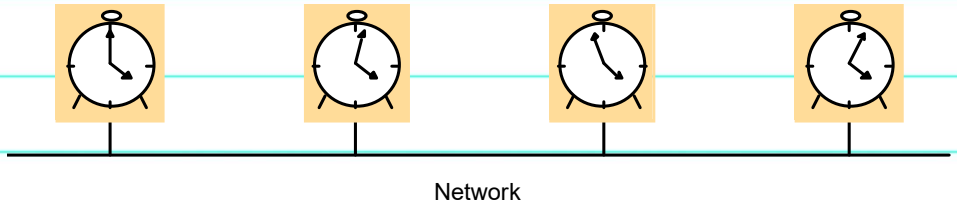
CSIE52400/CSIEM0140 Distributed Systems

Lecture 09: Coordination

Shiow-yang Wu (吳秀陽)
Department of Computer Science and Information Engineering
National Dong Hwa University

Clock Skew

- The instantaneous difference between the readings of any two clocks.
- Computer clock skew is caused by difference in frequency oscillations.



Physical Clocks

- Clock drift rate -- change in offset between clock and reference clock per unit of time
  - 10^{-6} seconds/second $\rightarrow$ 1 second every 11.6 days
- Reference clock:
  - astronomical time -- based on Earth's rotation on its axis and revolution about the sun
  - atomic time -- transitions of the Caesium-133(銫) atom
    - International Atomic Time (high-precision atomic coordinate time standard)
    - Coordinated Universal Time (UTC), successor of the Greenwich Mean Time (GMT).

Necessity for Clock Synchronization

- Need for accurate measure of time
  - e.g., time of day
- Algorithms may dependent on clock synchronization for
  - data consistency
  - check authenticity of requests to server
  - eliminate duplicate update processing
  - . . .

Clock Synchronization Requirements

- limit on
  - the deviation between clocks
  - the deviation between any clock and UTC
- clocks should only advance (monotonicity)
- only authorized principals may reset clocks

Modes of Synchronization

- External synchronization (accuracy)
  - synchronize with authoritative external source S (UTC)
  - for a bound α, $\forall t$, $|S(t) - C_i(t)| < \alpha$
  - C_i are accurate to within the bound α
- Internal synchronization (precision)
  - Clocks synchronize with one another.
  - for a bound π, $\forall t\ \forall i,j$ $|C_i(t) - C_j(t)| < \pi$
  - C_i agree within the bound π
- A set of clocks accurate within α will be precise within $\pi = 2\alpha$.

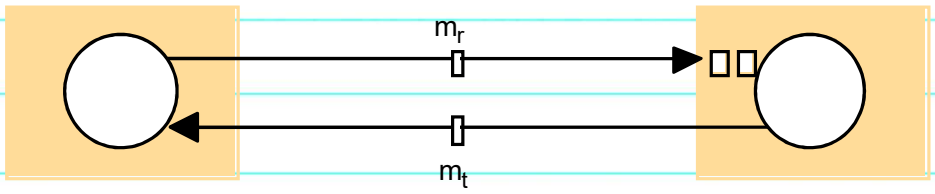
Clock Synchronization Algorithms

- **Cristian's algorithm**
  - use of time server which received UTC signals
- **Berkeley algorithm** (by Gusella and Zatti)
  - use of master/coordinator computer
- **Network Time Protocol** (NTP)
  - distribute time information over the Internet
- Clock Sampling Mutual Network Synchronization
  - CS-MNS, for distributed and mobile applications)
- **Precision Time Protocol** (PTP)
  - master/slave protocol for delivery of highly accurate time over local area networks

Cristian's Algorithm

- Given:
  - T_{round} : total roundtrip time to send and receive time message
  - t : time received from server
- Time estimate = $t + T_{round}/2$
- If the minimum transmission time is *min*, then the server's clock (when receiving the reply) is in
  $[t + min, t + T_{round} - min]$
- Therefore the accuracy is $\pm(T_{round}/2 - min)$

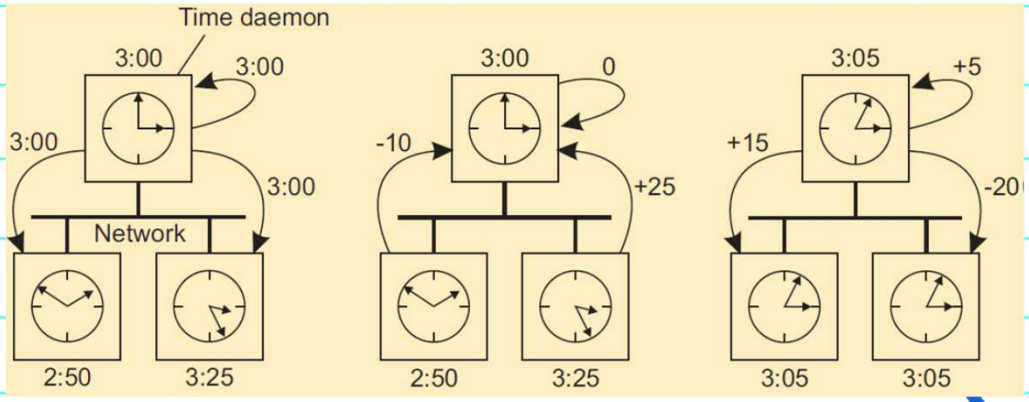
Time Server



Berkeley Algorithm

- by Gusella and Zatti (1989)
- Master polls other computers to send their clock values
- master calculates average time and sends each the necessary adjustment
- Faulty clocks may have significant adverse effect on the result.
- The master takes a fault-tolerant average by selecting only a subset of the clocks that are close enough.

