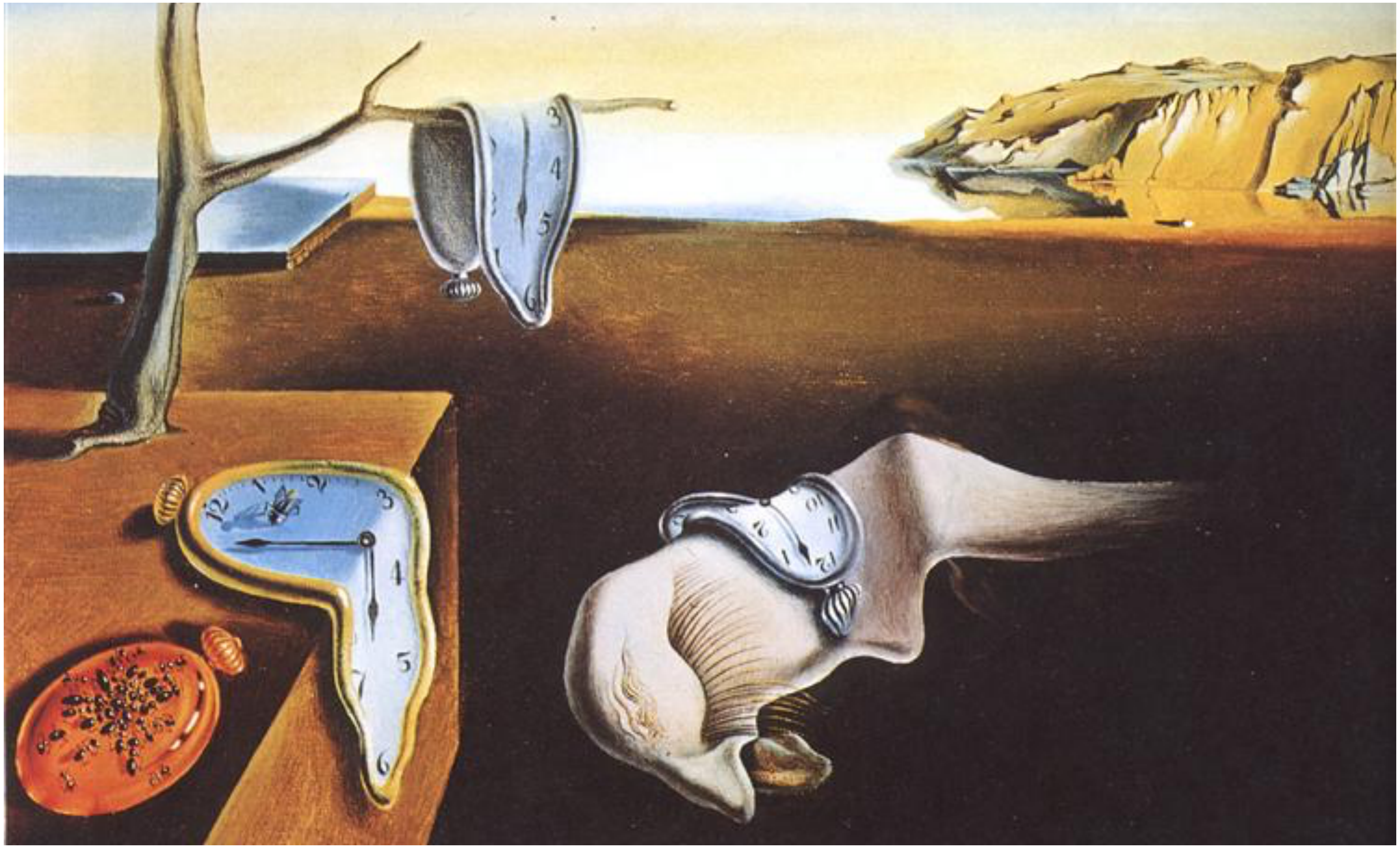
Coordination

Thomas Schwarz, SJ

Contents



Distributed Time

Thomas Schwarz, SJ

Physical Time

- System clock
  - Precisely machined quartz crystal
  - Counter and Holding register
    - Each oscillation of quartz decrements counter
    - When counter gets to zero, generate interrupt
      - Counter is reloaded from holding register
      - Time is incremented

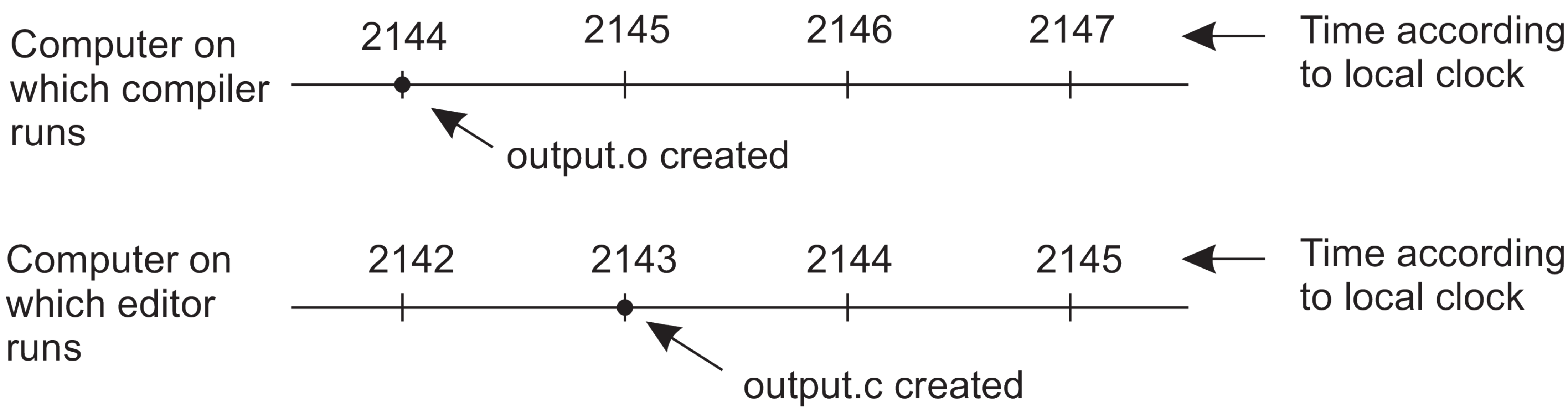
Physical Time

- On a single system:
  - Absolute time does not really matter
  - Important are relative times
  - Example:
    - Make will recompile *.c files if their modified time is later than the corresponding *.o file

Physical Time

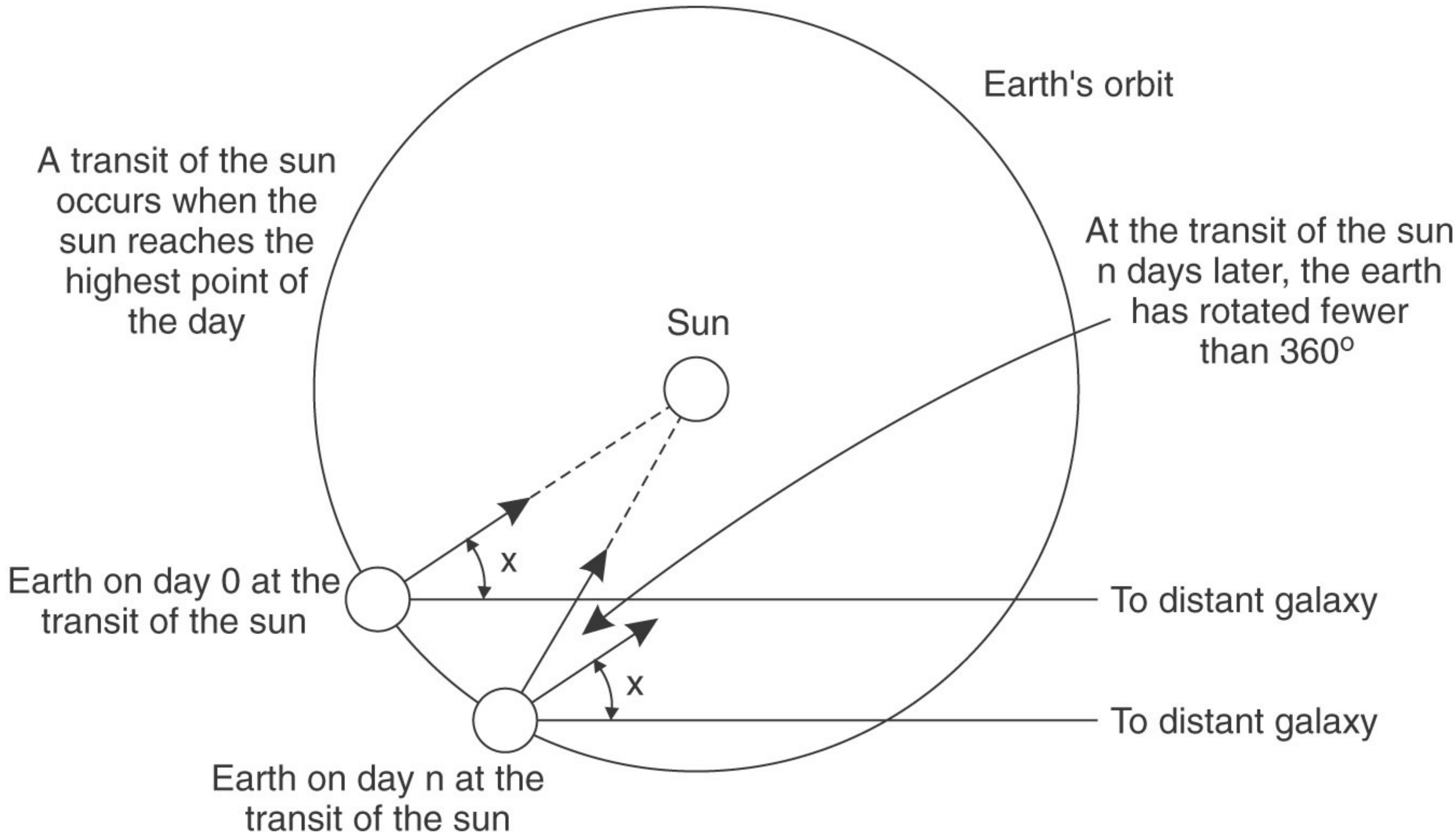
- Multiple CPU with their own clock
  - Distributed systems
    - Need to deal with clock skew
    - Is there a single notion of time?
      - Astronomical time
      - Atom clock time TAI
      - Leap seconds, UTC

Physical Time

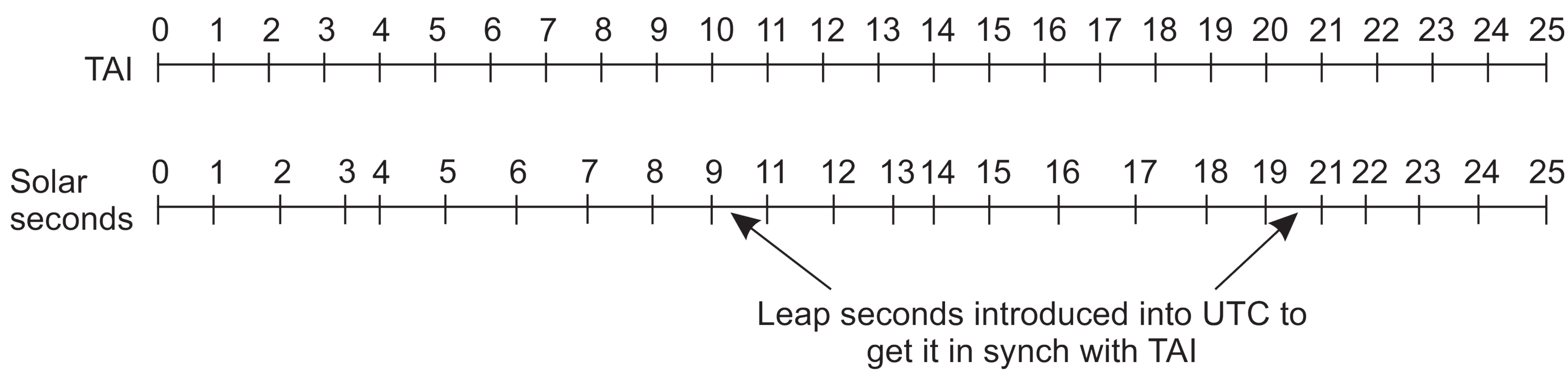


In a distributed system, applications often rely on a single notion of time

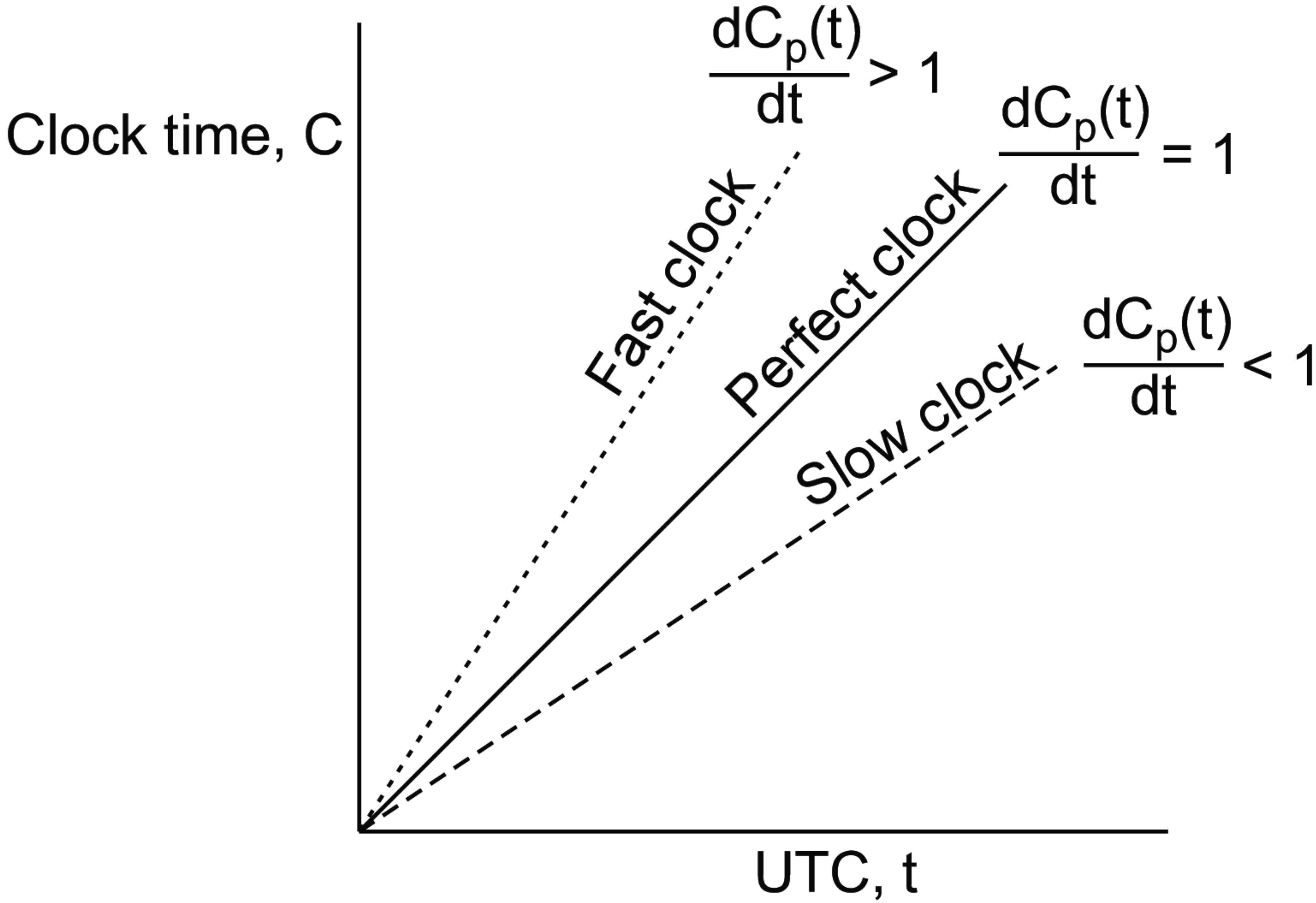
Physical Time



Physical Time

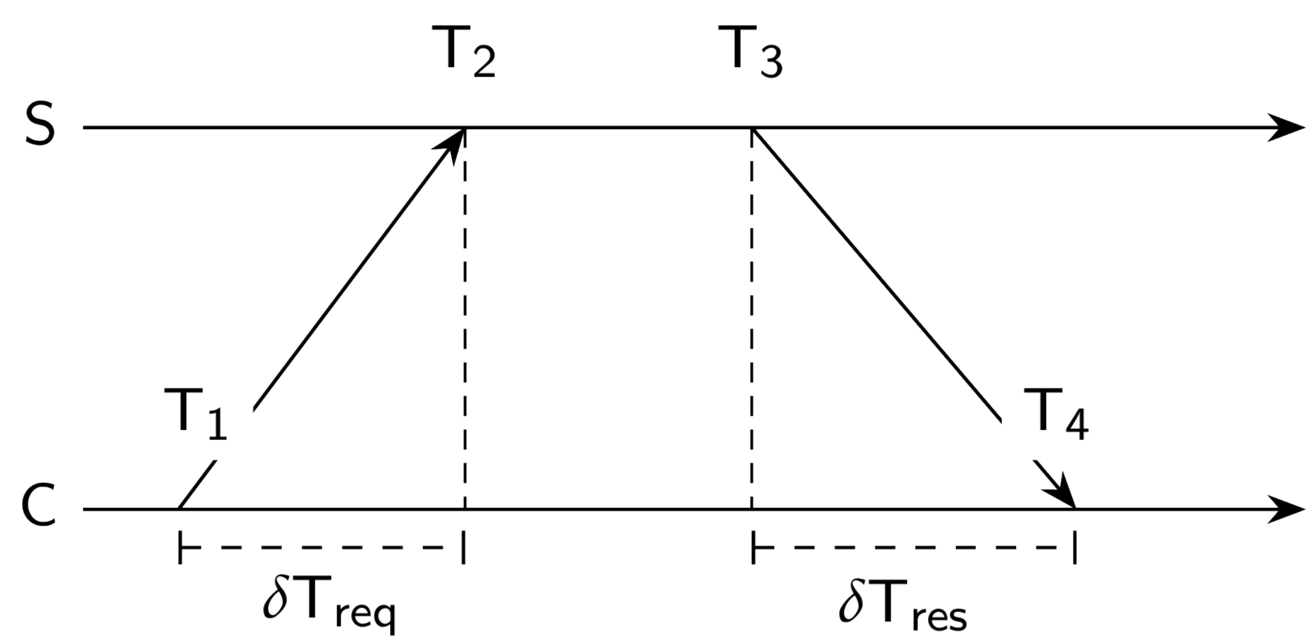


Physical Time



Physical Time

- Clock synchronization algorithms
  - Cristian's algorithm
  - Ask time server for time
  - Determine time



Physical Time

- To determine most likely time given a value send by a time server:
  - Repeat several times
  - Record request send and answer received times (in local time)
  - Record answer received
  - Eliminate outliers
  - Calculate delta
  - Average delta
  - Adjust local time

