Cristina Nita-Rotaru

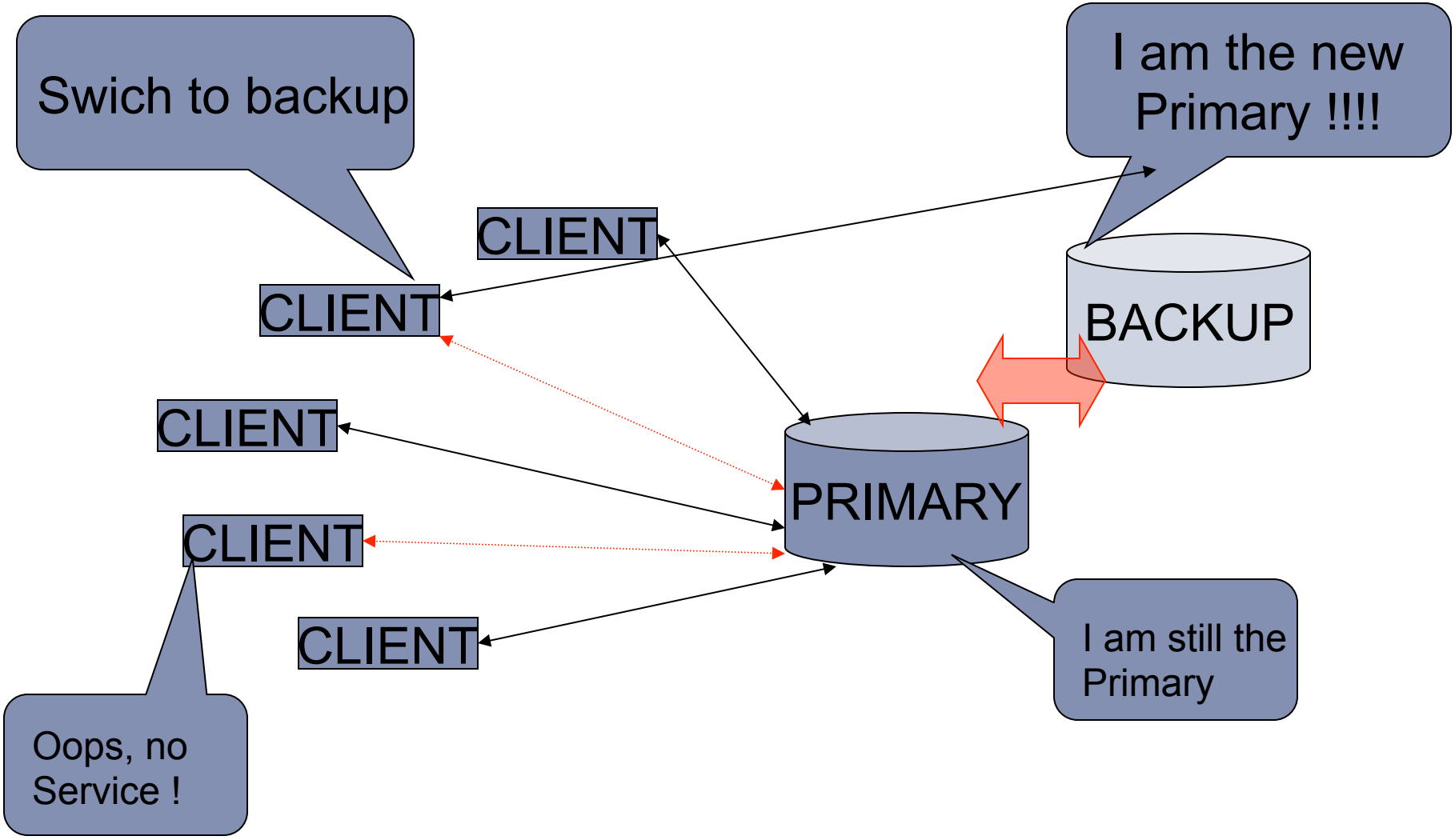
CS505: Distributed Systems

Ordering events. Lamport and vector clocks. Global states. Detecting failures.

Required reading for this topic…

- Leslie Lamport,"Time, Clocks, and the Ordering of Events in a Distributed System," 1978
- Colin J. Fidge "Timestamps in Message-Passing Systems That Preserve the Partial Ordering," 1988
- K. Mani Chandy and Leslie Lamport, ``Distributed Snapshots: Determining Global States of Distributed Systems," 1985

When things go wrong…



1: Clock synchronization

System Model Dimensions

- Non-deterministic processes
- Communication is through messages
- Network packets can be lost, duplicated, delivered very late or out of order, spied upon, replayed, corrupted, source or destination can lie
- Communication can be authenticated or not
- Execution model can be
  - Asynchronous: no synchronized clocks or time-bounds on message delays.
  - Synchronous: execution is partitioned in rounds, all messages send in a round are delivered in that round

Execution, Configuration, Events

- **Participants**: set of processes p_i, each process with a state s_i
- **Configuration** C_t: set of state of each process at some moment
- **Events**: send or deliver a message, events can change the state at a process
- **Execution**: sequence of configuration and events

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"

Example

- Successfully passing the exam for this course
- Safety: in case you take the exam, do not fail it
- Liveness: you eventually have to take the exam

Ordering of Events

- Single process: follow the sequence of events, each event has a timestamp and the causality relation between events is given by time
- Distributed processes: many events generated at different processes, how to order events?
- Time is essential for ordering events in a distributed system
  - Physical time: local clock; global clock
  - Logical time: partial ordering, total ordering

What is Time?