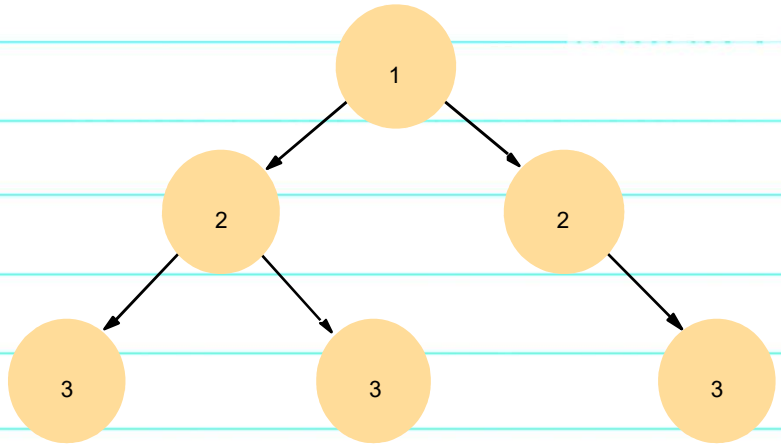
Berkeley Algorithm



Network Time Protocol (NTP)

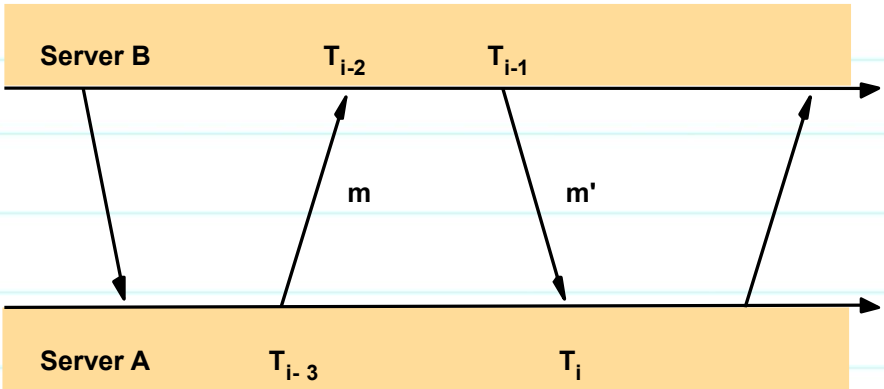
- servers organized into a synchronization subnet whose levels are called strata.
- primary servers → UTC
- secondary servers sync with primary servers
- sync modes with increasing accuracies
  - multicast mode (periodically multicast the time, used in high-speed LAN)
  - procedure-call mode (time server accepts requests from other computers)
  - symmetric mode (a pair of servers exchange timing info)

Synchronization Subnet



Note: Arrows denote synchronization control, numbers denote strata.

NTP Messages Exchange



NTP Clock Synchronization

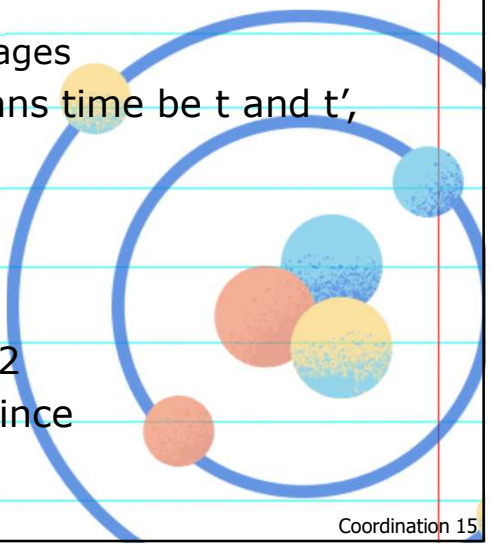
- For each pair of messages, NTP calculates
  - o_i : offset between two servers
  - t_i : delay, i.e. total trans time for two messages
- Let the true offset be o, and the actual trans time be t and t', then

  $T_{i-2} = T_{i-3} + t + o$ $T_i = T_{i-1} + t' - o$
- This leads to

  $d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$

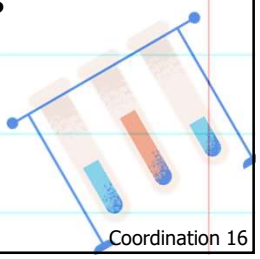
  $o = o_i + (t'-t)/2$ $o_i = (T_{i-2}-T_{i-3}+ T_{i-1}-T_i)/2$
- We can then use $<o_i, d_i>$ for clock sync, since

  $o_i - d_i/2 \le o \le o_i + d_i/2$



Logical Clocks

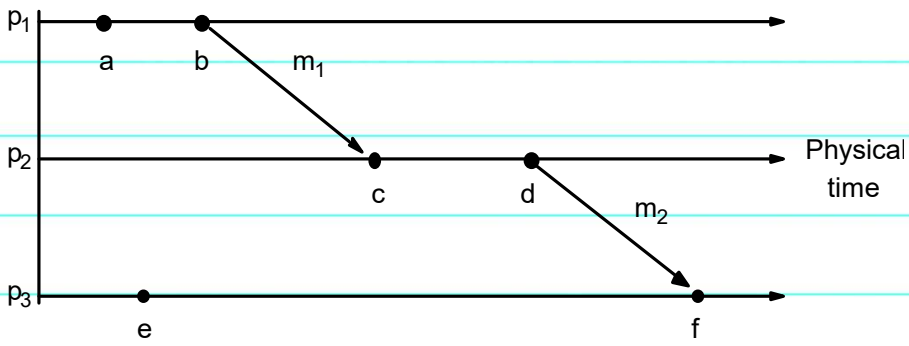
- Lamport(1978)
  - can't sync clocks perfectly
  - can't use physical time to order events
- monotonically increasing software counter
- value does not need to have any particular relationship to any physical clock
- use Lamport's algorithm to synchronize logical clocks
- uses the "happened-before" ordering



Happened-before Ordering (→)

- if *a* and *b* occurred in same process, and *a* occurs before *b*, then $a \rightarrow b$
- if message is sent between processes, then the send occurred before the receive: $send(m) \rightarrow receive(m)$
- transitivity property: $a \rightarrow b$, and $b \rightarrow c$, then $a \rightarrow c$
- Events that are not order by → are concurrent and write as a||e.
- This introduces a partial ordering of events in a system with concurrently operating processes.

