Time, Clocks, and State Machine Replication

Dan Ports, CSEP 552

Today's question

- **How do we order events in a distributed system?**
  - physical clocks
  - logical clocks
  - snapshots
  - (break)
  - application: state machine replication
    (Chain Replication / Lab 2)

Why do we need to order events?

Distributed Make

- Central file server holds source and object files
- Clients specify modification time on uploaded files
- Use timestamps to decide what needs to be rebuilt
    if object O depends on source S,
    and O.time < S.time, rebuild O

- What goes wrong?

Another example: Facebook

- Remove boss as friend
- Post “My boss is the worst, I need a new job!”

- Don’t want to get these in the wrong order!

Why would we get these in the wrong order?

- Data is not stored on one server - actually 100K+
- Privacy settings stored separately from post
- Lots of copies of data: replicas, caches in the data center, cross-datacenter replication, edge caches
- How do we update all these things consistently?
    - Can we just use wall clocks?

Physical clocks

- Quartz crystal can be distorted using piezoelectric effect, then snaps back
  => results in an oscillation at resonant frequency

- affected by crystal variations, temperature, age, etc

- Crystal oscillator (~1¢)
  5 min / yr

- Oven-controlled XO (~$50-100)
  1 sec / yr

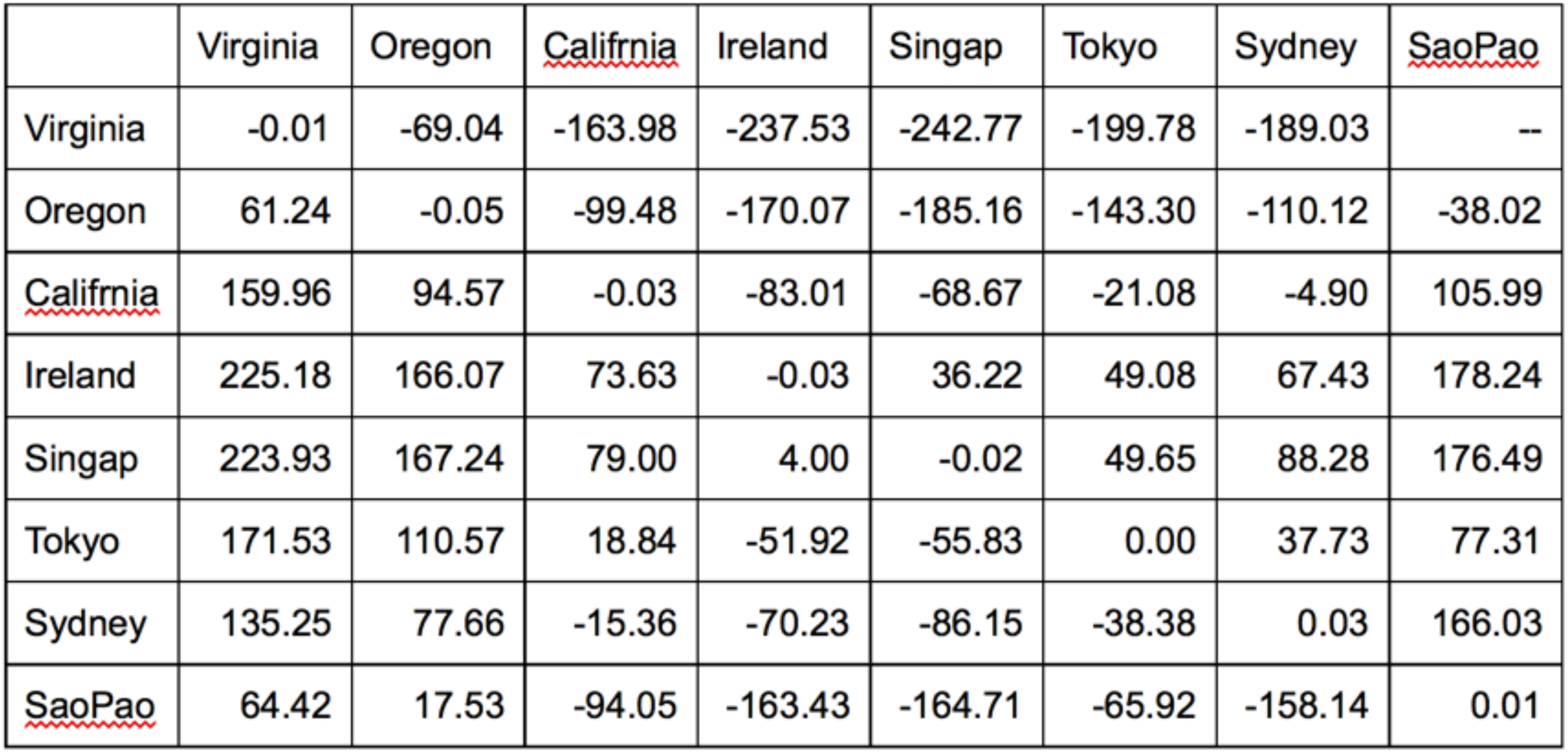
- Rubidium atomic clock (~$1k)
  <1 ms / yr

- Cesium atomic clock ($∞)
  100 ns / yr

How well are clocks synchronized in practice?

(measurements from Amazon EC2)

How well are clocks synchronized in practice?



	Virginia	Oregon	Califrnia	Ireland	Singap	Tokyo	Sydney	SaoPao
Virginia	-0.01	-69.04	-163.98	-237.53	-242.77	-199.78	-189.03	--
Oregon	61.24	-0.05	-99.48	-170.07	-185.16	-143.30	-110.12	-38.02
Califrnia	159.96	94.57	-0.03	-83.01	-68.67	-21.08	-4.90	105.99
Ireland	225.18	166.07	73.63	-0.03	36.22	49.08	67.43	178.24
Singap	223.93	167.24	79.00	4.00	-0.02	49.65	88.28	176.49
Tokyo	171.53	110.57	18.84	-51.92	-55.83	0.00	37.73	77.31
Sydney	135.25	77.66	-15.36	-70.23	-86.15	-38.38	0.03	166.03
SaoPao	64.42	17.53	-94.05	-163.43	-164.71	-65.92	-158.14	0.01

(measurements from Amazon EC2)

How well are clocks synchronized in practice?

