CS 5450

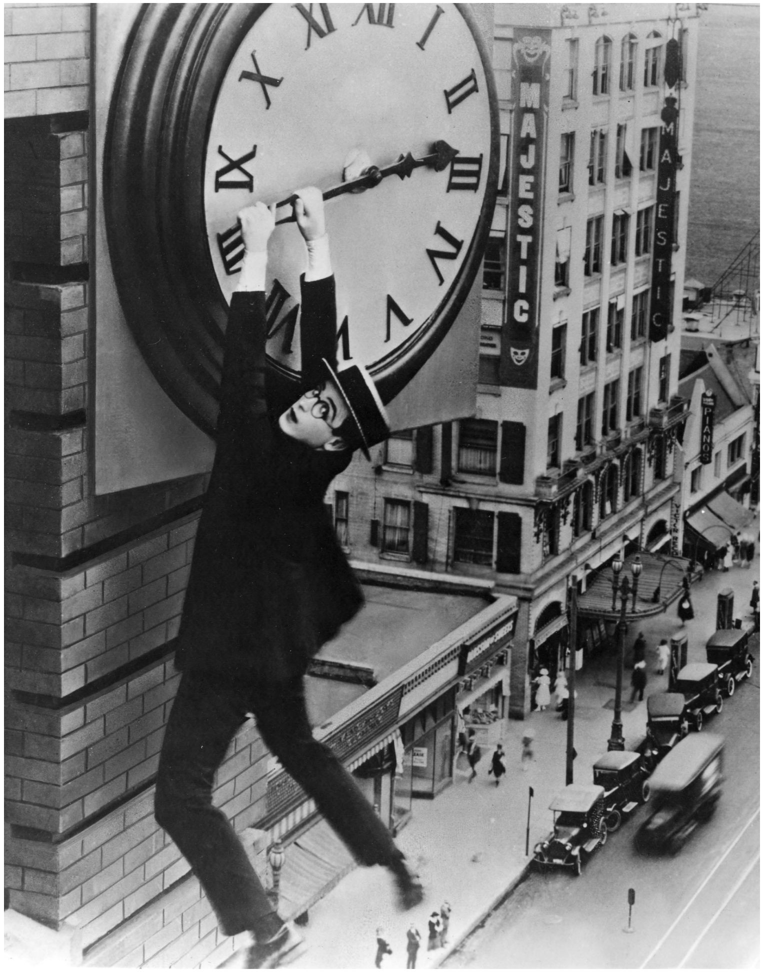
Time
Global State
Failure Detection

Vitaly Shmatikov

slides from Cristina Nita-Rotaru

Time

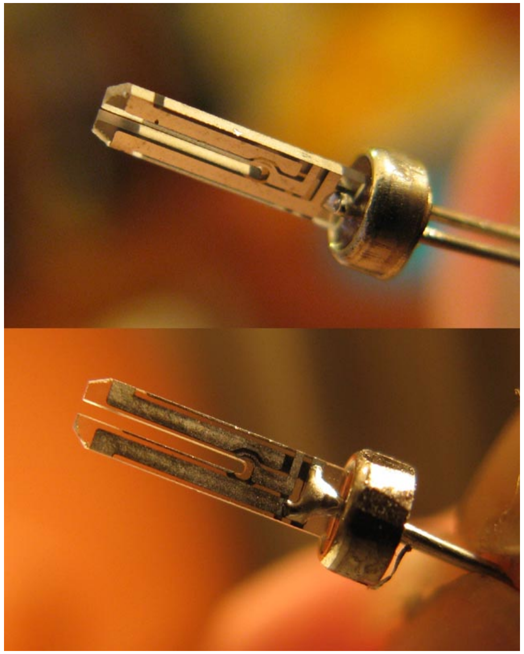
- Time is essential for ordering events in a distributed system
  - Physical time: local clock, global clock
  - Logical time: Lamport clocks, vector clocks



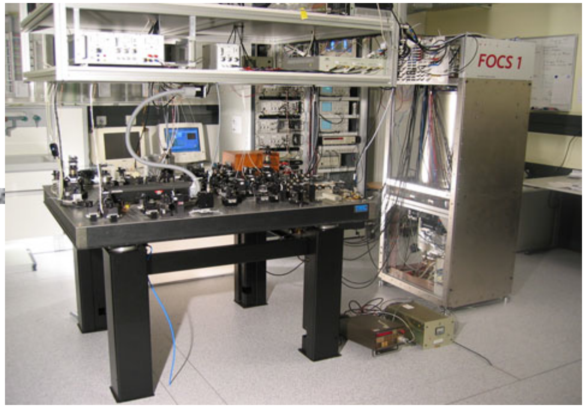
Historical Clocks

Electrical Clocks

- First developed in 1920s
  - Uses carefully shaped quartz crystal
  - Pass current, counts oscillations
- Most oscillate at 32,768/sec
  - Easy to count in hardware
  - Small enough to fit (~4mm)
- Typical quartz clock quite accurate
  - Within 15 sec/30 days (6e-6)
  - Can achieve 1e-7 accuracy in controlled conditions
  - Not good enough for today's applications