- The second is the duration of 9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state at the temperature of 0 K of the caesium 133 atom.
- Unit for measuring time

World Time

- International Atomic Time (TAI): is a continuous count of seconds based on atomic clocks around the world.
- Coordinated Universal Time (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

- **Global clock**: processes have access to a central global clock, each event will carry a timestamp
- **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

Synchronizing Physical Clocks

- **External synchronization**: Consider the source S and the synchronization bound B > 0, then none of the clocks drift with more than B from S, at any time
- **Internal synchronization**: Consider the synchronization bound B>0, then at any time, the difference between any two clocks is within B
- Skew: the instantaneous difference between (the readings of) two clocks
- Drift rate: the difference between the clock and a nominal perfect reference clock per unit of time
- Networks are asynchronous and unreliable

Cristian's Algorithm

- Assumes a time server has the accurate time and client synchronizes with it
- How it works:
  - Client asks the time server for time
  - Server sends its time Tserver
  - Client estimates how long it takes to receive answer from server as RTT/2 where:
  - RTT = (Tclient_receive – Tclient_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 [Tserver + min, Tserver + RTT – min]

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

Berkeley Algorithm

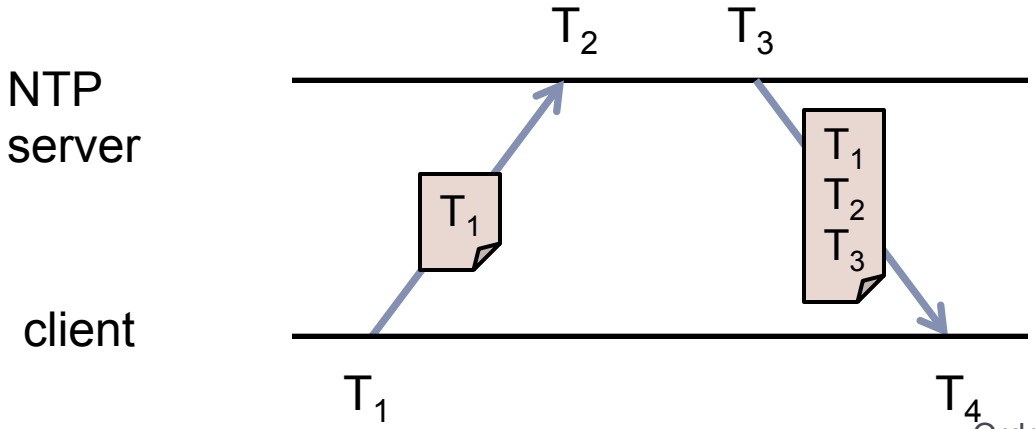
- Assumes no machine has an accurate time source; uses an elected master to synchronize
- Master coordinates:
  - queries the clients for their local time
  - estimates the clients' local time (similar to Cristian's Algorithm)
  - averages all times including its own, excluding the ones that are too 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 synchronizes a computer clock to other computers on the Internet or local time sources
- Has the goal of synchronizing clocks to less than a millisecond or two relative to Coordinated Universal Time (UTC)
- Trivia:
  - the most widely-used time synchronization protocol on the Internet today
  - there are over 2,000 NTP servers on the Internet today for public use
  - uses UDP as a transport protocol

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)]