- Within a datacenter: ~20-50 microseconds
- Across datacenters: ~50-250 **milli**seconds

- for comparison: can process a RPC in ~3us
  200ms is a user-perceptible difference

Two approaches

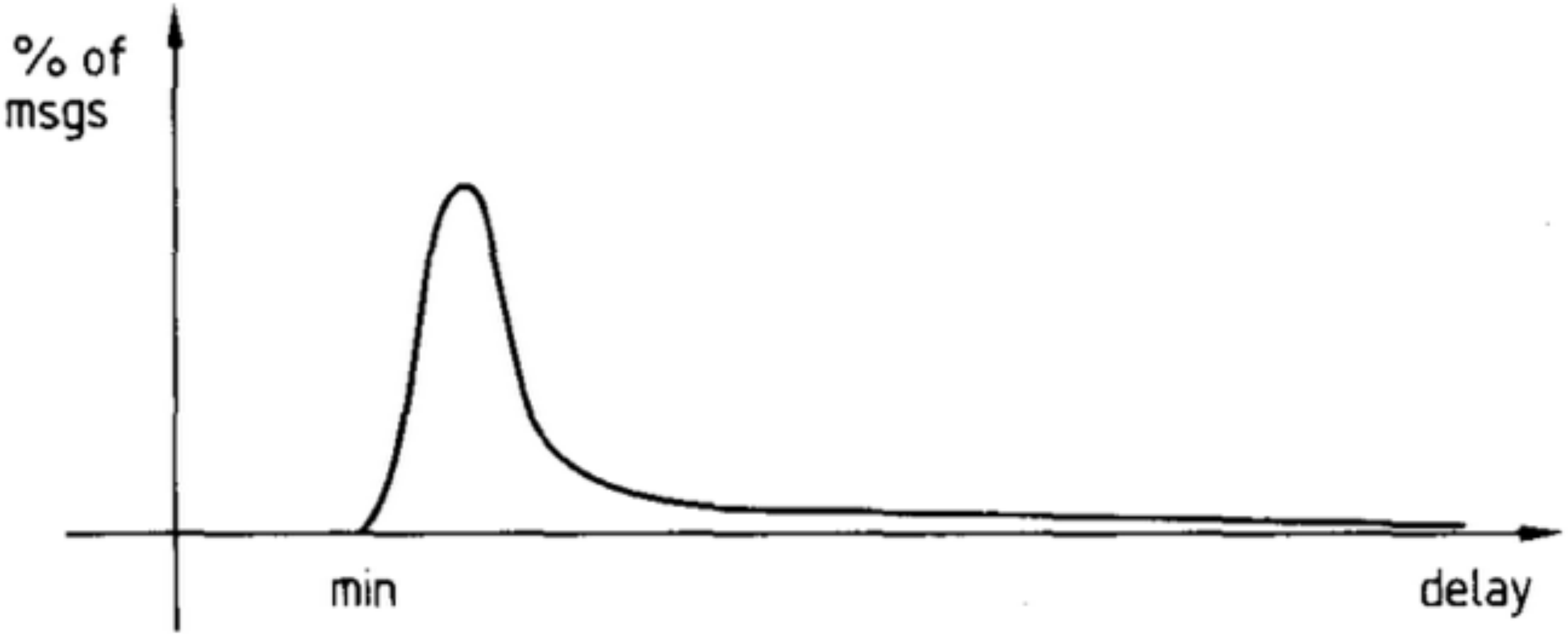
- Synchronize physical clocks
- Logical clocks

Strawman approach

- Designate one server as the master
  (How do we know the master's time is correct?)
- Master periodically broadcasts time
- Clients receive broadcast, set their clock to the value in the message
- Is this a good approach?

Network latency

- Have to assume **asynchronous network**: latency can be unpredictable and unbounded



Slightly better approach

- Designate one server as the master
  (How do we know the master's time is correct?)

- Master periodically broadcasts time

- Clients receive broadcast, set their clock to the value in the message **+ minimum delay**

- *Can we say anything about the accuracy?*

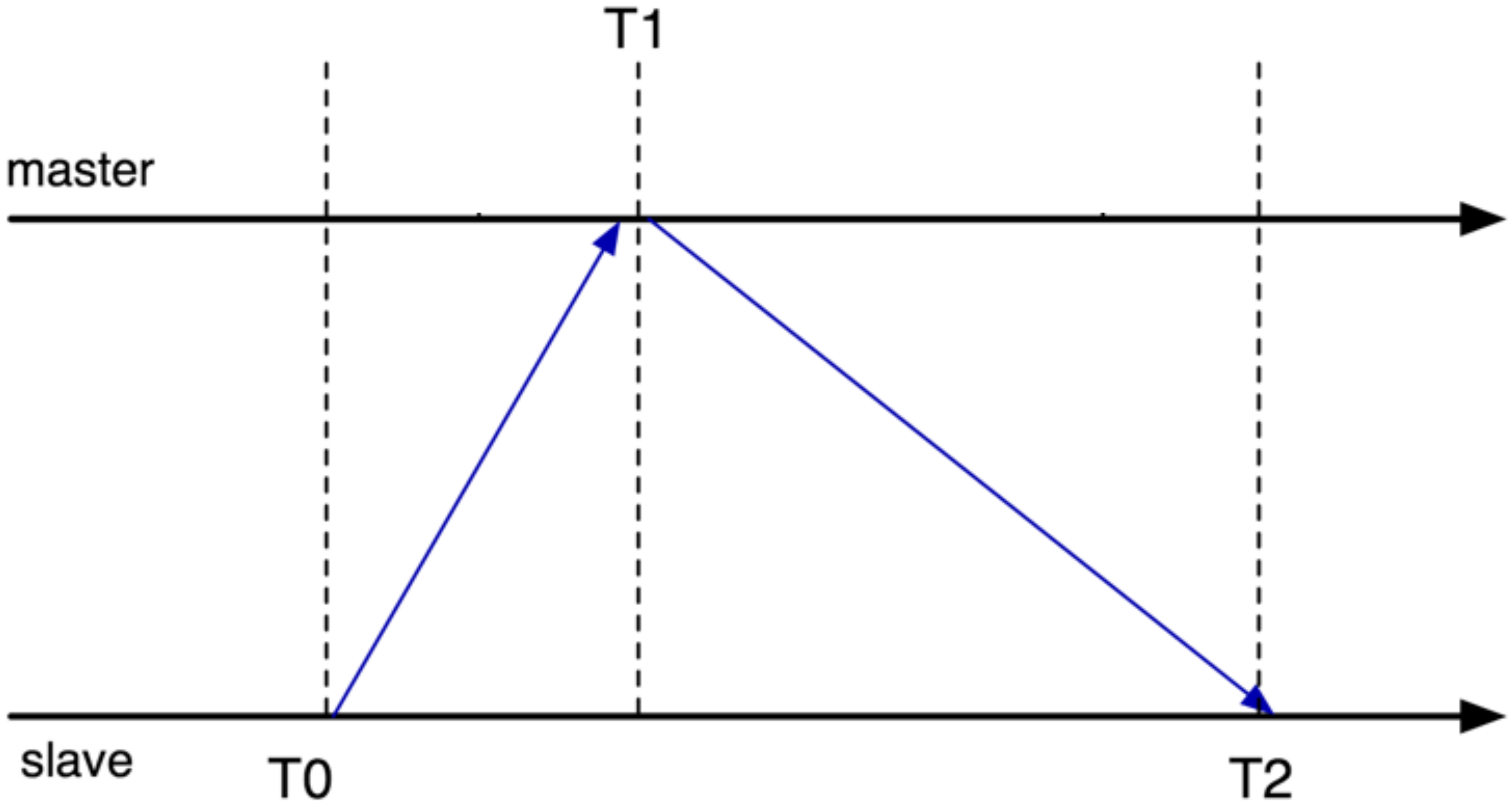
Slightly better approach

- Designate one server as the master
  (How do we know the master’s time is correct?)
- Master periodically broadcasts time
- Clients receive broadcast, set their clock to the value in the message **+ minimum delay**
- *Can we say anything about the accuracy?*

