Cristina Nita-Rotaru

7610: Distributed Systems

Physical and logical clocks. Global states. Failure detection.

QUESTIONS:

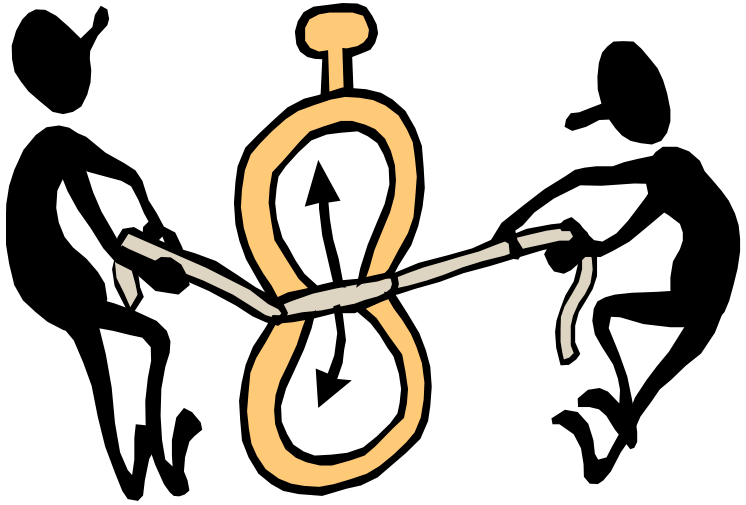
Please type in zoom chat window

Plan for today

- Why ordering events is needed in distributed systems
- Physical time
- Questions
- Logical time
- Questions
- Global state and snapshot, Chandi-Lamport
- Questions
- Failure detectors
- Questions

Ordering events in distributed systems

- Events in message-passing distributed systems
  - Send a message
  - Receive a message
- Time is essential for ordering events in a distributed system
  - Physical time:
    - global (wall) clock
    - logical clock
  - Logical time:
    - Lamport clocks,
    - vector clocks



Reading for this lecture

- L. Lamport for "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, July 1978, 21(7):558-565. E.W. Dijkstra Prize 2000, SIGOPS Hall of Fame.
- Mattern, F. "Virtual Time and Global States of Distributed Systems", in Cosnard, M., Proc. Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France: Elsevier, pp. 215–226, 1988
- K. M. Chandy and L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems, Vol. 3, No. 1, February, 1985, pp. 63-75. SIGOPS Hall of Fame.
- T. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems, 1996.

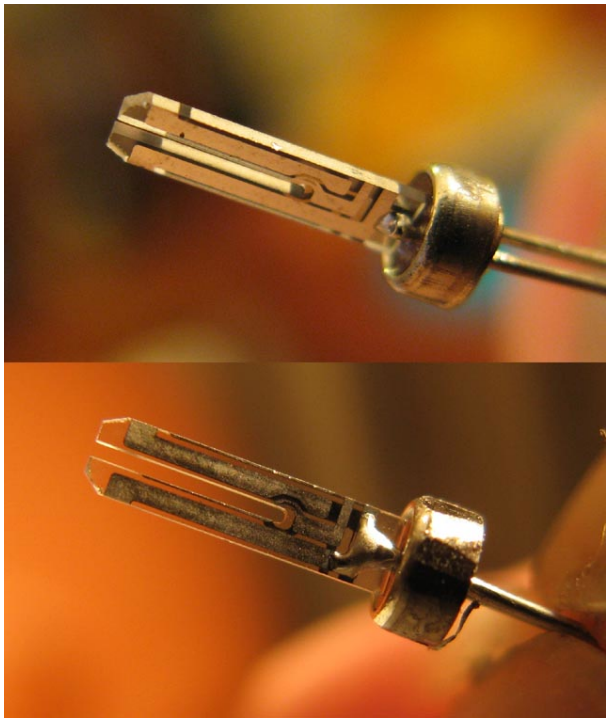
1: Physical time

Historical clocks

- Humans used a variety of devices to measure time
  - Sundials
  - Astronomical clocks
  - Candle clocks
  - Hourglasses
- Mechanical clocks developed in medieval ages
  - Typically maintained by monks (church bell tower)

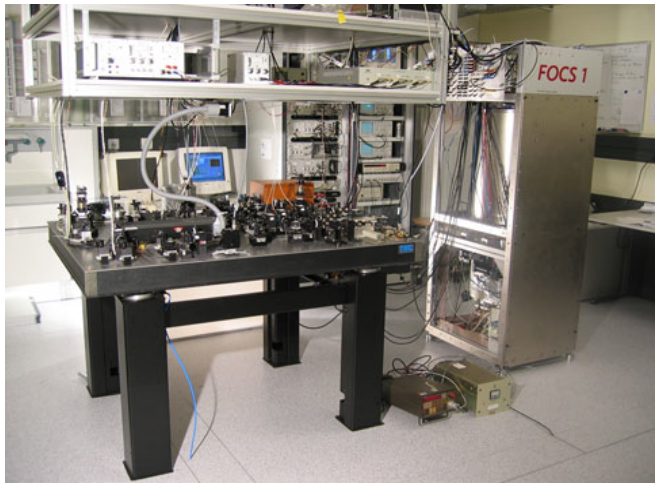
Electrical clocks

- 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 e.g., 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



GMT, UT1, and UTC

10

- **GMT**: Greenwich Mean Time
  - Originally, mean solar time at 0° longitude
  - This isn't really "noon" due to Earth's axial tilt
- **UT1**: Universal Time
  - Modernized version of GMT
  - Based on rotation of Earth, ~86,400 seconds/day
- **UTC**: Universal Coordinated Time
  - UT1 + leap seconds
  - Minutes can have 59-61 seconds
  - Since 1972, 25 leap seconds have been introduced

International Atomic Time

11

- Atomic clocks used to define a number of time standards
- TAI: International Atomic Time
  - Avg. of 200 atomic clocks, corrected for time dilation
- Essentially, a count of the number of seconds passed

- **Count was 0 on Jan. 1, 1958**
- UTC: since January 1, 1972, it has been 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 will carry 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 (Linux 2.6+ adjustable up to 1000 Hz)
  - Programmable Interval Timer (PIT) – Intel 8253, 8254
  - Interrupt service procedure adds 1 to a counter in memory
- **Quartz oscillators oscillate at slightly different frequencies, clocks do not agree in general !!!**

What does it mean for a clock to be correct?