Events at three processes



Logical Clocks

- How to maintain a global view that is consistent with the happened-before relation?
- Attach a timestamp *L(e)* to each event *e*, satisfying the following properties:
  1. If *a* and *b* are two events in the same process, and $a \rightarrow b$, then we demand that *L(a)* < *L(b)*.
  2. If *a* corresponds to sending a message *m*, and *b* to the receipt of that message, then also *L(a)* < *L(b)*.
- How to attach a timestamp to an event when there's no global clock ⇒ maintain a consistent set of logical clocks, one per process.

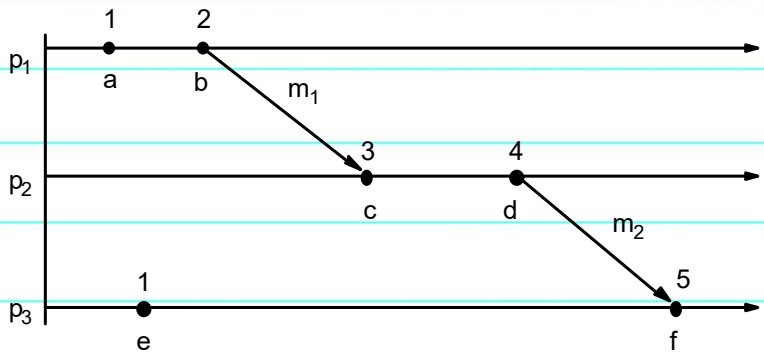
Lamport's Algorithm

- Each process p_i keeps its own logical clock L_i
- Each event *e* is timestamped at process p_i by $L_i(e)$
- Step 1: L_i incremented before each event is issued at p_i (satisfy property P1)
  - $L_i = L_i + 1$
- Step 2: (satisfy property P2)
  a) When p_i sends message *m*, $t = L_i$ is included
  b) On *receive(m, t)*, q_j sets $L_j = max(L_j, t)$; then applies Step 1 before timestamping the event *receive(m)*

Lamport's Algorithm

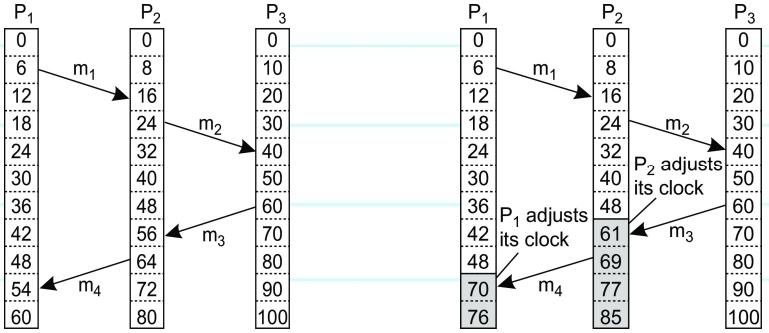
- If *send(m)* $\rightarrow$ *receive(m)*, then $L_{send(m)} < L_{receive(m)}$
- If $a \rightarrow b$, then $L_a < L_b$
- **Problem**: converse NOT true (why?)
- **Totally ordered logical clocks**:
  - define the global logical timestamp of event e at p_i with local timestamp T_i to be (T_i, i)
  - $(T_i, i) < (T_j, j)$ iff $T_i < T_j$, or $T_i = T_j$ and $i < j$.

Lamport Timestamps

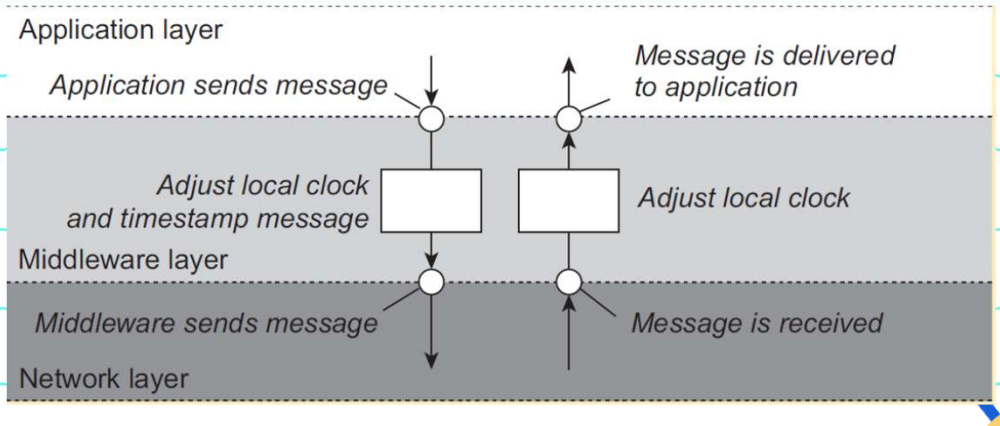


Logical Clocks: Example

- Consider three processes with event counters operating at different rates



Logical Clocks Implementation



Vector Clocks

- Developed by Mattern(1989) and Fidge(1991) to overcome the shortcoming of logical clock.
- Can capture causality.
- A vector clock (for N processes) is an array of N integers. Each process keeps its own clock.
- $V_i[i]$ is the local logical clock at process p_i.
- If $V_i[j] = k$ then p_i knows that k events have occurred at p_j.