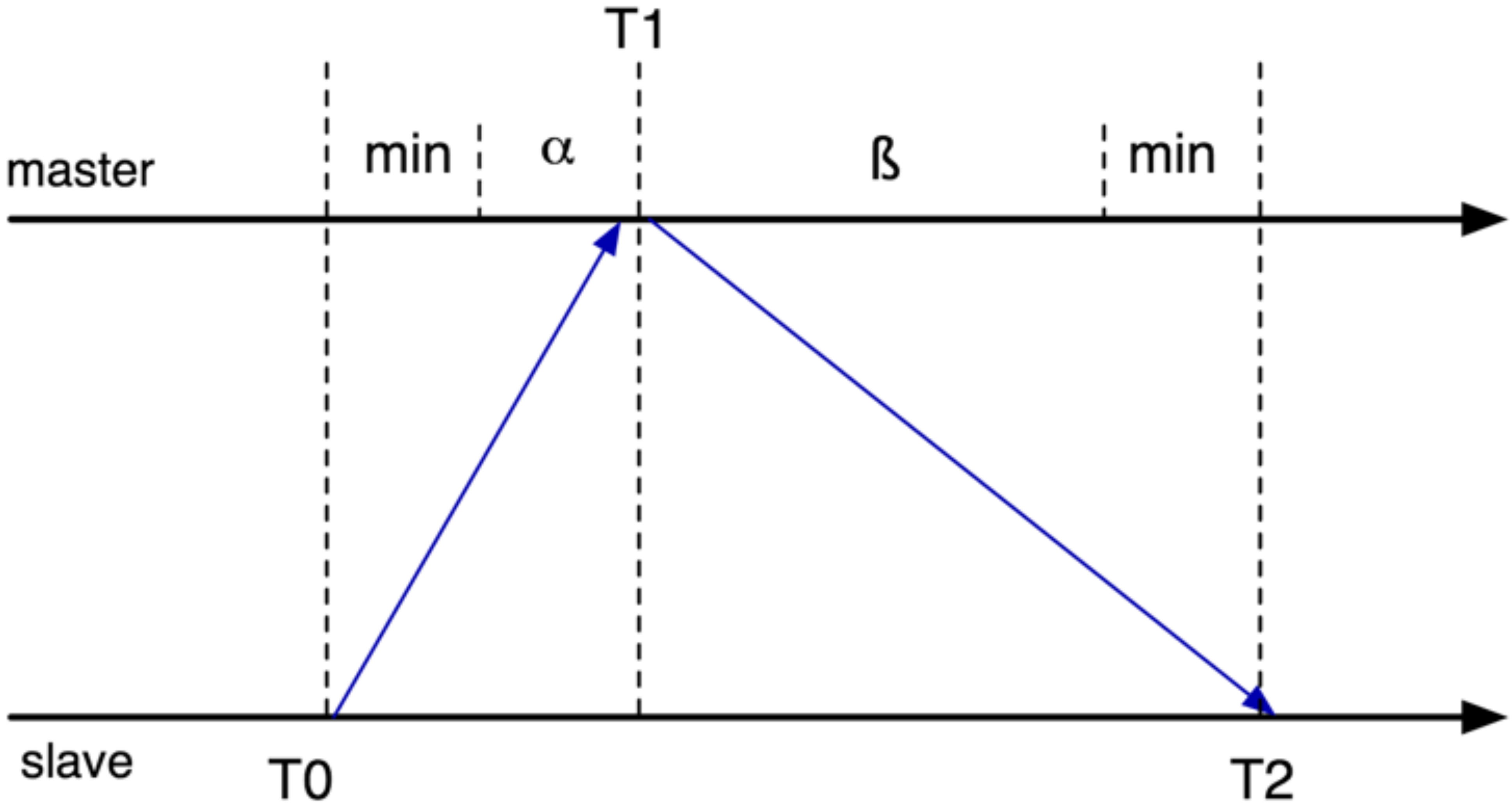
only that error ranges from 0 to (max-min)

Can we do better?

Interrogation-Based Protocol



Interrogation-Based Protocol



How accurate is this?

- No reliable way to tell where T1 lies between T0 and T2
- Best option is to assume the midpoint, set client's clock to T1 + (T2-T0)/2

- What is the maximum error?

How accurate is this?

- No reliable way to tell where T1 lies between T0 and T2
- Best option is to assume the midpoint, set client's clock to T1 + (T2-T0)/2
- What is the maximum error?

If we know the minimum latency: (T2-T0)/2 - min

Improving on this

- NTP uses an interrogation-based approach, plus:
  - taking multiple samples to eliminate ones not close to min RTT
  - averaging among multiple masters
  - taking into account clock *rate* skew
- PTP adds hardware timestamping support to track latency introduced in network

Are physical clocks enough?