Atomic Clocks



- Based on atomic physics
  - Cool atoms to near absolute zero
  - Bombard them with microwaves
  - Count transitions between energy levels
- Most accurate timekeeping devices
  - Accurate to within 10^{-9} seconds per day (loses 1 second in 30 million years)
- Standard International Second defined in terms of atomic oscillations
  - 9,192,631,770 transitions of cesium-133 atom

International Atomic Time

- Atomic clocks used to define several time standards
- TAI: International Atomic Time
  - Avg. of 200 atomic clocks, corrected for time dilation
  - Essentially, a count of the number of seconds passed since January 1, 1958
- UTC: since January 1, 1972, defined to follow TAI with an exact offset of an integer number of seconds, changing only when a leap second is added to keep clock time synchronized with the rotation of the Earth

Using Real Clocks to Order Events

- Each event carries a timestamp
- Global clock: processes have access to a central global clock
  - The global clock gives global ordering of events
- Local clock: each process has its own clock
  - What if the clocks are not synchronized?
  - What if events happened at the same time?

Clocks in Computers

- Real-time clock: CMOS clock (counter) circuit driven by a quartz oscillator with battery backup to continue measuring time when power is off
- OS generally programs a timer circuit to generate an interrupt periodically
  - e.g., 60, 100, 250, 1000 interrupts per second
  - Programmable Interval Timer (PIT) – Intel 8253, 8254
  - Interrupt handler adds 1 to a counter in memory
- Quartz oscillators oscillate at slightly different frequencies, clocks do not agree in general

When Is a Clock "Correct"?

- Relative to an "ideal" clock
  - Clock skew is magnitude
  - Clock drift is difference in rates
- Say clock is correct within p if
  $(1-p)(t'-t) \le H(t') - H(t) \le (1+p)(t'-t)$
  - $(t'-t)$ True length of interval
  - $H(t') - H(t)$ Measured length of interval
  - $(1-p)(t'-t)$ Smallest acceptable measurement
  - $(1+p)(t'-t)$ Largest acceptable measurement
- Monotonic property: $t < t' \Rightarrow H(t) < H(t')$

Monotonicity

- If a clock is running "slow" relative to real time…
  - Can simply re-set the clock to real time
  - Doesn't break monotonicity
- What if a clock is running "fast"?
  - Re-setting the clock back breaks monotonicity
  - Imagine programming with the same time occurring twice
- Instead, "slow down" clock
  - Maintains monotonicity

Network Time Protocol (NTP)

- NTP is a distributed service that…
  - Keeps machines synchronized to UTC
  - Deals with lengthy losses of connectivity
  - Enables clients to synchronized frequently (scalable)
  - Avoids security attacks
- NTP deployed widely today
  - Uses 64-bit value, epoch is 1/1/1900 (rollover in 2036)
  - Precision: 1ms on LANs, 10s of ms on Internet
  - **NTP pool** is a dynamic collection of 4000 servers that volunteer to provide time via NTP

Reference Clocks

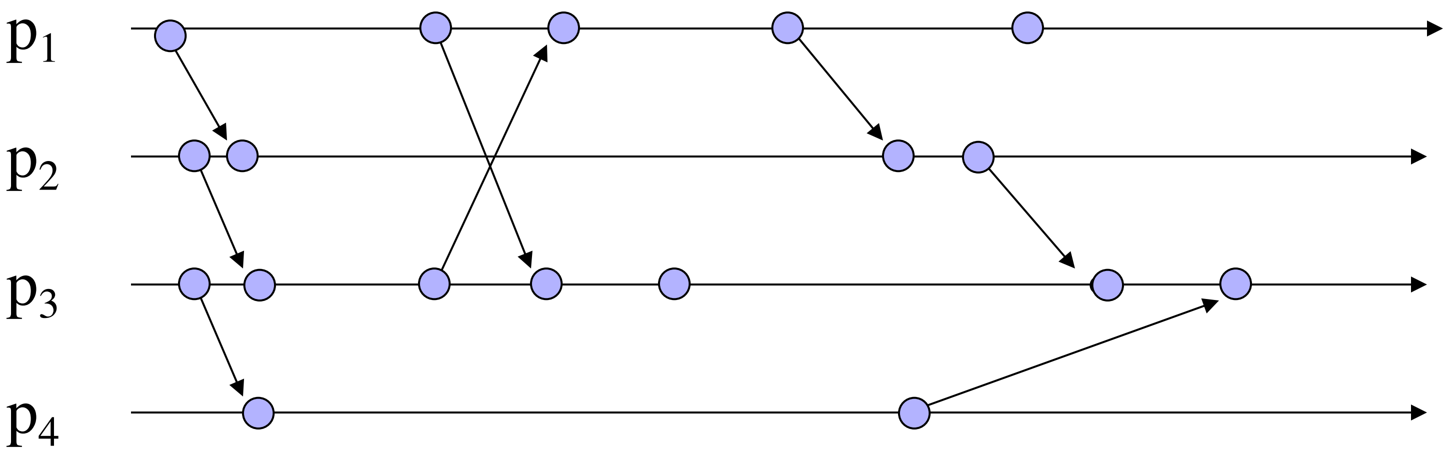
- Many NTP servers synchronize directly to UTC using specialized equipment
  - **Atomic clocks**: ultimately are the root source of time in NTP
  - **Global Positioning System (GPS)**: can synchronize with a satellite's atomic clock
  - **Code Division Multiple Access (CDMA)**: can synchronize with a local wireless provider (who in turn most likely synchronizes using GPS)
  - **Radio signals**: similar to CDMA, can synchronize with time/frequency radio stations