- Relative to an "ideal" clock
  - Clock skew is magnitude
  - Clock drift is difference in rates
- Say clock is correct within p if

$$(1-p)(t'-t) \leq H(t') - H(t) \leq (1+p)(t'-t)$$

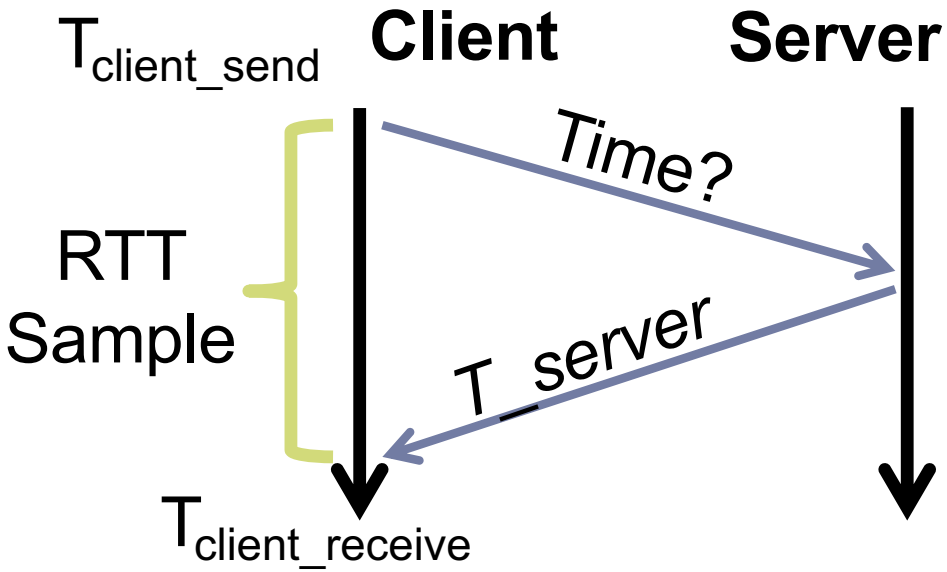
  - $(t'-t)$ True length of interval
  - $H(t') - H(t)$ Measured length of interval
  - $(1-p)(t'-t)$ Smallest acceptable measurement
  - $(1+p)(t'-t)$ Largest acceptable measurement
- Monotonic property: $t < t' \Rightarrow H(t) < H(t')$

Monotonicity

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

Cristian's Algorithm

- Assumes a time server has the accurate time and a client synchronizes with it
  - Client asks the time server for time
  - Server sends its time T_{server}
  - Client estimates how long it takes to receive answer from server as RTT/2 where:
  - $RTT = (T_{client\_receive} - T_{client\_send})$
  - Client adjusts its clock



$$T_{client} = T_{server} + (RTT / 2)$$

Cristian's Algorithm accuracy

- Assumes that it takes the same amount of time to send the request and receive the answer
- Minimum time to transmit a message one-way: min
- Time to receive the server's message is [min, RTT – min]
- Time at client [T_{server} + min, T_{server} + RTT – min]

accuracy is $\pm(RTT/2 - min)$

Berkeley Algorithm

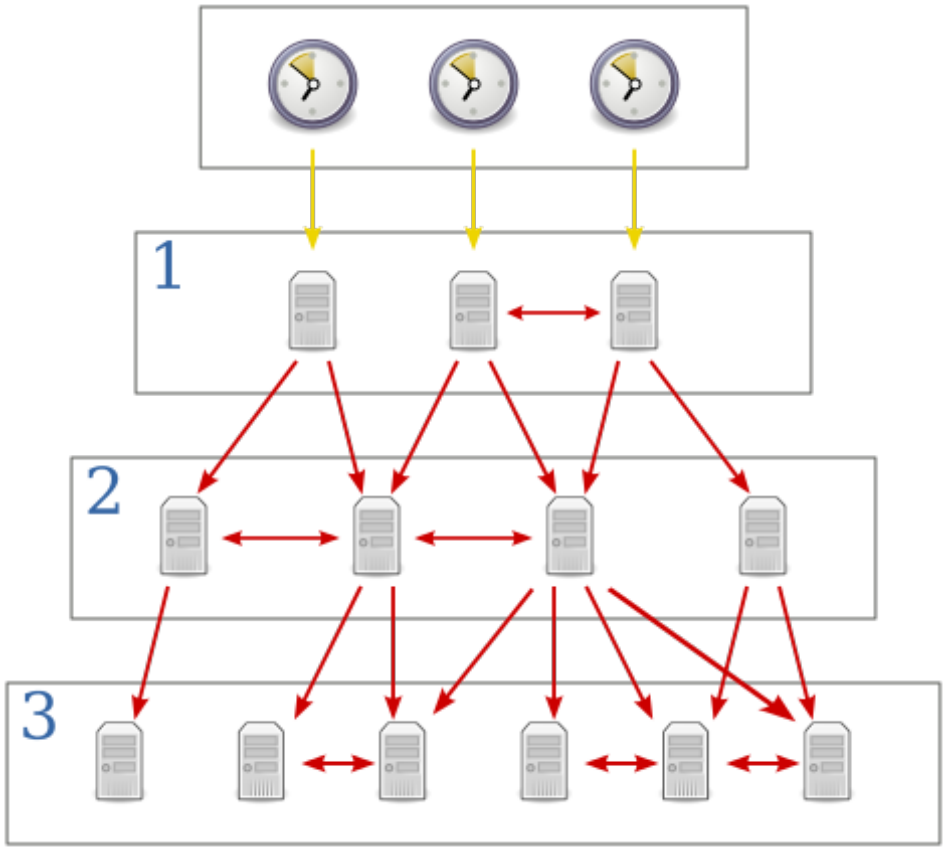
- Assumes no machine has an accurate time source; uses an elected master to synchronize
- Master coordinates:
  - Queries all clients for their local time
  - Estimates the clients' local time (as Cristian's Algorithm)
  - Averages all times including its own, excluding the ones that are too drifted
  - Tells each client the offset with each they need to adjust
- Some systems use multiple time servers
- Time is more accurate, but still drifts