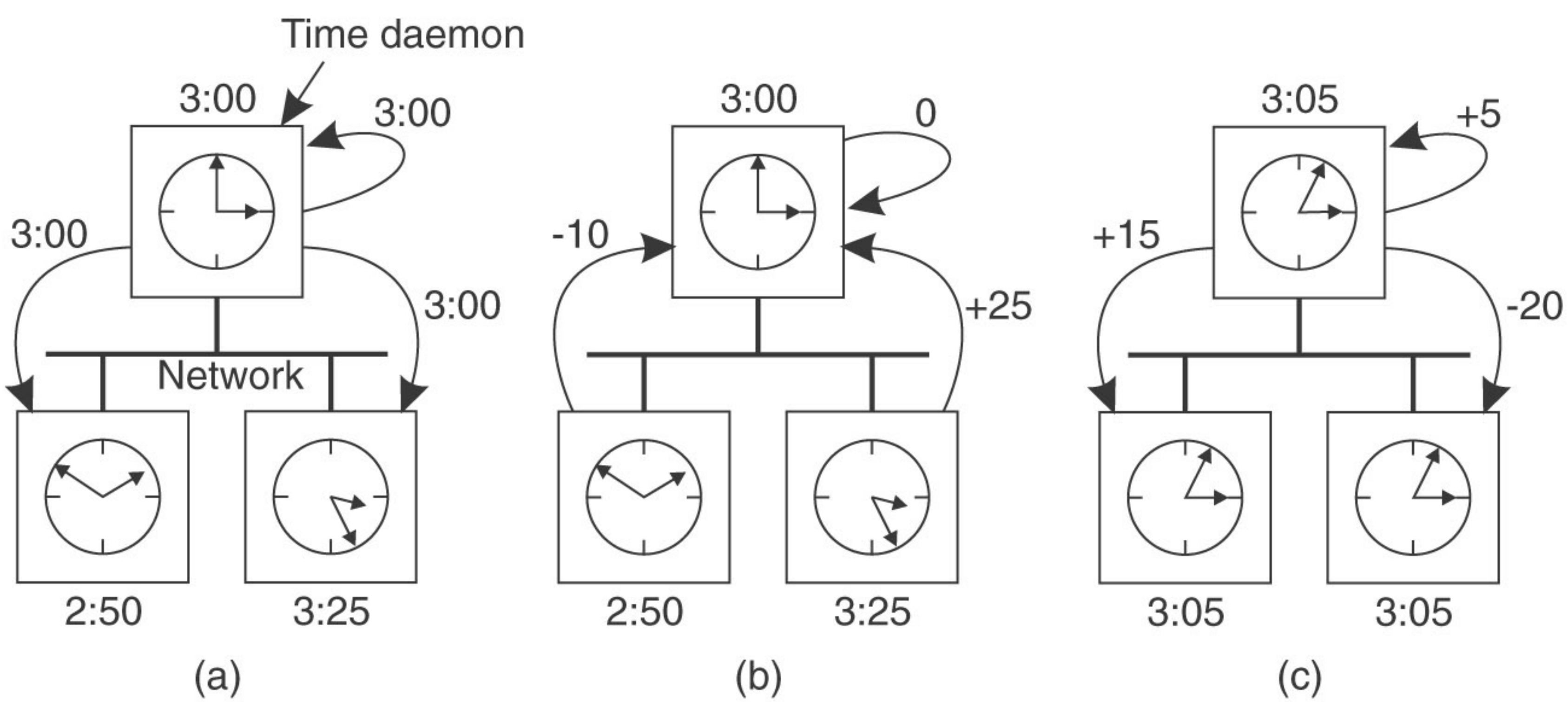
Physical Time

- Group quiz
  - Calculate clock adjustment

sent	received	value
300	350	410
450	495	555
600	750	620
800	845	915
1000	1055	1135
1200	1400	1590

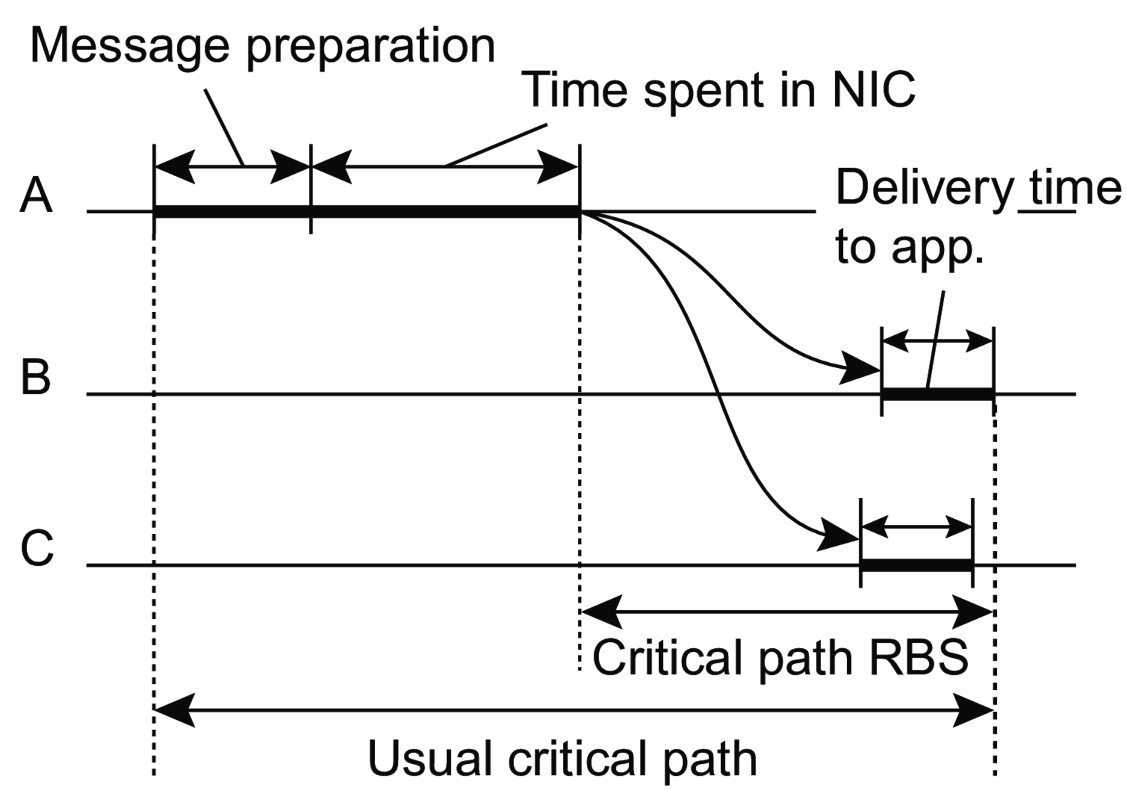
Physical Time

- Berkeley Algorithm
  - Time daemon polls all machines asking for their time
  - Calculates average time
  - Tells all other machines how to adjust



Physical Time

- Reference Broadcast Synchronization (RBS)
  - Assumes no routing, e.g. sensor network
  - A node broadcasts a reference message m
    - Only receivers synchronize their clocks



Physical Time

- Reference Broadcast Synchronization (RBS)
  - Two receivers then exchange their mutual, relative offsets several times
  - Use linear regression to estimate relative difference

Physical Time

- Google's TrueTime Server
  - Uses a number of time machines per data center
    - With different sources of time
  - Given a time stamp:
    - Service decides on:
      - Now — a range of values ~ 6msec long
      - After — definitely passed
      - Before — definitely in the future

Physical Time

- Google's TrueTime Server
  - Transactions can be time-stamped
  - Time service can determine whether and how two transactions are ordered

Logical Clocks

- Absolute time is rarely used
  - Exceptions: time outs
  - For distributed protocols:
  - Need Logical Time

Logical Clocks

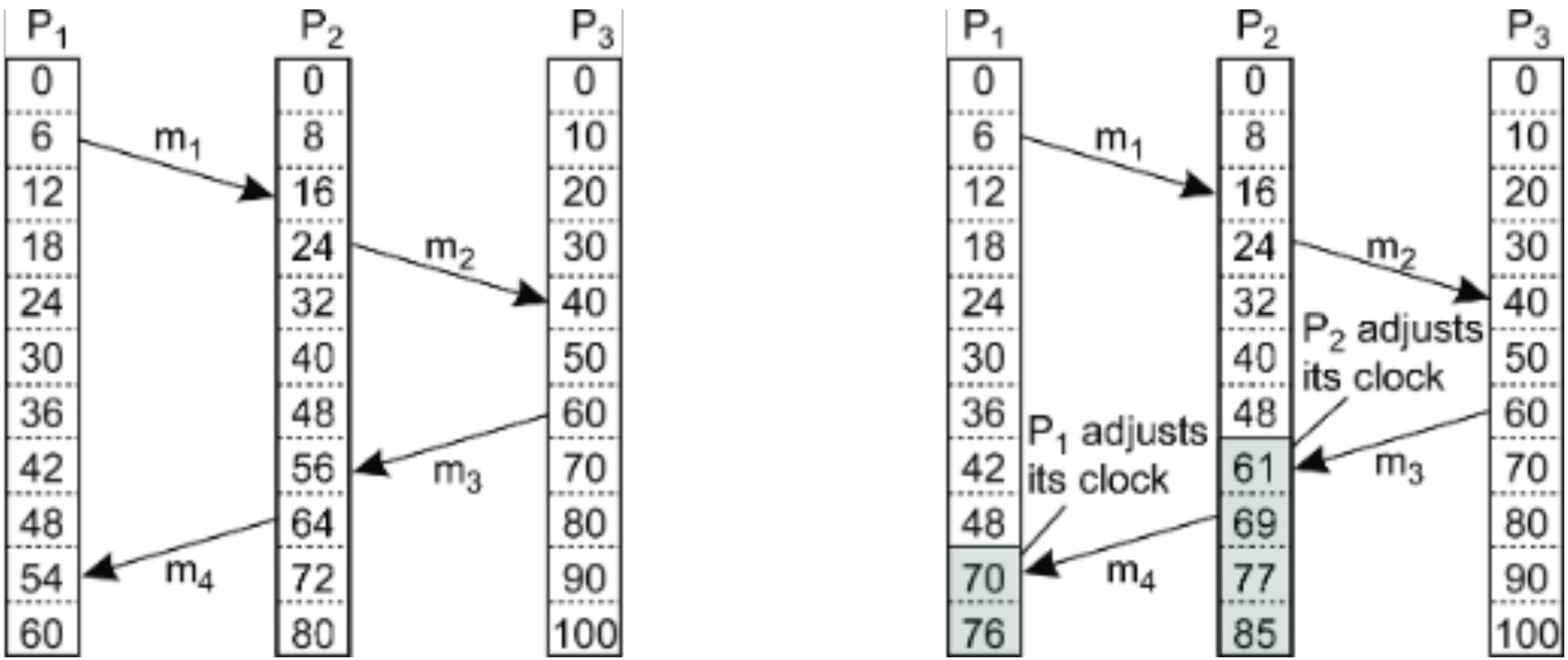
- Lamport time stamps
  - Happens before relationship between events:
    - a <* b
  - Axioms:
    - a, b events in the same process, and a happens before b, then a<*b
    - If
      - a — message being sent
      - b — message being received
    - then
    - a <* b

Logical Clocks

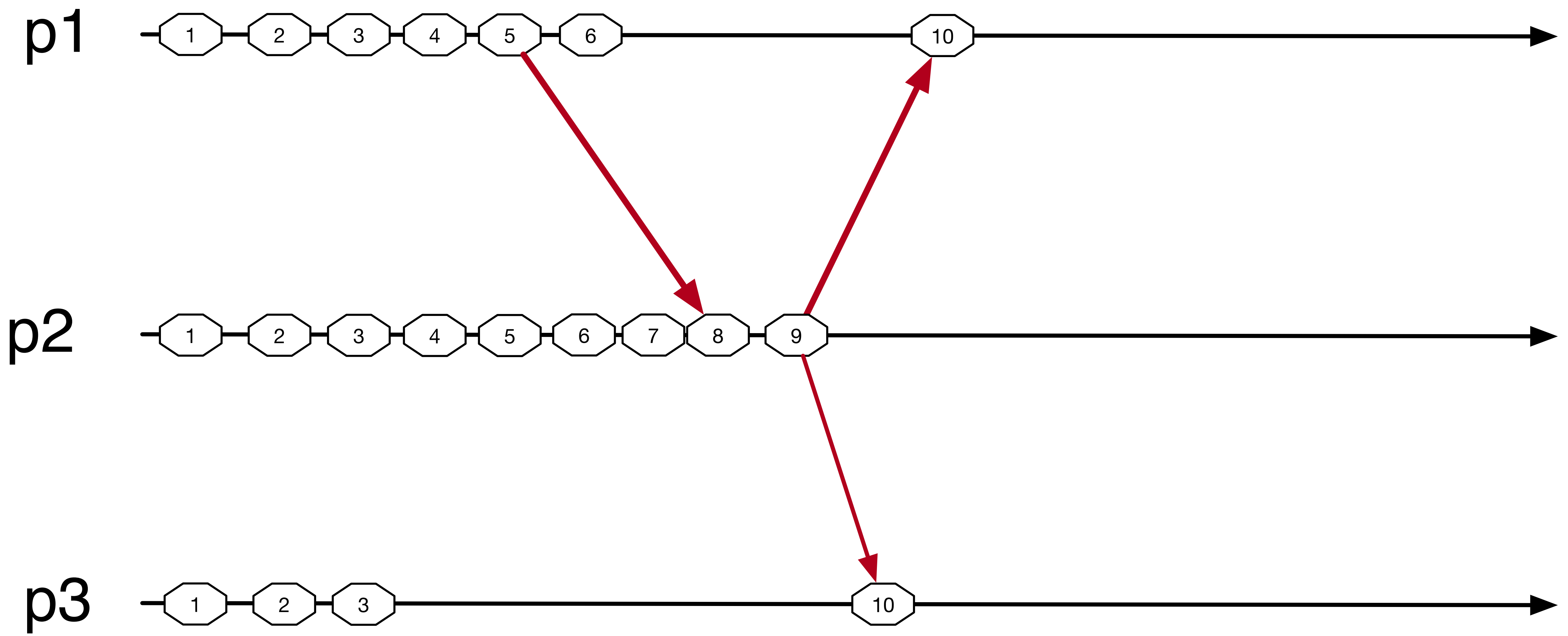
- Each system maintains its own logical time
  - Each local event advances the logical time by (at least one tick)
  - Each message is time-stamped
  - When a message is received, set local time to
    - MAX(local_time + 1, time_stamp + 1)
  - Properties
    - Local events have different times

Logical Clocks

- Consider three processes with event counters operating at different rates

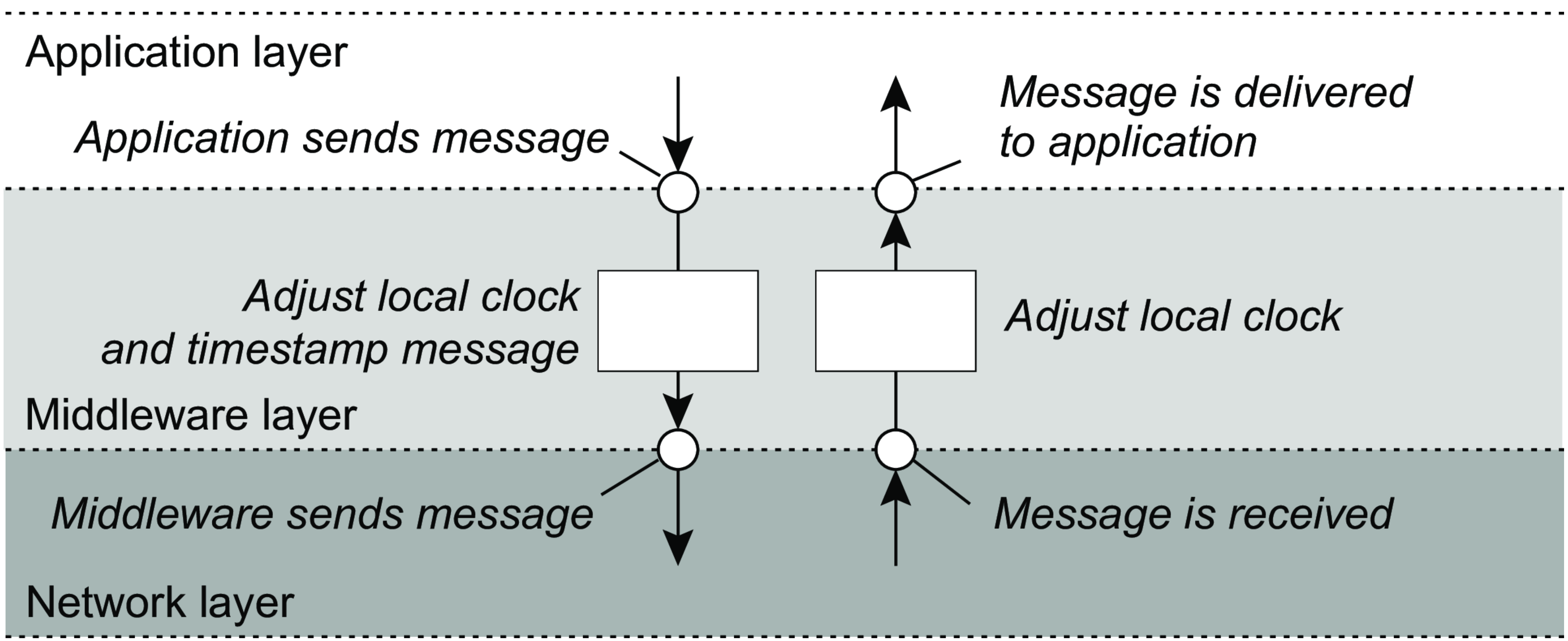


Group Quiz Solution



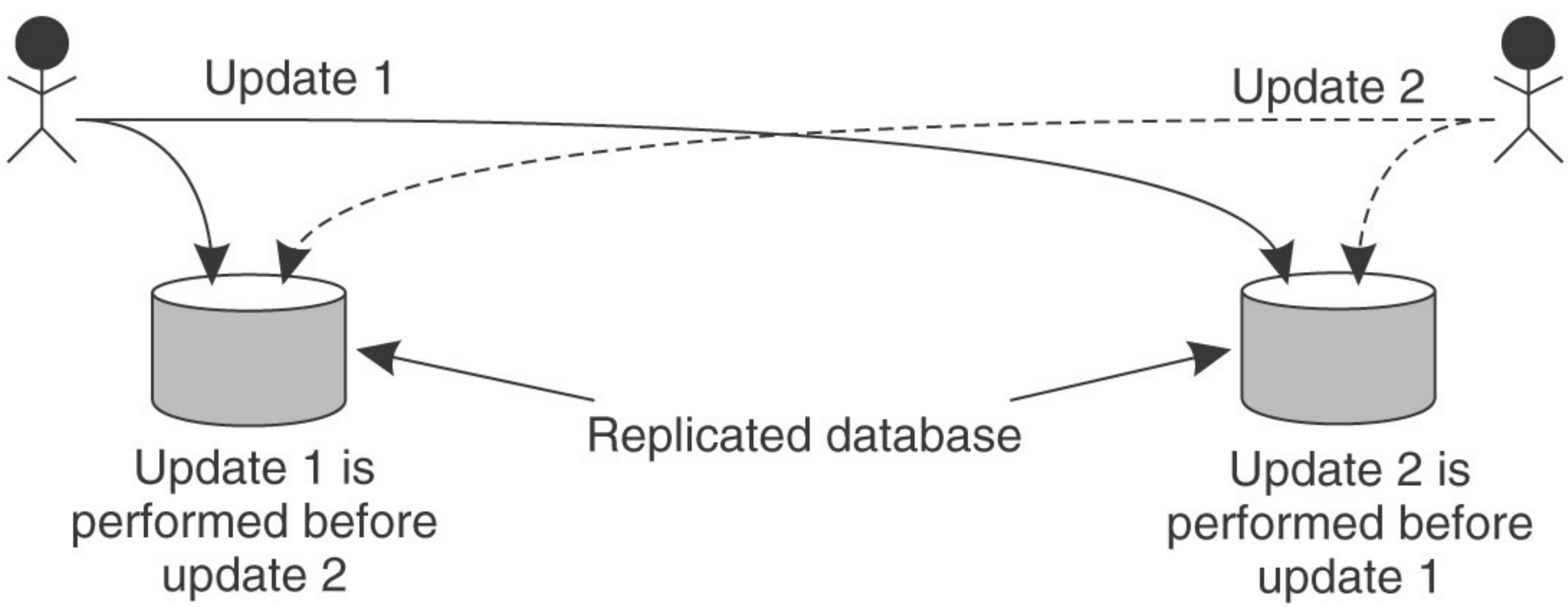
Logical Clocks

- Adjustments implemented in middleware



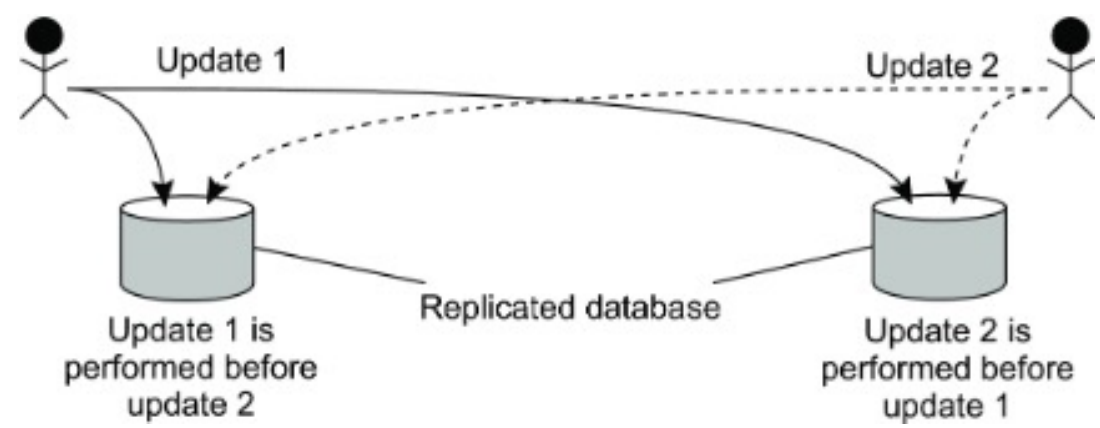
Logical Clocks

- Replica management
  - All replicas need to see the same sequence of updates



Logical Clocks

- Concurrent updates on a replicated database are seen in the same order everywhere
  - P1 adds $100 to an account (initial value: $1000)
  - P2 increments account by 1%
  - There are two replicas



  - In absence of proper synchronization: replica #1 ← $1111, while replica #2 ← $1110.