Alternative: logical clocks

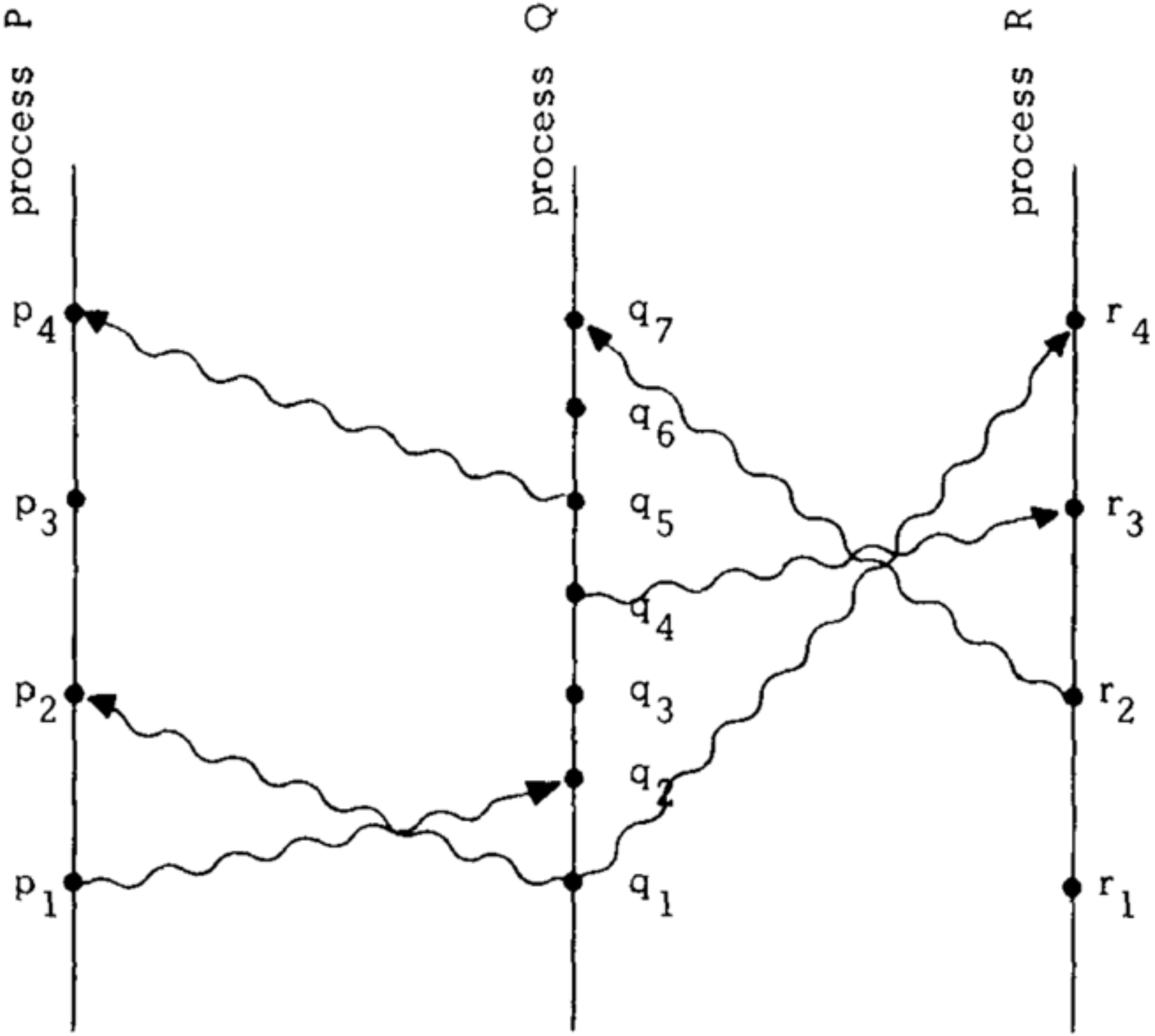
- another way to keep track of time
- based on the idea of causal relationships between events
- doesn't require any physical clocks

Definitions

- What is a process?
- What is an event?
- What is a message?

Happens-before relationship

- Captures logical (causal) dependencies between events
- Within a thread, P1 before P2 means P1 -> P2
- if a = send(M) and b = recv(M), a -> b
- transitivity: if a -> b and b -> c then a -> c


process P
process Q
process R
p4
p3
p2
p1
q7
q6
q5
q4
q3
q2
q1
r4
r3
r2
r1

What does -> mean?

What does -> mean?

- a -> b means “b could have been influenced by a”

What does -> mean?

- a -> b means “b could have been influenced by a”
- What about a -/-> b? Does that mean b -> a?

What does -> mean?

- a -> b means “b could have been influenced by a”
- What about a -/-> b? Does that mean b -> a?
- What does it mean, then? Events are *concurrent*

What does -> mean?

- a -> b means “b could have been influenced by a”
- What about a -/-> b? Does that mean b -> a?
- What does it mean, then? Events are *concurrent*
- What does it mean for events to be concurrent?

What does -> mean?

- a -> b means “b could have been influenced by a”
- What about a -/-> b? Does that mean b -> a?
- What does it mean, then? Events are *concurrent*
- What does it mean for events to be concurrent?
- **Key insight: no one can tell whether a or b happened first!**

Abstract logical clocks

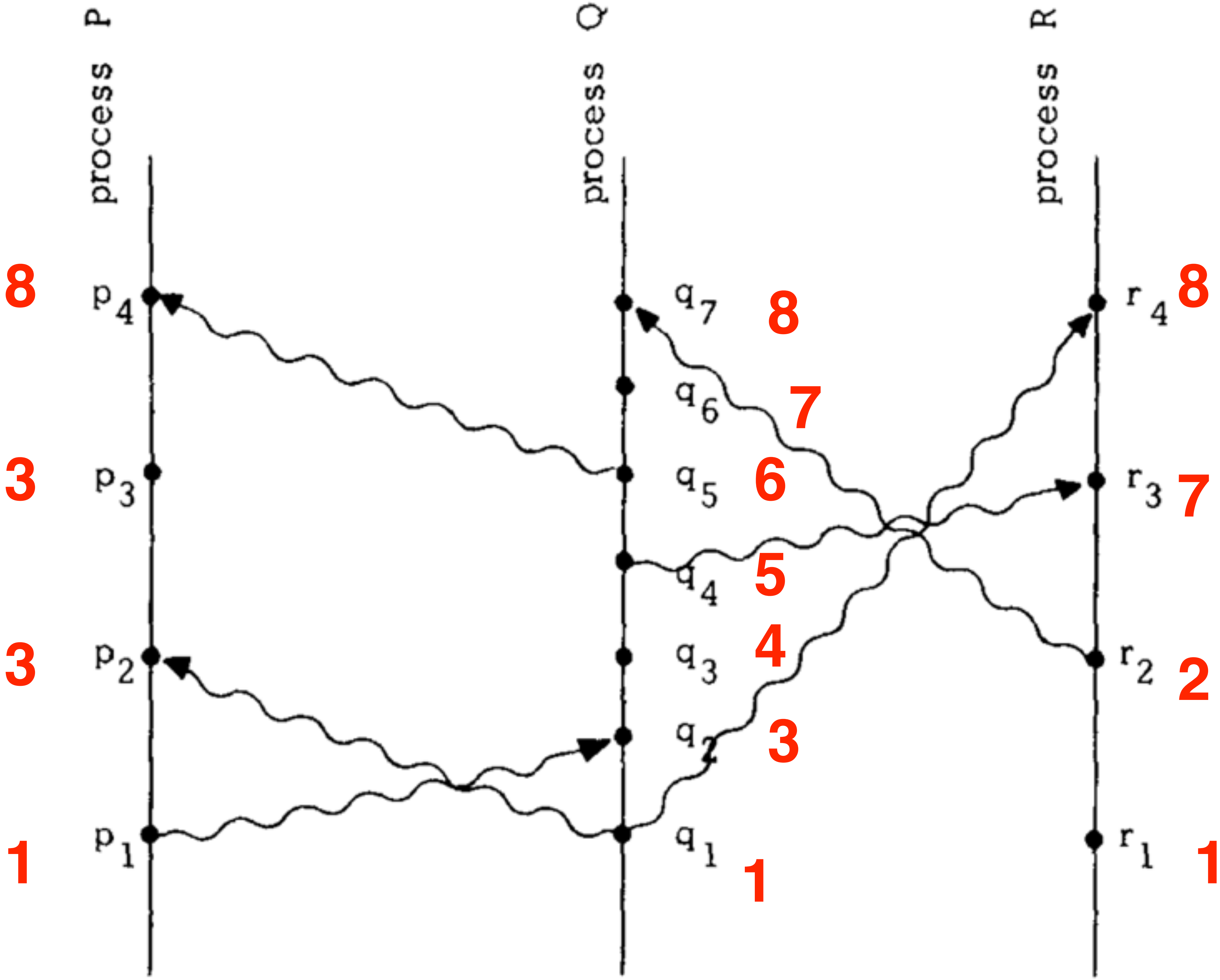
- Goal: if a -> b, then C(a) < C(b)
- Clock conditions:
  - if a and b are on the same process i,
    Ci(a) < Ci(b)
  - if a = process i sends M, and
    b = process j receives m
    Ci(a) < Cj(b)

