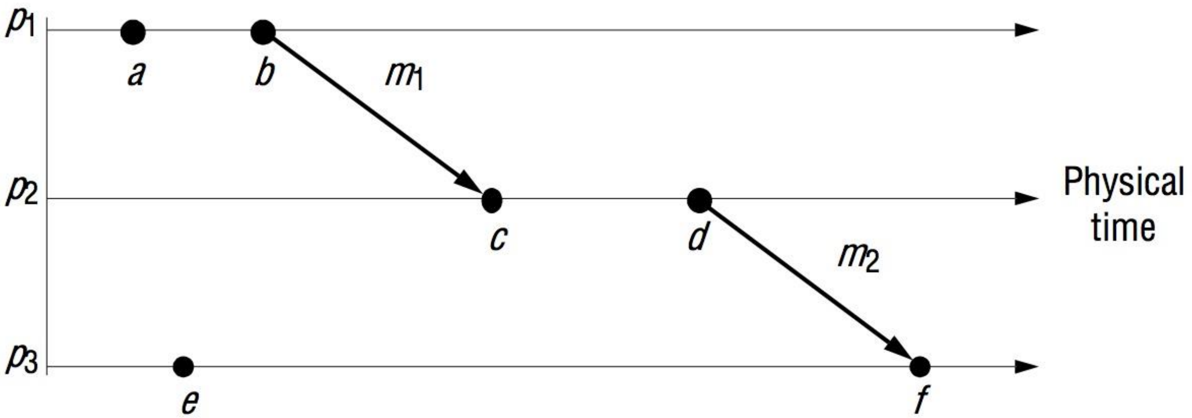
HB2: For any message m: $send(m) \rightarrow receive(m)$

HB3: If e, e', and e'' are events such that $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Two events are said to be concurrent if:

$$e \nrightarrow e' \text{ and } e' \nrightarrow e$$

A simple example



HB1: a ⟶ b, c ⟶ d, e ⟶ f

HB2: b ⟶ c, d ⟶ f

HB3: a ⟶ b ⟶ c ⟶ d ⟶ f

No ordering for e.g., b and e

They are concurrent, denoted b || e

Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – **Figure 14.5**

How can we use the “→” relation when implementing systems?

Lamport’s logical clocks

Lamport's logical clocks

- Monotonically increasing counter
  - Counter serves as a timestamp
- Each process has a counter that increases when an event occurs (*send* and *receive*)
- Counter is sent with message
  - Recipient sets own clock to `max(own, received)` and then increases its own counter

Details

Denote timestamp of event e at p_i by $L_i(e)$ and globally $L(e)$

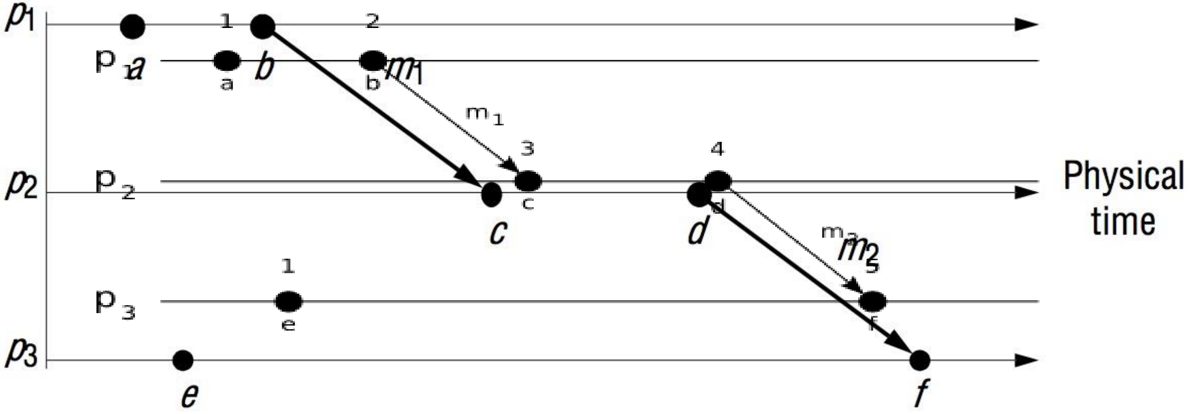
LC1: Increment L_i before each event at p_i , $L_i = L_i + 1$

LC2 : *(m* is a message, *t* is a timestamp*)*

a) When p_i sends `m`, it sends along the value $t = L_i$

b) On receiving `(m, t)`, p_j computes $L_j = \max(L_j, t)$ and then applies LC1 before time stamping the received event `receive(m)`

What can we say about our simple example



Ordering of events:

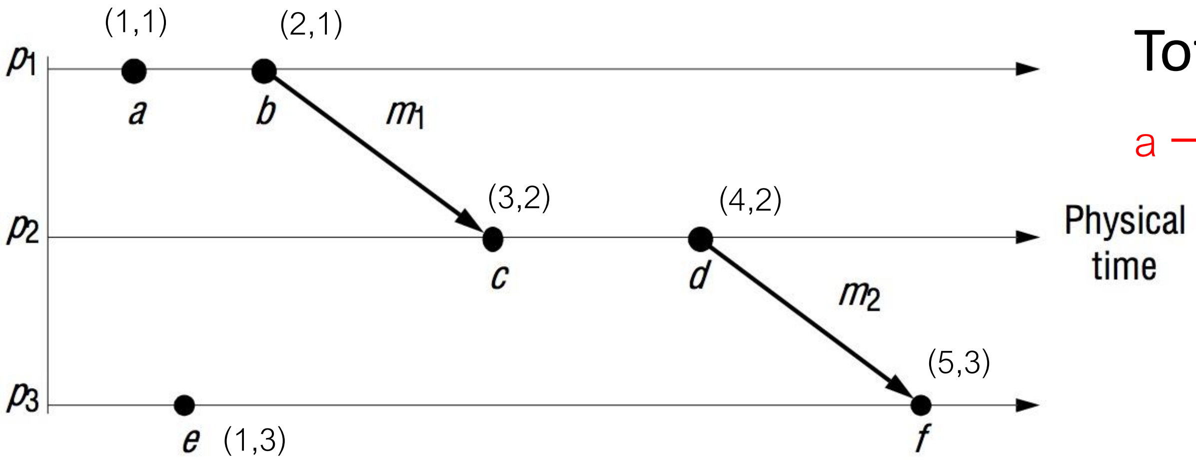
Evident that $e \rightarrow e' \Rightarrow L(e) < L(e')$

But, the opposite does not hold!

- e.g., $L(b) > L(e)$, but $b \parallel e$

Figure adapted from Instructor’s Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – **Figure 14.6**

How can we create a total order?



Total order:

a ⟶ e ⟶ b ⟶ c ⟶ d ⟶ f

Define global timestamps for `e` and `e'` to be (T_i, i) and (T_j, j) and $(T_i, i) < (T_j, j)$ iff $T_i < T_j$, or $T_i = T_j$ and $i < j$

Figure adapted from Instructor’s Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – **Figure 14.5**

But coming back to $L(e) < L(e') \nRightarrow e \rightarrow e'$

Vector clocks

Vector clocks

- Keep track of known events at all processes (a vector or array of timestamps)
- Each process keeps a vector clock to timestamp local events
- Send vector clock with message
  - Receiver merges clocks by setting own values to the maximum of own values and received ones

Formally

VC1: Initially, $V_i[j] = 0$, for `i, j = 1, 2, …, N`

VC2: Just before p_i timestamps `e`, it sets $V_i[i] = V_i[i] + 1$

VC3: p_i includes `timestamp =` V_i in every `send(m, timestamp)`

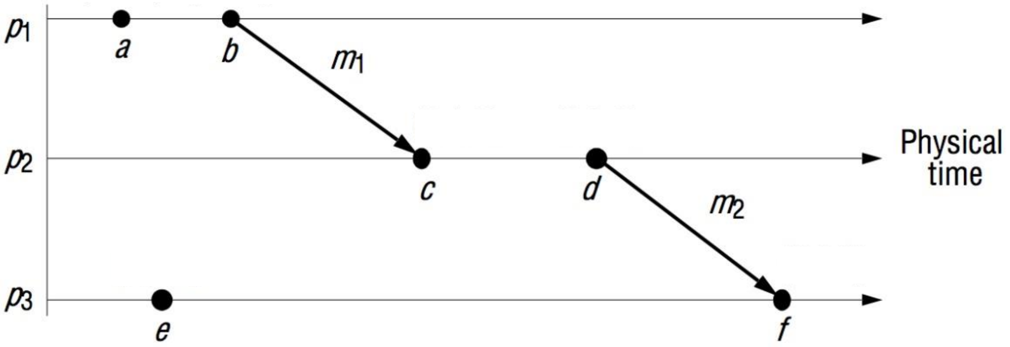
VC4: When p_i receives `timestamp` in a message, it sets

$V_i[j] = max\ (V_i[j]\ ,\ timestamp[j]\ )$, for `j = 1, 2, …, N`

Back to our simple example

$V = (L_1, L_2, L_3)$

$V_1 = (0,0,0)$
$V_2 = (0,0,0)$
$V_3 = (0,0,0)$



Vector clocks can be ordered

- V=V' if all values are the same
- V≤V' if all values in V are ≤ those in V'
- V<V' if V≤V' and V and V' are non-equal

Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – **Figure 14.7**

Concurrent events

$$\mathrm{e} \rightarrow \mathrm{e}' \Rightarrow \mathrm{V(e)} < \mathrm{V(e')} \; and \; \mathrm{V(e)} < \mathrm{V(e')} \Rightarrow \mathrm{e} \rightarrow \mathrm{e}'$$

- Concurrent events (`b || e`):
  - *Neither* `V(b) < V(e)` *nor* `V(e) < V(b)`

Vector clocks have nice properties

- Causal paths can be visualized
  - Causal paths help learn updates that occurred on other processes previous to an event
- However…
  - They use more space
    - expensive in terms of memory and bandwidth (`O(N)` in both cases)
    - no upper bound on clock size
  - It is better if processes don’t change dynamically

Summary

- ✓ We don’t have universal or global time
- ✓ Logical clocks are based on events in processes and the inter-event relationships (between processes)
  - ➢ Detect causal relationships – capability of one event to affect another event either directly or transitively
  - ➢ Happened-before relation
  - ➢ Some events are concurrent

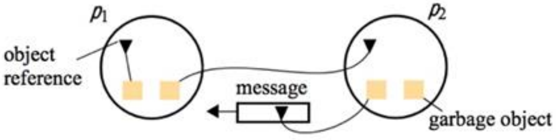
Summary (2)

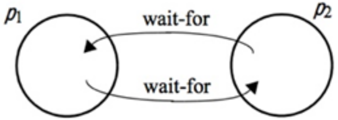
- ✓ Lamport's logical clocks are simple, but have problems with concurrent events
    - ➢ Can derive total order, but with no physical significance
    - ➢ Completely distributed
    - ➢ Fault tolerant
    - ➢ Impose minimal overhead
- ✓ Vector clocks are more powerful, but also more costly
    - ➢ Can differentiate when two events are concurrent

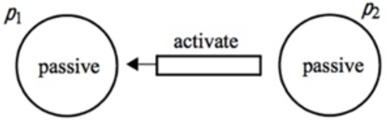
Global states

We often need to know the state of the entire distributed system of knowing if a particular property is true for the system as it executes

- Distributed garbage collection
- Stable property detection: distributed deadlocks, distributed termination detection
- Checkpointing

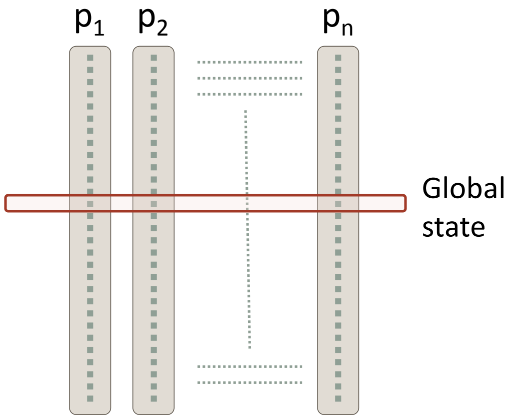






What prevents us from observing a global state in a Distributed System?

- ❑ Non-instantaneous communication
  - ➢ The view of a global state of a system depends on the observer
- ❑ Relativistic effects
  - ➢ Synchronization by time is not a reliable mechanism
- ❑ Interruptions
  - ➢ Different machines don’t react at the same time



Simple with global time!

Just issue “report state at time X”

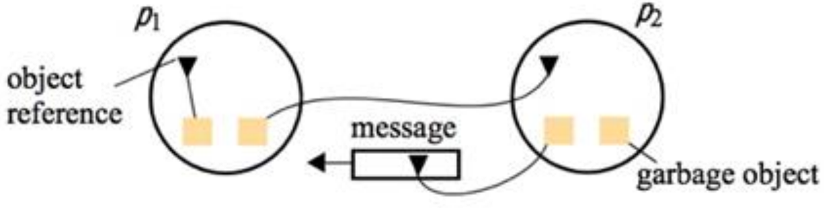
…we do not have this luxury

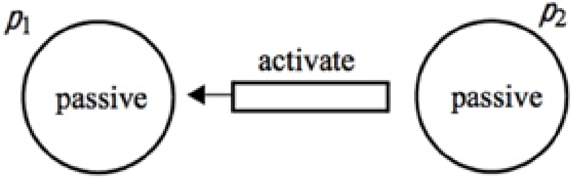
A simple approach

- Collect the state of each process one by one

Just process states are not enough!