Logical Clocks

- Totally Ordered Multicast
  - Multicast in which all messages are delivered in the same order to receivers
  - Group of processes multicasting to each other
    - Each message is time-stamped
    - Messages are received in order sent
    - Messages are sent to everyone, including the sender
    - Messages are put in local queues ordered by timestamp
    - Receiver multicast acknowledgments to the other processes
  - Lamport clock algorithm assures that all messages have different timestamps
  - All processes eventually have the same messages in their queue

```python
class Process:
  def __init__(self, chanID, procID, procIDSet):
    self.chan.join(procID)
    self.procID     = int(procID)
    self.otherProcs.remove(self.procID)
    self.queue      = []                        # The request queue
    self.clock      = 0                         # The current logical clock

  def requestToEnter(self):
    self.clock = self.clock + 1                               # Increment clock value
    self.queue.append((self.clock, self.procID, ENTER)) # Append request to q
    self.cleanupQ()                                           # Sort the queue
    self.chan.sendTo(self.otherProcs, (self.clock, self.procID, ENTER)) # Send request

  def ackToEnter(self, requester):
    self.clock = self.clock + 1                               # Increment clock value
    self.chan.sendTo(requester, (self.clock, self.procID, ACK)) # Permit other

  def release(self):
    tmp = [r for r in self.queue[1:] if r[2] == ENTER]  # Remove all ACKs
    self.queue = tmp                                          # and copy to new queue
    self.clock = self.clock + 1                               # Increment clock value
    self.chan.sendTo(self.otherProcs, (self.clock, self.procID, RELEASE)) # Release

  def allowedToEnter(self):
    commProcs = set([req[1] for req in self.queue[1:]]) # See who has sent a message
    return (self.queue[0][1] == self.procID and len(self.otherProcs) == len(commProcs))
```

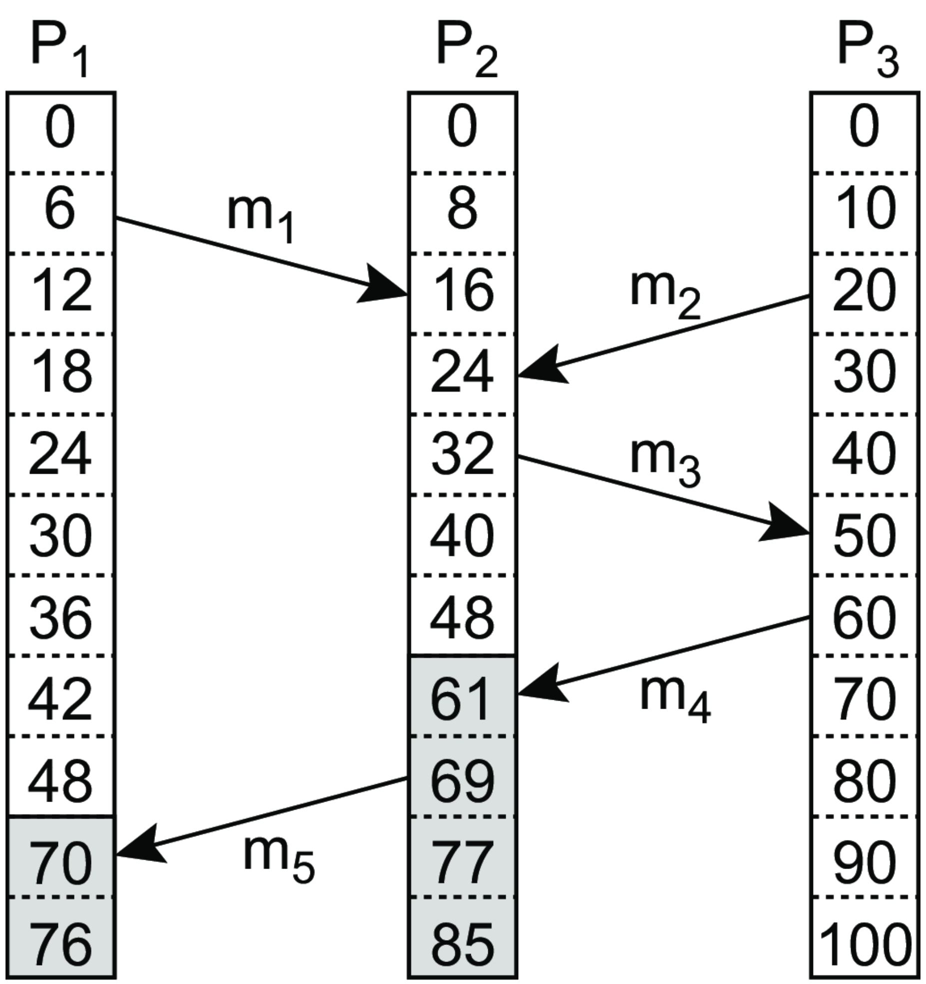
```
def receive(self):
  msg = self.chan.recvFrom(self.otherProcs)[1]        # Pick up any message
  self.clock = max(self.clock, msg[0])                # Adjust clock value...
  self.clock = self.clock + 1                         # ...and increment
  if msg[2] == ENTER:
    self.queue.append(msg)                            # Append an ENTER request
    self.ackToEnter(msg[1])                           # and unconditionally allow
  elif msg[2] == ACK:
    self.queue.append(msg)                            # Append a received ACK
  elif msg[2] == RELEASE:
    del(self.queue[0])                                # Just remove first message
  self.cleanupQ()                                     # And sort and cleanup
```

Vector Clocks

- Causal dependency
  - We assume that all local events before sending a message might have ***caused*** events at the receiver after receiving the message

Vector Clocks

- Observation
  - Lamport's clocks do not guarantee that if C(a) < C(b) that a causally preceded b.



m_1 is received at time 16

m_2 is received at time 24

But:
There is no causal connection between these events

Vector Clocks

- Lamport timestamps do not capture causality
  - Vector timestamps better capture causality
    - Label the k^{th} event at process P_i as $p_{i,k}$
    - If two events happen at P_i, then the ***causal history*** of $p_{i,2}$ at P_i is $H(p_2) = \{p_{i,1}, p_{i,2}\}$

Vector Clocks

- Now assume P_i sends a message to P_j.
  - Sending the message is event $p_{i,2}$
  - Assume the history at P_j is $\{p_{j,1}\}$
  - Receiving the message by P_j is event $p_{j,2}$.
  - Upon arrival, P_j updates its history to include the history at P_i. This gives
    - $\{p_{i,1}, p_{i,2}, p_{i,3}, p_{j,1}, p_{j,2}\}$

Vector Clocks

- An event p causally precedes an event q if
  - $H(p) \subsetneq H(q)$

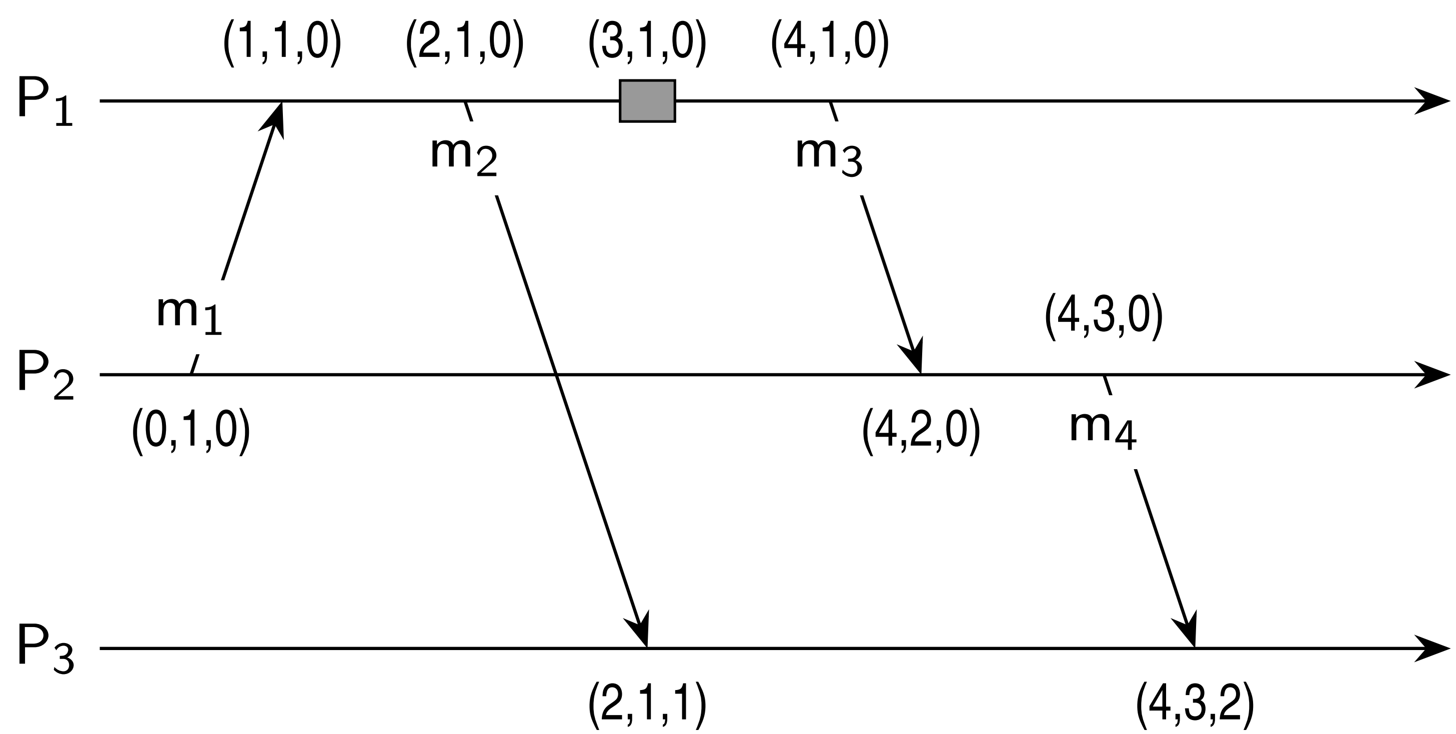
Vector Clocks

- Capturing potential causality
  - Each P_i maintains a vector VC_i
    - $VC_i[i]$ is the local logical clock at process P_i
    - If $VC_i[j] = k$ then P_i knows that k events have occurred at P_j

Vector Clocks

- Maintaining vector clocks
  - Before executing an event, P_i executes $VC_i[i]++$
  - When process P_i sends a message *m* to P_j, it sets the timestamp of m $ts(m)$ to VC_i
  - Upon receipt of a message m, process P_j sets
    - $VC_j[k] = \max\{VC_j[k], ts(m)[k]\}$
    - then increments its clock $VC_j[j]++$
    - then delivers the message to the application

Vector Clocks



$ts(m_2) = (2,1,0)$

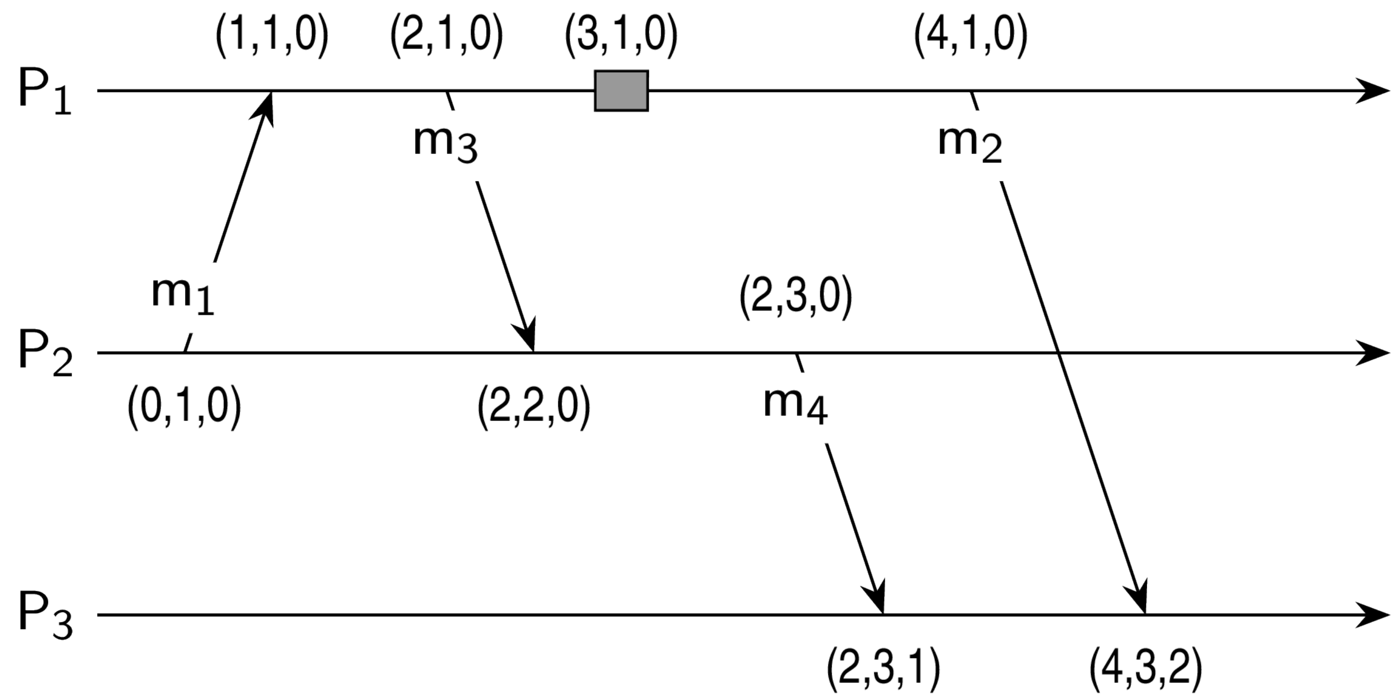
$ts(m_4) = (4,3,0)$

$ts(m2) < ts(m_4)$

$ts(m_2) \not> ts(m_4)$

m_2 may causally precede m_4

Vector Clocks



$ts(m_2) = (4,1,0)$
$ts(m_4) = (2,3,0)$

$ts(m2) \nless ts(m_4)$
$ts(m_2) \ngtr ts(m_4)$

m_2 and m_4 might conflict

Vector Clocks

- Strictly ordered multicasting:
  - All processes receive messages in exactly the same order
- Causally ordered multicasting:
  - Message that are not related to each other can be delivered in any order

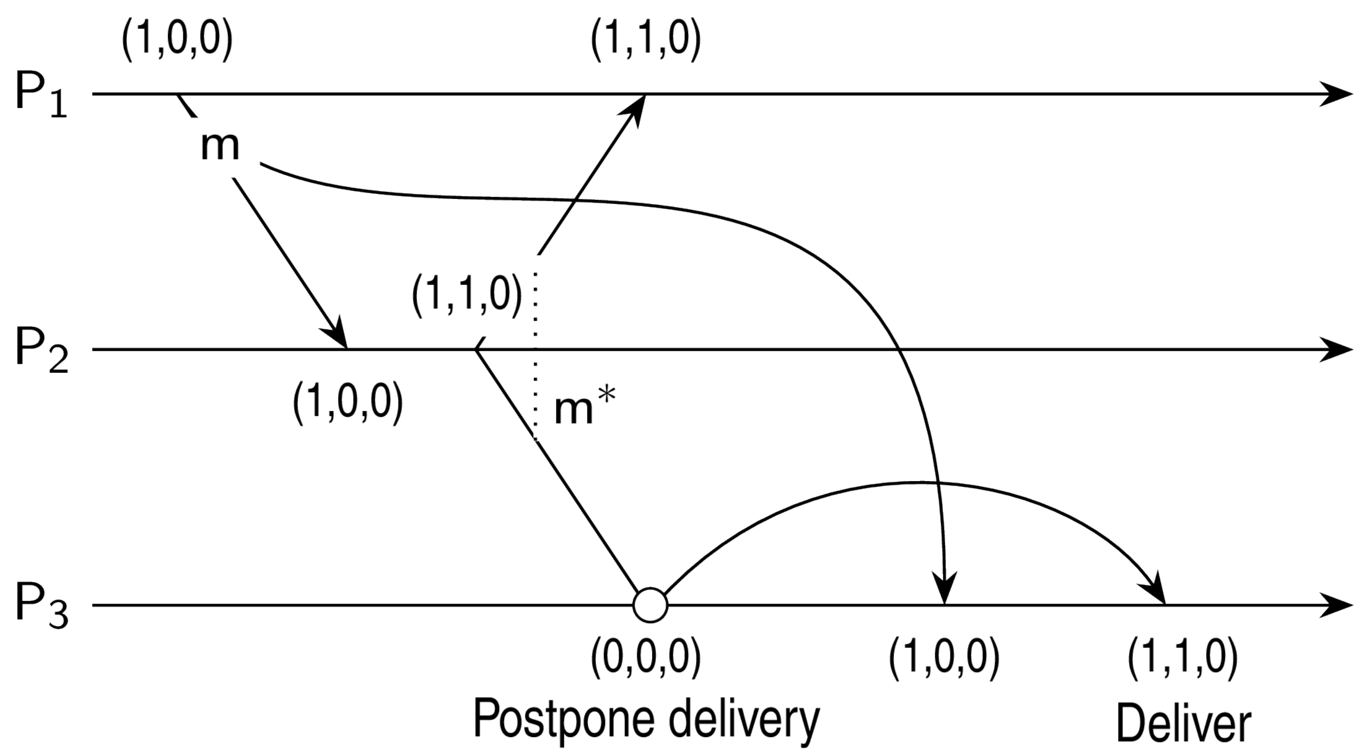
Vector Clocks

- Causally ordered multicasting
  - Observation:
    - We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.
  - Adjustment
    - P_i increments $VC_i[i]$ only when sending a message
    - P_j adjusts VC_j only when delivering a message

Vector Clocks

- Causally ordered multicasting
  - Adjustment
    - P_i increments $VC_i[i]$ only when sending a message
    - P_j adjusts VC_j only when delivering a message
    - P_j postpones delivery until:
      - $ts(m)[i] = VC_j[i] + 1$
      - $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$