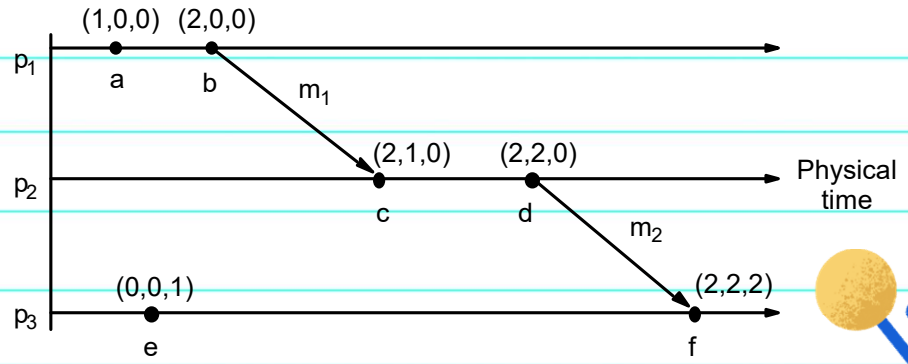
Vector Clocks Operations

- VC1: initially, $V_i[j] = 0$, $\forall j=1, 2,\ldots,N$
- VC2: before process p_i timestamps an event,
  sets $V_i[i] = V_i[i]+1$
- VC3: p_i includes V_i in every message sent
- VC4: when p_i receives a message with t, from p_j
  delay delivery until
  ✓ $t[j] = V_i[j]+1$
  ✓ $t[k] <= V_i[k]$ for $k <> i$
- VC5 (merge): when p_i delivers a message with t,
  sets $V_i[j] = \max(V_i[j], t[j])$

Vector Timestamps

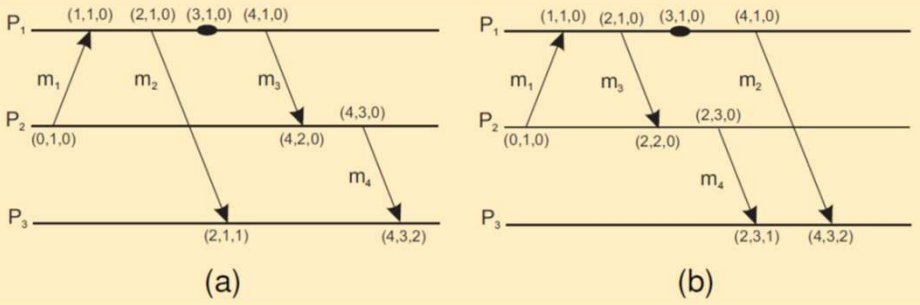
- To compare vector timestamps:
  - $V = V'$ iff $V[j] = V'[j]$, $\forall j=1, 2,\ldots,N$
  - $V \leq V'$ iff $V[j] \leq V'[j]$, $\forall j=1, 2,\ldots,N$
  - $V < V'$ iff $V \leq V' \wedge V \neq V'$
- We can show that
  - $e \rightarrow e' \Rightarrow V(e) < V(e')$ (the same as Lamport's logical clock), AND
  - $V(e) < V(e') \Rightarrow e \rightarrow e'$ (overcome the problem of logical clock)
- Exercise: Prove the claim above.
- Any disadvantage of vector clocks compared with logical clock?

Vector Timestamps



Vector Clocks: More Examples

Capturing potential causality when exchanging messages

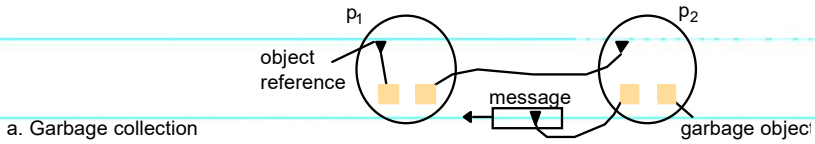


Situation	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	Conclusion
(a)	(2, 1, 0)	(4, 3, 0)	Yes	No	m_2 may causally precede m_4
(b)	(4, 1, 0)	(2, 3, 0)	No	No	m_2 and m_4 may conflict

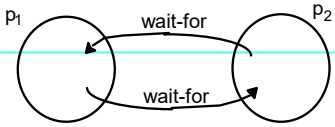
Global States

- In many cases, we need to have a good observation of the current global state of a distributed system.
  - distributed garbage collection
  - distributed deadlock detection
  - distributed termination detection
  - distributed debugging
  - . . .
- We call the global state of a distributed system in a particular moment the snapshot of the system.

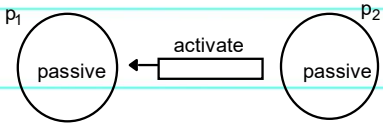
Detecting Global Properties



a. Garbage collection



b. Deadlock



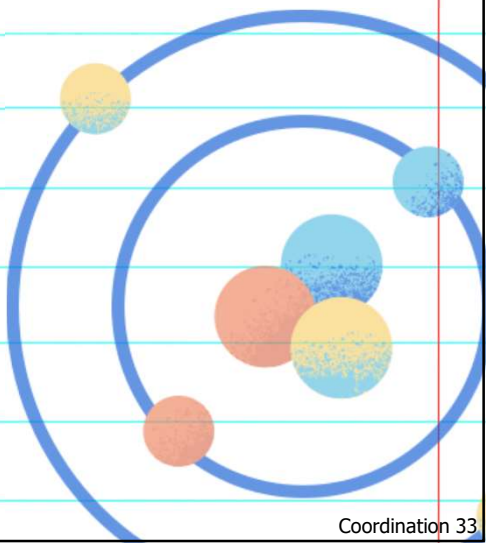
c. Termination

Determine Global States

- It is quite easy to observe and record the succession of states of a single process.
- It is not that easy to determine the global state of a distributed system.
- The essential problem is the absence of global time.
- We therefore want to assemble a meaningful global state from local states of different processes.