Messages currently in the channels





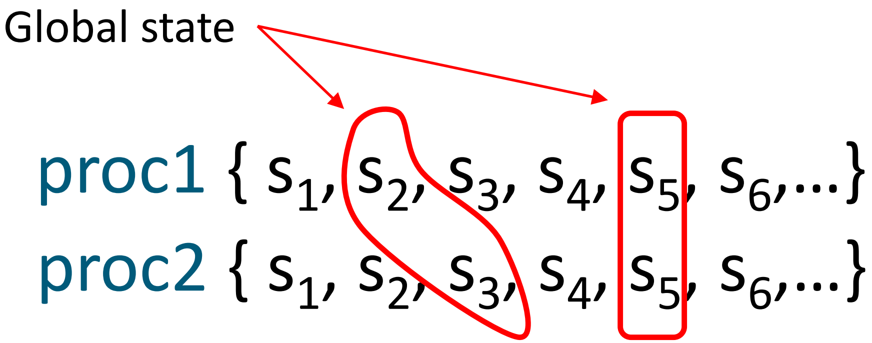
Motivation

```
a = 10                S1
b = 2                 S2
c = a + b             S3
send(c, proc2)        S4
a = 4                 S5
receive(b, proc1)     S6
      …               …
```

proc1

```
a = 5                 S1
b = 2                 S2
c = a - b             S3
receive(b, proc1)     S4
c = 9                 S5
send(b, proc1)        S6
      …               …
```

proc2



Each process maintains own history

- ☐ We could create global history by just taking union of all local histories

We only want to consider such global states *S* that may have occurred at some point in time

We can be more formal

Let’s remember that events at p_i defined a *history*

$history(p_i) = h_i = <e_i^0, e_i^1, e_i^2, ...>$

each process changes state accordingly

$s_i = <s_i^0, s_i^1, s_i^2, ...>$

The global history is the union of processes histories:

$$H = h_0 \bigcup h_1 \bigcup \dots$$

Let’s consider a prefix (first κ events) of a process histories

$h_i^k = <e_i^0, e_i^1, ..., e_i^k>$

Cuts

A *cut* is a union of prefixes of process histories:

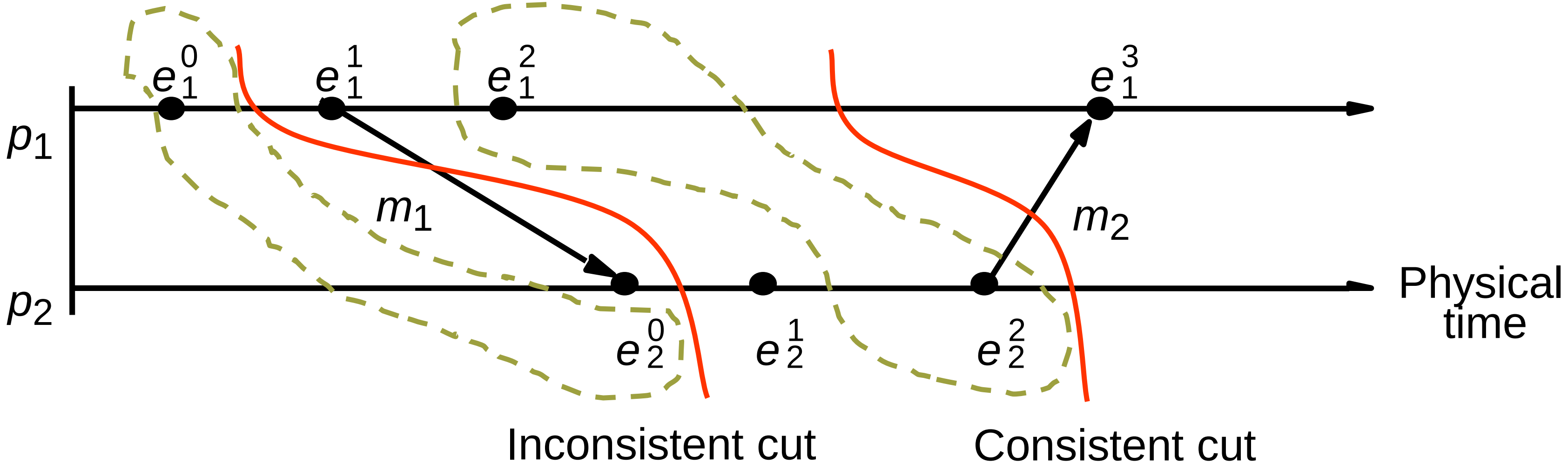
$$C = h_1^{c_1} \bigcup h_2^{c_2} \bigcup \dots \bigcup h_N^{c_N}$$

Frontier of the cut

States in which each process is after processing the last event in the cut:

$$\{e_i^{c_i} : i = 1,2, \dots , N\}$$

A simple example



According to the definition, we can make any cut that we want, including ones that make no sense!

Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – **Figure 14.9**

Consistent cuts and global states

- A cut is *consistent* if for each event in the cut
  - all events that happened before are also in the cut

$$e \in C,\ f \rightarrow e \Rightarrow f \in C$$

- We want to only consider *consistent cuts*
- Consistent global states correspond to consistent global cuts
  - We only move between consistent global states during execution: $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \dots$

Linearization and runs

- Total orderings of all events in the global history
  - A *run* is only consistent with the ordering of each process' own local history
  - A *linearization* is consistent with the (global) happened-before relation
- Runs do not have to pass through consistent global states, but all linearizations do
  - `s'` is *reachable* from `s` if ∃ a linearization from `s` to `s'`

Snapshot algorithm (Chandy-Lamport algorithm)

Snapshot algorithm

- ❑ Chandy and Lamport, distributed algorithm for determining global states of a distributed system
- ❑ Constructs a snapshot of the global state (both processes and channels)
  - ➢ Ensures that the global state is *consistent*
  - ➢ Makes **no guarantee** that the system was actually in the recorded state!

Assumptions

- ❑ Neither channel nor processes fail
  - ➢ Communication is reliable
- ❑ There’s a communication path between any two processes
  - ➢ Unidirectional channels with FIFO message delivery
- ❑ Any process may initiate a global snapshot at any time
- ❑ Algorithm does not interfere with the normal execution of the processes

How does the algorithm works?

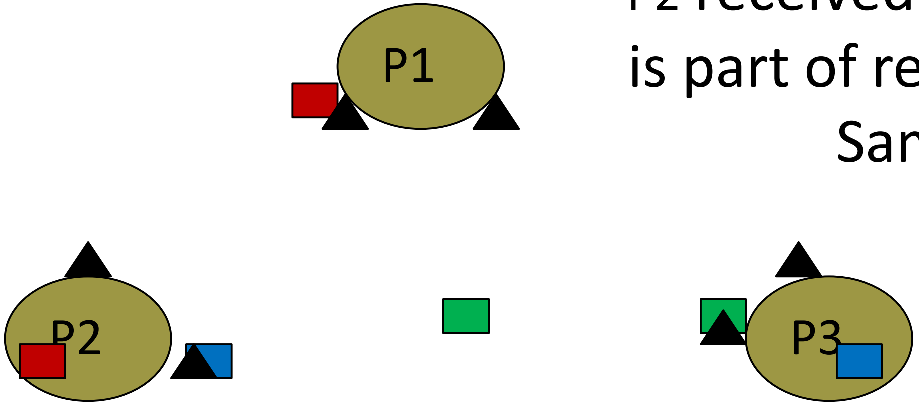
- Each process records its local state and the state of the incoming channels
- The algorithm works by using markers for two purposes:
  - As a signal for saving a process state
  - As a means of determining which messages belong to the channel state
- State is recorded at each process,
  - Global state is formed by collecting states from all processes

Algorithm

```
Marker receiving rule for process $p_i$
On $p_i$'s receipt of a marker message over channel $c$:
    if ($p_i$ has not yet recorded its state) it
    records its process state now;
    records the state of $c$ as the empty set;
    turns on recording of messages arriving over other incoming channels;
    else
     $p_i$ records the state of $c$ as the set of messages it has received over $c$
    since it saved its state.
    end if
Marker sending rule for process $p_i$
After $p_i$ has recorded its state, for each outgoing channel $c$:
     $p_i$ sends one marker message over $c$
    (before it sends any other message over $c$).
```

Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – **Figures14.10**

Snapshot example



P2 received marker on P1→P2 after 🟥, so it is part of recorded state

Same for P3 and 🟦

However, P3 sent out 🟩 before the marker, and P2’s state snapshot does not include it

Note that 🟩 is neither part of the state of P3 nor of P2 at this point!

Algorithm concludes that 🟩 was in transit between P3 and P2

Summary

- ✓ There are some cases where it is necessary to know the global state of a system
  - ➢ Lacking a global clock makes this difficult
- ✓ Global state encompasses both processes and channels states
- ✓ We Introduced the concept of cuts and consistent cuts
- ✓ We learned how to captured consistent global states corresponding to consistent cuts
  - ➢ Snapshot algorithm (Chandy & Lamport)

Next Lecture

Mutual exclusion and Elections