From Physical to Logical Clocks

- Synchronized clocks are great if we have them
- Why do we need the time anyway?
- In distributed systems, we care about "what happened before what"

- Message-based systems, two type of events
  - Send a message
  - Receive a message

"Happened Before"

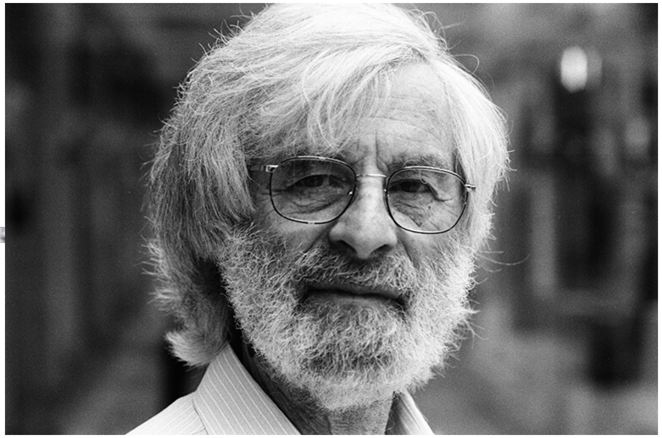


- If events a and b take place at the same process and a occurs before b (physical time), then we have $a \rightarrow b$
- If a is a send event of message m at p_1 and b is a deliver event of the same m at p_2, $p_1 \neq p_2$ then $a \rightarrow b$
- If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

Reminder: Partial and Total Order

- A relation R over a set S is a partial order iff for each a, b, and c in S:
  - aRa (reflexive)
  - $aRb \wedge bRa \Rightarrow a = b$ (antisymmetric)
  - $aRb \wedge bRc \Rightarrow aRc$ (transitive)
- A relation R over a set S is total order if for each distinct a and b in S, R is antisymmetric, transitive and either aRb or bRa (completeness)

Lamport Clocks (1978)

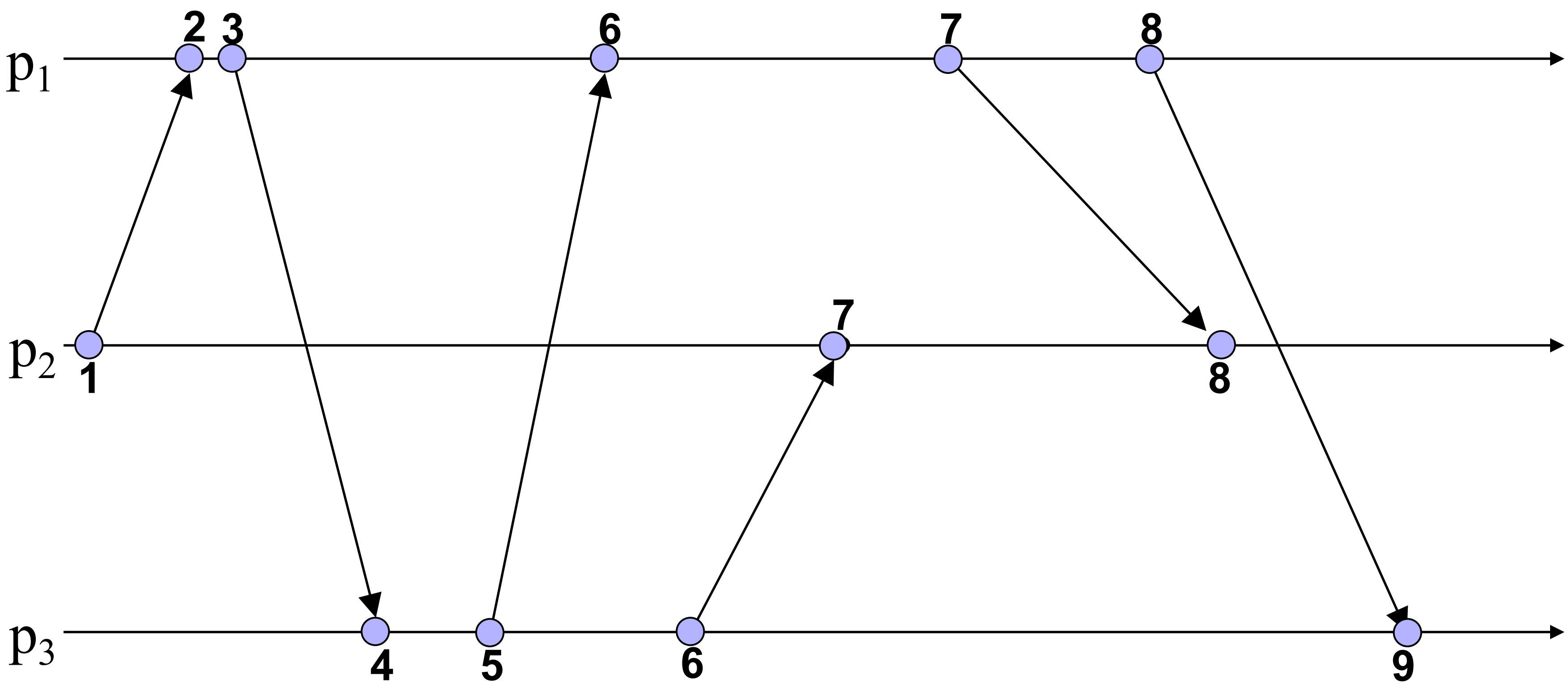


- Each process maintains its own clock C_i (a counter)
- For any events a and b in process p_i

  if $a \rightarrow b$ then $C_i(a) < C_i(b)$

- Implementation:
  - Each p_i increments C_i between any successive events
  - On sending a message m, attach local clock $T_m = C_i(a)$
  - On receiving a message m, process p_k sets C_k to $C_k = \max(C_k, T_m) + 1$