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)
  - LANs: Precision to 1ms
  - Internet: Precision to 10s of ms
  - **NTP pool** is a dynamic collection of computers that volunteer to provide time via the NTP, about 4000 servers

NTP Hierarchy

- Based on hierarchy of accuracy, called strata
  - Stratum 0: High-precision atomic clocks
  - Stratum 1: Hosts directly connected to atomic clocks
  - Stratum 2: Hosts that run NTP with stratum 1 hosts
  - Stratum 3: Hosts that run NTP with stratum 2 hosts
  - …
- Stratum *x* hosts often synch with other stratum *x* hosts
  - Provides redundancy

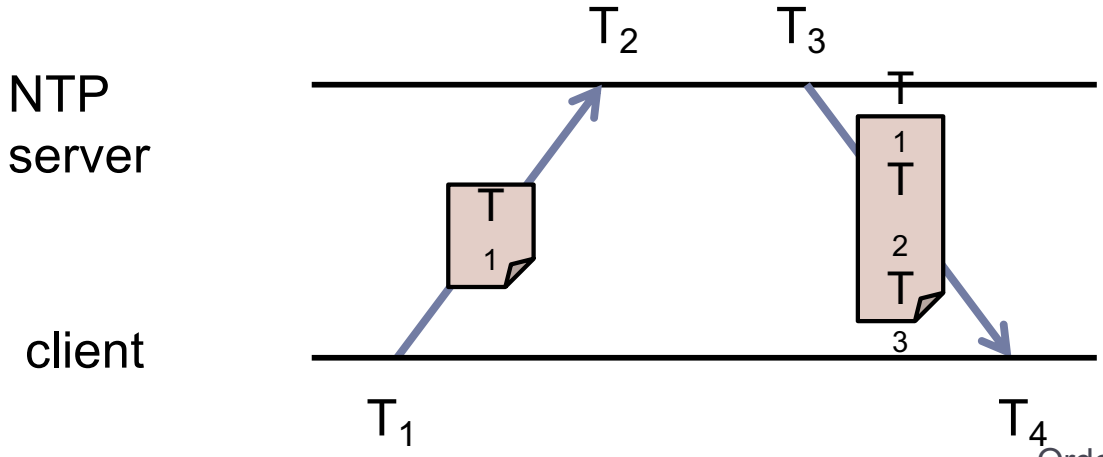


Reference clocks

- 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

On-the-wire protocol

- Client initiates request by recording timestamp T1, placing in packet, then sending to NTP server
- NTP Server records timestamp T2 when receiving request packet (and can do other processing if needed)
- When ready to send a reply, the NTP server records timestamp T3, places T1, T2, T3 in reply and sends back to client
- Client receives reply and records timestamp T4



Updating the clock

- Client calculates offset between his clock and server's clock, and updates his clock by that amount
- To synchronize exactly, client needs to know one-way delay between server and client
  - This is difficult in practice to ascertain, so NTP assumes path is symmetrical and one-way delay is half of round trip time
  - Offset is calculated to be: ½ [(T2 - T1) + (T3 - T4)]

NTP in practice

- Run on UDP port 123
  - Most Internet hosts support NTP
- Accuracy on general Internet is ~10ms
  - Up to 1ms on local networks, ideal conditions
- Many networks run local NTP servers
  - E.g., *time.ccs.neu.edu*
- NTP has been a vector for DDoS attacks
  - Best practice is for servers to filter requests outside local network

QUESTIONS:

Please type in zoom chat window

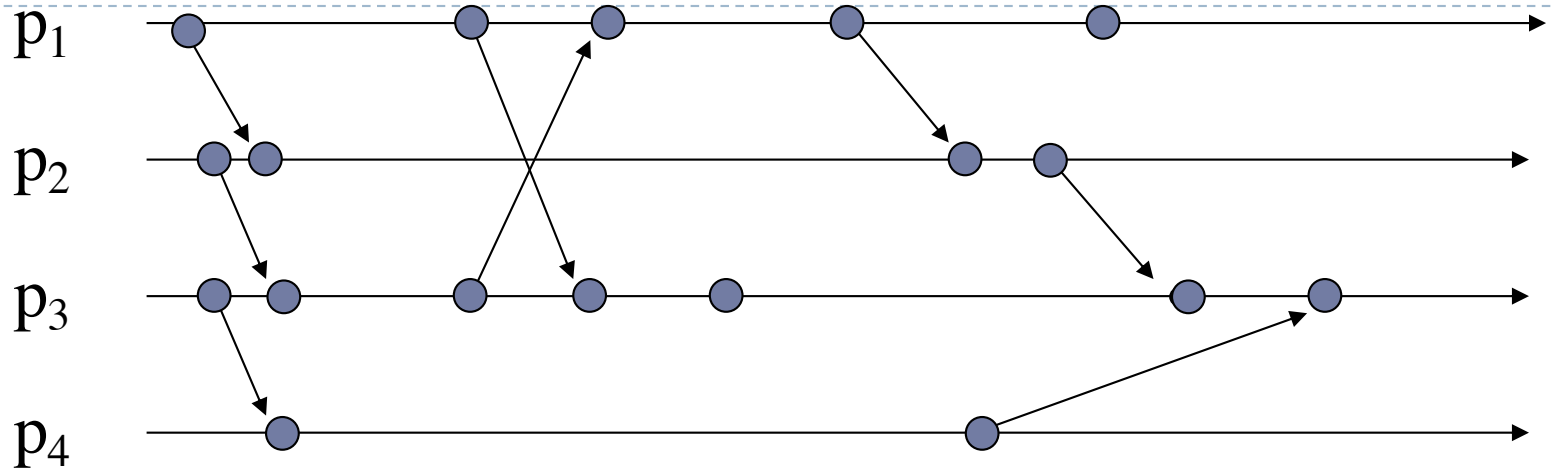
2: Logical time

From physical clocks 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" $\rightarrow$



- 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

- **Definition**: 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).
- **Definition**: 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).