Vector Clocks

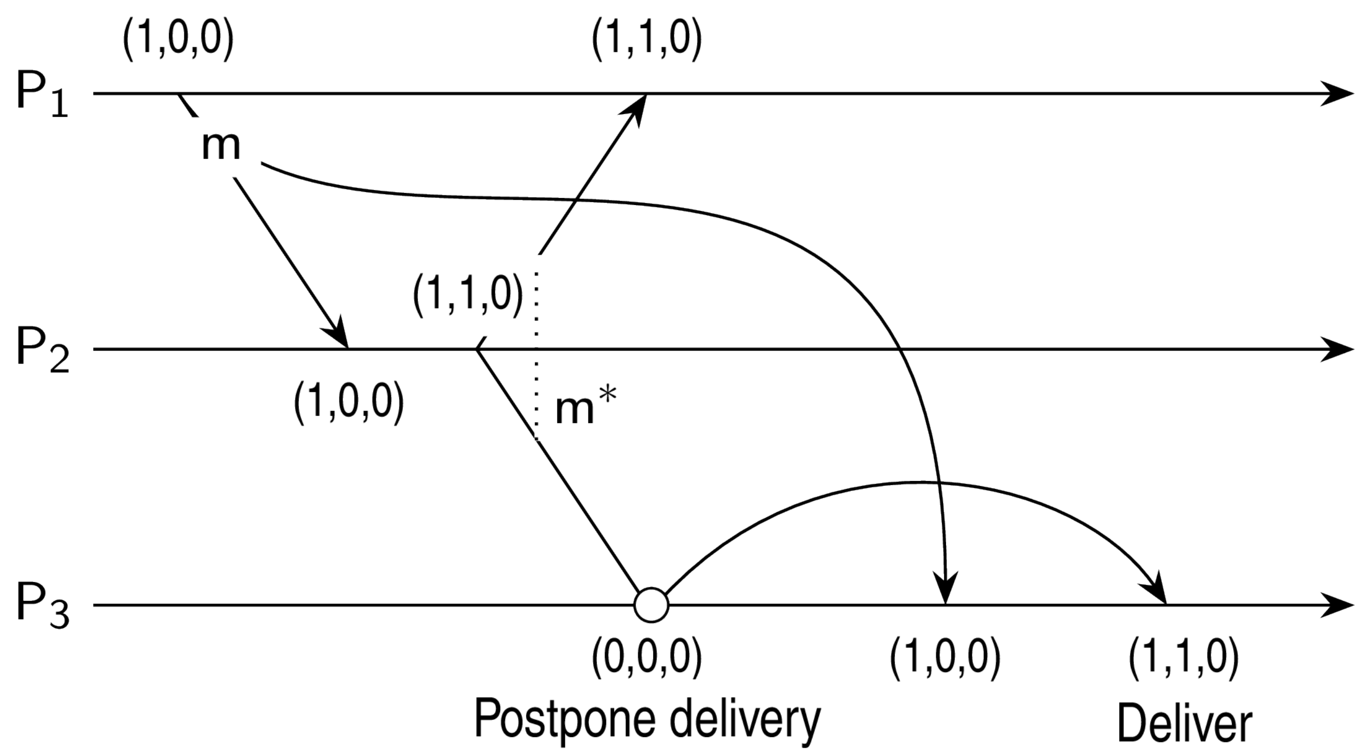


P_1 multicasts m to the other processes at time $(1,0,0)$

$ts(m) = (1,0,0)$

P_2 receives m with $VC_2 = (1,1,0)$

Vector Clocks

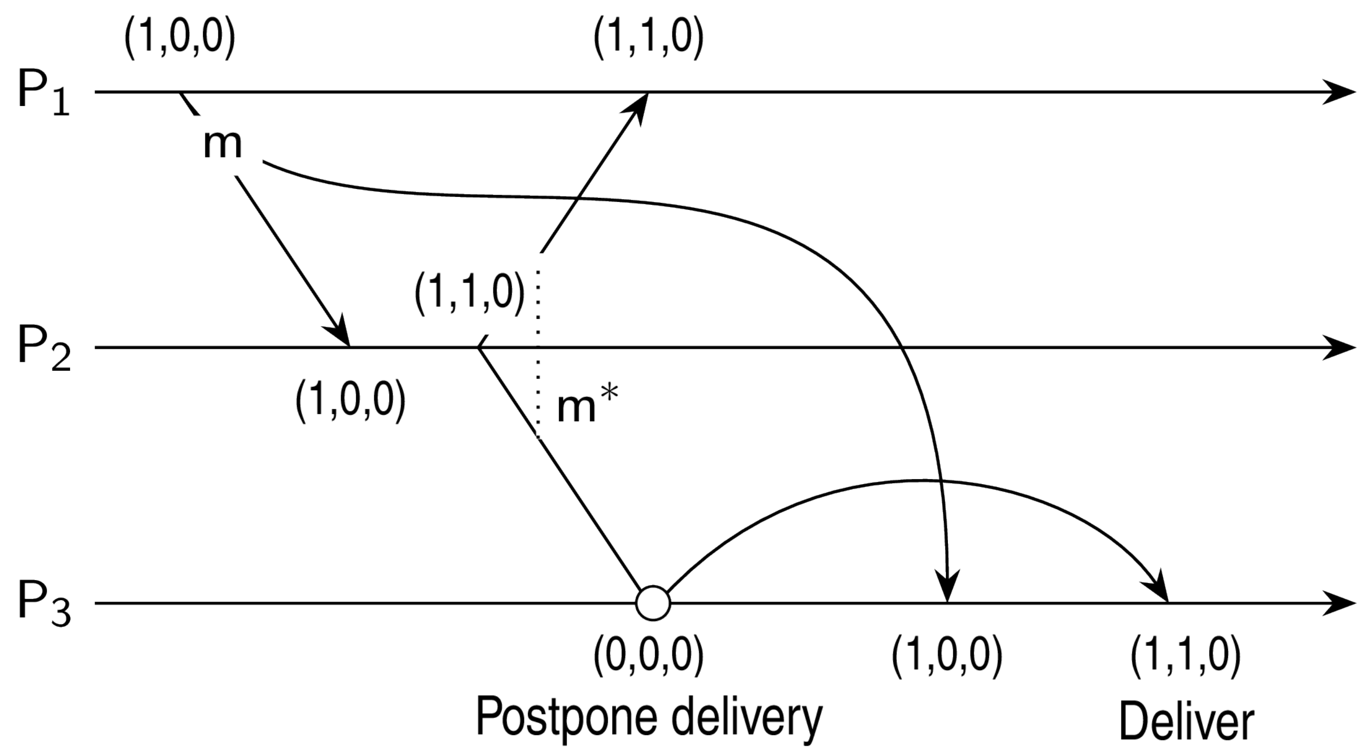


P_2 multicasts m^* to the other processes at time $(1,1,0)$

$ts(m^*) = (1,1,0)$

P_1 receives m^*

Vector Clocks



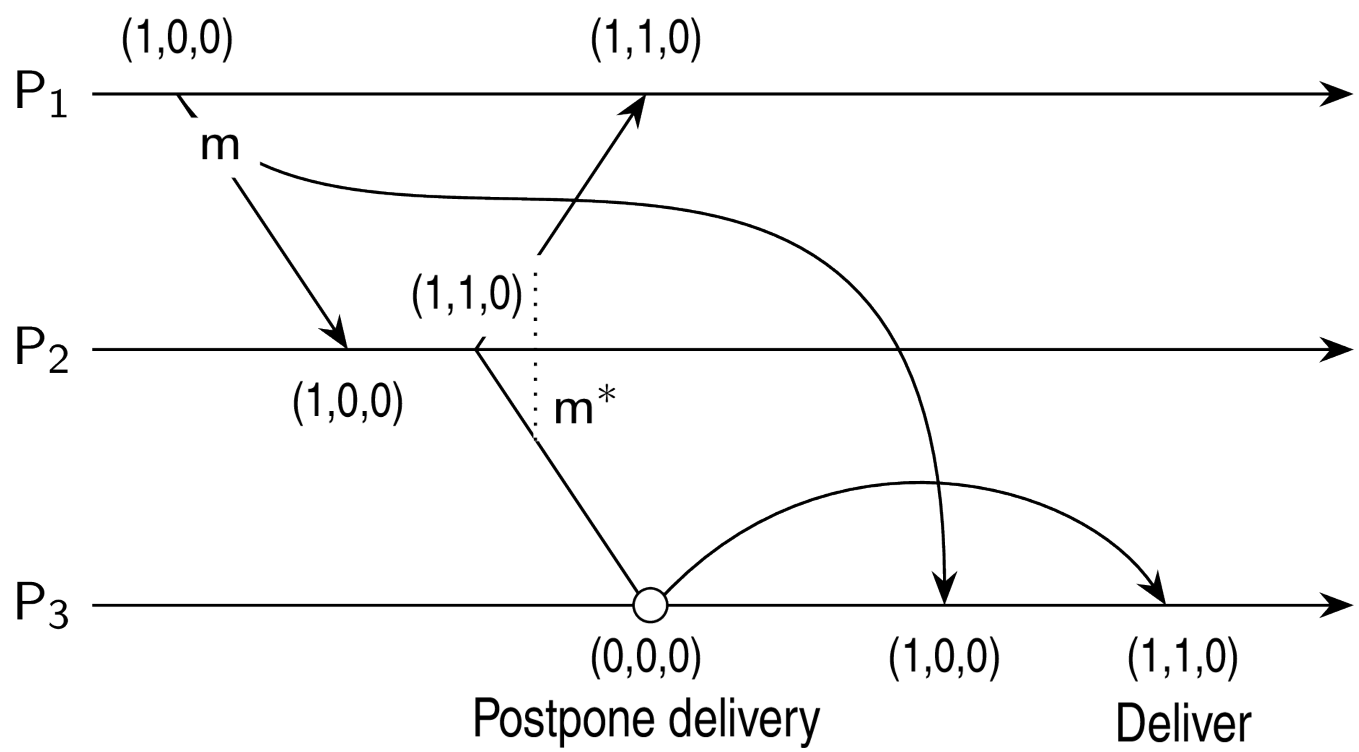
m^* arrives at P_3 before m

$ts(m^*) = (1,1,0)$

P_3 compares $ts(m^*) = (1,1,0)$ with its clock $VC_3 = (0,0,0)$

This shows that P_3 is missing a message from P_1 and m is delayed

Vector Clocks



m arrives with $ts(m) = (1,0,0)$

P_3 compares with $VC_3 = (0,0,0)$ and delivers m

P_3 now has $VC_3 = (1,0,0)$

Now m^* can be delivered and $VC_3 = (1,1,0)$

Vector Clocks

- Some middleware (ISIS, Horus) support totally and causally ordered multicasting
  - Controversy over which to choose
- Middleware can only capture ***potential*** causality
- Not all causality is captured because of out-of-band messaging

Mutual Exclusion

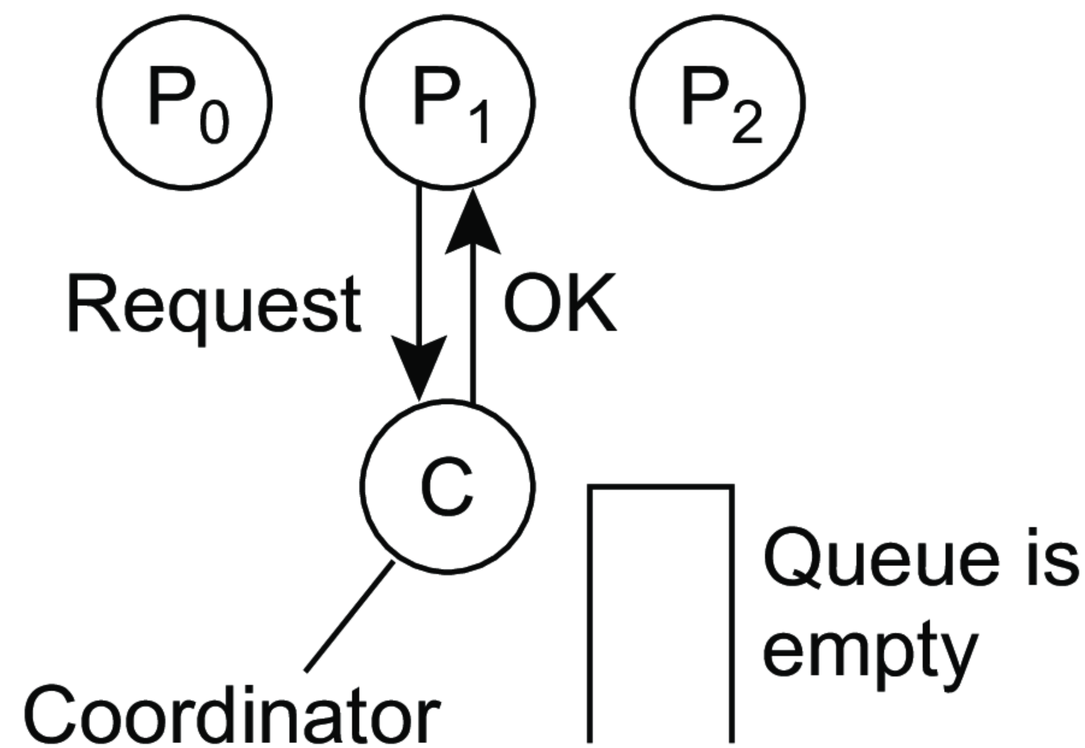
Mutual Exclusion

- Problem: Several processes want exclusive access to some resource

- Solutions:
  - Permission-based: A process wanting to grab a resource needs permission from the other processes
  - Token-based: A token is passed between processes. Only the one who has the token can grab the resource, but if it does not need the resource, pass it on to another process.

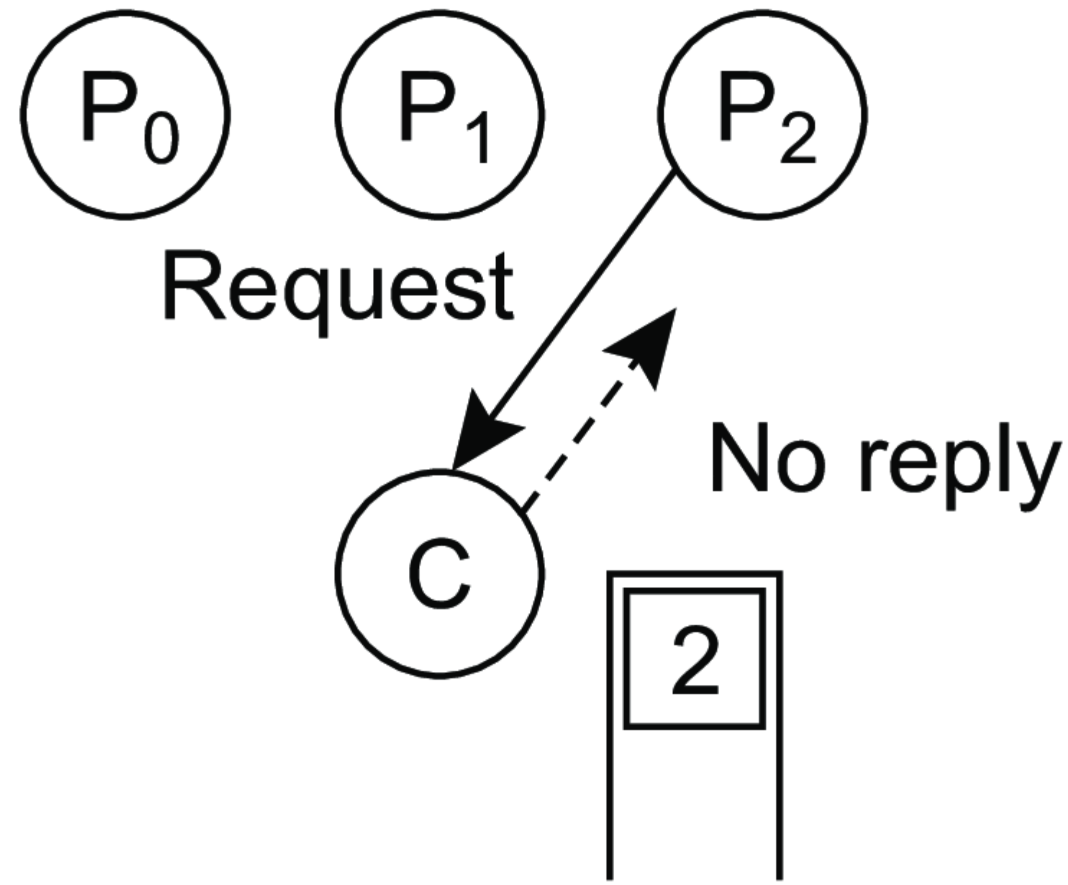
Mutual Exclusion

- A centralized solution
  - Have a single coordinator
  - Processes request from the coordinator and wait for an OK



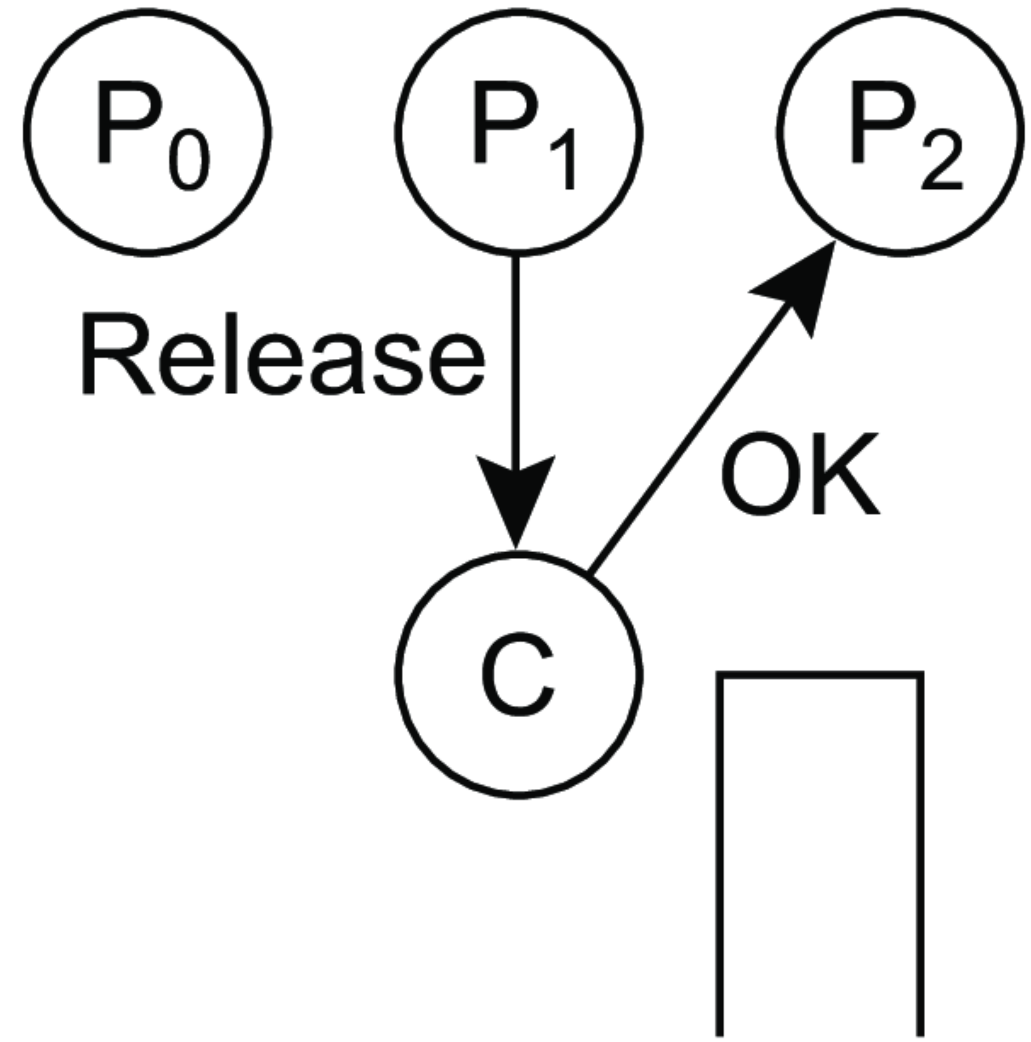
Mutual Exclusion

- If requests overlap:
  - Coordinator queues request



Mutual Exclusion

- When the resource is released (by P_1), then the request is dequeued and process P_2 obtains the resource