Reference Clocks

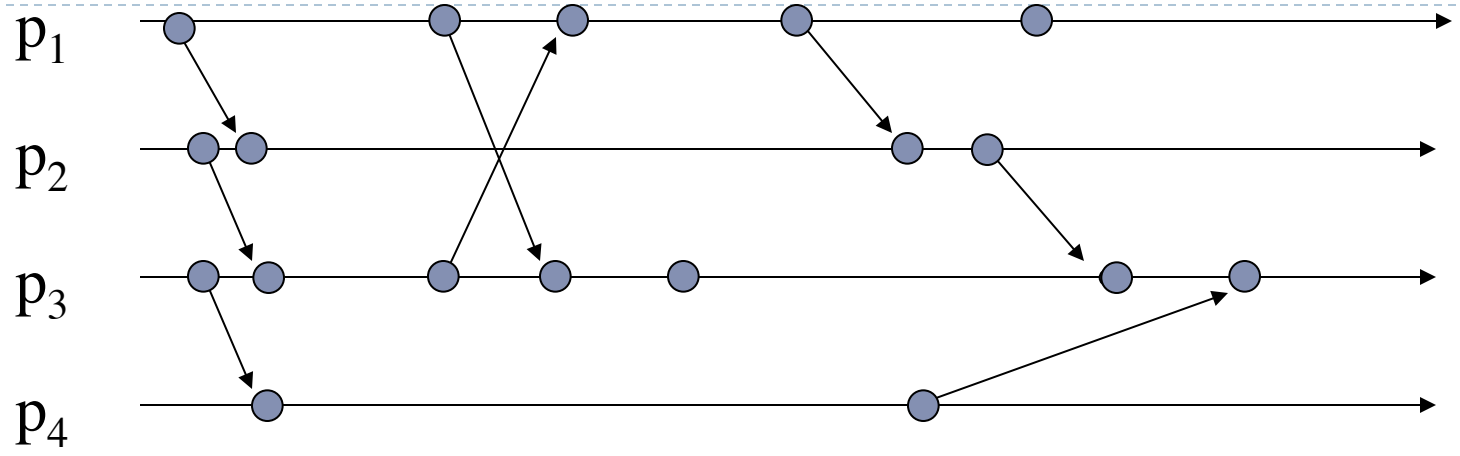
- Many NTP servers synchronize directly to UTC using specialized equipment
  - Atomic clocks: Ultimately are the root source of time in NTP
  - Global Positioning System (GPS): can synchronize with a satellite's atomic clock
  - Code Division Multiple Access (CDMA): can synchronize with a local wireless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

From Physical 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’

2: Lamport clocks

``HAPPENED BEFORE''



- If events a and b take place at the same process and a occurs before b then $a \rightarrow b$
- If a is a send event of m at p1 and b is a deliver event of the same m at p2, $p1 \neq p2$ 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 ∧ bRa ⇒ a = b (antisymmetric).
  - aRb ∧ bRc ⇒ 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

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

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

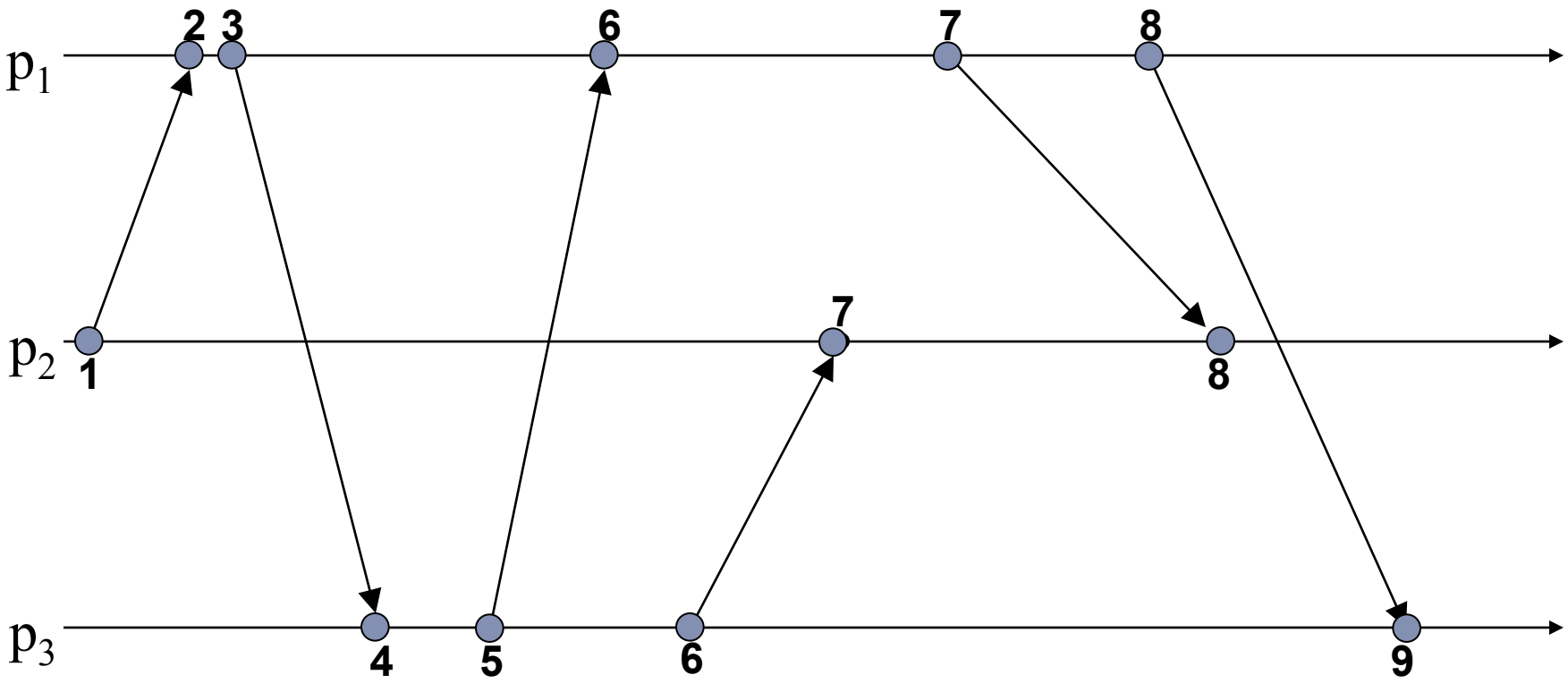
- Implementation:
  - each process p_i increments C_i between any successive events
  - on sending a message m, attach to the message local clock

