

Distributed Systems

Clocks, Ordering, and Global Snapshots

Björn Franke
2015/2016

University of Edinburgh

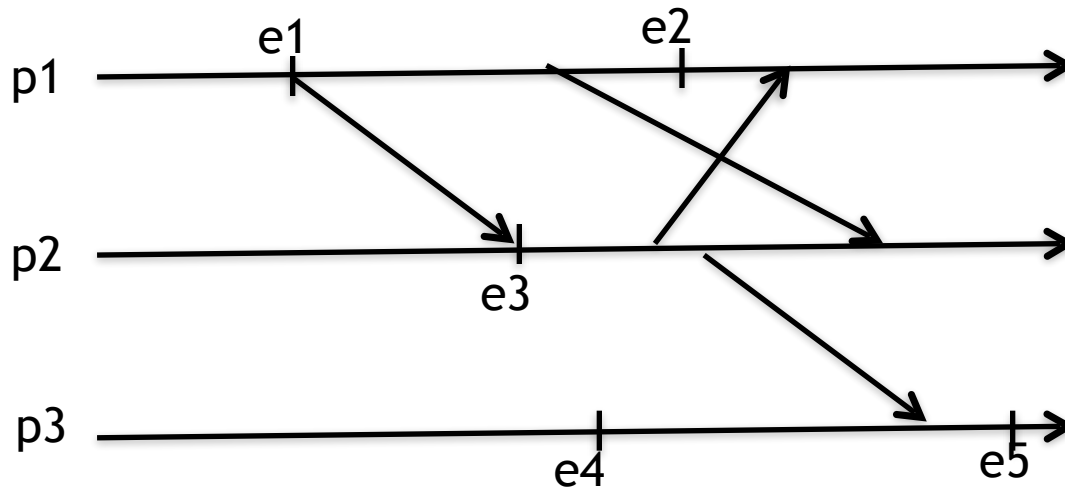
Logical clocks

- Why do we need clocks?
 - To determine when one thing happened before another
- Can we determine that without using a “clock” at all?
 - Then we don’t need to worry about synchronisation, millisecond errors etc..

Happened before

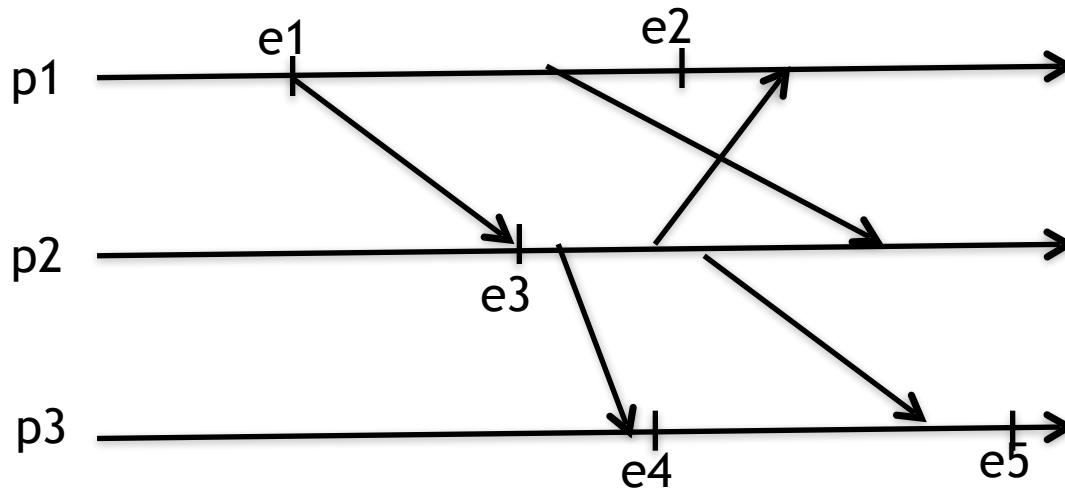
- $a \longrightarrow b$: a happened before b
 - If a and b are successive events in same process then $a \longrightarrow b$
 - Send before receive
 - If a : “send” event of message m
 - And b : “receive” event of message m
 - Then $a \longrightarrow b$
 - Transitive: $a \longrightarrow b$ and $b \longrightarrow c \implies a \longrightarrow c$