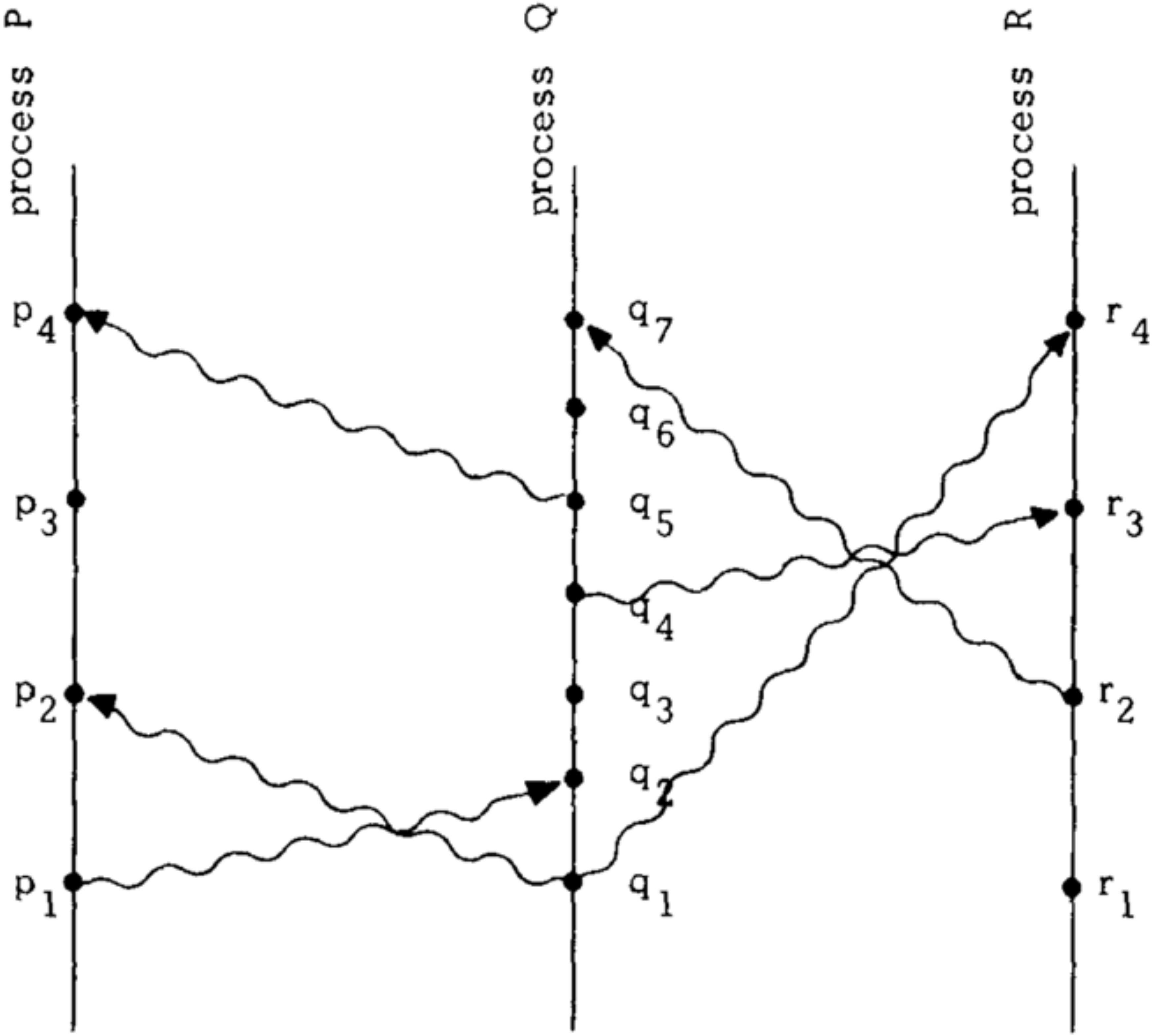
(One) Algorithm

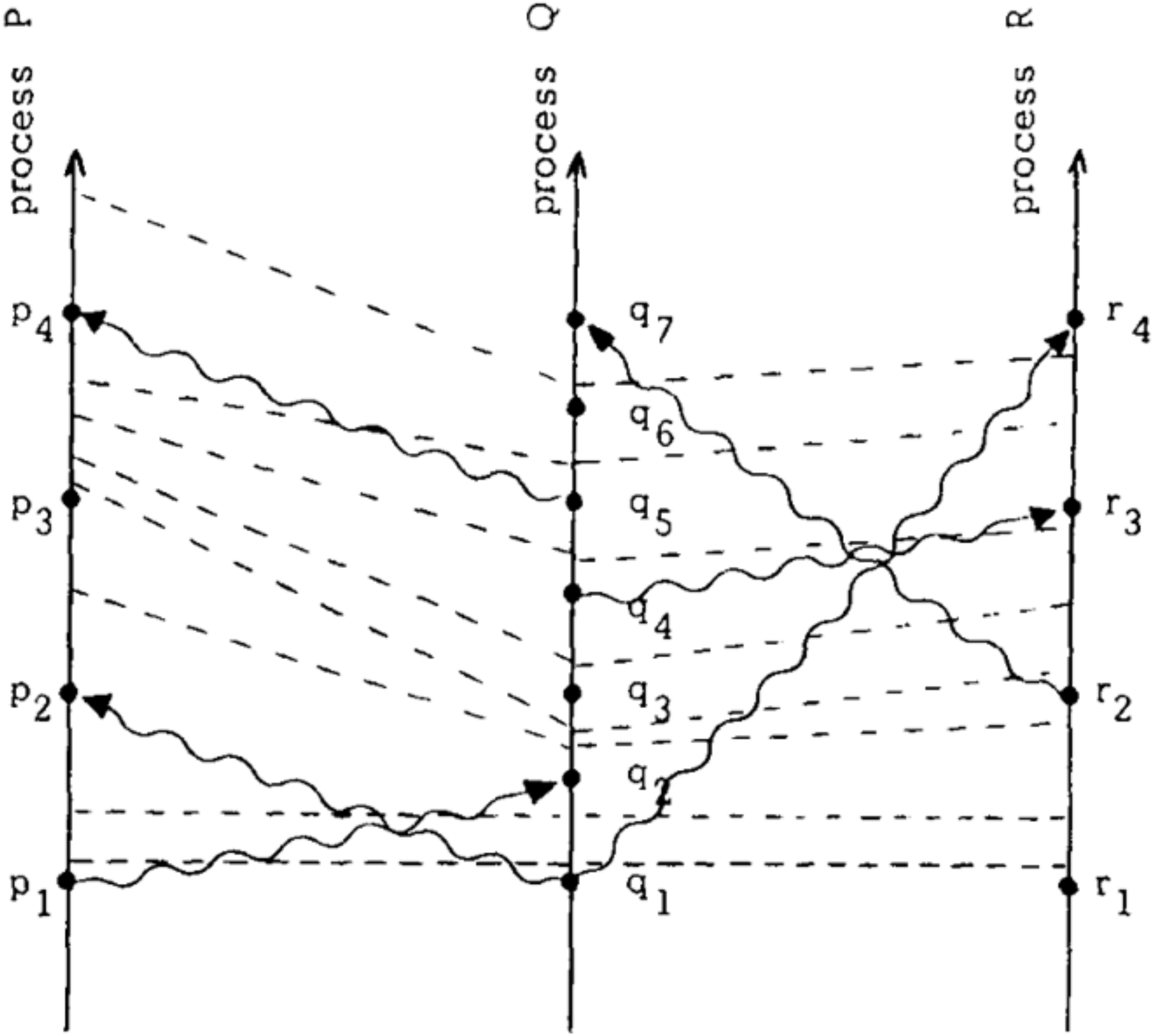
- Each process i increments counter Ci between two local events