    $T_m = C_i(a)$
  - on receiving of message m process p_k sets C_k to

    $C_k = \max(C_k, T_m) + 1$

Lamport Clocks: Example

$C_k = \max(C_k, T_m) + 1$



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 pi and b is event in p then

$$a \rightarrow b \quad \text{iff}$$

  - $C_i(a) < C_j(b)$ 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

3: Vector clocks

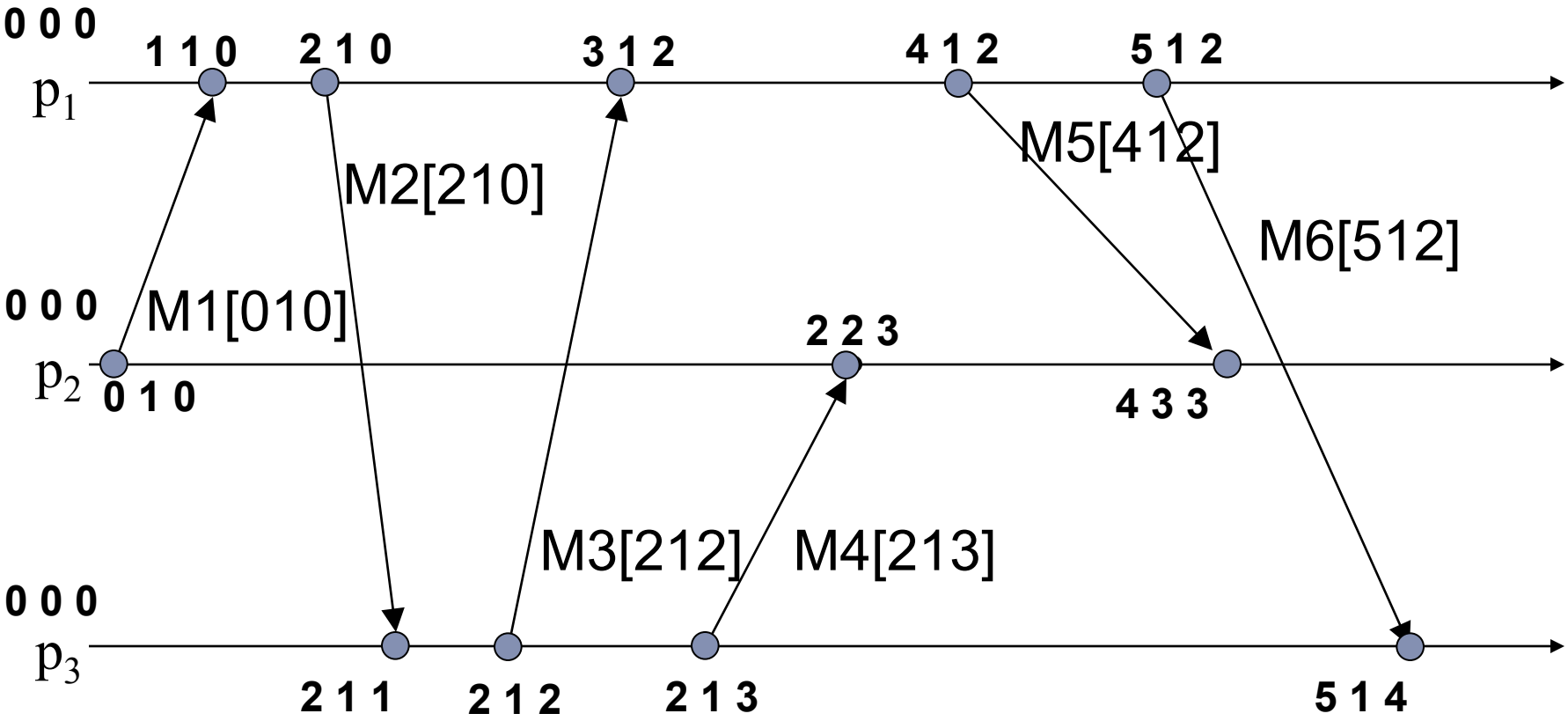
Vector Clocks

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

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

- 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
  $\forall\ j: 1 \leq j \leq n, j \neq i: Ci[j] = \max(Ci[j], m.C[j])$
  $Ci[i] = Ci[i] + 1.$