Logical Clocks: Lamport Clocks (1978)

- Each process maintains his own clock C_i (a counter)
- Clock Condition: for any events a and b at process p_i

$$\textbf{if } a \rightarrow b \textbf{ then } C_i(a) < C_i(b)$$

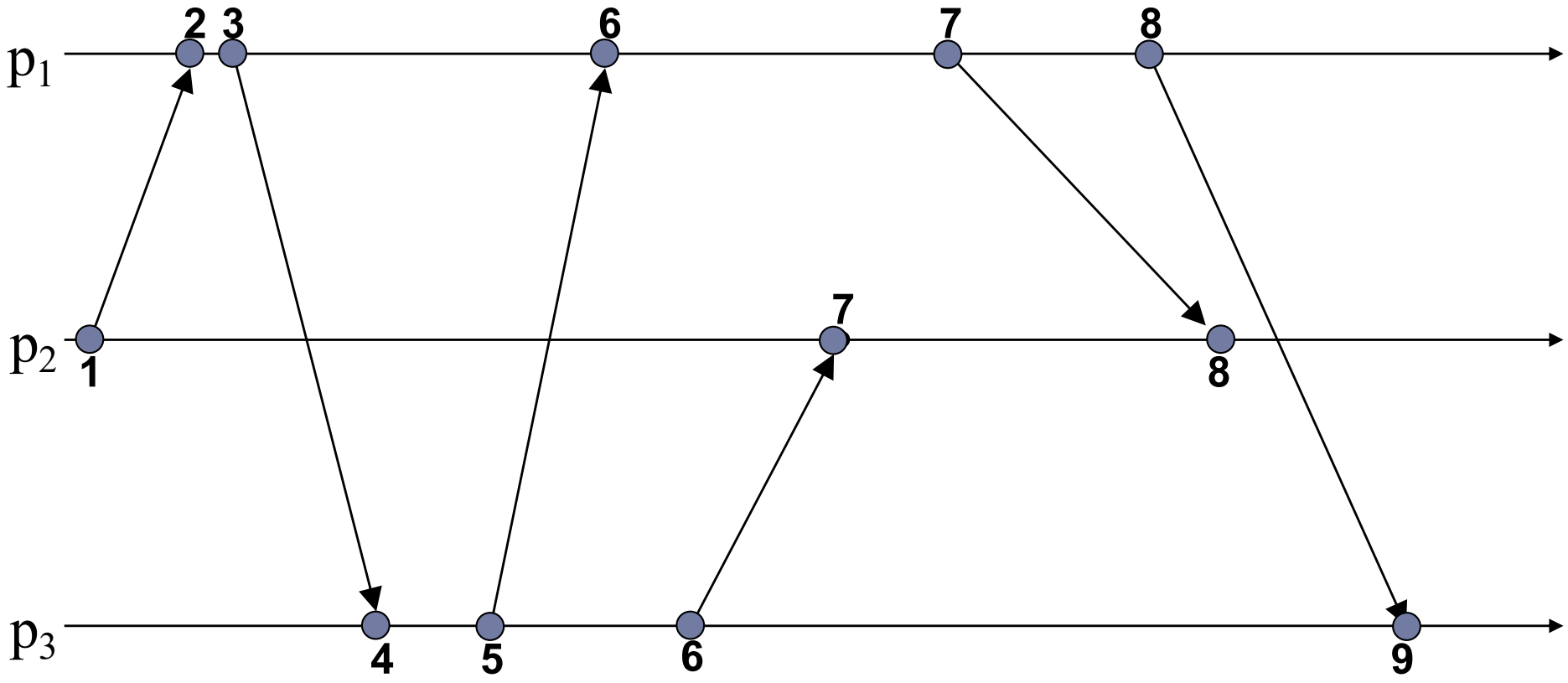
- Implementation: each process p_i
  - increments C_i between any successive events
  - on sending a message m, attaches to it local clock C_i

$$T_m = C_i(a)$$

  - on receiving of message m from process p_k sets C_i to

$$C_i = \max(C_i, T_k) + 1$$

Lamport Clocks: Example



Lamport Clocks: Total Order

- Logical Clocks only provide partial order
- Create Total Order by breaking the ties
- Example to break ties, use process identifiers, have an order on process identifiers:
  - 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

- 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, in other words 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