Define Global States

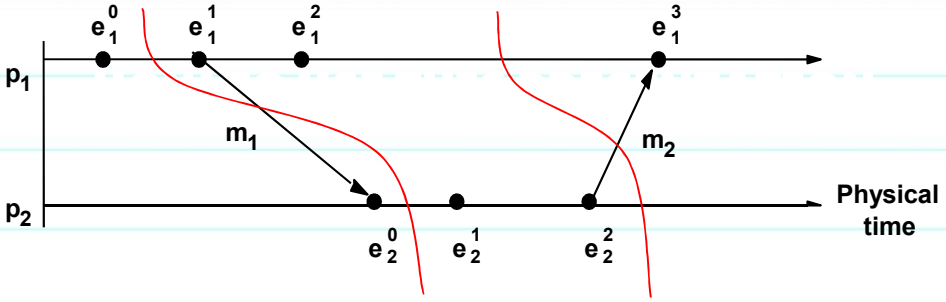
- A system P of N processes p_i (i = 1, 2, …, N)
- Local history and finite prefix
  history(p_i)= h_i = <e_i^0, e_i^1, e_i^2, …>
  h_i^k = <e_i^0, e_i^1, … , e_i^k>
  s_i^k :the state of process p_i immediately before the event e_i^k occurs
- Global history
  $H = h_1 \cup h_2 \cup \ldots \cup h_N$
- Global state
  $S = (s_1, s_2, \ldots, s_N)$



Consistent Global States

- A global state corresponds to initial prefixes of h_i (i = 1, 2, …, N)
- A cut is a union of prefixes of h_i :
  $C = h_1^{c1} \cup h_2^{c2} \cup \ldots \cup h_N^{cN}$
- The frontier of the cut
  $\{ e_i^{ci} : i = 1, 2, \ldots, N \}$
- Consistent cut C
  $\forall e \in C, f \rightarrow e \Rightarrow f \in C$ ($\rightarrow$:happened-before)(what does it mean?)
- A consistent global state is a global state that corresponds to a consistent cut.

Cuts



Inconsistent cut

Consistent cut

State Transitions and Runs

- The execution of a distributed system can be considered as a series of transitions between global states:

  $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$
- A run is a total ordering of events in a global history that is consistent with each local ordering $\rightarrow_i$ (i = 1, 2, …, N)
- A linearization (consistent run) is a run that is consistent with the ordering $\rightarrow$ on H.
- S′ reachable from S if there is a linearization from S to S′.

Global State Predicates

- A global state predicate P: $S \rightarrow$ {True, False}
- A stable predicate P: once in a state S in which P is True, it remains True in all future states reachable from S. Otherwise, P is non-stable.
- Safety w.r.t an undesirable property α : α is False for all states S reachable from S_0(the initial state).
- Liveness w.r.t a desirable property β : for any linearization L started from S_0, β is True for some state S_L reachable from S_0.

Distributed Snapshot Algorithm

- Assume each process communicates with another process using unidirectional point-to-point channels (e.g, TCP connections)
- Any process can initiate the algorithm
  - Checkpoint local state
  - Send marker on every outgoing channel
- On receiving a marker
  - Checkpoint state if first marker and send marker on outgoing channels, save messages on all other channels until:
  - Subsequent marker on a channel: stop saving state for that channel

Distributed Snapshot

- A process finishes when
  - It receives a marker on each incoming channel and processes them all
  - State: local state plus state of all channels
  - Send state to initiator
- Any process can initiate snapshot
  - Multiple snapshots may be in progress. Each is separate, and each is distinguished by tagging the marker with the initiator ID (and sequence number)

Chandy and Lamport's Algorithm

Marker receiving rule *for process* p_i

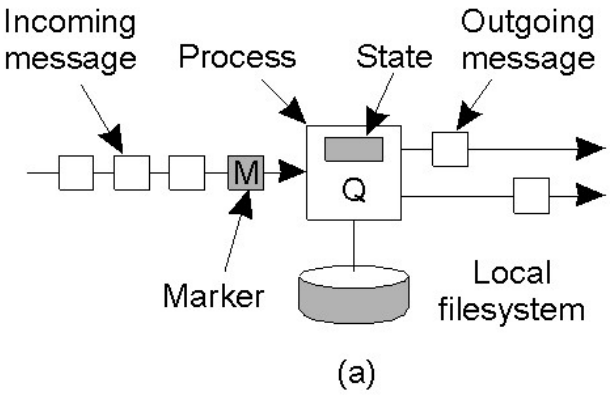
On p_i's receipt of a *marker* message over channel c:
  if (p_i has not yet recorded its state) it
    records its process state now;
    records the state of c as the empty set;
    turns on recording of messages arriving over other incoming channels;
  else
    p_i records the state of c as the set of messages it has received over c since it saved its state.
  end if

Marker sending rule *for process* p_i

After p_i has recorded its state, for each outgoing channel c:
  p_i sends one marker message over c (before it sends any other message over c).

Distributed Snapshot Algorithm

a) Organization of a process and channels for a distributed snapshot

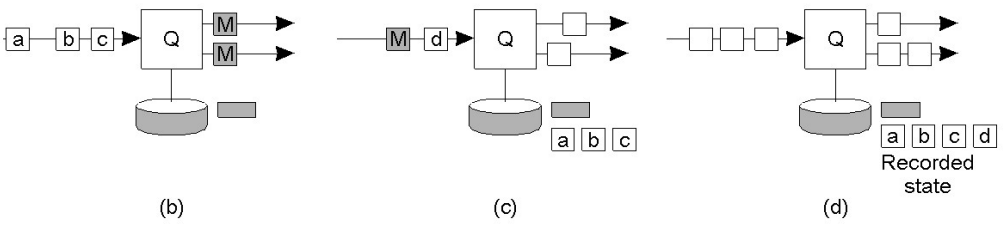


Distributed Snapshot

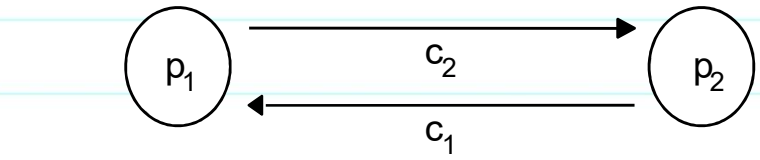
b) Process Q receives a marker for the first time and records its local state