- When i sends a message m, it includes a timestamp $Tm = $ (Ci at the time message was sent)

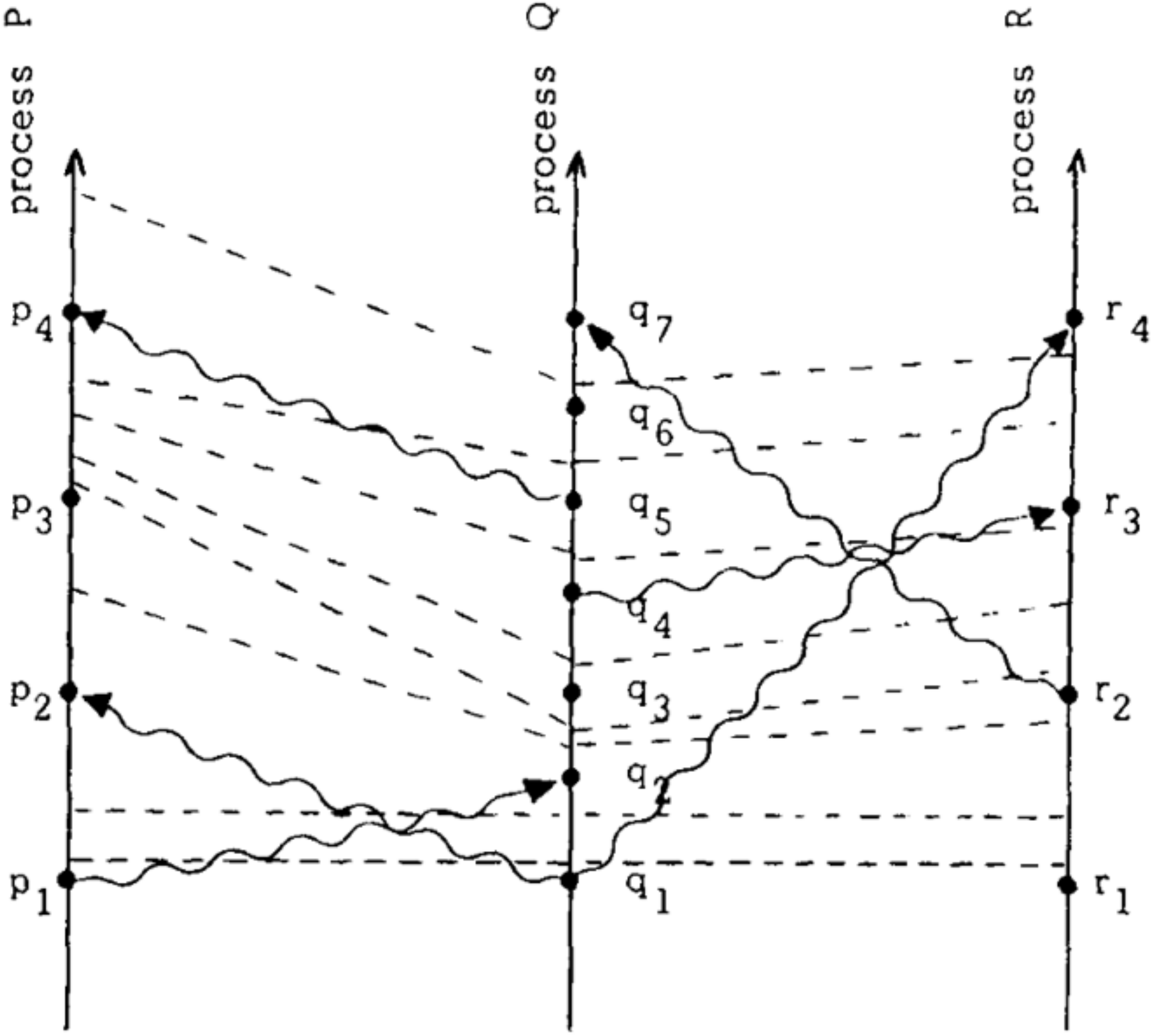
- On receiving m, process j updates its clock:
  $Cj = \max(Cj, Tm + 1) + 1$