Lamport Clocks: Example



Lamport Clocks: Total Order

- Logical clocks only provide partial order
- Create total order by breaking the ties
- Example: have an order on process identifiers, use them to break ties
  - If a is event in p_i and b is event in p_j then

    $a \rightarrow b$ iff
  - $C_i(a) < C_j(b)$ or
  - $C_i(a) = C_j(b)$ and $p_i < p_j$

Concurrent Events

- If $a \rightarrow b$ and $b \rightarrow a$ then a and b are concurrent
- Logical clocks assign order to events that are causally independent
  - Events that are causally independent appear as if they happened in a certain order
- For some applications (e.g. debugging) it is important to capture independence

Vector Clocks

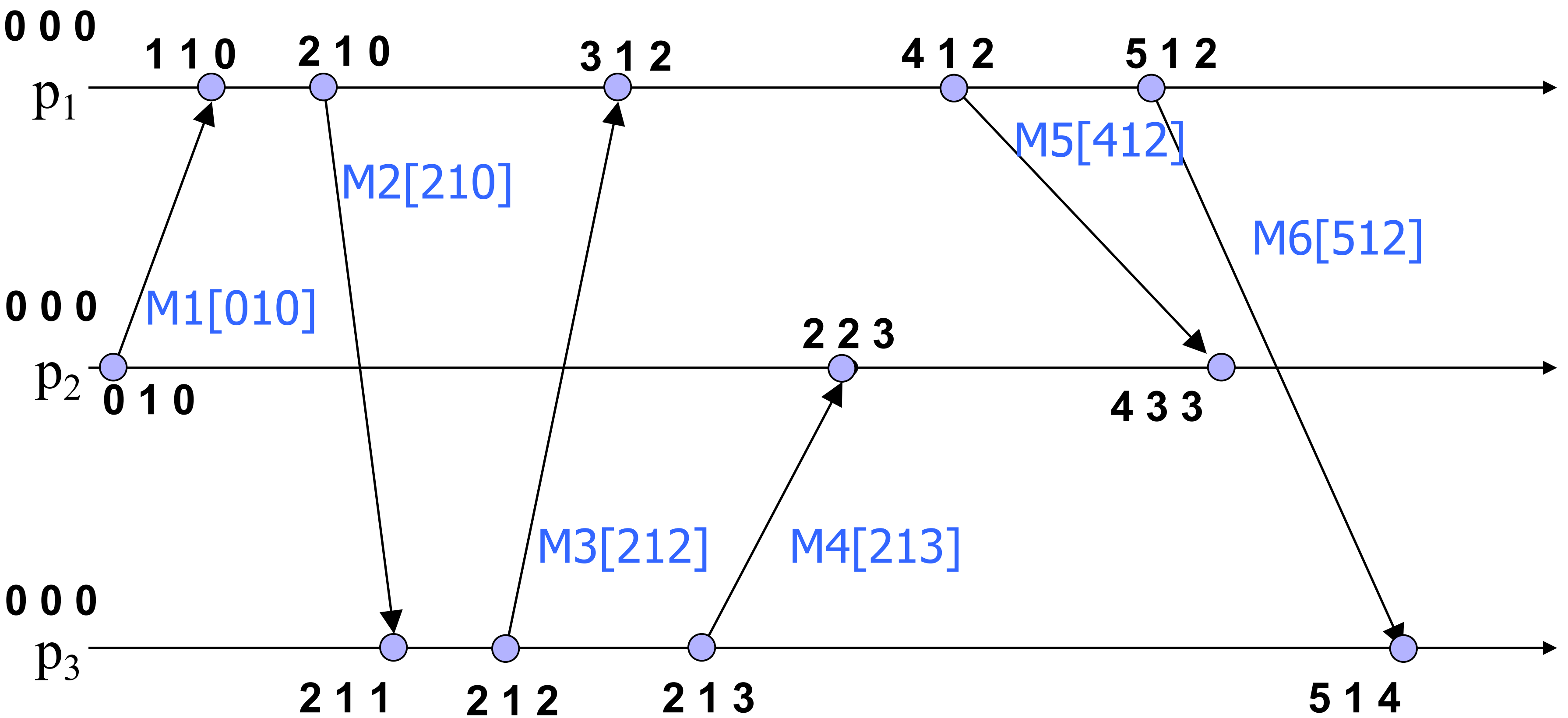
[Fidge and Mattern (independently), 1988]

- Each process p_i maintains a vector C_i
- When p_i executes an event, it increments its own clock $C_i[i]$
- When p_i sends a message m to p_j, it attaches its vector C_i
- When p_i receives a message m, increments its own clock and updates the clock for the other processes