c) Q records all incoming message

d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel



Processes and Initial States



$1000	(none)	$50	2000
account	widgets	account	widgets

Processes and Global States

1. Global state S_0: <$1000, 0> p_1 — c_2 (empty) → p_2 <$50, 2000>; c_1 (empty)

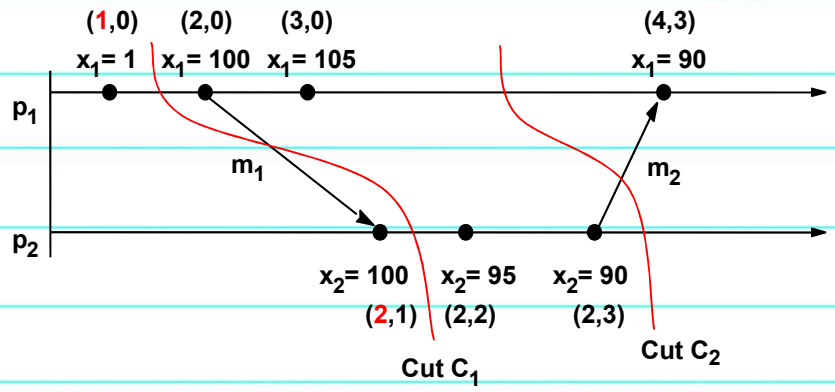
2. Global state S_1: <$900, 0> p_1 — c_2 (Order 10, $100), M → p_2 <$50, 2000>; c_1 (empty)

3. Global state S_2: <$900, 0> p_1 — c_2 (Order 10, $100), M → p_2 <$50, 1995>; c_1 (five widgets)

4. Global state S_3: <$900, 5> p_1 — c_2 (Order 10, $100) → p_2 <$50, 1995>; c_1 (empty)

(M = marker message)

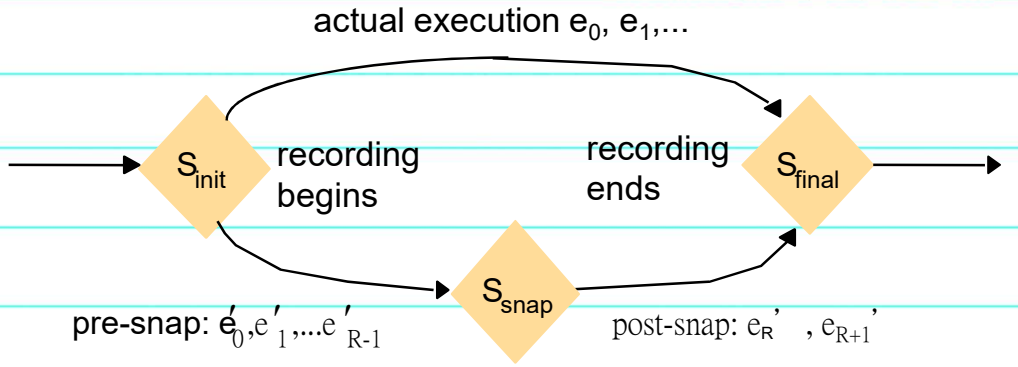
Timestamps and Variable Values



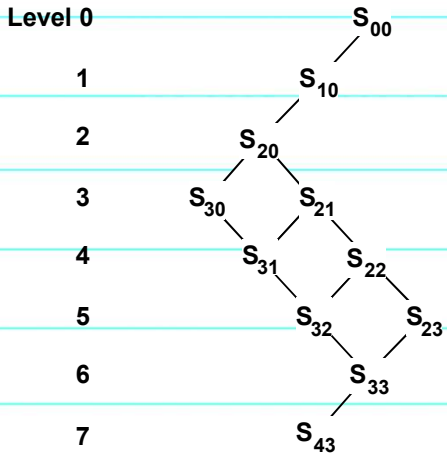
Termination Detection

- Detecting the end of a distributed computation
- Notation: let sender be *predecessor*, receiver be *successor*
- Two types of markers: Done and Continue
- After finishing its part of the snapshot, process *Q* sends a Done or a Continue to its predecessor
- Send a Done only when
  - All of *Q*'s successors send a Done
  - *Q* has not received any message since it check-pointed its local state and received a marker on all incoming channels
  - Else send a Continue
- Computation has terminated if the initiator receives Done messages from everyone

Reachability between States



The lattice of global states



S_{ij}= global state after i events at process 1 and j events at process 2

Global Predicates: *Possibly* ϕ and *Definitely* ϕ

1. *Evaluating possibly* ϕ *for global history* H *of* N *processes*

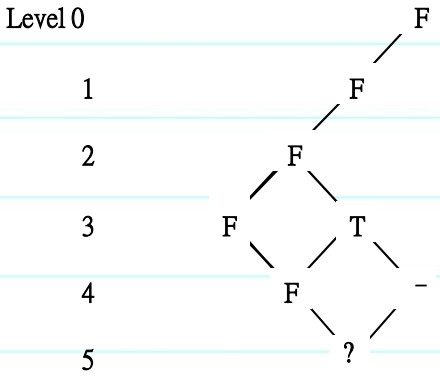
$L := 0$;
$States := \{ (s_1^0, s_2^0, \ldots, s_N^0) \}$;
while $(\phi(S) = False$ for all $S \in$ States$)$
    $L := L + 1$;
    $Reachable := \{S' : S'$ reachable in H from some $S \in$ States $\wedge$ level$(S') = L\}$;
    $States := Reachable$
end while
output "*possibly* ϕ";