Vector Clocks: Example



How to Order with Vector Clocks

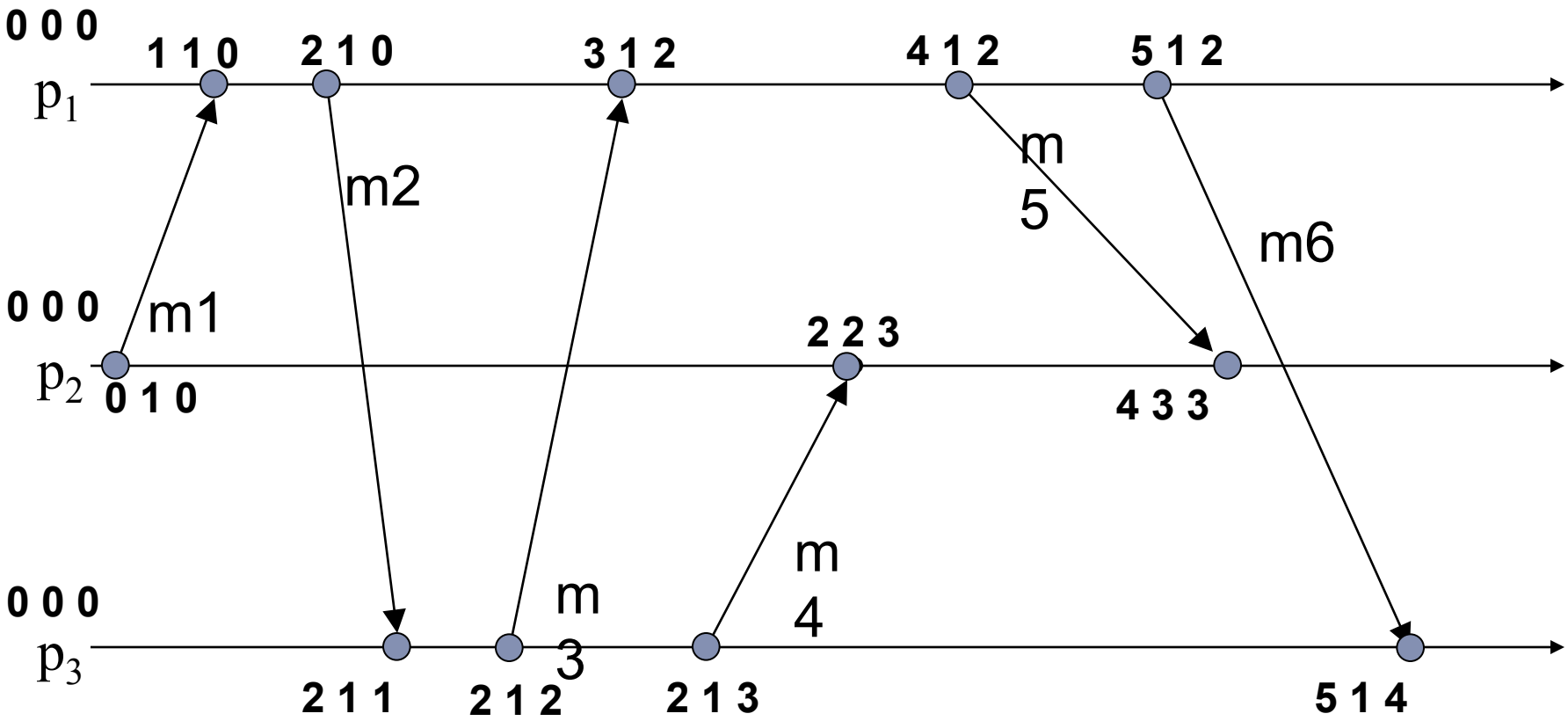
- Given two events a and b, $a \rightarrow b$ if and only if

- V(a) is less than or equal to V(b) for all process indices, and at least one of those relationships is strictly smaller.

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

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

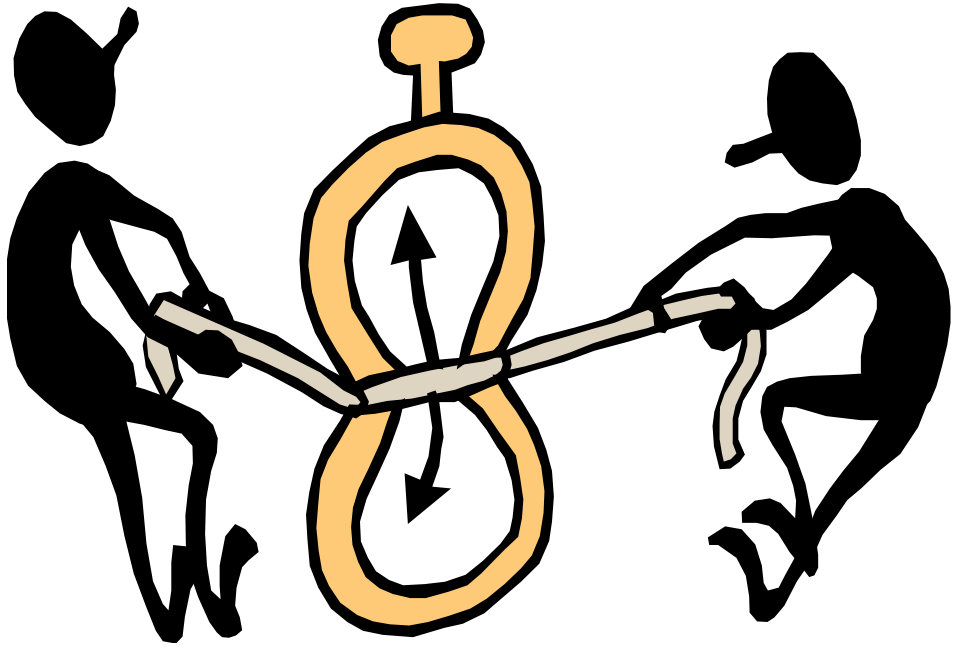
What Events Are Independent?