  $\forall\ j: 1 \le j \le n, j \ne i: C_i[j] = \max(C_i[j], m.C[j])$

  $C_i[i] = C_i[i] + 1.$

Vector Clocks: Example



How to Order with Vector Clocks

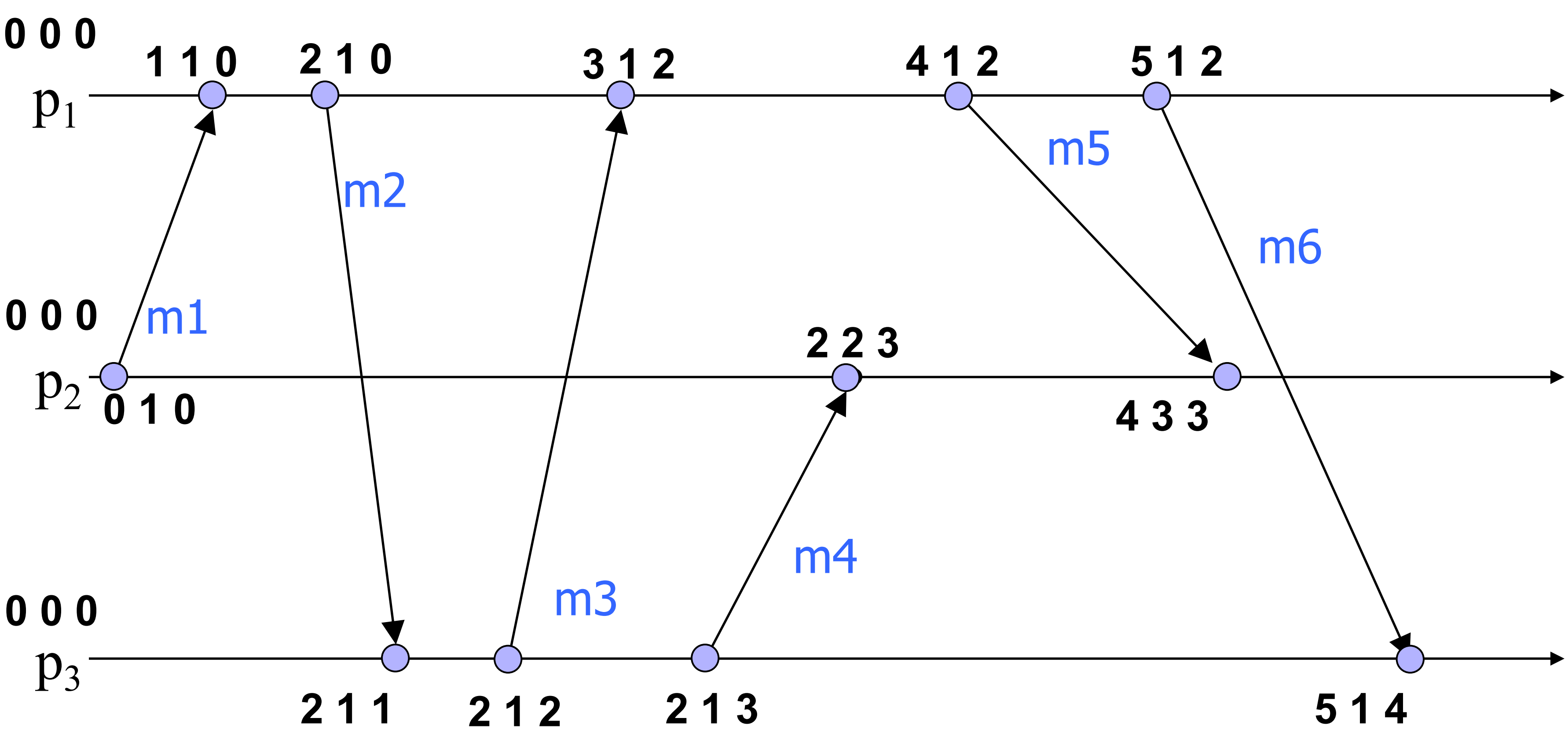
Given two events a and b, a $\rightarrow$ b if and only if V(a) is less than or equal to V(b) for all process indices, and at least one of those relationships is strictly smaller

- $a \rightarrow b \equiv \forall\ i: 1 \leq i \leq n: V(a)[i] \leq V(b)[i] \wedge$
  $\exists\ i: 1 \leq i \leq n: V(a)[i] < V(b)[i]$

Otherwise, they are concurrent or independent

- $a \parallel b \equiv \exists\ i: 1 \leq i \leq n: V(a)[i] < V(b)[i] \wedge$
  $\exists\ j: 1 \leq j \leq n: V(b)[j] < V(a)[j]$

What Events Are Independent?



Why We Need Global Snapshots

- Checkpointing: save the state and restart the distributed application after a failure
- Garbage collection of objects: objects at servers that don't have any other objects (at any servers) with pointers to them
- Deadlock detection: debugging for database transaction systems
- Termination of computation: useful for batch computing systems

Recording Global Snapshots

- If synchronized clocks are available, each process records its state at a known time t
  - How to obtain the state of the messages that transit the channels?
- If synchronized clocks are not available?
  - How to determine when a process takes its snapshot?
  - How to distinguish between the messages to be recorded in the snapshot from those not to be recorded?