Global Predicates: *Possibly* ϕ and *Definitely* ϕ

2. *Evaluating definitely* ϕ *for global history* H *of* N *processes*

$L := 0$;
if $(\phi(s_1^0, s_2^0, \ldots, s_N^0))$ *then* $States := \{\}$ *else* $States := \{ (s_1^0, s_2^0, \ldots, s_N^0) \}$;
while $(States \neq \{\})$
    $L := L + 1$;
    $Reachable := \{S' : S'$ reachable in H from some $S \in$ States $\wedge$ level$(S') = L\}$;
    $States := \{S \in Reachable : \phi(S) = False\}$
end while
output "*definitely* ϕ";

Evaluating *definitely* ϕ



F = (ϕ(S) = False); T = (ϕ(S) = True)

Mutual Exclusion

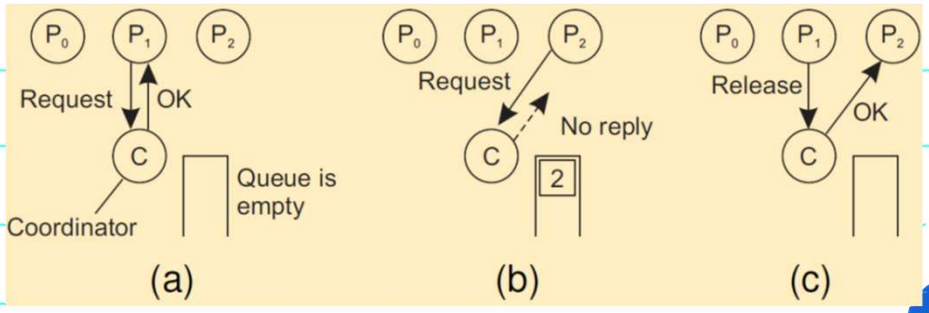
- **Problem**: A number of processes in a distributed system want exclusive access to some resource.
- **Basic solutions**:
  - Permission-based: A process wanting to enter its critical section, or access a resource, needs permission from other processes.
  - Token-based: A token is passed between processes. The one who has the token may proceed in its critical section, or pass it on when not interested.

Permission-based (centralized algorithm)

a) Process P1 asks the coordinator for permission to access a shared resource. Permission is granted.

b) Process P2 then asks permission to access the same resource. The coordinator does not reply.

c) When P1 releases the resource, it tells the coordinator, which then replies to P2.

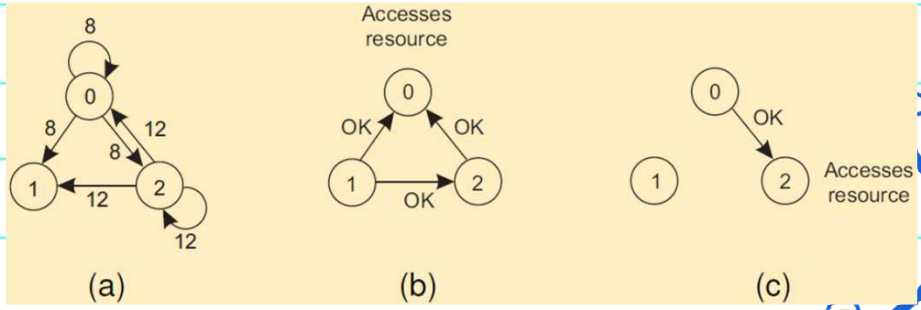


Mutual Exclusion (Ricart & Agrawala)

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

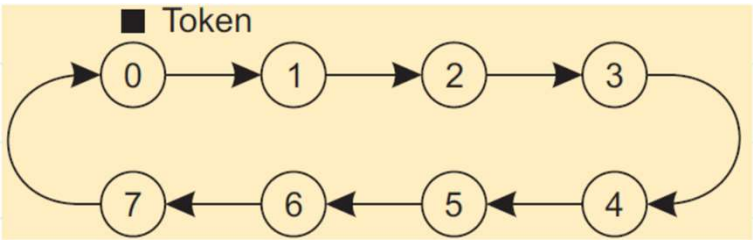
b) P0 has the lowest timestamp, so it wins.

c) When process P0 is done, it sends an OK also, so P2 can now go ahead.



Mutual Exclusion: Token ring algorithm

- Organize processes in a logical ring, and let a token be passed between them. The one that holds the token is allowed to enter the critical region (if it wants to).



Decentralized Mutual Exclusion

- Assume every resource is replicated N times, with each replica having 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.

Decentralized Mutual Exclusion (Robustness)

- Let $p = \triangle t / T$ be the probability that a coordinator resets during a time interval t, while having a lifetime of T.
- The probability $P[k]$ that k out of m coordinators reset during the same interval is

$$\mathbb{P}[k] = \binom{m}{k} p^k (1-p)^{m-k}$$

- f coordinators reset $\Rightarrow$ correctness is violated when there is only a minority of nonfaulty coordinators: when $m - f \leq N / 2$, or, $f \geq m - N / 2$.
- The probability of a violation is $\sum_{k=m-N/2}^{N} \mathbb{P}[k]$

Violation Probabilities

- So, what can we conclude?

N	m	p	Violation
8	5	3 sec/hour	$< 10^{-15}$
8	6	3 sec/hour	$< 10^{-18}$
16	9	3 sec/hour	$< 10^{-27}$
16	12	3 sec/hour	$< 10^{-36}$
32	17	3 sec/hour	$< 10^{-52}$
32	24	3 sec/hour	$< 10^{-73}$

N	m	p	Violation
8	5	30 sec/hour	$< 10^{-10}$
8	6	30 sec/hour	$< 10^{-11}$
16	9	30 sec/hour	$< 10^{-18}$
16	12	30 sec/hour	$< 10^{-24}$
32	17	30 sec/hour	$< 10^{-35}$
32	24	30 sec/hour	$< 10^{-49}$

Mutual Exclusion: Comparison

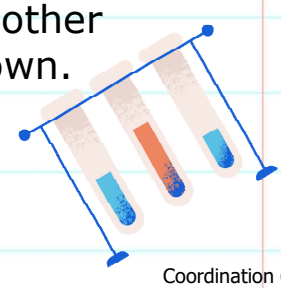
Algorithm	Messages per entry/exit	Delay before entry (in message times)
Centralized	3	2
Distributed	$2 \cdot (N-1)$	$2 \cdot (N-1)$
Token ring	$1, \ldots, \infty$	$0, \ldots, N-1$
Decentralized	$2 \cdot m \cdot k + m, k = 1, 2, \ldots$	$2 \cdot m \cdot k$

Election Algorithms

- 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 by hand (e.g. file servers). This leads to centralized solutions) single point of failure.
- If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?
- Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?