3: Determining global states

Ordering Events in Distributed Systems

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



History of Events: Some Definitions

- 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

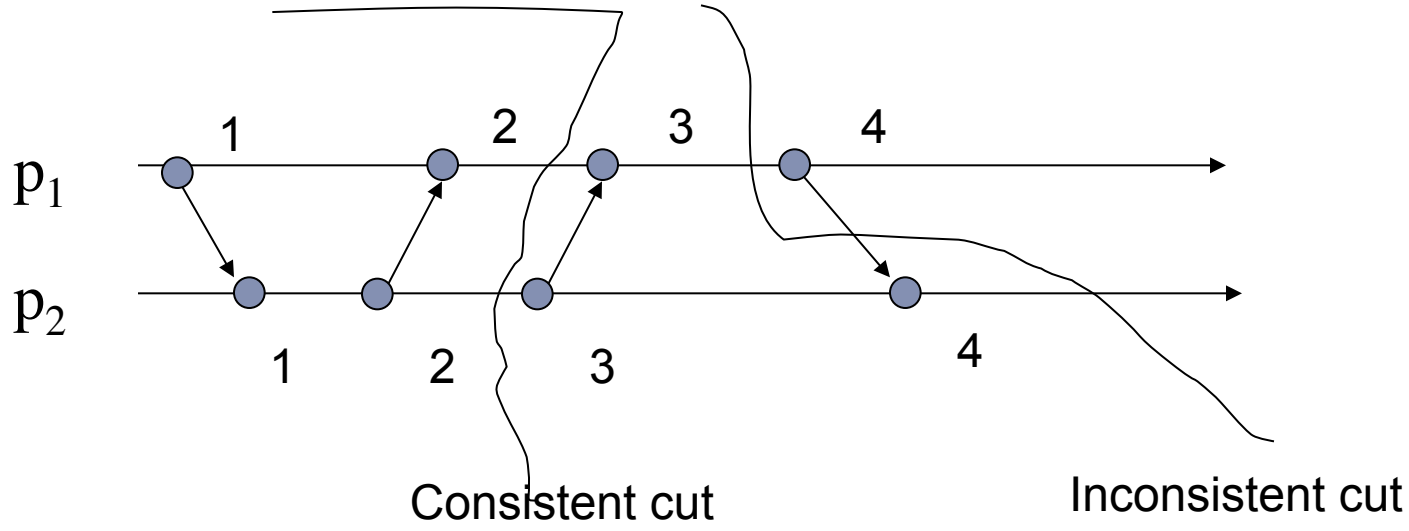
History of Events: More Definitions

- 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

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



Global States: 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 Predicate

- 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"

Remember 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

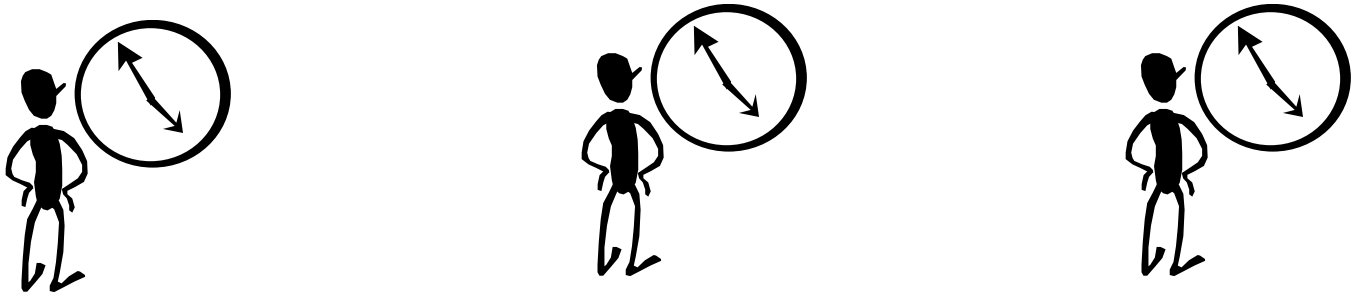
- Look for desirable properties, "good things"
- Assume that a "good think" GT (for example reaching termination) is a global-state-predicate and S_0 is the initial state of the system then

  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

Determining Global States

- 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?
- What if synchronized clocks are not available?