Chandy-Lamport Algorithm

records a consistent global state of an asynchronous system.

- System model
  - No failures and all messages arrive intact and only once
  - Communication channels are unidirectional and FIFO
  - There is a communication path between any two processes
- Other assumptions
  - Any process may initiate the snapshot algorithm
  - The snapshot algorithm does not interfere with the normal execution of the processes
  - Each process records its local state and the state of its incoming channels

Chandy-Lamport Algorithm

- A process needs to know
  - When to start recording (if it was not the one that initiated the algorithm)
  - What messages to include in the snapshot
  - When did all other processes record their snapshot
- Key design: a control message, **marker**
  - To separate messages to be included from messages not to be included
  - To inform other processes that it has recorded snapshot
  - To tell other processed to start recording: a process must record its snapshot no later than when it receives a marker on any of its incoming channels

Chandy-Lamport Algorithm

- Any process can initiate by executing the “Marker Sending Rule”
- On receiving a marker on some channel c, a process executes the “Marker Receiving Rule”
  - If not yet recorded its local state, record the state of channel c as empty and execute the “Marker Sending Rule” to record its local state
- The algorithm terminates after each process has received a marker on all of its incoming channels
- Sum of all local snapshots = global state

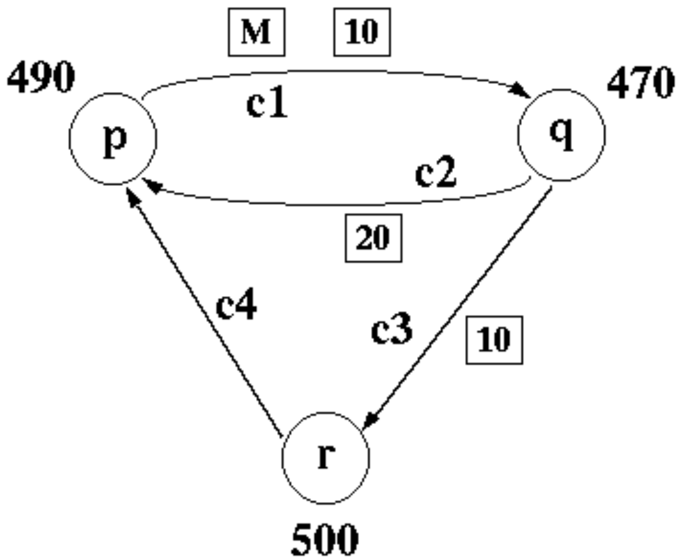
Chandy-Lamport Snapshot Algorithm

- Marker-sending rule for a process
  - Record its local state
  - Send a marker to all other processes on the corresponding channels before sending any other message
- Marker-receiving rule for a process q on channel c
  - If not yet recorded its local state, then
    - Record its local state
    - Record the state of channel c as "empty"
    - Turn on recording of messages over other incoming channels
    - Send a market on each outgoing channel
  - Else
    - Record the state of incoming channel as all messages received over it after q recorded its state and before it received the marker along c

Chandy-Lamport Algorithm: Example

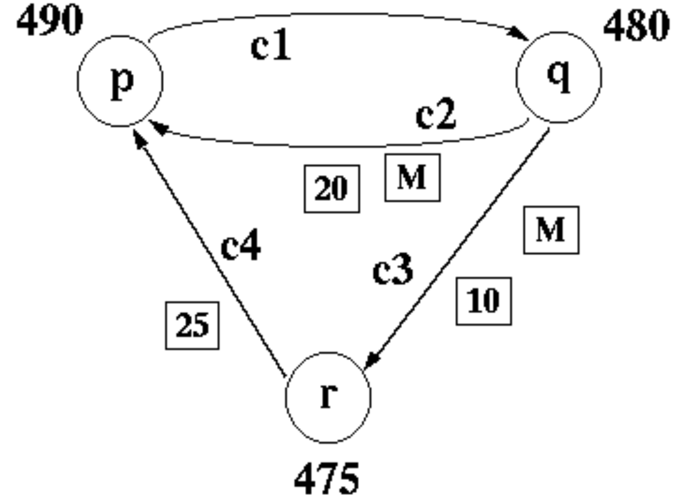
- Setup
  - Three processes p, q and r
  - Communication channels: c1 (p to q), c2 (q to p), c3 (q to r), and c4 (r to p)
  - All start with state = $500 and the channels are empty. The stable property is that the total amount of money is $1500.
- Process p sends $10 to q and then starts the snapshot algorithm: records its current state $490 and sends out a marker on c1
- Meanwhile q has sent $20 to p along c2 and 10 to r along c3

Snapshot/State Recording Example (Step 1)