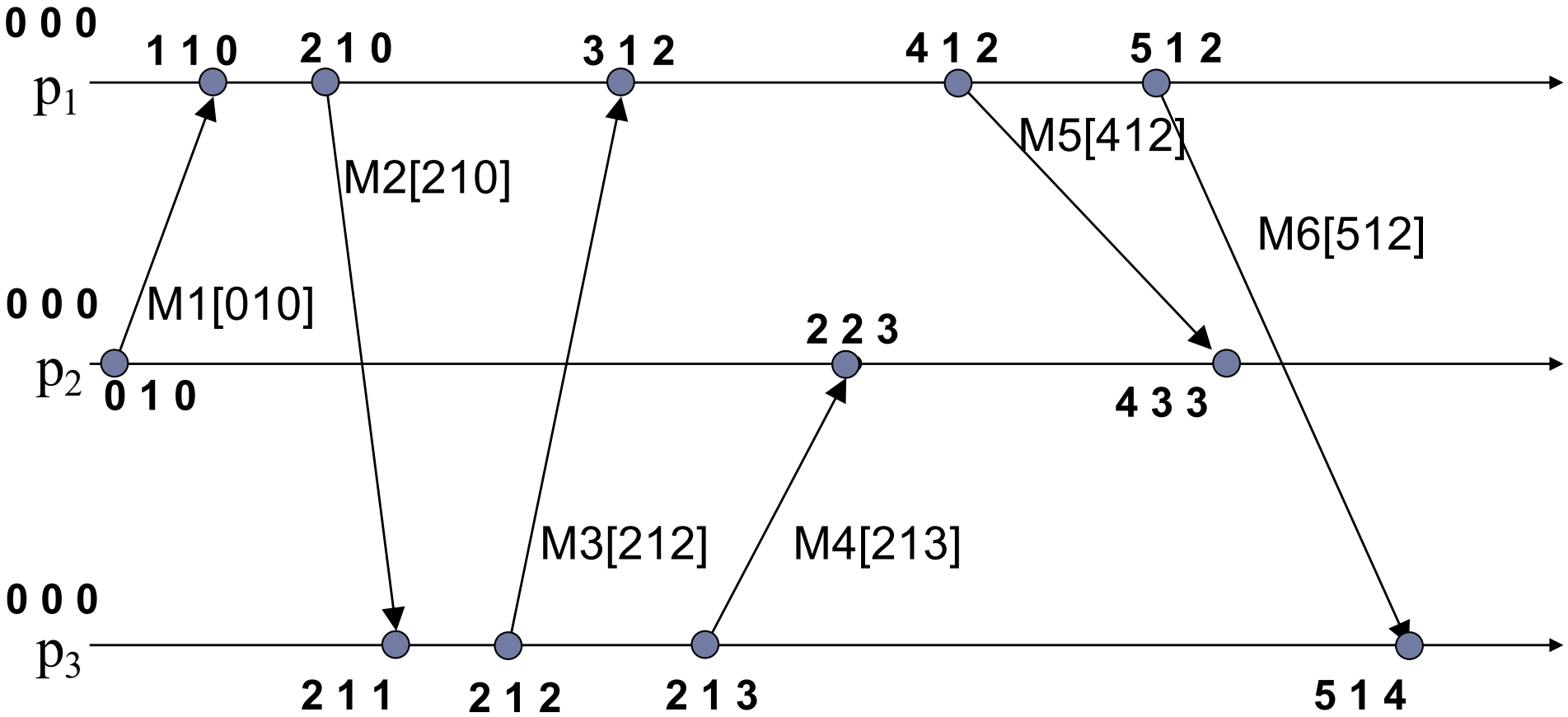
Vector Clocks

- Independently developed by Colin Fidge and Friedemann Mattern in 1988.
- Each process p_i maintains a vector C_i

$$C_i = [0, 0, ..., 0].$$

- 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 on m.
- When p_i receives a message m, increments its own clock and updates the clock for the other processes as follows
  $\forall\ j: 1 \leq j \leq n, j \neq i: C_i[j] = \max(C_i[j], m.C[j])$
  $C_i[i] = C_i[i] + 1.$

Vector Clocks: Example



How to Order with Vector Clocks

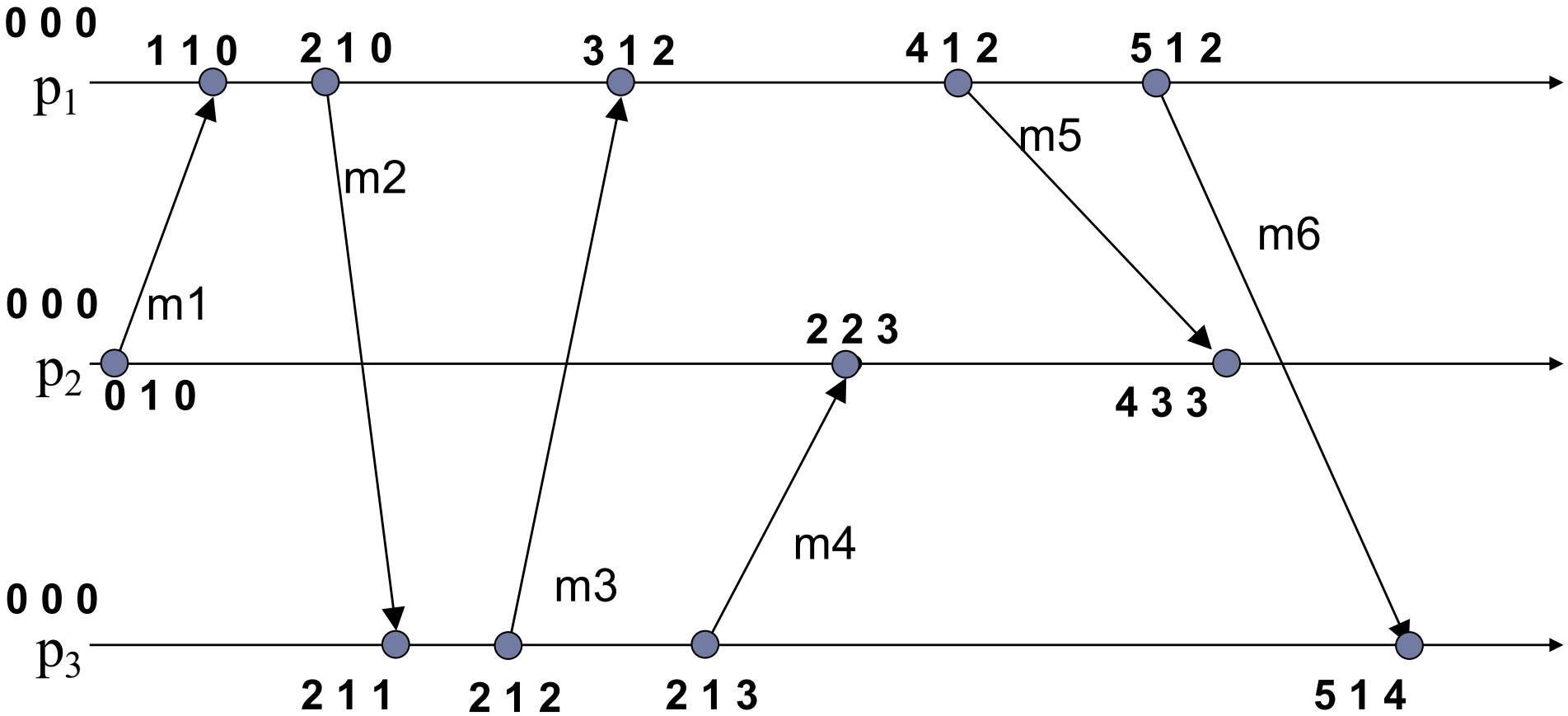
- Given two events a and b, $a \rightarrow b$ if and only if

- V(a) is less than or equal to V(b) for all process indices, and at least one of those relationships is strictly smaller.
- Otherwise, we say they are concurrent or independent ||

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

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

What Events Are Independent?