Mutual Exclusion

- Centralized solution without queue:
  - If the resource is held by another process, then permission is denied
  - Denied processes back-off and have to ask later

Mutual Exclusion

- Central Coordinator Protocol
    - guarantees mutual exclusions
    - is ***fair***: all processes have equal chance to obtain access to the resource

Mutual Exclusion

- Central Coordinator Protocol
  - has a single point of failure
    - Processors cannot distinguish between a dead coordinator and a busy resource
  - can become a bottleneck
  - relies on reliable messaging

Mutual Exclusion

- Lamport clocks
  - Use Lamport clocks for totally ordered multicasting
  - If a process wants to get the resource:
    - Sends a multicast REQUEST message to all other processes
    - Waits for answers from all other processes

Mutual Exclusion

- Lamport clocks
  - When a process receives a REQUEST:
    - If the process is not interested in the resource:
      - Send GO_AHEAD to the process with earliest REQUEST timestamp
    - If the process is interested in the resources: Check
      - whether it has already sent a GO_AHEAD to another process
      - whether it has made a REQUEST itself at an earlier time
      - Then do not send a GO_AHEAD

Mutual Exclusion

- Lamport clocks
  - After request, wait for a GO_AHEAD from all other processes
  - Access the resource when this is true

Mutual Exclusion

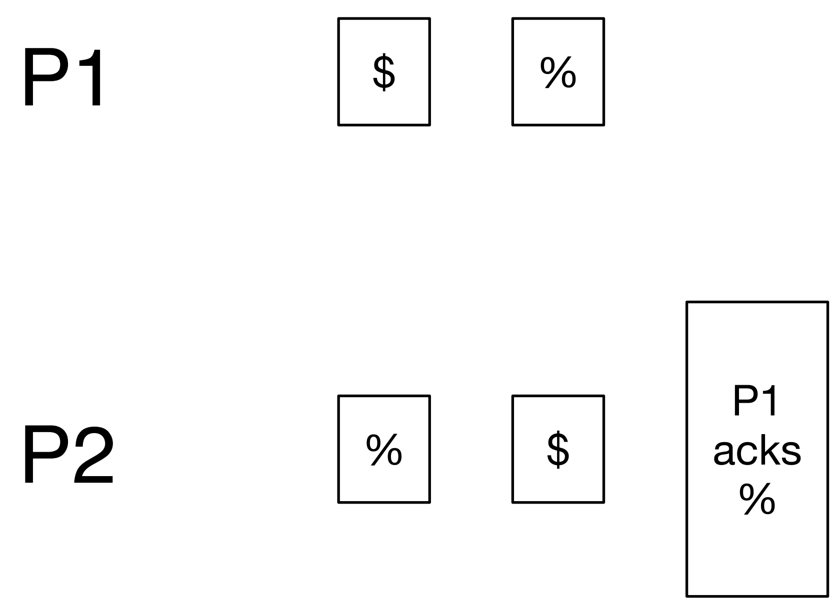
- Lamport clocks
  - If a process no longer needs the resource
    - Send RELEASE to all other processes
  - If a process receives a RELEASE
    - Remove REQUEST from the message queue
    - Remove GO_AHEADs for this request

Mutual Exclusion

- Servers with replicas but messages can be out of order
- Totally ordered multicast (almost correct)
  - At a replica: On receiving an update from a client, broadcast to other servers
  - On receiving an update from another replica:
    - Add it to your local queue
    - Broadcast an acknowledgement message to every replica
  - On receiving an acknowledgment:
    - Mark corresponding update acknowledged in your queue
  - Remove and process updates everyone has acknowledged from the head of the queue

Mutual Exclusion

- Totally ordered multicast (almost correct)
  - P_1 queues \$, P_2 queues %
  - P_1 queues % and acks %
  - P_2 marks % fully acked



P2 processes %

and eventually P1 will process \$ and then %

Mutual Exclusion

- So, this does not work. Correct version
  - At a replica: On receiving an update from a client, broadcast to other servers
  - On receiving an update from another replica:
    - Add it to your local queue
    - Broadcast an acknowledgement message to every replica for the head of the queue only
  - On receiving an acknowledgment:
    - Mark corresponding update acknowledged in your queue
  - Remove and process updates everyone has acknowledged from the head of the queue

Mutual Exclusion

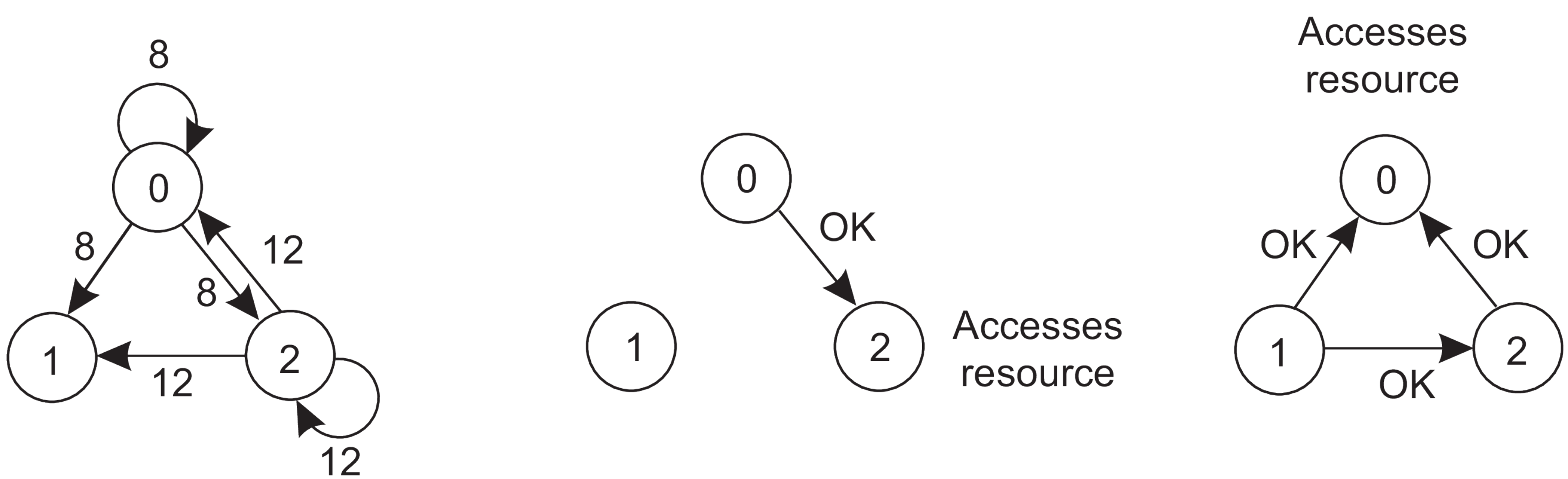
- Why is this correct?

Mutual Exclusion

- A simpler version by Ricart and Agrawala
  - Assume a total ordering of events
  - If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.
  - If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
  - If the receiver wants to access the resource as well but has not yet done so: it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message has a lower timestamp, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

Mutual Exclusion

- Example:



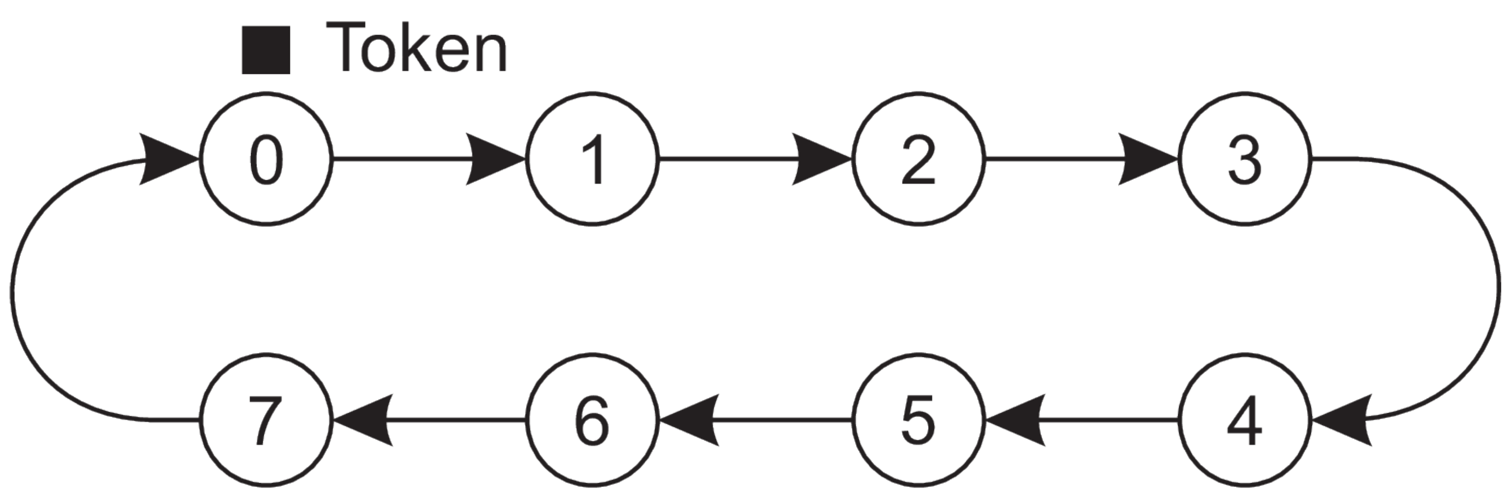
a. Two processes want to access a shared resource at the same moment.

b. P_0 has the lowest timestamp, so it wins.

c. When process P_0 is done, it sends an OK also, so P_2 can now go ahead.

Mutual Exclusion

- Token Ring Algorithm



An overlay network organized as a ring

- Only the process with the token can gain access to the resource

Mutual Exclusion

- A voting algorithm
  - Assume N replica with its own coordinator
  - Access requires a majority vote from $m > N/2$ coordinators. A coordinator always responds immediately to a request.
  - When a coordinator crashes, it will recover quickly, but will have forgotten about permissions it had granted.

Mutual Exclusion

- A voting algorithm
  - How robust is this system?
    - Processor restarts and has forgotten that it already gave permission to someone else.
  - Let $p = \Delta t / T$ be the probability that a coordinator resets during a time interval ΔT, while having a lifetime of T.
  - The probability that k coordinators out of m have reset is $\binom{m}{k} p^k (1-p)^{m-k}$
  - Need a majority of functioning coordinators: $N - (m - f) \geq m \iff f \geq 2m - N$

Mutual Exclusion

- This gives the probabilities for a violation

$$\sum_{k=2m-N}^{m} \binom{m}{k} p^k (1-p)^{m-k}$$

N	m	p	Violation
8	5	3 sec/hour	$< 10^{-5}$
8	6	3 sec/hour	$< 10^{-11}$
16	9	3 sec/hour	$< 10^{-4}$
16	12	3 sec/hour	$< 10^{-21}$
32	17	3 sec/hour	$< 10^{-4}$
32	24	3 sec/hour	$< 10^{-43}$

N	m	p	Violation
8	5	30 sec/hour	$< 10^{-3}$
8	6	30 sec/hour	$< 10^{-7}$
16	9	30 sec/hour	$< 10^{-2}$
16	12	30 sec/hour	$< 10^{-13}$
32	17	30 sec/hour	$< 10^{-2}$
32	24	30 sec/hour	$< 10^{-27}$

Mutual Exclusion

- Number of messages

Algorithm	**Messages per entry/exit**	**Delay before entry (in message times)**
Centralized	3	2
Distributed	$2 \cdot (N-1)$	$2 \cdot (N-1)$
Token ring	$1, \ldots, \infty$	$0, \ldots, N-1$
Decentralized	$2 \cdot m \cdot k + m, k = 1, 2, \ldots$	$2 \cdot m \cdot k$

Mutual Exclusion

- Delays:
  - Measured in **Message Transfer Time Units** (MTTU)
    - Centralized: 2 MTTU
    - Distributed: $N-1$ request messages and $N-1$ grant messages: $2(N-1)$ MTTU
    - Token ring: 0 MTTU to $N-1$ MTTU
    - Decentralized: depends on the number of votes that have to be taken

Mutual Exclusion

- Zookeeper
  - Developed for various coordination tasks:
    - locking
    - leader election
    - monitoring
    - …

Mutual Exclusion

- Zookeeper basics
  - No blocking:
    - Client sends messages to ZooKeeper and immediately receives a response
  - Zookeeper uses a namespace
    - Organized as a tree
      - Creating and deleting nodes
      - Reading and updating data in a node
        - Partial updates are not possible
      - Checking whether a node exists

Mutual Exclusion

- Zookeeper example:
  - To acquire a lock:
    - Create a node *lock* if the node does not already exist
    - Release a lock by deleting the node
- Zookeeper nodes can be *ephemeral or persistent*
  - Persistent nodes need to be explicitly created and deleted
  - Ephemeral nodes need to be explicitly created, but vanish if there is no contact with the client

Mutual Exclusion

- Zookeeper client should not have to poll
    - Clients can subscribe to notifications for updates on nodes or subtrees

Mutual Exclusion

- Notification problem with locking:

  (1) A client C_1 creates a node /lock .

  (2) A client C_2 wants to acquire the lock but is notified that the associated node already exists.

  (3) Before C_2 subscribes to a notification, C_1 releases the lock, i.e., deletes /lock

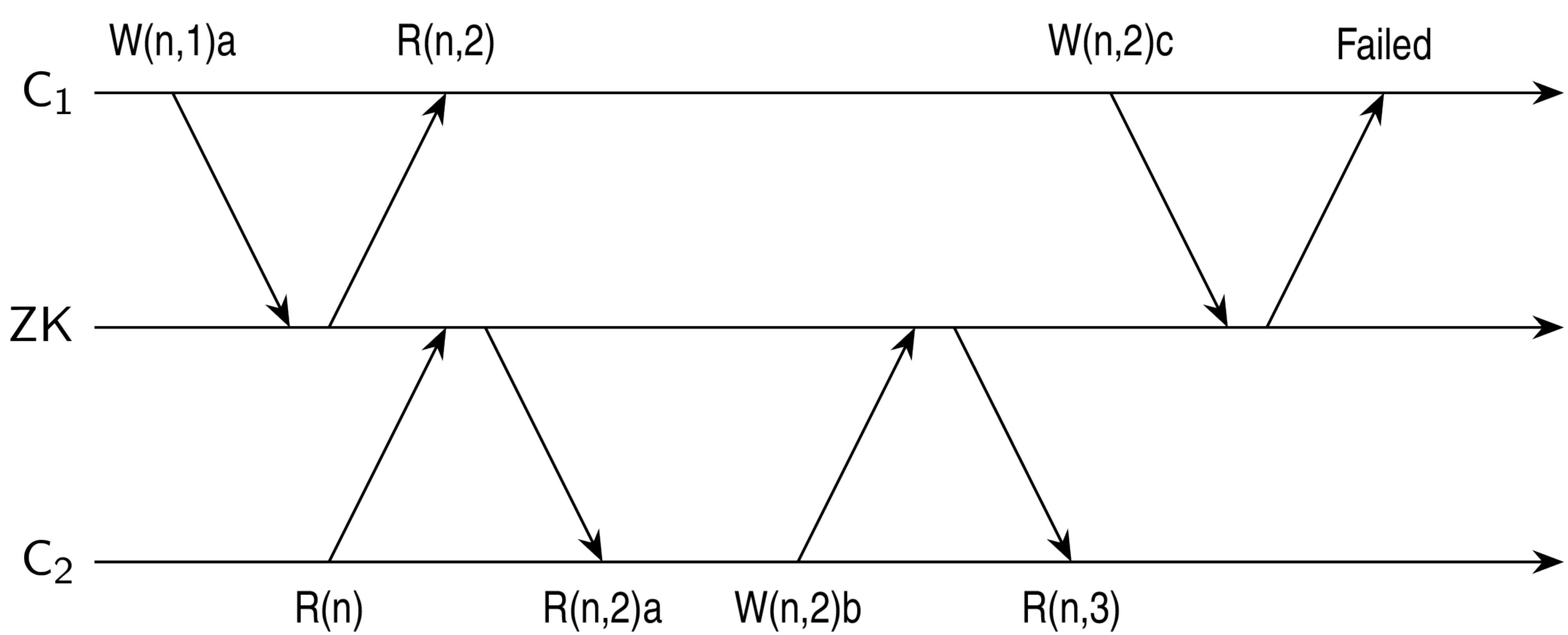
  (4) Client C_2 subscribes to changes to /lock and blocks locally.

Mutual Exclusion

- Zookeeper
  - Need to prevent a scenario, where a node is changed twice
    - Client is notified for the first update and should react to that update

Mutual Exclusion

- Zookeeper
    - Need to prevent the scenario where a client reads a node, its value changes, and the client now updates
    - User version numbers



Mutual Exclusion

- Zookeeper locking:
- First lock: A client C_1 creates a node /lock .
- Second lock: A client C_2 wants to acquire the lock but is notified that the associated node already exists
  - $\Rightarrow C_2$ subscribes to notification on changes of /lock .
- Unlock: Client C_1 deletes node /lock
  - $\Rightarrow$ all subscribers to changes are notified.

Leader Election

Election Algorithms

- Principle
  - An algorithm requires that some process acts as a coordinator. The question is how to select this special process dynamically.
  - In many systems, the coordinator is chosen manually (e.g., file servers). This leads to centralized solutions and a single point of failure

Election Algorithms

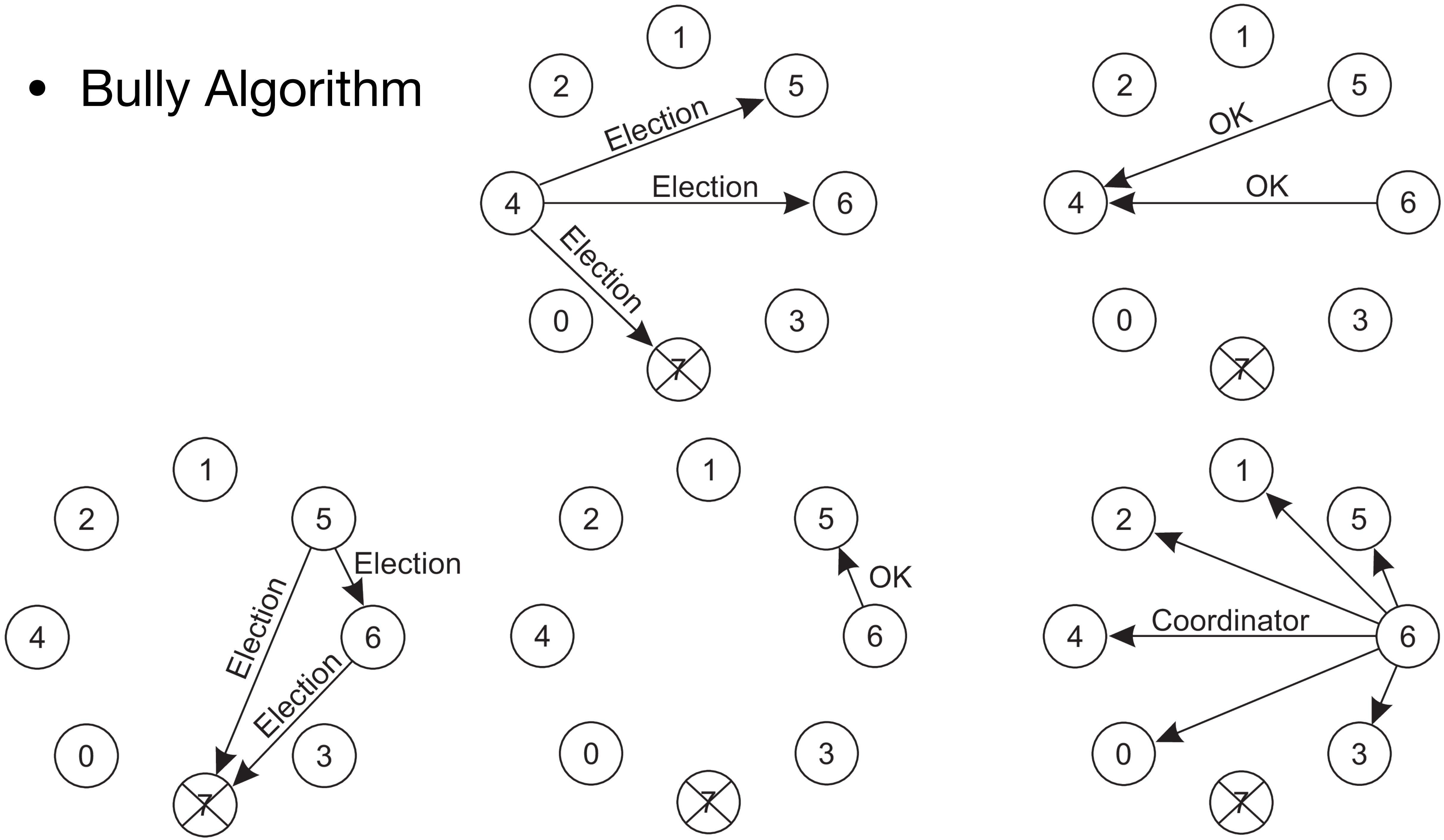
- All processes have unique id's
- All processes know id's of all processes in the system (but not if they are up or down)
- Election means identifying the process with the highest id that is up

Election Algorithms

- Bullying Algorithm
  - Consider N processes $\{P_0, P_1, \ldots, P_{N-1}\}$ and let $\mathrm{id}(P_k) = k$. When a process P_k notices that the coordinator is no longer responding to requests, it initiates an election:
    - P_k sends an ELECTION message to all processes with higher identifiers: $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$.
  - If no one responds, P_k wins the election and becomes coordinator.
  - If one of the higher-ups answers, it takes over and P_k's job is done.