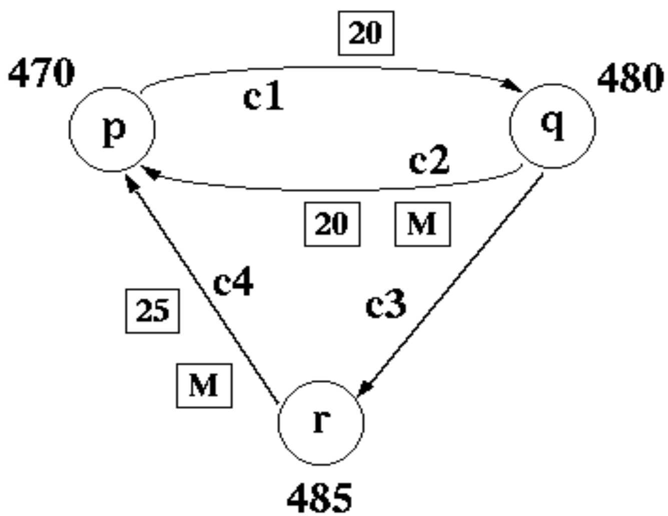
Node	Recorded State				
	state	c1	c2	c3	c4
p	490		{}		{}
q		{}			
r				{}	

Snapshot/State Recording Example (Step 2)



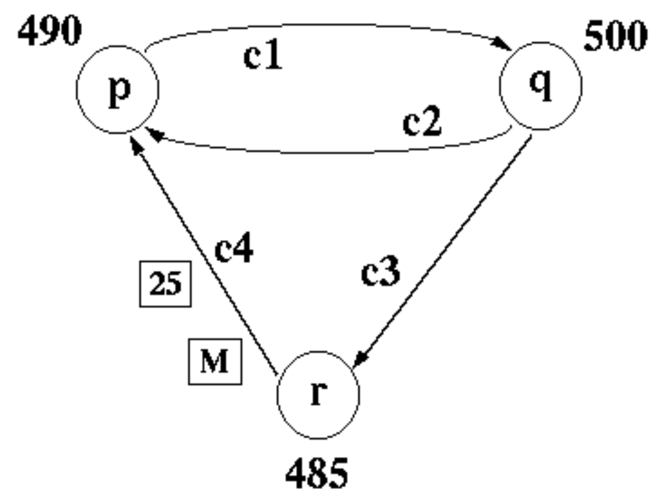
Node	Recorded State				
	state	c1	c2	c3	c4
p	490		{}		{}
q	480	{}			
r				{}	

Snapshot/State Recording Example (Step 3)



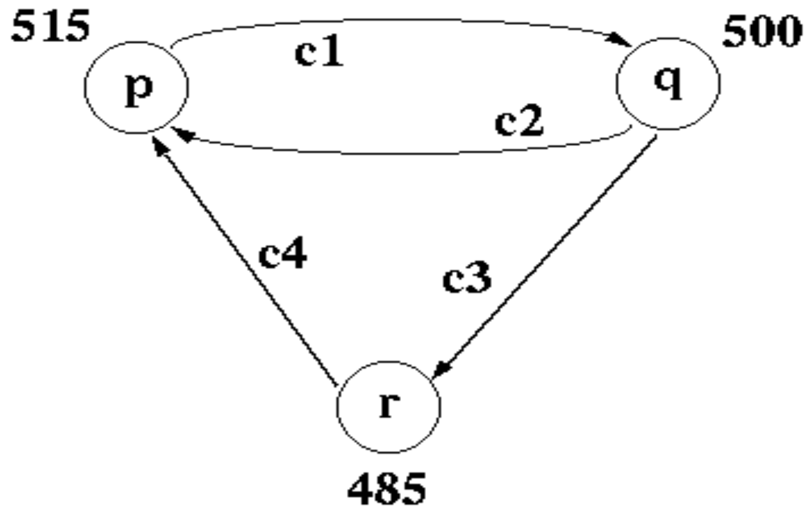
Node	Recorded State				
	state	c1	c2	c3	c4
p	490		{}		{}
q	480	{}			
r	485			{}	

Snapshot/State Recording Example (Step 4)



Node	Recorded State				
	state	c1	c2	c3	c4
p	490		{20}		{}
q	480	{}			
r	485			{}	

Snapshot/State Recording Example (Step 5)



Node	Recorded State				
	state	c1	c2	c3	c4
p	490		{20}		{25}
q	480	{}			
r	485			{}	

Correctness for Chandi-Lamport

- How do we define correctness?
- Records a consistent global state of an asynchronous system