QUESTIONS:

Please type in zoom chat window

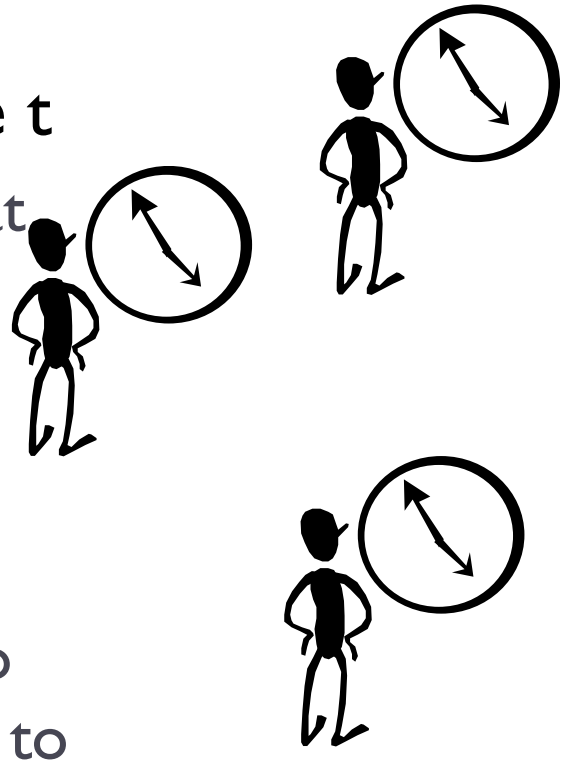
3: Global states and Chandi-Lamport Snapshot Algorithm.

Why do we need global snapshots?

- Global snaphot gives you the "global view" of the system Examples of applications where global snapshots are useful:
  - 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: Model

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 ordered
  - 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 (in case it was not the one that initiated the algorithm)
  - What messages to include in the snapshot
  - When did all the other processes recorded their snapshot
- Key design: uses a control message, **marker**
  - separate messages between those to be included in the snapshot from those not to be recorded in the snapshot.
  - inform other processed that it has recorded its snapshot
  - inform 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

- Can be initiated by any process by executing the "Marker Sending Rule"
- A process executes the "Marker Receiving Rule" on receiving a marker.
  - If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes 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.
- All the local snapshots get disseminated to all other processes and all the processes can determine the global state.

Chandy/Lamport Snapshot Algorithm

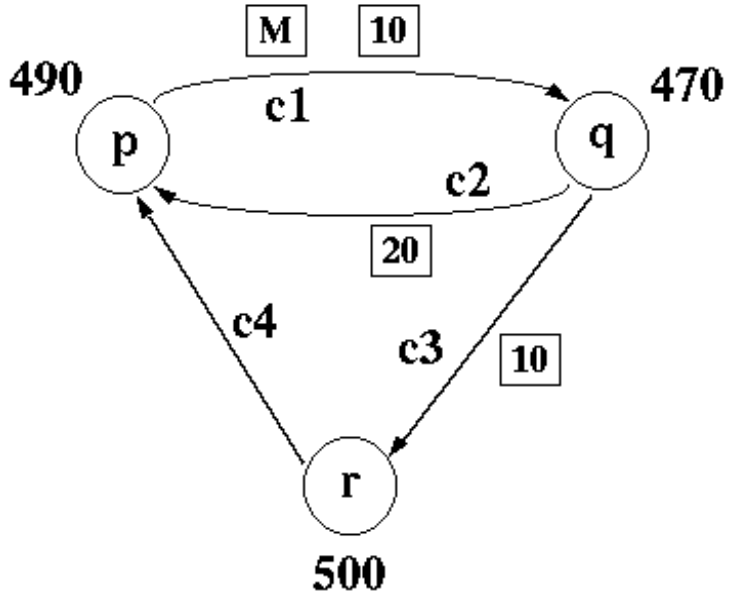
- Marker-sending rule for a process p:
  - Saves its own local state
  - Sends a marker to all other processes on their corresponding channels before sending any other message
- Marker-receiving rule for a process q on channel c
  - If q has not recorded its state then
    - q records its state
    - q record the state of incoming channel c as "empty"
    - turn on recording of messages over other incoming channels
    - for each outgoing channel c, send a marker on c
  - else
    - q records the state of incoming channel c as all the messages received over c after q recorded its state and before q received the marker along c

Example of Chandy-Lamport Algorithm

- 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). They all start with state = $500 and the channels are empty. The stable property is that the total amount of money is $1500.

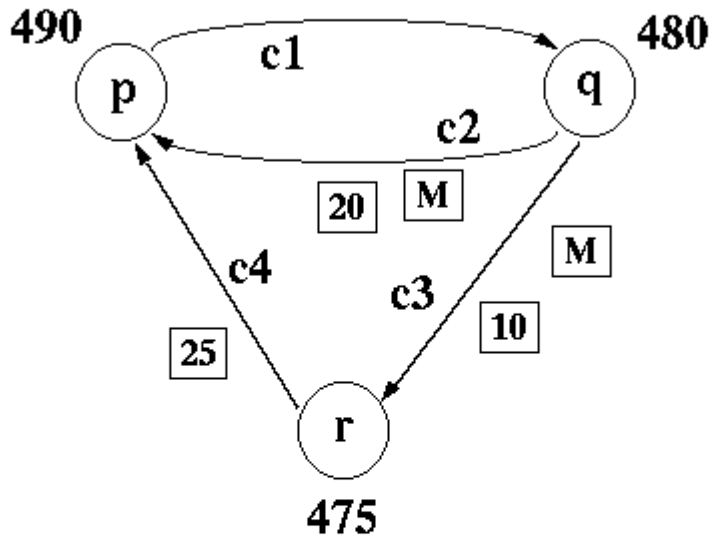
- Process p sends $10 to q and then starts the snapshot algorithm: records its current state 490 and sends out a marker on c1.
- Meanwhile q has sent $20 to p along c2 and 10 to r along c3.