Example



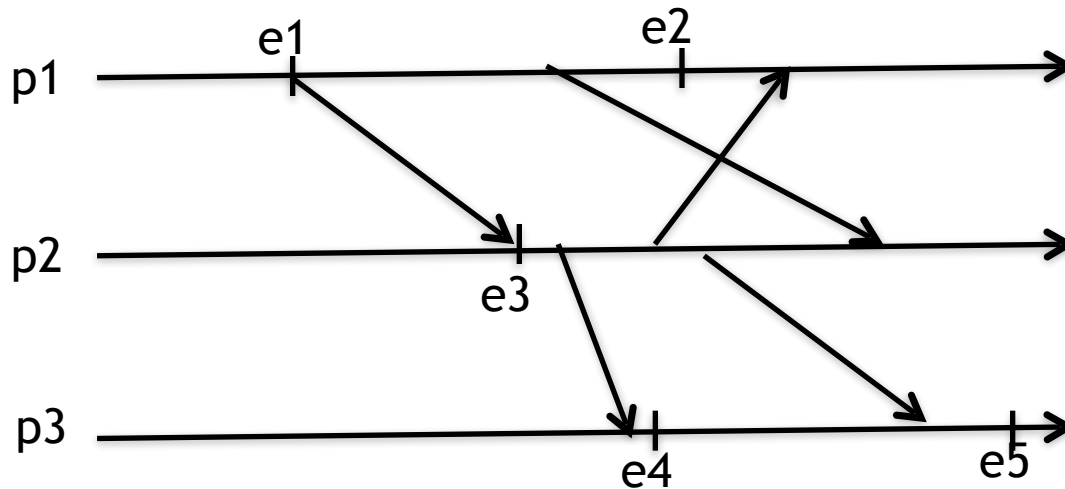
Example

- Events without a happened before relation are “concurrent”
- $e1 \longrightarrow e2$, $e3 \longrightarrow e4$, $e1 \longrightarrow e5$, $e5 \parallel e2$



Example

- Events without a happened before relation are “concurrent”
- Happened before is a partial ordering



Happened before & causal order

- Happened before == could have caused/ influenced
- Preserves causal relations
- Implies a partial order
 - Implies time ordering between certain pairs of events
 - Does not imply anything about ordering between concurrent events