Election Algorithms

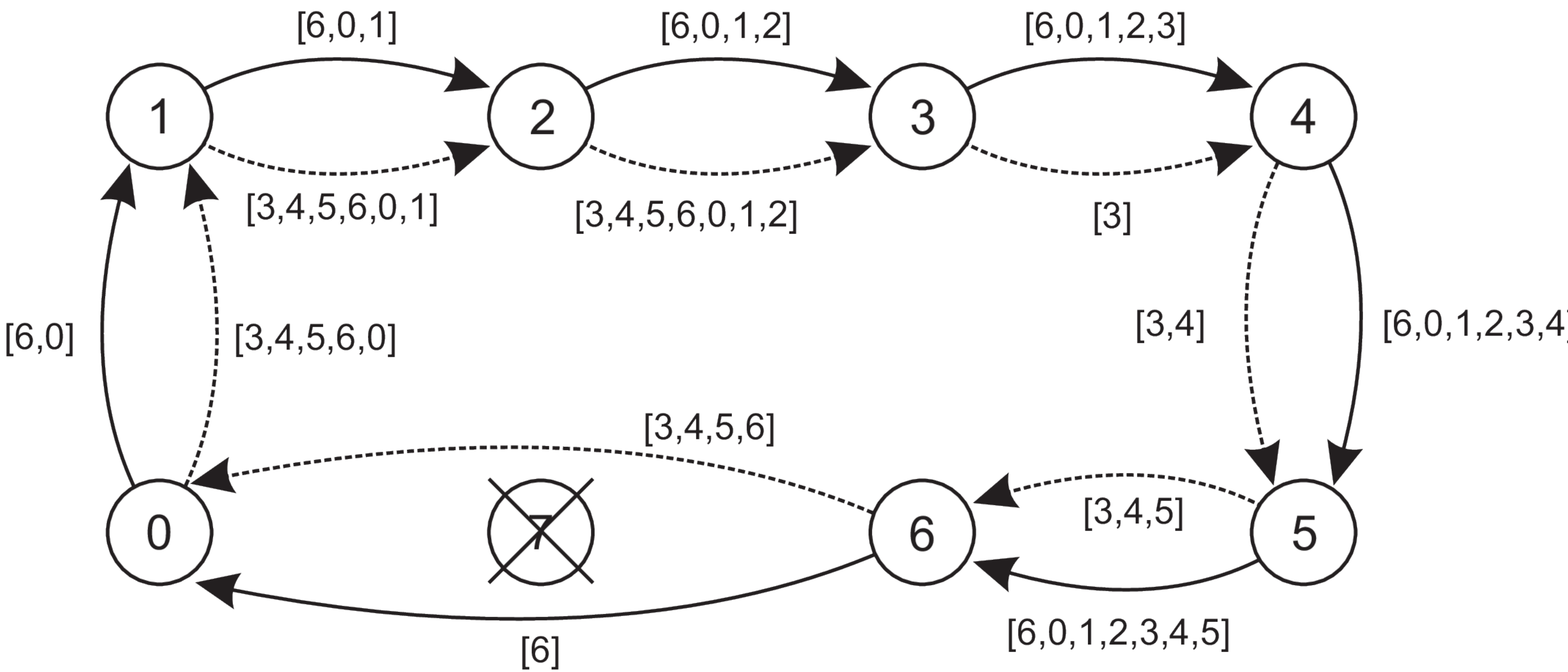
- Bully Algorithm



Election Algorithms

- Election in a ring principle
  - Process priority is obtained by organizing processes into a (logical) ring. The process with the highest priority should be elected as coordinator.
- Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on to the next successor.
- If a message is passed on, the sender adds itself to the list. When it gets back to the initiator, everyone had a chance to make its presence known.
- The initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority is elected as coordinator.

Election Algorithms



Processes 3 and 6 start procedure

Election Algorithm in Zookeeper

- Each server s in the server group has an identifier id(s)
- Each server has a monotonically increasing counter $\tau(s)$ of the latest transaction it handled (i.e., series of operations on the namespace).
- When follower s suspects leader crashed, it broadcasts an ELECTION message, along with the pair (voteID,voteTX). Initially,
  - voteID ← id(s);   voteTX ← $\tau(s)$
- When $s*$ believes it should be the leader, it broadcasts $<$ id$(s*), \tau(\ *\ ) >$.
  - This is bullying.

Election Algorithm in Zookeeper

- Each server maintains two variables:
  - leader(): records the server that s believes may be final leader. Initially, leader(s) ← id(s).
  - lastTX(s): what s knows to be the most recent transaction. Initially, lastTX(s) $\leftarrow \tau(s)$.

Election Algorithm in Zookeeper

- When s^* receives (voteID,voteTX)
  - If lastTX(s^*) $<$ voteTX , then s^* just received more up-to-date information on the most recent transaction, and sets
  - leader(s^*) ← voteID; lastTX(s^*) ← voteTX
- If lastTX(s$_*$) = voteTX and leader(s^*) $<$ voteID, then s^* knows as much about the most recent transaction as what it was just sent, but its perspective on which server will be the next leader needs to be updated:
  - leader(s^*) ← voteID

Election Algorithm in Zookeeper

- Coming to the conclusion that one server is now the leader is difficult
  - If a server is no longer able to become a leader, it becomes a follower of the alleged leader
  - If a server has enough followers, it promotes itself the leader

Election Algorithm in Zookeeper

- Newer algorithms uses the node creation mechanism
  - ZAB (ZooKeeper Atomic Broadcast) protocol

```
class Process:
  def __init__(self, chanID, procID, procIDSet, initTX):
    self.txID       = initTX       # Your own most recent transaction
    self.leader     = self.procID  # Who you believe may become leader
    self.lastTX     = self.txID    # What is the most recent transaction
    self.noleader   = False        # Are you still in the race for leader?

  def receive(self):
    while True:
      msg             = self.chan.recvFrom(self.otherProcs)
      sender, payload = msg[0], msg[1]
      if payload[0] == ELECTION: # A process started an election
        voteID, voteTX = payload[1], payload[2]

        if self.lastTX < voteTX: # You're not up to date on most recent transaction
          self.leader = voteID   # Record the suspected leader
          self.lastTX = voteTX   # As well as the likely most recent transaction

        elif (self.lastTX == voteTX) and (self.leader < voteID): # Wrong leader
          self.leader = voteID   # Update your suspected leader

        elif (self.procID > voteID) and (self.txID >= voteTX) and (not self.noleader):
          # At this point, you may very well be the new leader (having a sufficiently
          # high process identifier as well as perhaps the most recent transaction).
          # No one has told you so far that you could not be leader. Tell the others.
          self.chan.sendTo(self.otherProcs, (LEADER, self.procID, self.txID))

      if payload[0] == LEADER:
        # Check if the sender should indeed be leader
        if ((self.lastTX < payload[2]) or
            ((self.lastTX == payload[2]) and (self.leader <= payload[1]))):
          # The sender is more up-to-date than you, or is equally up-to-date but
          # has a higher process identifier. Declare yourself follower.
          self.chan.sendTo(sender, (FOLLOWER, self.procID))
        else:
          # Sender is wrong: you have information that the sender based its decision
          # on outdated information
          self.chan.sendTo(sender, (NOLEADER))
```

Election Algorithm in RAFT

- Raft is an improvement on Paxos
  - We have a (relatively small) group of servers
  - A server is in one of three states: follower, candidate, or leader
  - The protocol works in terms, starting with term 0
  - Each server starts in the follower state.
  - A leader is to regularly broadcast messages (perhaps just a simple heartbeat)

Election Algorithm in RAFT

- Selecting a new leader
- When follower s^* hasn't received anything from the alleged leader s for some time, s^* broadcasts that it volunteers to be the next leader, increasing the term by 1. s^* enters the candidate state. Then:
- If leader s receives the message, it responds by acknowledging that it is still the leader. s^* returns to the follower state.
- If another follower s^{**} gets the election message from s^*, and it is the first election message during the current term, s^{**} votes for s^*. Otherwise, it simply ignores the election message from s^*. When s^* has collected a majority of votes, a new term starts with a new leader.

Election in Large Scale Systems

- E.g. in permission-less blockchains
- Proof by Work:
  - Validators run a race. The first one to finish is the leader and get a payment
    - Each validator computes the hash of its block of validated transaction H
    - Race: Validator finds a **nonce** N such that hash(M, N) has a certain number of leading zeroes

Election in Large Scale Systems

- The number of zeroes controls the difficulty
  - Bitcoin: Race finishes in about 10 minutes
  - Since blocks have 2500 transactions, Bitcoin can handle four transactions per second
  - If races are too short, we can have a fork that needs to be dealt with
    - Also: this gives rise to insecurity

Election in Large Scale Systems

- Bitcoin uses 127 terawatt-hours of electricity
  - More than Norway
  - Alternative: ***proof of stake***
    - Assume that each transaction has one or more tokens
    - Each token has a unique owner
    - There are N tokens per blockchain

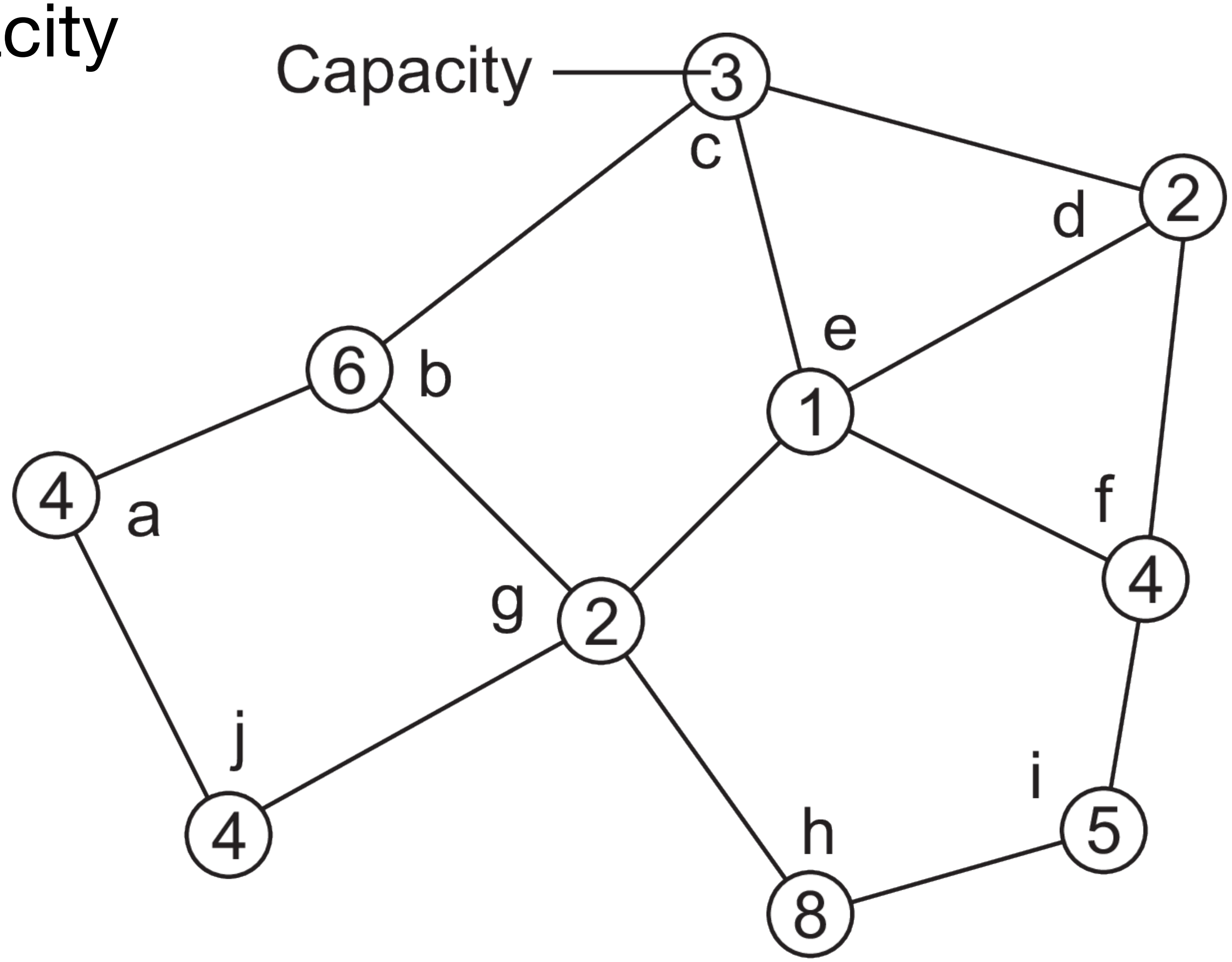
Election in Large Scale Systems

- Proof of stake:
  - Public process generates a random number generator for a number between 1 and N
  - Owner of the corresponding token is made the leader

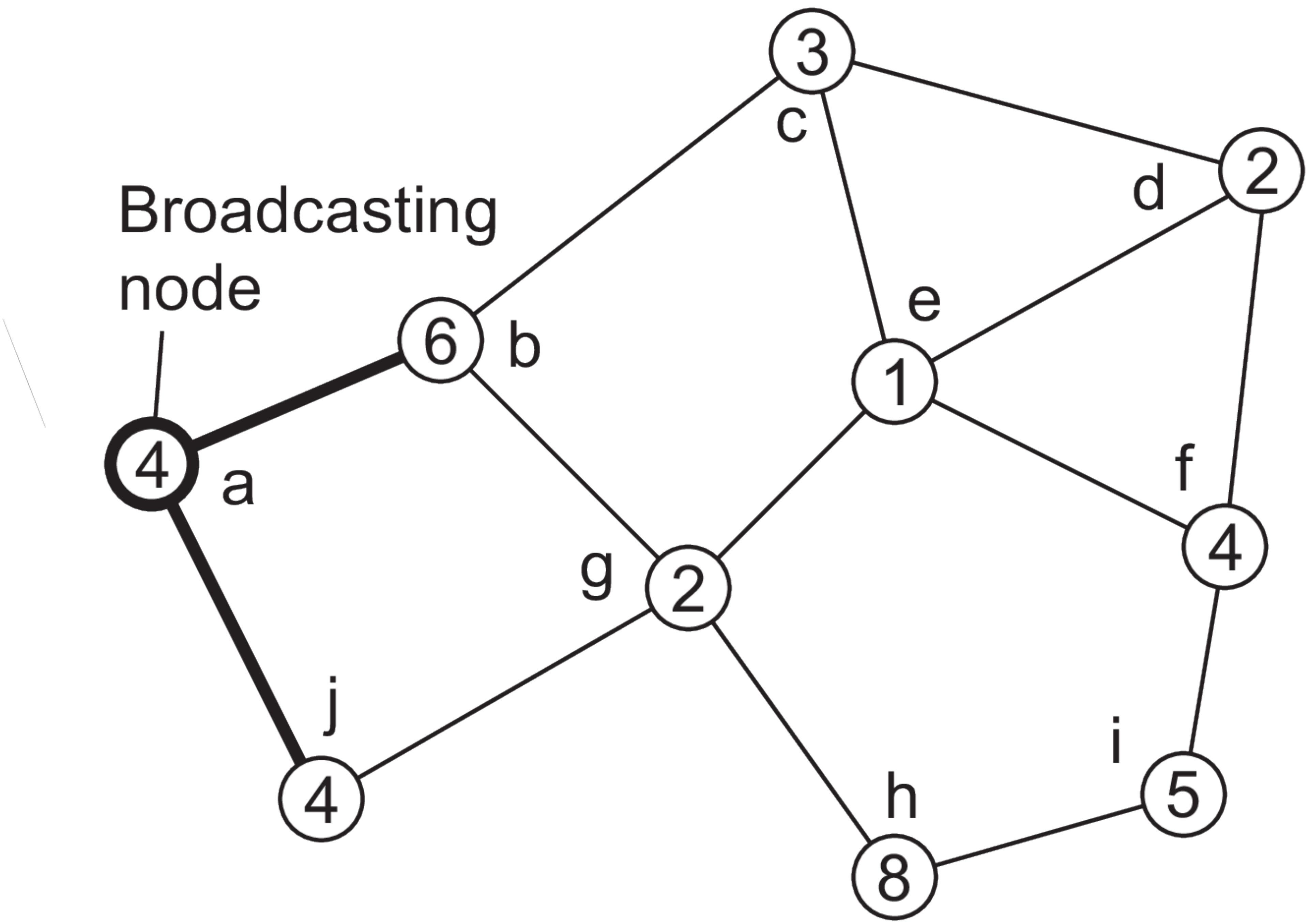
  - Lots of problems remain

Electing the Best Leader

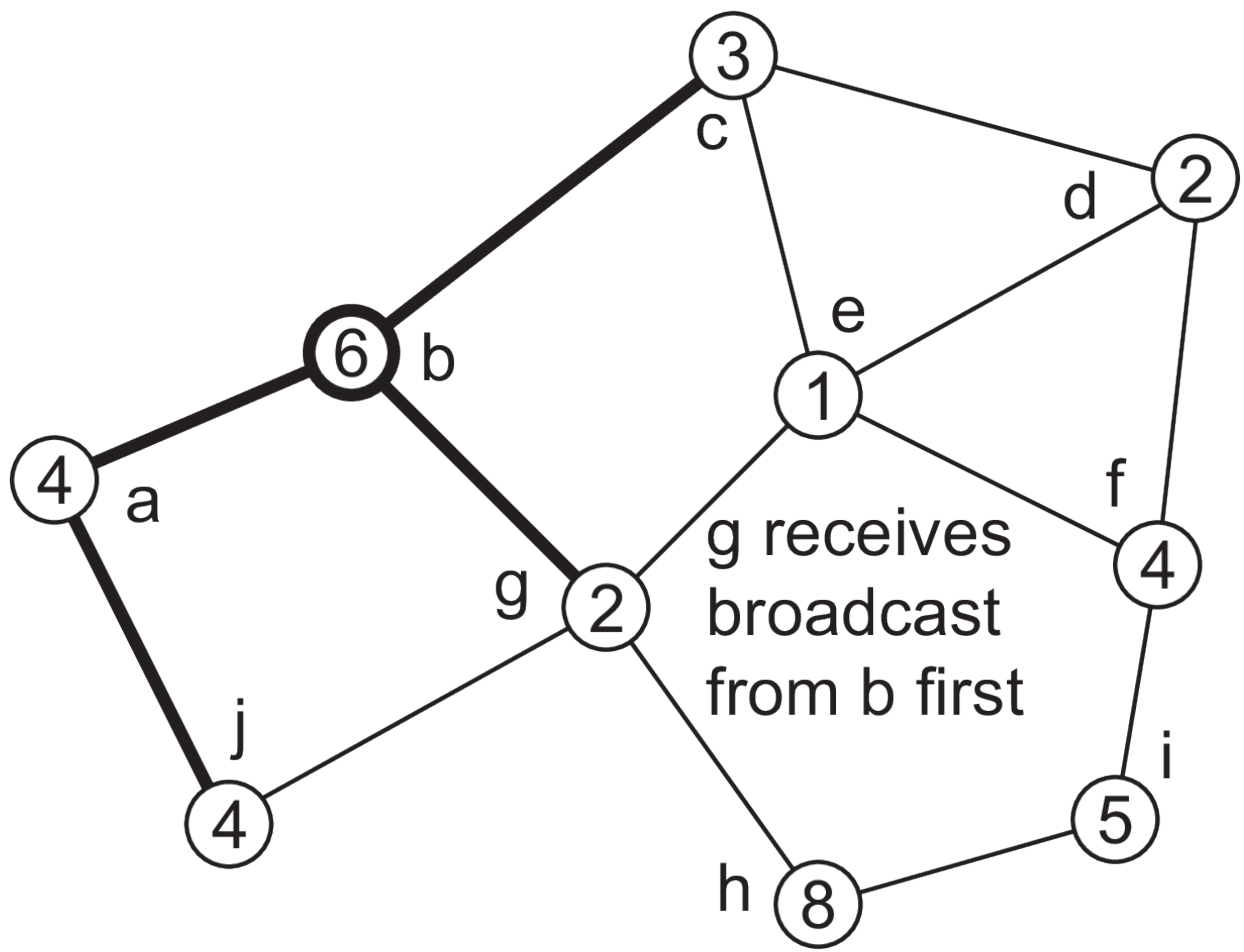
- Wireless networks: Find the ***best*** leader
  - Vasudevan: Best leader in a wireless network — max capacity