Snapshot/State Recording Example (Step 1)



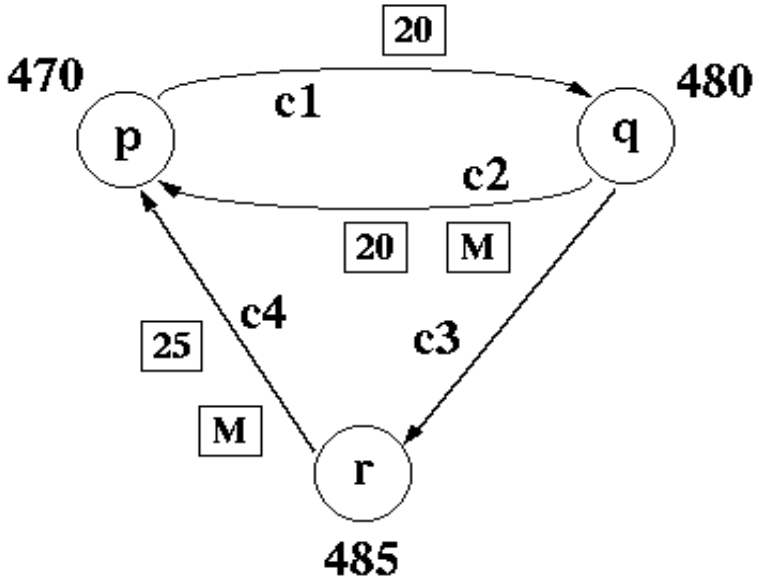
Node	Recorded State				
	state	c1	c2	c3	c4
p	490		{}		{}
q		{}			
r				{}	

Snapshot/State Recording Example (Step 2)



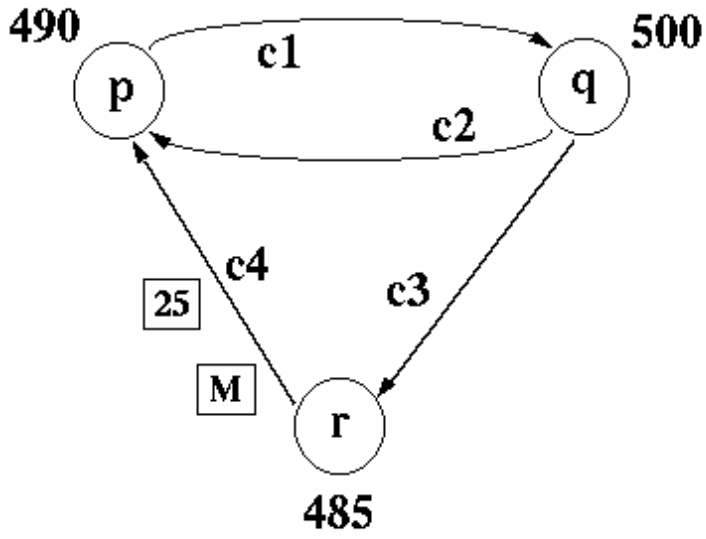
Node	Recorded State				
	state	c1	c2	c3	c4
p	490		{}		{}
q	480	{}			
r				{}	

Snapshot/State Recording Example (Step 3)



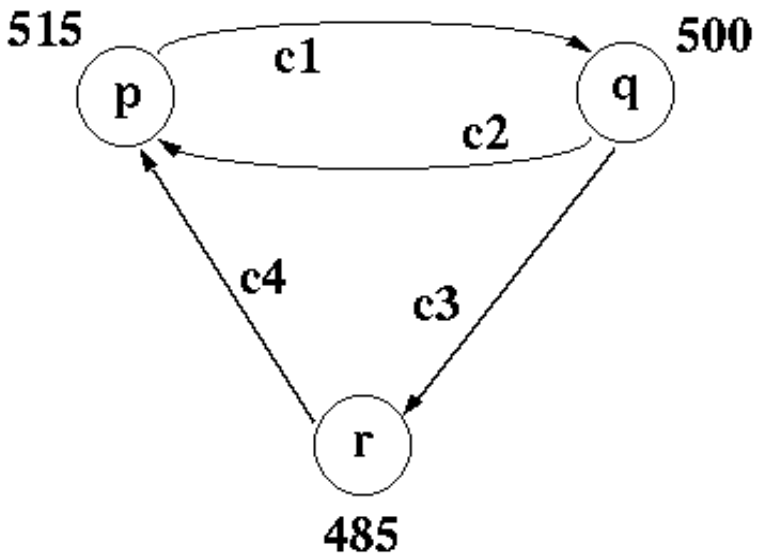
Node	Recorded State				
	state	c1	c2	c3	c4
p	490		{}		{}
q	480	{}			
r	485			{}	

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 in this case?
- Records a consistent global state of an asynchronous system.
- We need some definitions

History of events

- Given a process p_i
- Event e_i^j is the event j at process i
- History of process p_i, h_i is a sequence of events that happened at p_i

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

- Prefix history at p_i up to k, is the history of p_i 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

History of events: More definitions

- Given a set of processes
- Global history: the set of all processes' histories
  - $H = \cup_i (h_i)$
- Global state: the set of states at each process