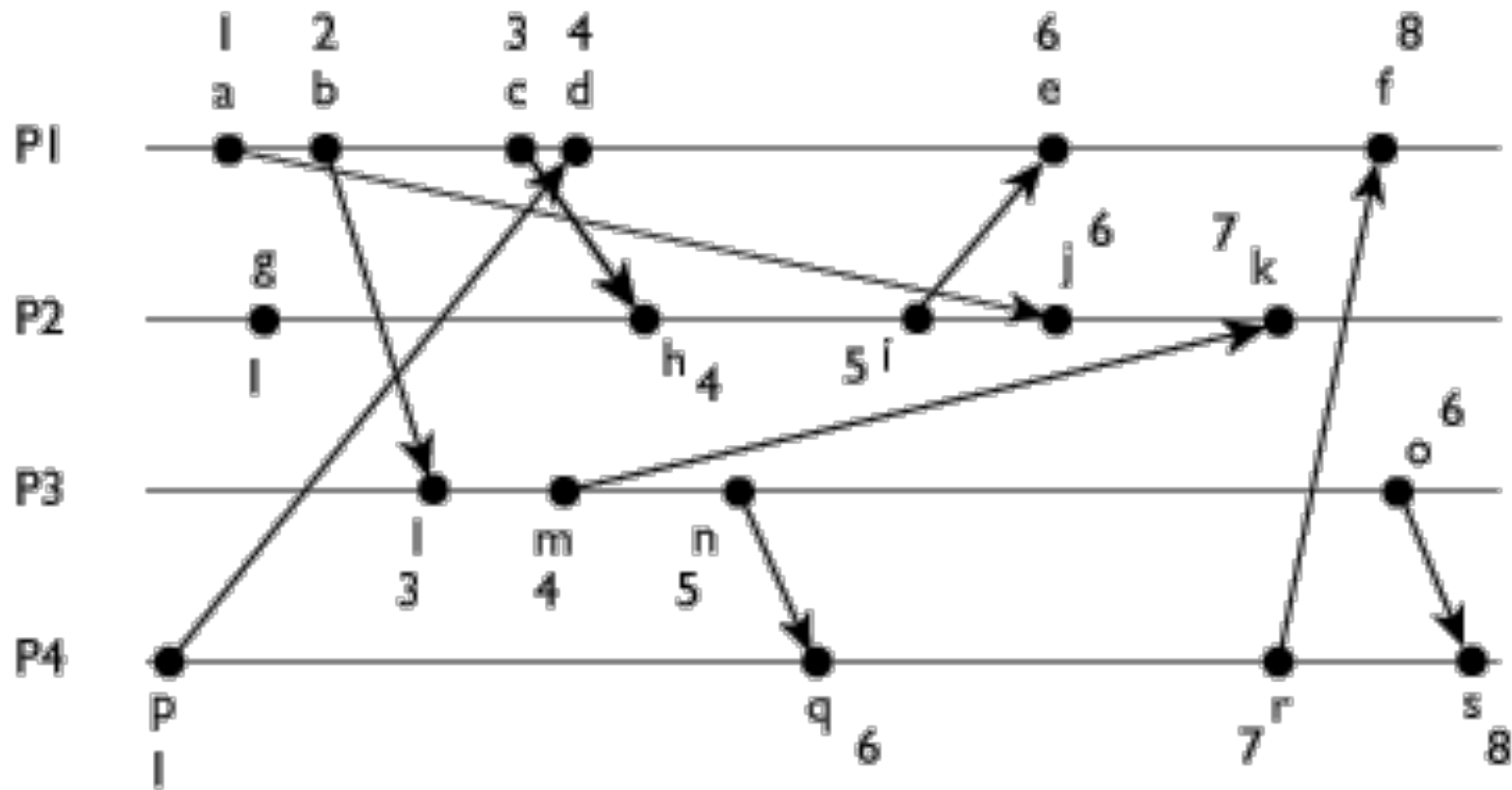
Logical clocks

- Idea: Use a counter at each process
- Increment after each event
- Can also increment when there are no events
 - Eg. A clock
- An actual clock can be thought of as such an event counter
- It counts the states of the process
- Each event has an associated time: The count of the state when the event happened

Lamport clocks

- Keep a logical clock (counter)
- Send it with every message
- On receiving a message, set own clock to $\max(\{\text{own counter, message counter}\}) + 1$
- For any event e , write $c(e)$ for the logical time
- Property:
 - If $a \longrightarrow b$, then $c(a) < c(b)$
 - If $a \parallel b$, then no guarantees

Lamport clocks: Example



Concurrency and Lamport clocks

- If $e1 \longrightarrow e2$
 - Then no Lamport clock C exists with $C(e1) == C(e2)$

Concurrency and Lamport clocks

- If $e1 \longrightarrow e2$
 - Then no Lamport clock C exists with $C(e1) == C(e2)$
- If $e1 \parallel e2$, then there exists a Lamport clock C such that $C(e1) == C(e2)$

The Purpose of Lamport Clocks

The Purpose of Lamport Clocks

- If $a \longrightarrow b$, then $c(a) < c(b)$
- If we order all events by their Lamport clock times
 - We get a partial order, since some events have same time
 - The partial order satisfies “causal relations”

The purpose of Lamport clocks

- Suppose there are events in different machines
 - Transactions, money in/out, file read, write, copy
- An ordering of events that guarantees preserving causality

Total order from Lamport clocks

- If event e occurs in process j at time $C(e)$
 - Give it a time $(C(e), j)$
 - Order events by $(C, \text{process id})$
 - For events e_1 in process i , e_2 in process j :
 - If $C(e_1) < C(e_2)$, then $e_1 < e_2$
 - Else if $C(e_1) == C(e_2)$ and $i < j$, then $e_1 < e_2$
- Leslie Lamport. Time, clocks and ordering of events in a distributed system.