Recording Global States

- How to distinguish between the messages to be recorded in the snapshot from those not to be recorded?
- How to determine when a process takes its snapshot?

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 in the system records its local state and the state of its incoming channels

Chandy-Lamport Algorithm

- Uses a control message, **marker**, to separate messages in the channels between those to be included in the snapshot from those not to be recorded in the snapshot.
- After a process has recorded its snapshot, it sends a marker before sending out any more messages.
- 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

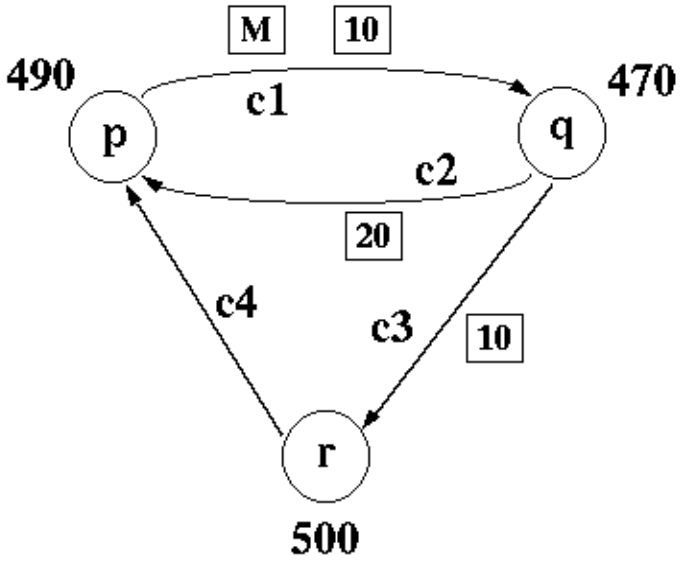
- Marker-sending rule for a process p:
  - Saves its own local state
  - Sends a marker to all other processes on their corresponding channels before sending any other message
- Marker-receiving rule for a process q
  - If q has not recorded its state then
    - q records records its state
    - q record 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 incomming 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. Communications chanels, 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.

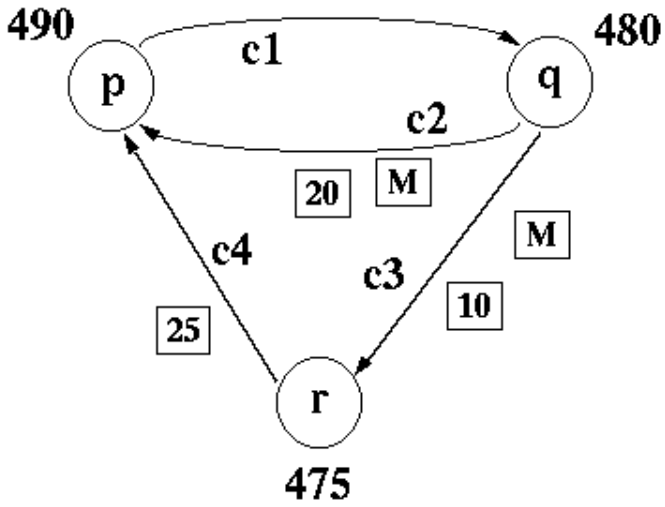
- Process p sends $10 to q and then starts the snapshot algorithm: records its current state 490 and sends out a marker on c1.
- Meanwhile q has sent $20 to p along c2 and 10 to r along c3.

Snapshot/State Recording Example (Step 1)



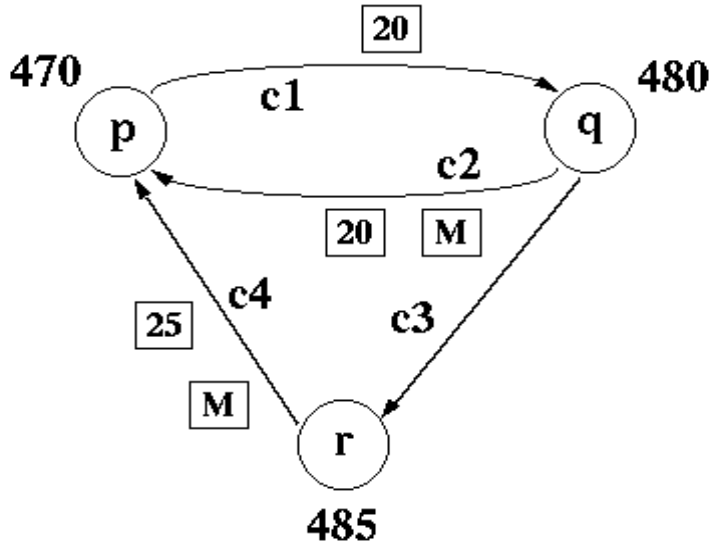
Node	Recorded State				
	state	c1	c2	c3	c4
p	490		{}		{}
q		{}			
r				{}	