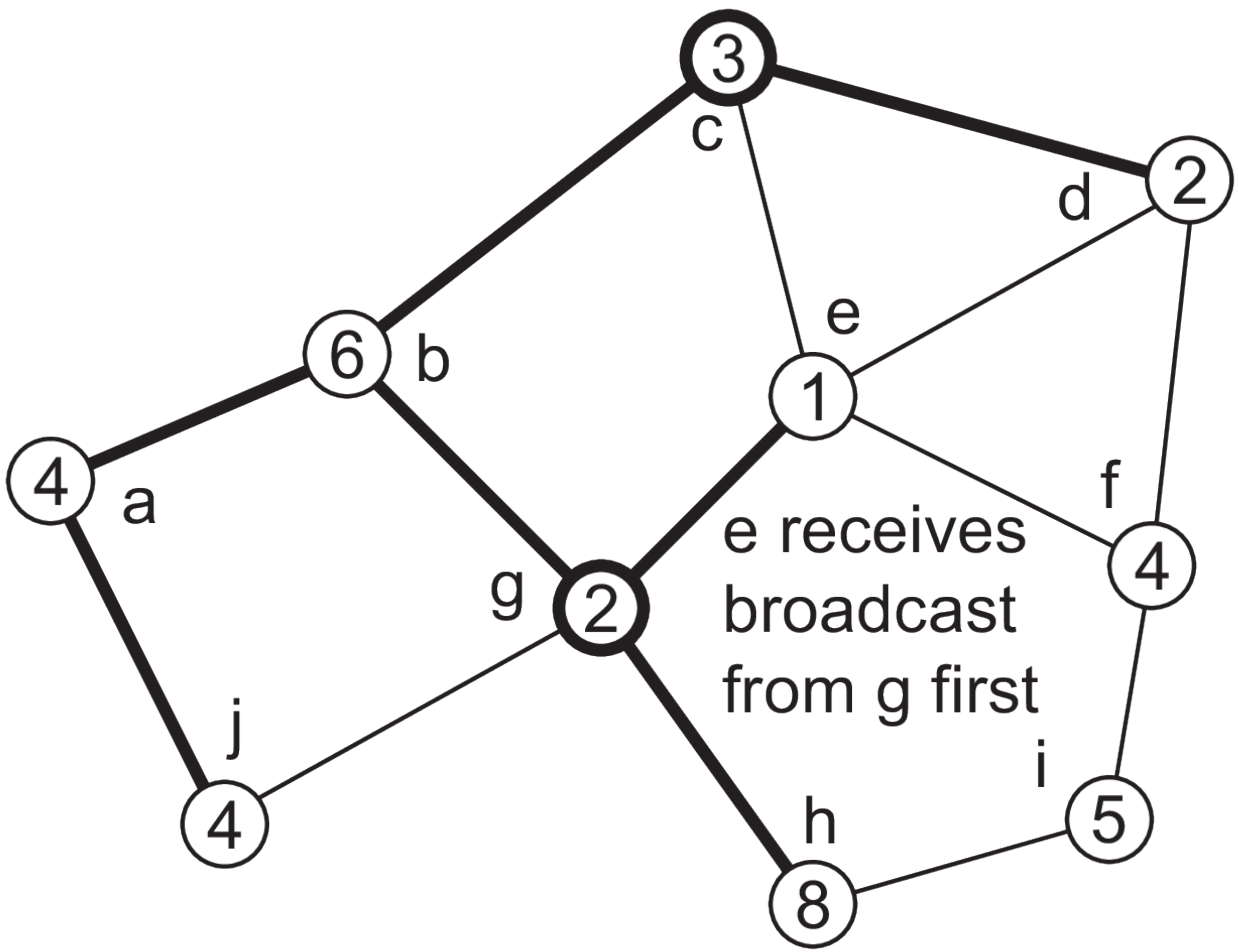
Electing the best leader



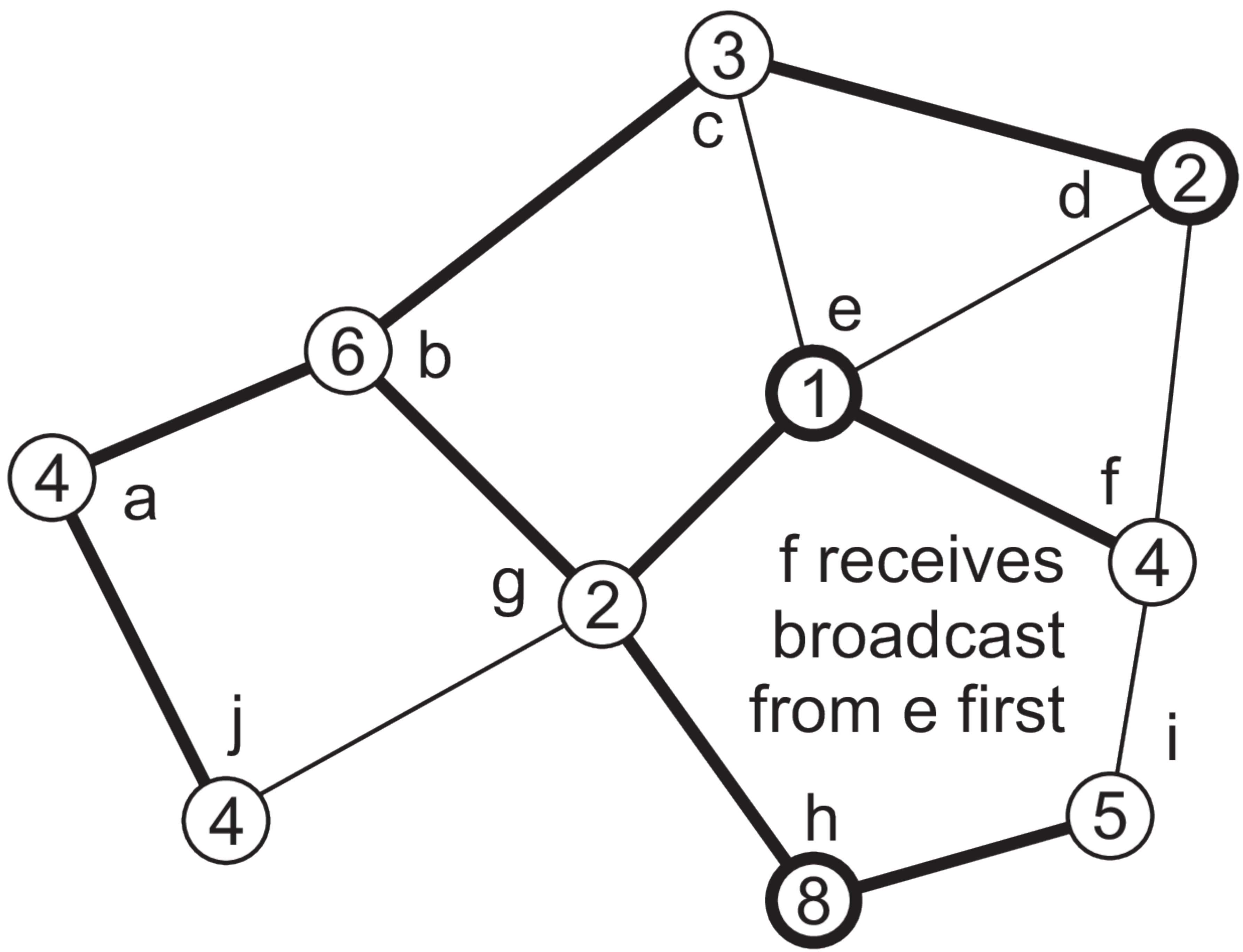
Electing the best leader



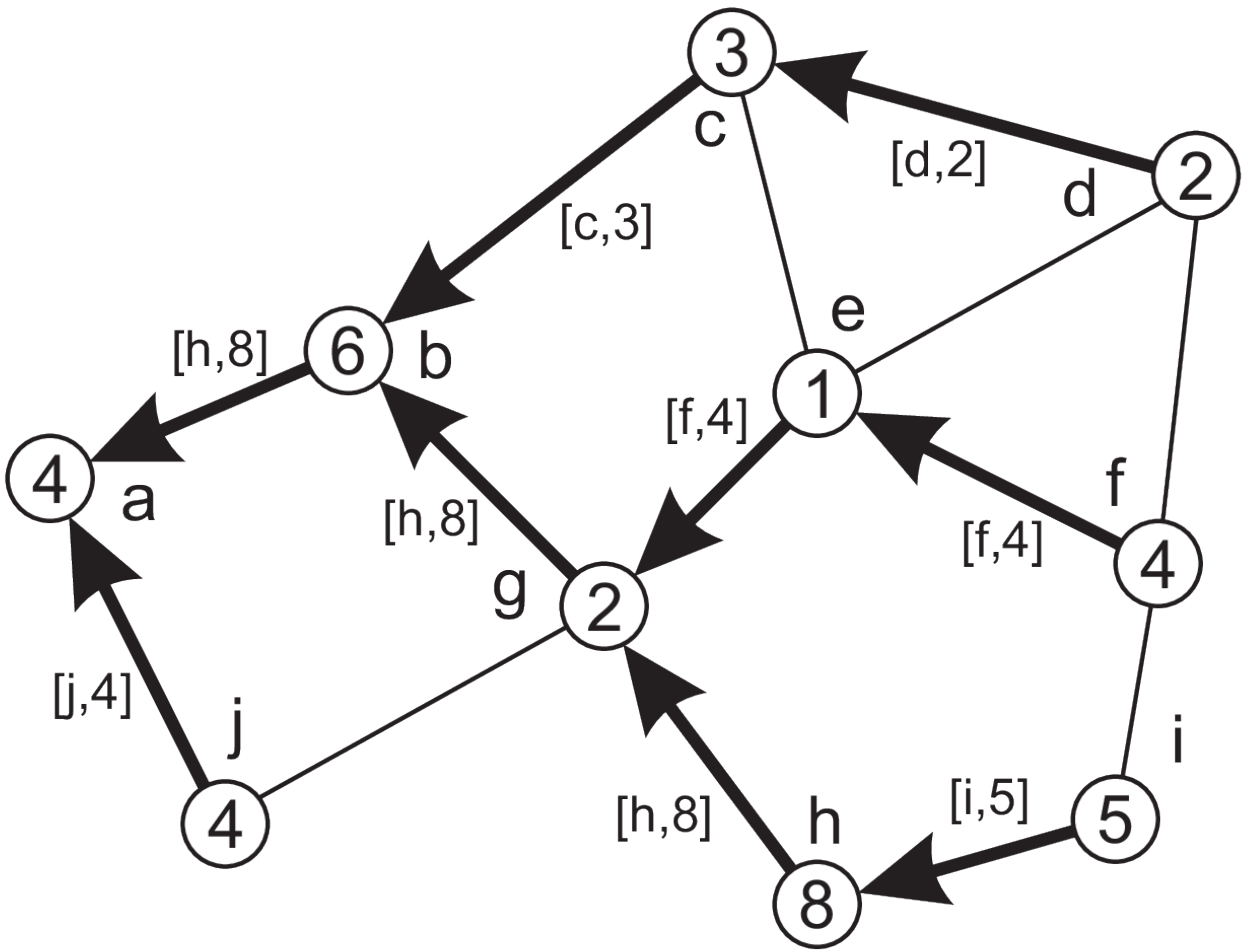
Electing the best leader



Electing the best leader



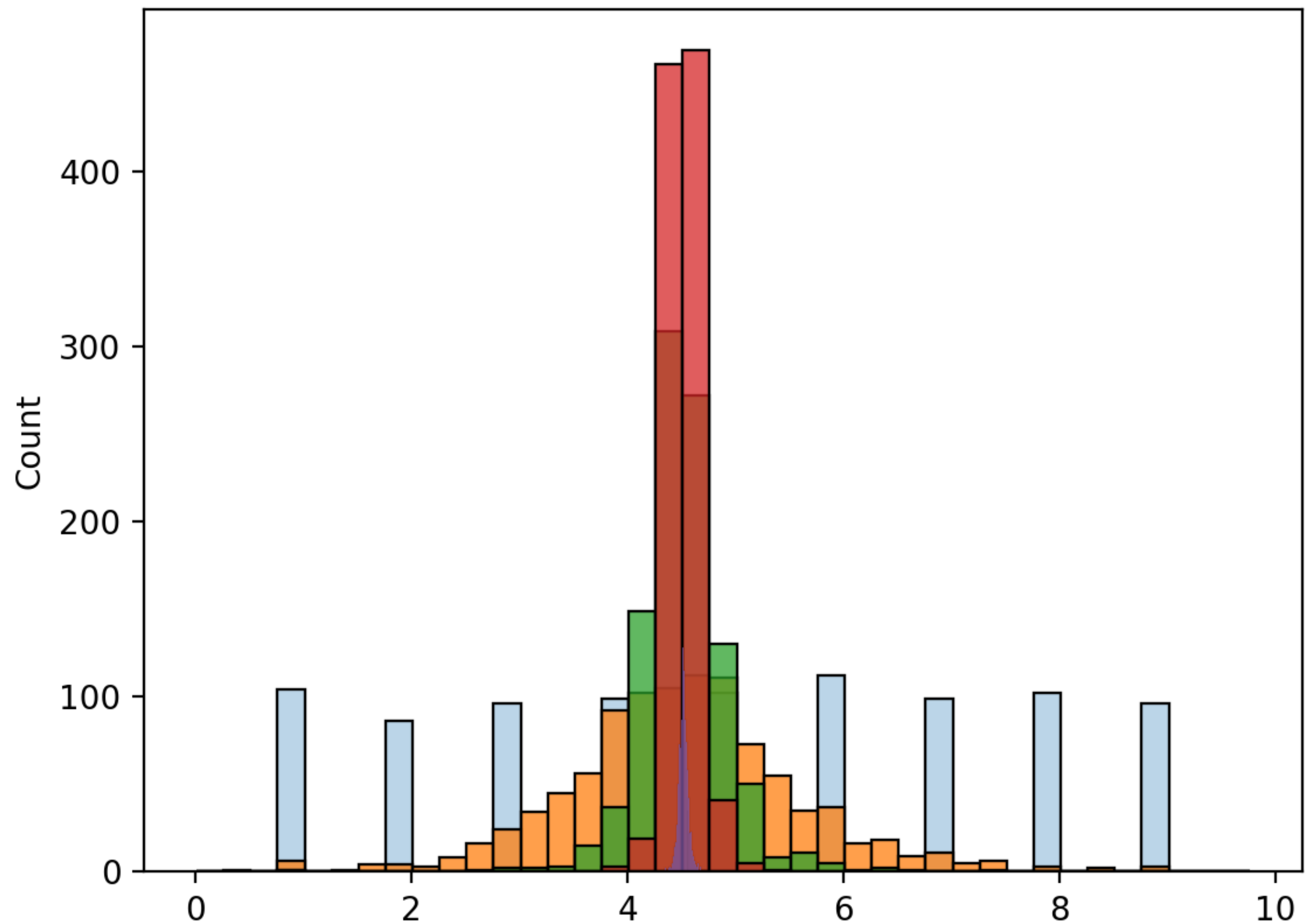
Electing the best leader



Gossip-based Coordination

- Data dissemination: Perhaps the most important one.
- Aggregation: Let every node P_i maintain a variable v_i .
  - When two nodes gossip, they each reset their variable to $v_i, v_j = \frac{v_i + v_j}{2}, \frac{v_i + v_j}{2}$
  - Result: in the end each node will have computed the average $\frac{\sum_{i=1}^{N} v_j}{N}$

Gossip-based Coordination



Histogram of values at peers original, after 2000, 4000, 6000, and 8000 interchanges of gossip

Gossip-based Coordination

- Estimating the number *N* of peers:
  - One peer sets value to one
  - All others set value to zero, when contacted
  - Gives 1/*N*

Gossip-based Coordination: Peer Sampling

- **Problem**: For many gossip-based applications, you need to select a peer uniformly at random from the entire network. In principle, this means you need to know all other peers. Impossible?
- Basics:
  - Each node maintains a list of c references to other nodes
  - Regularly, pick another node at random (from the list), and exchange roughly c/2 references
  - When the application needs to select a node at random, it also picks a random one from from its local list.
- Observation:
  - Statistically, it turns out that the selection of a peer from the local list is indistinguishable from selecting uniformly at random peer from the entire network

Gossip-based Coordination: Peer Sampling

- Task: have a node P choose another node Q *at random*
- ***Peer-Sampling Service*** *(PSS)*
  - Each node maintains a list of *c* nodes (with age of entry) (***partial view***)
  - Algorithm updates the partial view as follows:
    - selectPeer: Randomly select a neighbor from the local partial view
    - selectToSend: Select some other entries from the partial view, and add to the list intended for the selected neighbor.
    - selectToKeep: Add received entries to the partial view, remove repeated items, and shrink the view to *c* items.