Vector Clocks

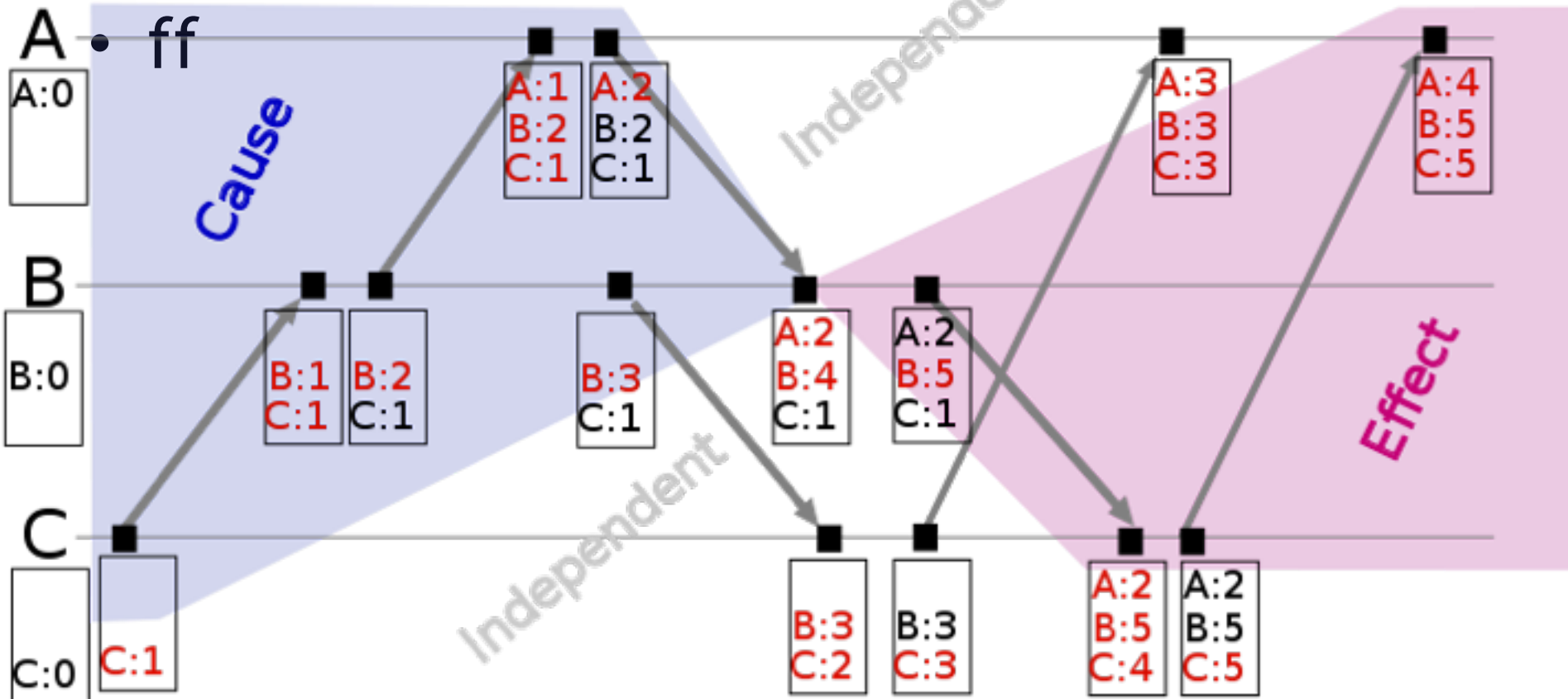
- We want a clock such that:
 - If $a \longrightarrow b$, then $c(a) < c(b)$
 - AND
 - If $c(a) < c(b)$, then $a \longrightarrow b$
- Ref: Coulouris et al., V. Garg