History of Events

Given a process p_i

- e_i^j is the j^{th} event
- History is a sequence of events

$$h_i = \langle e_i^0, e_i^1, \ldots \rangle$$

- Prefix history is the history up to the k^{th} event

$$h_i^k = \langle e_i^0, e_i^1, \ldots, e_i^k \rangle$$

- State S_i^k is the state of process p_i immediately before the k^{th} event

More Definitions

- Global history: the set of all processes' histories

$$H = \cup_i (h_i)$$

- Global state: the set of all processes' states

$$S = \cup_i (S_i^{k_i})$$

- Cut: a set of prefix histories

$$C \subseteq H = h_1^{c1} \cup h_2^{c2} \cup \ldots \cup h_n^{cn}$$

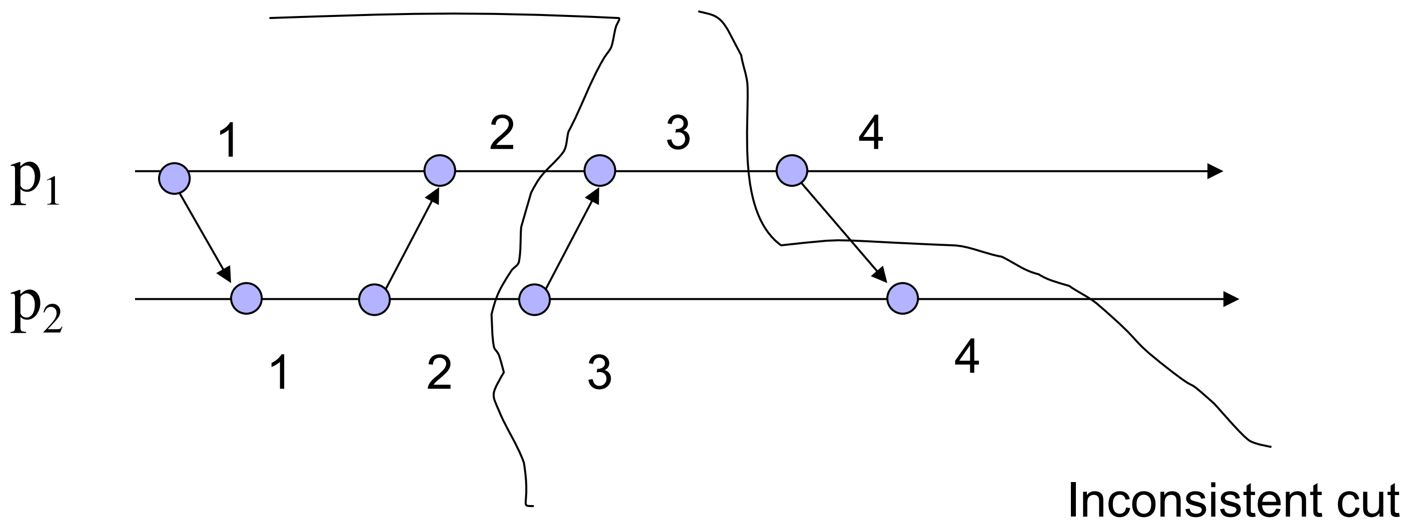
- Frontier of a cut: the set of last events that happened in each prefix history

$$C = \{e_i^{ci}, i = 1,2, \ldots n\}$$

Consistent Cuts

A cut C is consistent if for any event e in the cut, if an event f 'happened before' e, then f is also in the cut C

$$\forall_{e \in C} (\text{if } f \rightarrow e \text{ then } f \in C)$$



Using Global States

- Consistent global state: a global state that corresponds to a consistent cut
- Run: a total ordering of events in history H that is consistent with each process history h_i
- Linearization: a run consistent with happens-before relation in H; linearizations pass through consistent global states
- Reachability: a global state S_k is reachable from global state S_i, if there is a linearization, L, that passes through S_i and then through S_k

Global State Predicates

- Global state predicate: a function from the set of global states to {TRUE, FALSE}
- Stable global state predicate: one that once it becomes true, it remains true in all future states reachable from that state
- Examples:
  - “the system is deadlocked”
  - “all tokens in a token ring have disappeared”
  - “the computation has finished”

Safety and Liveness

- Safety: a condition that must hold in every finite prefix of a sequence (from an execution)

  "nothing bad happens"

- Liveness: a condition that must hold a certain number of times

  "something good happens"

Stable Global States and Safety

Assume that a “bad thing” BT (for example deadlock) is a global state predicate and S_0 is the initial state of the system, then

“Safety with respect to BT” means

$\forall S$ reachable from S_0, $BT(S) = FALSE$

Stable Global States and Liveness

Assume that a "good thing" GT (for example, reaching termination) is a global state predicate and S_0 is the initial state of the system, then

Liveness with respect to GT means

For any linearization L starting at S_0 $\exists$ state S_L reachable from S_0 such that $GT(S_L) = TRUE$

Failure Detectors as Abstraction

- Failure detector: distributed oracle that makes guesses about process failures
  - Failure = crash (only for now!)
- Accuracy: failure detector makes no mistakes when labeling processes as crashed
- Completeness: failure detector "eventually" (after some time) suspects every process that actually crashed
- Used to solve different distributed systems problems

Completeness

- Strong completeness: There is a time after which every process that crashes is suspected by EVERY correct process
- Weak completeness: There is a time after which every process that crashes is suspected by SOME correct process

Accuracy

- Strong accuracy: No process is suspected before it crashes
- Weak accuracy: At least one correct process is never suspected
- Eventual strong accuracy: There is a time after which correct processes are not suspected by any correct process
- Eventual weak accuracy: There is a time after which some correct process is never suspected by any correct process

Perfect Failure Detector

- A perfect failure detector has strong accuracy and strong completeness… such detector is IMPOSSIBLE
- We have to live with unreliable failures detectors

Failure Detection Implementation

- Push: processes keep sending heartbeats “I am alive” to the monitor. If no message is received for awhile from some process, that process is suspected as being dead (faulty)
- Pull: monitor asks the processes “Are you alive?”, and process will respond “Yes”. If no answer is received from some process, the process is suspected as being dead (faulty)
- What are advantages and disadvantages of these two approaches?

Failure Detection Implementation

- Every process must know about who failed
- How to disseminate the information?
- What if not every node can communicate directly with another node?
  - Centralized
  - All-to-all
  - Gossip based: provides probabilistic guarantees

Metrics for failure detectors

- Detection time
- Mistake recurrence time
- Mistake duration
- Average mistake rate
- Query accuracy probability
- Good period duration
- Network load