Gossip-based Coordination: Peer Sampling

```
selectPeer(&Q);
selectToSend(&bufs);
sendTo(Q, bufs);

receiveFrom(Q, &bufr);
selectToKeep(p_view, bufr);
```

```
receiveFromAny(&P, &bufr);
selectToSend(&bufs);
sendTo(P, bufs);
selectToKeep(p_view, bufr);
```

Initiator P

Selected Peer Q

Gossip-based Coordination: Peer Sampling

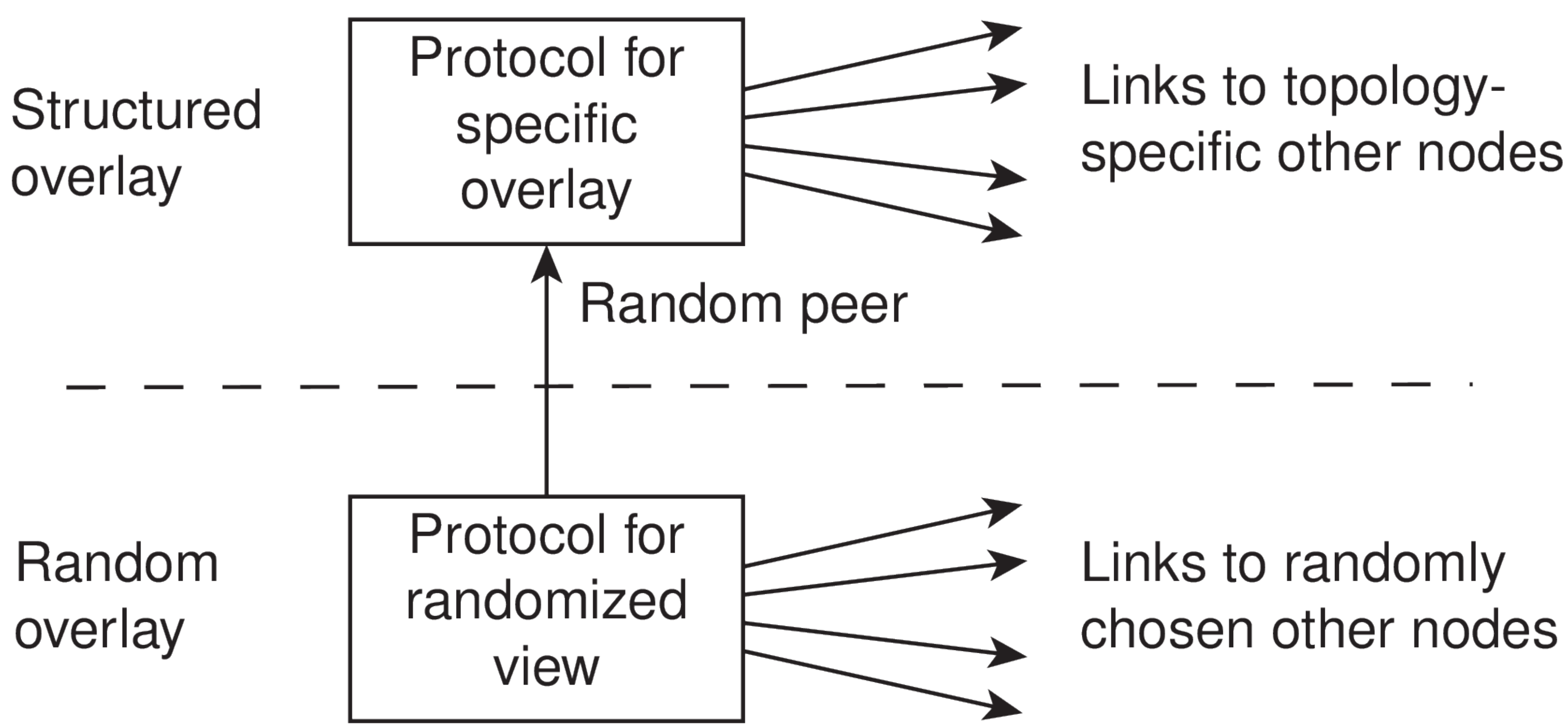
- Constructing the new partial views
  - First approach:
    - Discard the nodes sent to each other. Swaps parts of the partial view.
  - Second approach:
    - Discard the oldest peers

Gossip-based Coordination: Peer Sampling

- If selecting a random peer from a partial view happens at the same frequency as the exchange of partial views, peer selection is statistically indistinguishable from random selection

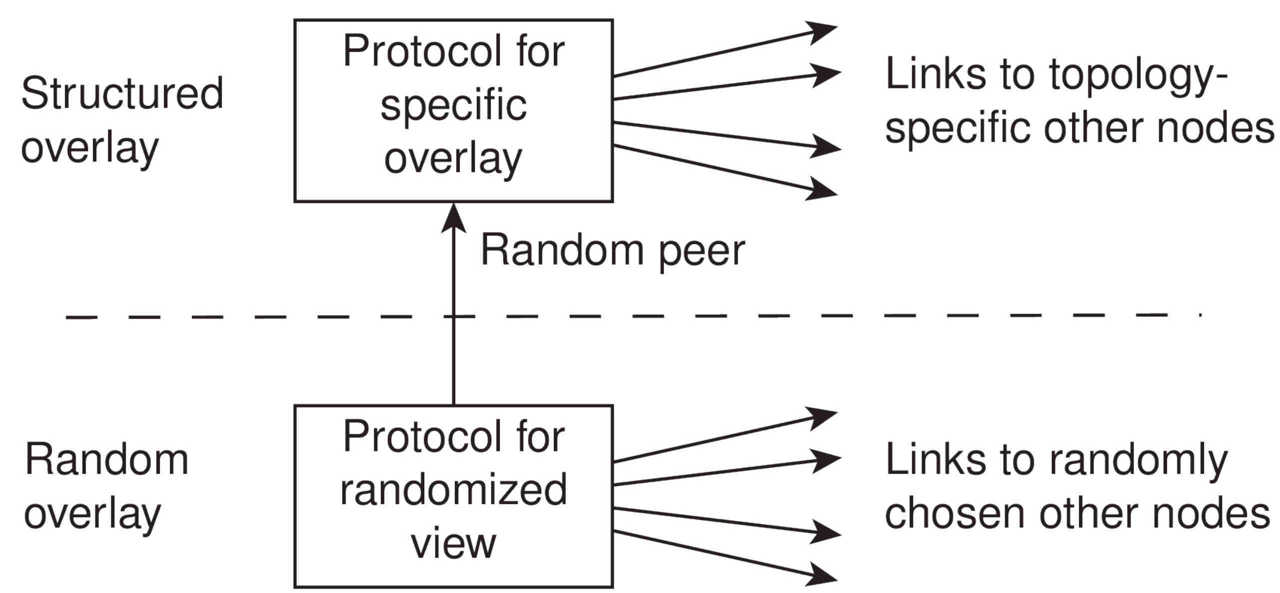
Gossip-based Coordination: Peer Sampling

- Application: Overlay construction
  - Uses a two-layer approach



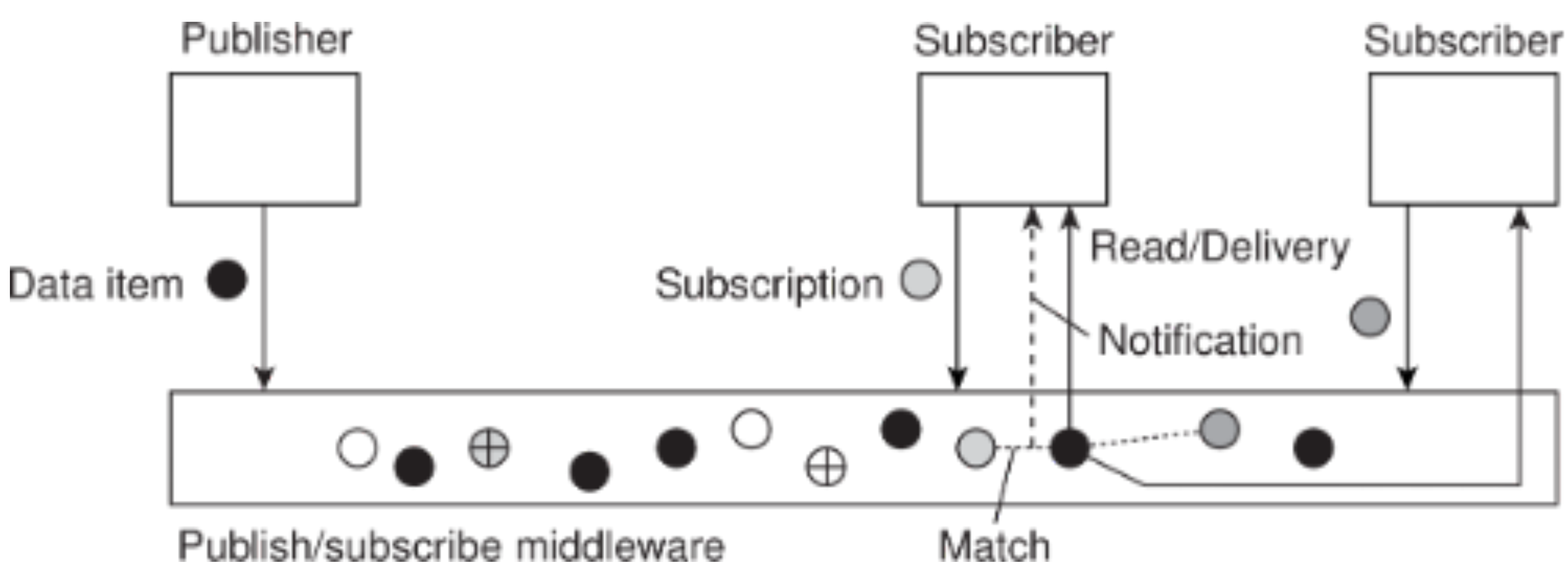
Gossip-based Coordination: Peer Sampling

- Lower layer passes on a list of randomly chosen nodes to upper layer
- Use Ranking in order to select peers
- Example:
    - Nearest nodes in terms of network hops



Distributed Event Matching

- A process specifies in which events it is interested (subscription *S*)
- When a process publishes a notification N we need to see whether *S* matches *N*.



Distributed Event Matching

- Event matching a.k.a. ***notification filtering*** is used for publish-subscribe systems
  - A process specifies through a subscription *S* in which events it is interested.
  - When a process publishes a notification *N* on the occurrence of an event, the system needs to see if *S* matches *N*.
  - In the case of a match, the system should send the notification *N*, possibly including the data associated with the event that took place, to the subscriber.

Distributed Event Matching

1. Match subscriptions against events
2. Notify subscriber in case of a match

Distributed Event Matching

- Centralized implementation
  - E.g. a centralized server is the canonical solution for implementing Linda tuple spaces

Distributed Event Matching

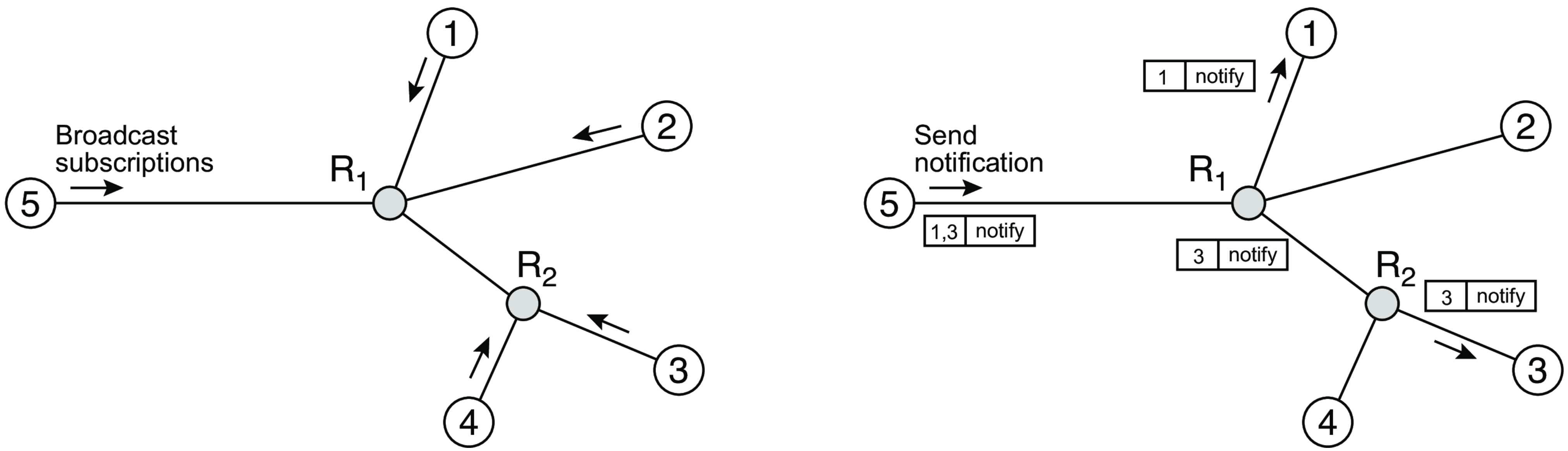
- Deterministically divide work between servers
    - Needs a function *sub2node*(*S*), which takes a subscription *S* and maps it to a nonempty subset of servers
    - Needs a function *not2node*(*N*), which takes a notification *N* and maps it to a nonempty subset of servers.
- Ensure that $sub2node(S) \cap not2node(N) \neq \emptyset$.
- In topic based subscription services, use a hash on the topic to map to servers

Distributed Event Matching

- Use brokers:
  - Organized in an overlay network
  - Use flooding to ensure that notifications reach subscribers
    - Store each subscription at each broker and send a notification to only one broker
    - Store subscription at only one broker and flood notifications to all brokers

Distributed Event Matching

- Use brokers with selective routing



- Routers receive list of subscriptions and make routing decision based on these lists

Distributed Event Matching

- Gossiping based solutions: Sub2Sub
  - Goal: To realize scalability, make sure that subscribers with the same interests form just a single group
  - Model: There are N attributes $a_1, a_2, \ldots, a_N$. An attribute value is always (mappable to) a floating-point number.
  - Subscription: Takes forms such as $S = < a_1 \rightarrow 0.3, a_4 \rightarrow [0.0, 0.5) >$
    - a1 should be 0.3; a4 should lie between 0.0 and 0.5; other attribute values don't matter.

Distributed Event Matching

- Sub2Sub
  - Each subscription s_i specifies a subset $S_i \subset \mathbb{R}^n$
  - Notifications in $\mathbb{S} = \cup_i S_i$ are only ones of interest
  - Use gossiping to partition $\mathbb{S}$ into disjoint subspaces
    - each part is in a subscription set S_i

Distributed Event Matching

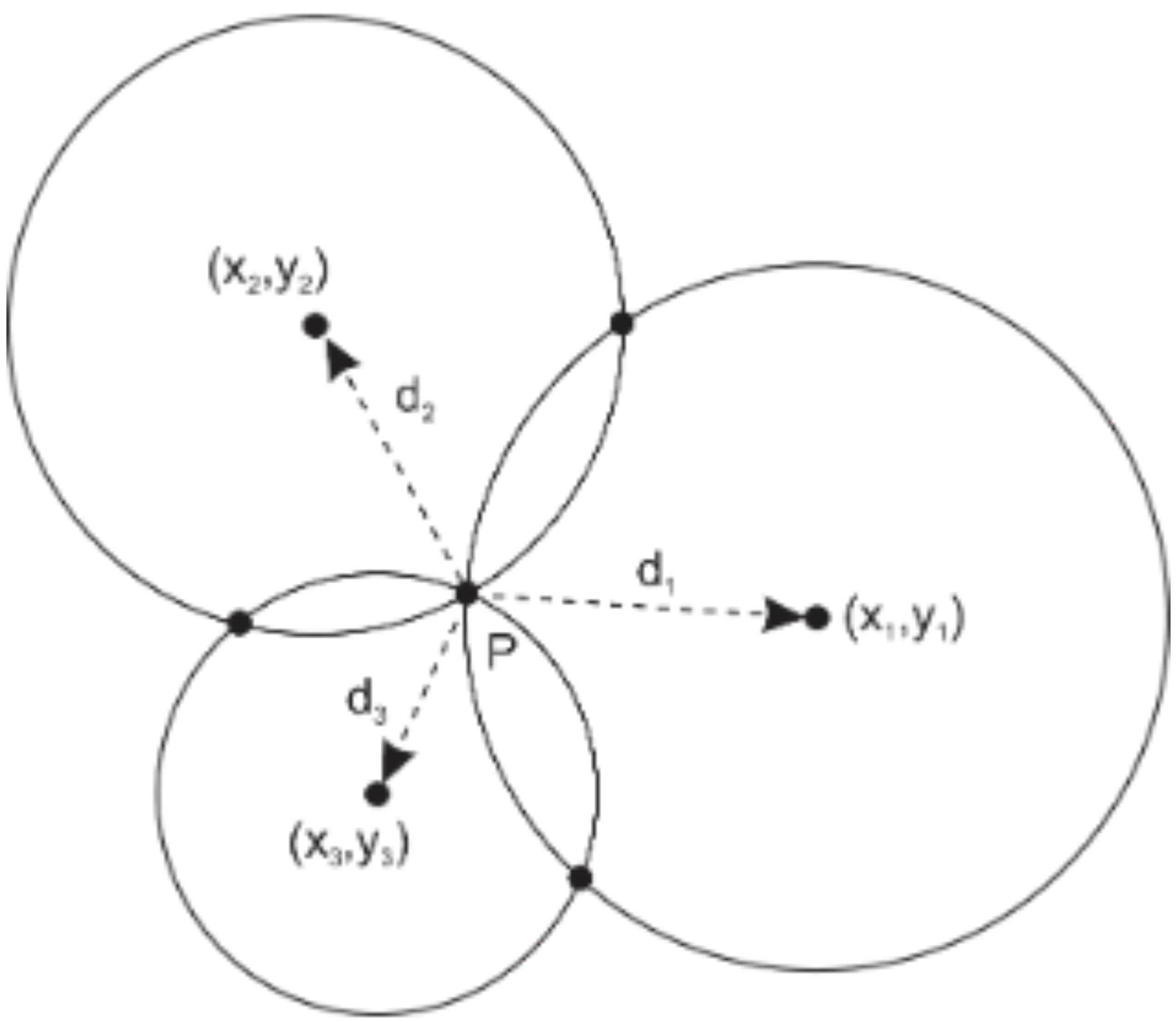
- Sub2Sub
  - Nodes regularly exchange subscriptions though gossiping
    - If their subscriptions intersect, keep references to each other
    - If $S_{ijk} = S_i \cap S_j \cap S_k \neq \emptyset$ and $S_{ij} - S_{ijk} \neq \emptyset$ then
      - nodes *i, j, k* are grouped in a single overlay network for S_{ijk}
      - nodes i, j are grouped in a single overlay network for S_{ij}.

Positioning

- In large-scale distributed systems, we often need to take some notion of proximity or distance into account
  - It starts with determining a (relative) location of a node.

Computing Position

- A point needs $d + 1$ ***landmarks*** to compute its position in d-space



$$d_i = \sqrt{(x_i - x_P)^2 + (y_i - y_P)^2}$$

Global Positioning System

- Assuming that the clocks of the satellites are accurate and synchronized
  - It takes a while before a signal reaches the receiver
  - The receiver's clock is definitely out of sync with the satellite

Global Positioning System

Basics

- Δ_r: unknown deviation of the receiver's clock.
- x_r, y_r, z_r: unknown coordinates of the receiver.
- T_i: timestamp on a message from satellite i
- $\Delta_i = (T_{now} - T_i) + \Delta_r$: measured delay of the message sent by satellite i.
- Measured distance to satellite i: $c \times \Delta_i$ (c is speed of light)
- Real distance: $d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$

4 satellites ⇒ 4 equations in 4 unknowns
(with Δ_r as one of them)

WIFI based location system

- Basic idea
  - Assume we have a database of known access points (APs) with coordinates
  - Assume we can estimate distance to an AP
  - Then: with 3 detected access points, we can compute a position.

WIFI based location system

- War driving: locating access points
  - Use a WiFi-enabled device along with a GPS receiver, and move through an area while recording observed access points.
  - Compute the centroid: assume an access point AP has been detected at N different locations $\{x_1, x_2, \ldots, x_N\}$ with known GPS location.
  - Compute location of AP as the mean $\frac{\sum_{i=1}^{N} x_i}{N}$

WIFI based location system

- ***Geometric Overlay Networks***
  - Each node is given a position in an N-dimensional space
  - Distance between nodes reflect network-latency
    - ***Network Coordinate System***