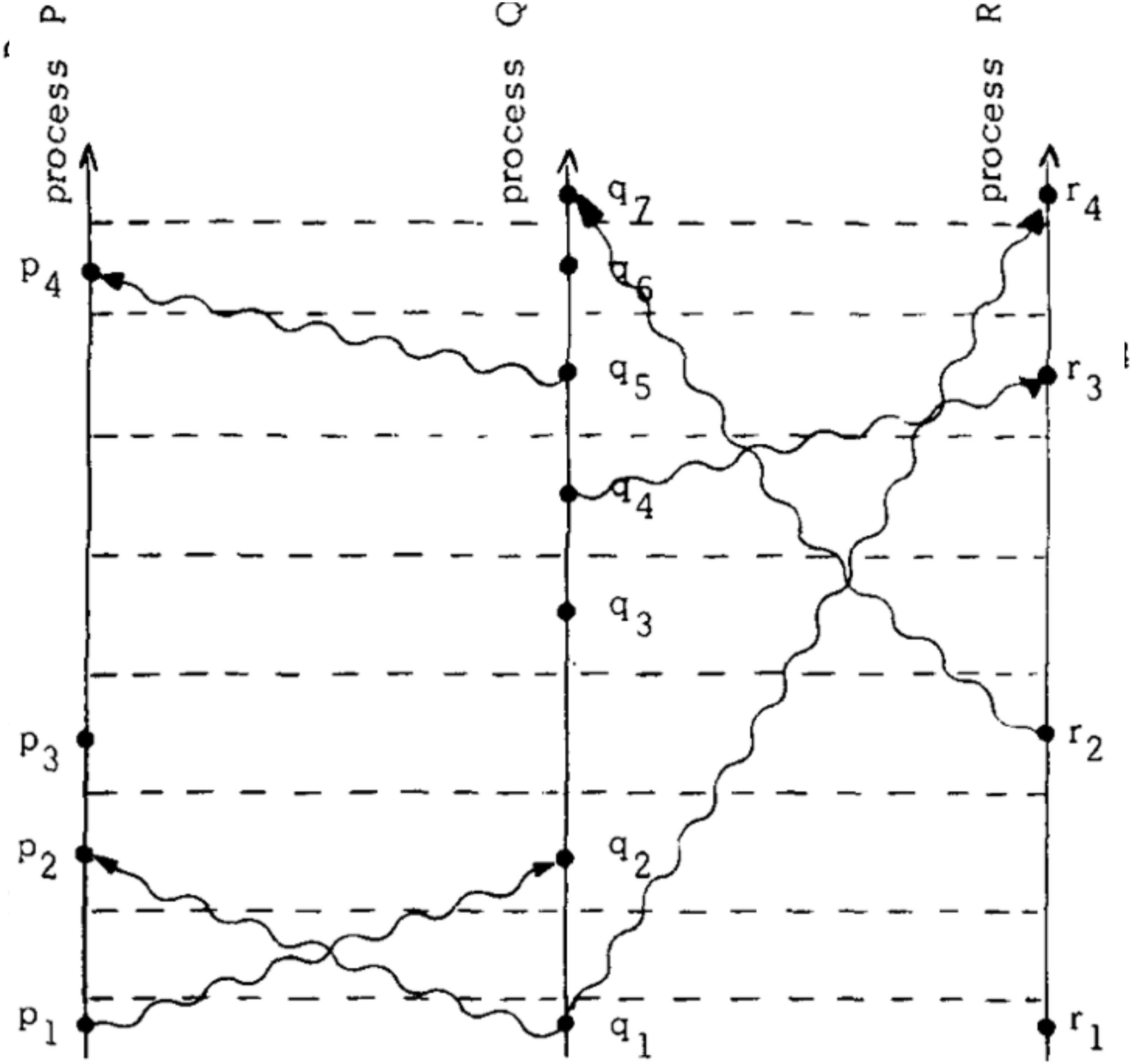

process P
process Q
process R
8 p_4
q_7 8
r_4 8
q_6 7
3 p_3
q_5 6
r_3 7
q_4 5
3 p_2
q_3 4
r_2 2
q_2 3
1 p_1
q_1 1
r_1 1


process P
process Q
process R
p4
p3
p2
p1
q7
q6
q5
q4
q3
q2
q1
r4
r3
r2
r1



process P
process Q
process R
p4
p3
p2
p1
q7
q6
q5
q4
q3
q2
q1
r4
r3
r2
r1


process P
process Q
process R
p4
p3
p2
p1
q7
q6
q5
q4
q3
q2
q1
r4
r3
r2
r1


process P
process Q
process R
q_7
r_4
p_4
q_6
q_5
r_3
q_4
q_3
r_2
p_3
p_2
q_2
p_1
q_1
r_1

What does this mean?

What does this mean?

- If a -> b, then C(a) < C(b)

What does this mean?

- If a -> b, then C(a) < C(b)
- Is the converse true: if C(a) < C(b) then a -> b?

What does this mean?

- If a -> b, then C(a) < C(b)
- Is the converse true: if C(a) < C(b) then a -> b?
    - no, they could also be concurrent

What does this mean?

- If a -> b, then C(a) < C(b)
- Is the converse true: if C(a) < C(b) then a -> b?
  - no, they could also be concurrent
  - if we were to use the Lamport clock as a global order, we would induce some unnecessary ordering constraints

Could we build a better logical clock?

Could we build a better logical clock?

- One where the converse is true,
  C(a) < C(b) => a -> b

Could we build a better logical clock?

- One where the converse is true,
  $C(a) < C(b) \Rightarrow a \rightarrow b$

- Note that there must still be concurrent events:
  sometimes neither $C(a) < C(b)$ or $C(b) < C(a)$

Could we build a better logical clock?

- One where the converse is true,
  C(a) < C(b) => a -> b

- Note that there must still be concurrent events:
  sometimes neither C(a) < C(b) or C(b) < C(a)

- Strawman: keep a dependency list,
  i.e. a list of all previous events

Could we build a better logical clock?

- One where the converse is true,
  $C(a) < C(b) \Rightarrow a \rightarrow b$

- Note that there must still be concurrent events:
  sometimes neither $C(a) < C(b)$ or $C(b) < C(a)$

- Strawman: keep a dependency list,
  i.e. a list of all previous events

- Better answer: vector clocks (later!)

Snapshots

Motivating Example: PageRank

- Long-running computation on thousands of servers
  - each server holds some subset of webpages
  - each page starts out with some reputation
  - each iteration: transfer some of a page's reputation to the pages it links to
- **What do we do if a server crashes?**

Suppose we want to take a snapshot for fault tolerance.

How often would we need to snapshot each machine?

Consistent Snapshots

- We want processes to record their snapshots at “about the same time”
- If a process’s checkpoint reflects receiving message m, then the sending process’s checkpoint should reflect sending it
  - or if a channel’s checkpoint contains a message
- If a process’s checkpoint reflects sending a message, the message needs to be reflected in the receiver’s or channel’s checkpoint
  - i.e., can’t lose messages

Put another way:

- Process checkpoints are ***logically concurrent***
- i.e., no process checkpoint happens-before another!
- alternatively*:*
  if a -> b, and b is in some checkpoint, so is a

Chandy-Lamport algorithm

- Assumptions
  - finite set of processes and channels
  - strongly connected graph between processes
  - channels are infinite buffers,
    error-free,
    in-order delivery,
    finite delay
  - processes are deterministic
- Why do we need each of these?

The Algorithm

- Start: some process sends itself a “take snapshot” token
- When i receives a token from j:
    - i checkpoints its process state
    - i sends token on all outgoing channels
    - i records that channel from j is empty
    - i starts recording messages on other channels until receiving a token on that channel
- Done when every process has received a token on every channel

Why does this work?

Why does this work?