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

- Cut: a set of prefix histories

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

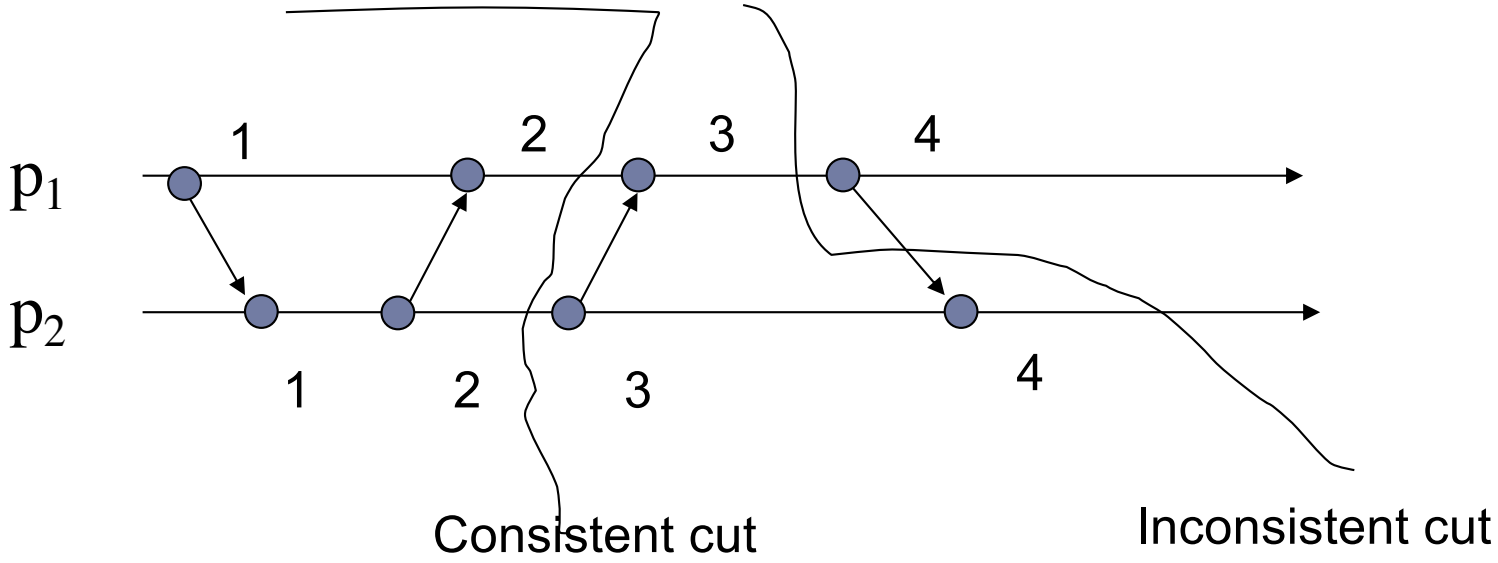
- Frontier of a cut: the set of last event that happened in each prefix history

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

Consistent Cuts

Definition: 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

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



How do we use global states?

- Need more definitions:
- **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's ordering
- **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

- How do we use global states to reason about distributed systems?
- 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

- Look for undesirable properties, "bad things"
- 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

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

Stable Global States and Liveness

- Look for desirable properties, "good things"
- Assume that a "good think" 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

QUESTIONS:

Please type in zoom chat window

4: Detecting Failures

Failure detectors as an abstraction

- Failure detector: distributed oracle that makes guesses about process failures
- **Accuracy**: the failure detector makes no mistakes when labeling processes as faulty
- **Completeness**: the failure detector "eventually" (after some time) suspects every process that actually crashes
- Detectors classified based on their properties
- 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**: Some correct process is never suspected. (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
- THIS IS AN ABSTRACTION
- IT IS IMPOSSIBLE TO HAVE A PERFECT FAILURE DETECTOR
- We have to live with … unreliable failures detectors…

Unreliable failure detectors

- Unreliable failure detectors can make mistakes !!!
- A process is suspected that it was faulty, that can be true or false, if false the list of alive processes is modified.
- Failure detectors can add/remove processes from the list of suspects; different processes have different lists.
- The assumptions are that:
  - After a while the network becomes stable so the failure detector does not make mistakes anymore.
  - In the unstable period, the failure detector can make mistakes.

Failure detection implementation

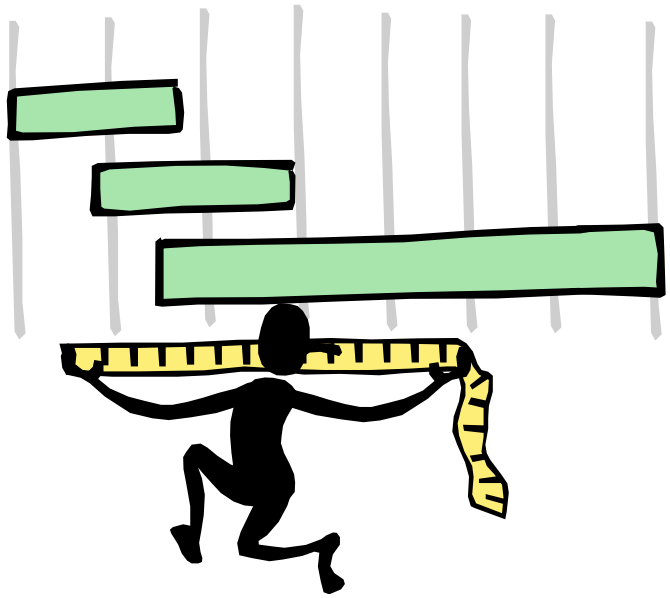
- Push: processes keep sending heartbeats "I am alive" to the monitor. If no message is received for awhile from some process, that process is suspected as being dead (faulty).
- Pull: monitor asks the processes "Are you alive?", and process will respond "Yes, I am alive". If no answer is received from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

Failure detectors implementation (2)

- Every process must know about who failed
- How to disseminate the information
- How about 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



Summary

- Ordering events with logical clocks
  - Lamport clocks uses a single clock per process,
  - Vector clocks – each process maintains a clock for all the other processes
- Determining global states
  - Chandi-Lampprt algorithm for asynchronous systems, no failures and communication FIFO unidirectional.
- Detecting failures
    - There are no perfect failure detectors, both accurate and complete; push or pull methods

QUESTIONS:

Please type in zoom chat window