Vector Clocks

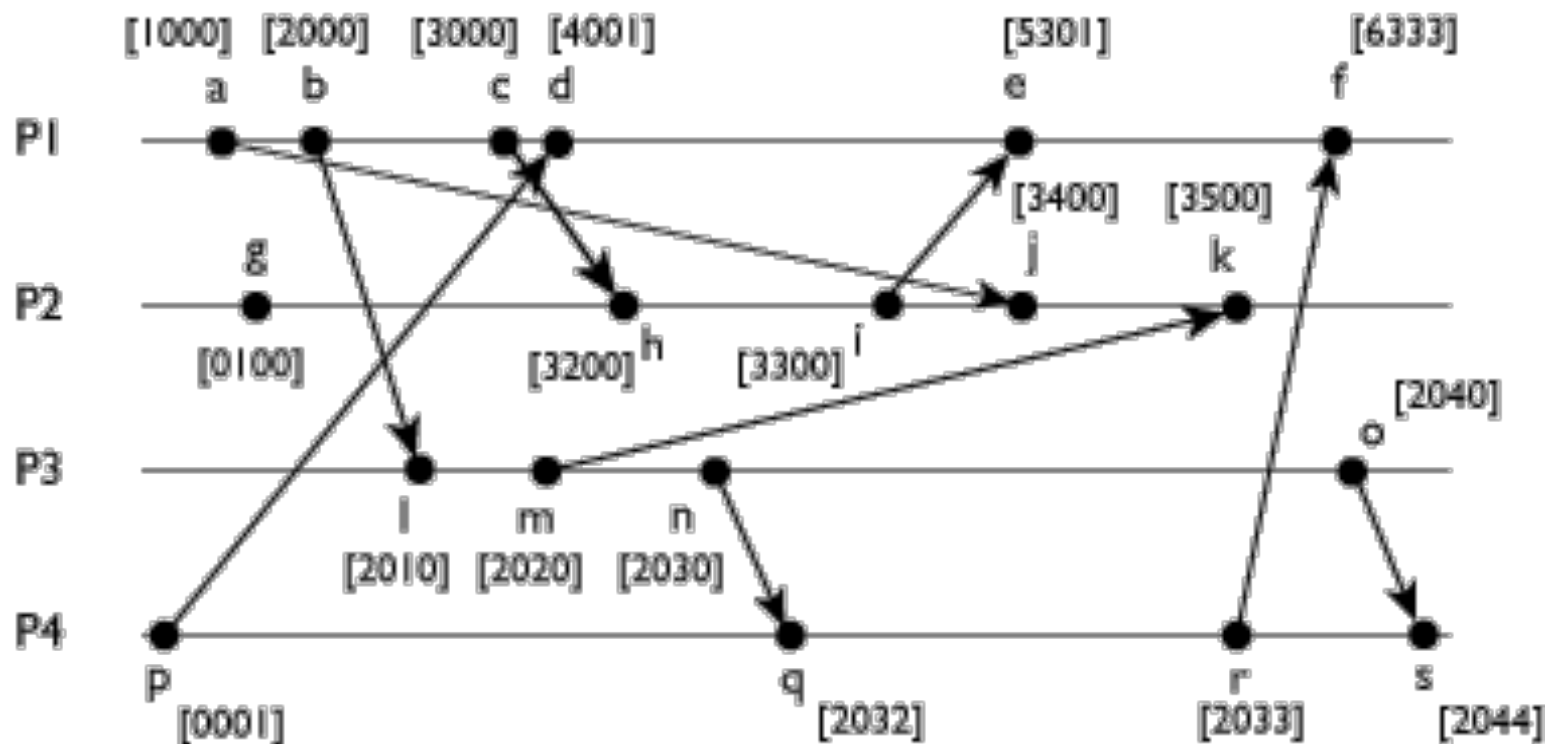
- Each process i maintains a vector V_i
- V_i has n elements
 - keeps clock $V_i[j]$ for every other process j
 - On every local event: $V_i[i] = V_i[i] + 1$
 - On sending a message, i sends entire V_i
 - On receiving a message at process j :
 - Takes max element by element
 - $V_j[k] = \max(V_j[k], V_i[k])$, for $k = 1, 2, \dots, n$
 - And adds 1 to $V_j[j]$

Example

Time →



Another Example



Comparing Timestamps

- $V = V'$ iff $V[i] == V'[i]$ for $i=1,2,\dots,n$
- $V < V'$ iff $V[i] < V'[i]$ for $i=1,2,\dots,n$

Comparing Timestamps

- $V = V'$ iff $V[i] == V'[i]$ for $i=1,2,\dots,n$
- $V < V'$ iff $V[i] < V'[i]$ for $i=1,2,\dots,n$
- For events a, b and vector clock V
 - $a \longrightarrow b$ iff $V(a) < V(b)$
- Is this a total order?