- Tokens separate logical time into
  “before the snapshot” from “after the snapshot”

- if process i records state that includes receiving a message from j
  then j’s state includes sending that message

Discussion

- Is this the best way to snapshot systems?
- Can we use this technique for other purposes?

State Machine Replication

(Chain Replication & Lab 2)

How do we build a system that tolerates server failures?

- **Replication!**

- Goal: tolerate up to f server failures
  by using (at least) f+1 copies

- Goal: look just like one copy to the client

- Challenge: coordinating operations so they are applied to all replicas with the same result

State Machine Replication

- Incredibly powerful abstraction
- Idea: model the system as a state machine
    - service maintains some amount of state
    - transition function: (input, state) -> new state
    - output function: (input, state) -> output
- i.e., system state/output entirely determined by input sequence

Key idea:
If the system is a state machine,
keeping the replicas consistent means
agreeing on the order of operations

Are all real systems state machines?

Are all real systems state machines?

- Needs to be deterministic
  - what about clocks? randomness?
  - parallel execution within a single machine (multicore)
- Need to be careful to capture all inputs?

Ordering operations

- **Goal**: achieve a consistent order of operations
  on all replicas
- What does “consistent” mean here?
- Single-copy serializability: it appears to all clients as though operations were executed sequentially on a single machine
  - i.e, total order of operations doesn’t change
- Strict serializability (linearizability): adds real time req:
  if a finishes before b starts, a is ordered before b

State machine replication

- Many ways to achieve this:
- Primary copy approaches
  - chain replication is one example
  - Lab 2 is a simplified version
- Quorum approaches, e.g. Paxos (two weeks)

Primary Copy Replication

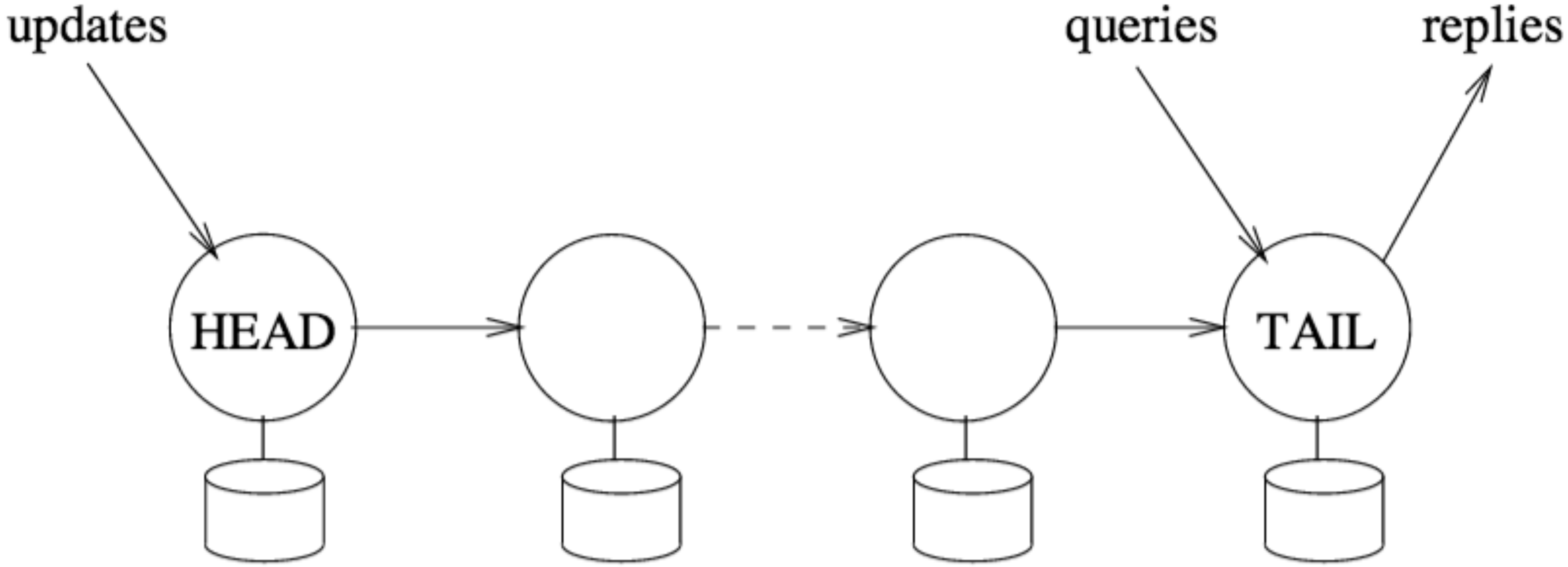
- Key idea: have a designated primary that assigns order to requests
- All replicas execute requests in primary’s order
- Client sees results consistent with that order
- Client doesn’t see results until executed by “enough” replicas (here, all f+1)
- When primary fails, replace it — but make sure the new primary respects the order of all successful operations (*this is the hard part!*)

Chain Replication Assumptions

Chain Replication Assumptions

- f+1 nodes to tolerate f failures
- nodes fail only by crashing, and crashes are detected
- fault-tolerant master service keeps track of system membership
- operations are read or write

Chain Replication



Normal Case Processing

- Updates sent to head, propagated down chain, response comes from tail
- Key invariant: each node has seen a superset of operations seen by all following nodes in the chain
- What is the commit point of an operation?

Failures in the Chain

- What happens if the tail fails?
- What happens if the head fails?
- What happens if a node in the middle fails?
- What happens if we add a node?
- What happens if the master fails?

Performance

- Alternative: primary sends to all other replicas in parallel, waits for responses
    - could use f+1 replicas and wait for responses from all, or 2f+1 and wait for responses from majority
- Throughput: chain replication best (2 msgs per node)
- Latency: chain replication worst
  - need to execute at every replica in sequence
  - need to wait for slowest replica

Lab 2

- Simplified version of chain replication:
  chain always two nodes (primary & backup)
- Part A: implement the view service (master)
- Part B: implement a primary/backup key-value store

View Service Behavior

- What state does the master need?
  - list of alive replicas, last ping time
  - view number, primary and backup for that view

- View transitions
  - initial state -> make some node primary in view 1
  - primary, no backup -> add a backup
  - primary, backup -> backup fails
  - primary, backup -> primary fails, replace with backup

View Service Behavior

- Servers periodically ping master
  - n missed pings => server dead
  - 1 successful ping => server alive
  - primary dead => promote backup
  - no backup, some live server => add it as backup

Primary/Backup

- Need to ensure that the new primary has up-to-date state
- Only promote previous backup (not an idle server)
- What if the previous backup didn't have time to get the state from the old master?
  - primary must acknowledge new view to view server
  - if it doesn't, can't move to a new view
    even if the primary fails!

Multiple Primaries

- Can more than one replica think it's the primary?
- How do we keep other replicas from *acting* as the primary?
- Operations need to be forwarded to the backup to succeed
- Backup will always be the primary in the next view, so it rejects forwarded ops from the old primary