Snapshot/State Recording Example (Step 2)



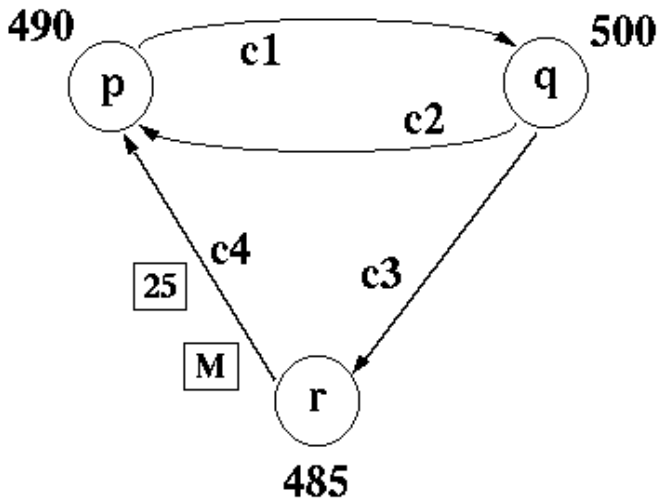
Node	Recorded State				
	state	c1	c2	c3	c4
p	490		{}		{}
q	480	{}			
r				{}	

Snapshot/State Recording Example (Step 3)



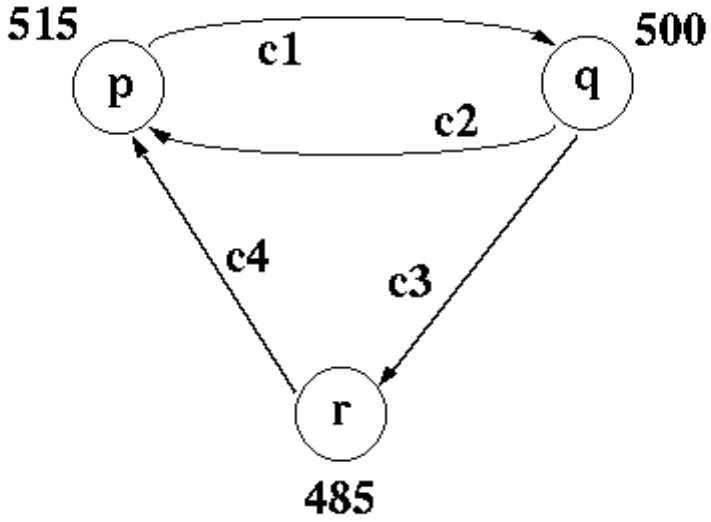
Node	Recorded State				
	state	c1	c2	c3	c4
p	490		{}		{}
q	480	{}			
r	485			{}	

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			{}	

4: Detecting failures

Detecting failures

- Impossibility result: it is impossible to design an asynchronous fault-tolerant consensus algorithm, even when only one process can crash. (FLP85)
- Proof Idea: It is shown how an infinite sequence of events can be constructed such that the algorithm never terminates (stays indecisive forever).
- The impossibility comes from the fact that in an asynchronous system, it is impossible to distinguish between a faulty-process and a slow process.
- We will come back to the proof

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

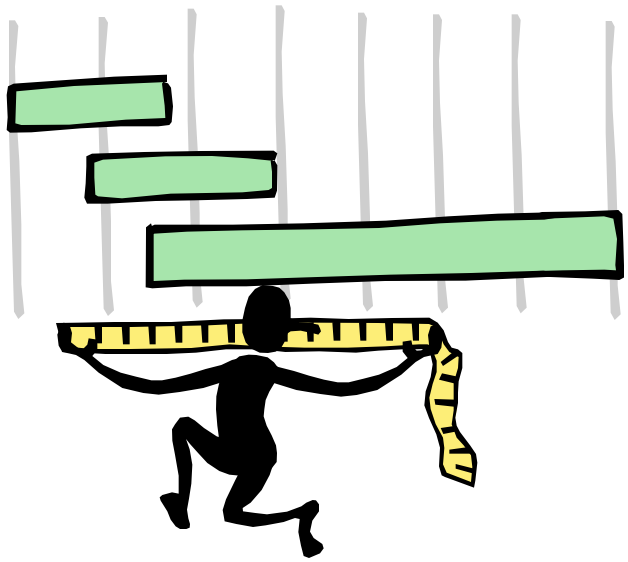
- Push: processes keep sending heartbeats "I am alive" to the monitor. If no message is received for awhile from some process, that process is suspected as being dead (faulty).
- Pull: monitor asks the processes "Are you alive?", and process will respond "Yes, I am alive". If no answer is received from 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-Lamprt algorithm for asynchronous systems, no failures and communication FIFO unidirectional.
- Detecting failures
  - There are no perfect failure detectors, both accurate and complete