Comparing Timestamps

- $V = V'$ iff $V[i] == V'[i]$ for $i=1,2,\dots,n$
- $V \leq V'$ iff $V[i] \leq V'[i]$ for $i=1,2,\dots,n$
- For events a, b and vector clock V
 - $a \longrightarrow b$ iff $V(a) \leq V(b)$
- Two events are concurrent if
 - Neither $V(a) < V(b)$ nor $V(b) < V(a)$

Vector Clock Examples

- $(1,2,1) \leq (3,2,1)$ but $(1,2,1) \not\leq (3,1,2)$
- Also $(3,1,2) \not\leq (1,2,1)$
- No ordering exists

Vector Clocks

- What are the drawbacks?
- What is the communication complexity?

Vector Clocks

- What are the drawbacks?
 - Entire vector is sent with message
 - All vector elements (n) have to be checked on every message
- What is the communication complexity?
 - $\Omega(n)$ *per message*
 - Increases with time

Logical Clocks

- There is no way to have perfect knowledge on ordering of events
 - A “true” ordering may not exist..
 - Logical and vector clocks give us a way to have ordering consistent with causality

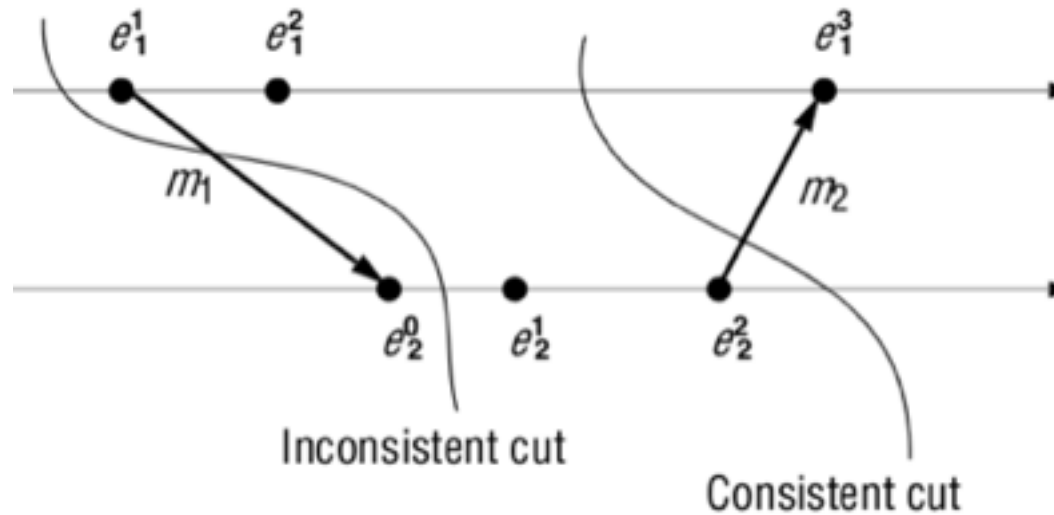
Distributed Snapshots

- Take a “snapshot” of a system
- E.g. for backup: If system fails, it can start up from a meaningful state
- Problem:
 - Imagine a sky filled with birds. The sky is too large to cover in a single picture.
 - We want to take multiple pictures that are consistent in a suitable sense
 - Eg. We can correctly count the number of birds from the snapshot

Distributed Snapshots

- Global state:
 - State of all processes and communication channels
- Consistent cuts:
 - A set of states of all processes is a consistent cut if:
 - For any states s, t in the cut, $s \mid \mid t$
- If $a \longrightarrow b$, then the following is not allowed:
 - b is before the cut, a is after the cut

Consistent Cut



Distributed Snapshot Algorithm

- Ask each process to record its state
- The set of states must be a consistent cut
- **Assumptions:**
 - Communication channels are FIFO
 - Processes communicate only with neighbors
 - (We assume for now that everyone is neighbor of everyone)
 - Processes do not fail

Global Snapshot

Chandy and Lamport Algorithm

- One process initiates snapshot and sends a marker
- Marker is the boundary between “before” and “after” snapshot

Global snapshot: Chandy and Lamport algorithm

- Marker send rule (Process i)
 - Process i records its state
 - On every outgoing channel where a marker has not been sent:
 - i sends a marker on the channel
 - before sending any other message
- Marker receive rule (Process i receives marker on channel C)
 - If i has not received the marker before
 - Record state of I
 - Record state of C as empty
 - Follow marker send rule
 - Else:
 - Record the state of C as the set of messages received on C since recording i's state and before receiving marker on C
- Algorithm stops when all processes have received marker on all incoming channels

Complexity

- Message?
- Time?