Election Algorithms

- Many distributed algorithms require one process (any one) to be the coordinator.
- Election algorithms try to locate the process with the highest process number to be the coordinator.
- The goal is to ensure that all processes agreeing on who the new coordinator is.
- Each process knows the process number of every other process but doesn't know which ones are up or down.



Election Requirements

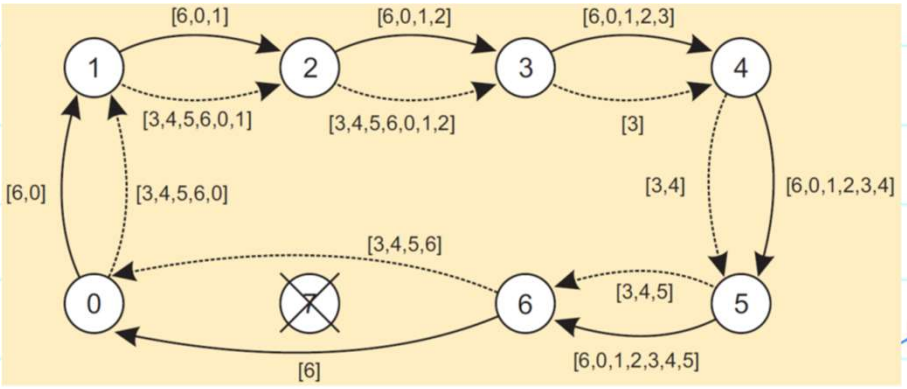
- Any process can call an election but does not call more than one election at a time.
- Concurrent elections are allowed.
- Elected process must be unique with the highest id, even if
  - several processes call elections concurrently
  - processes fail during the election
- A participant process is either undecided or concludes with the highest id process at the end (**safety**)
- All processes eventually agree (**liveness**)

Election in a Ring

- Process priority is obtained by organizing processes into a (logical) ring. 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.
- 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.

A Ring-based Election

- The solid line shows the election messages initiated by P6
- The dashed one the messages by P3



Bully(土匪) Algorithm

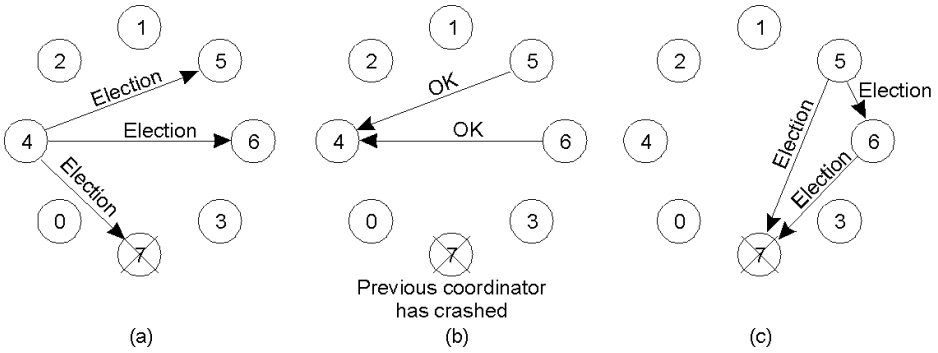
- Each process has a unique numerical ID
- Processes know the IDs and address of every other process
- Communication is assumed reliable
- ***Key Idea***: select process with highest ID
- Process initiates election if it just recovered from failure or if coordinator failed
- 3 message types: *Election*, *OK*, *I won*
- Several processes can initiate an election simultaneously
  - Need consistent result
- $O(n^2)$ messages required with n processes

Bully Algorithm Details

- Any process *P* can initiate an election
- *P* sends *Election* messages to all process with higher IDs and awaits *OK* messages
- If no *OK* messages, *P* becomes coordinator and sends *I won* messages to all process with lower IDs
- If it receives an *OK*, it drops out and waits for an *I won*
- If a process receives an *Election* msg, it returns an *OK* and starts an election
- If a process receives a *I won*, it treats sender an coordinator

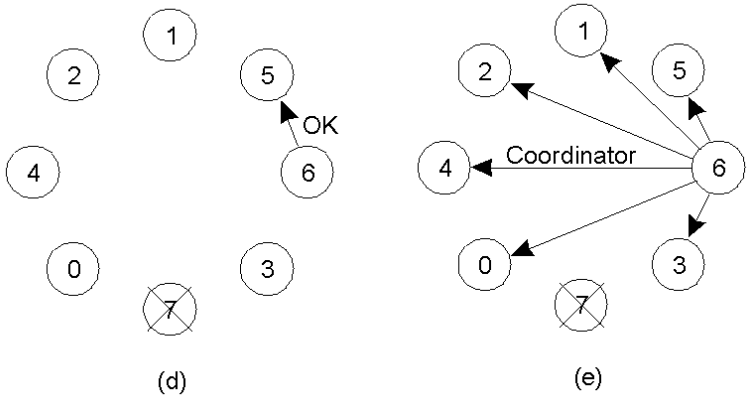
Bully Algorithm (1)

a) Process 4 holds an election by sending ELECTION messages to all processes with higher numbers
b) Process 5 and 6 respond, telling 4 to stop
c) Now 5 and 6 each hold an election

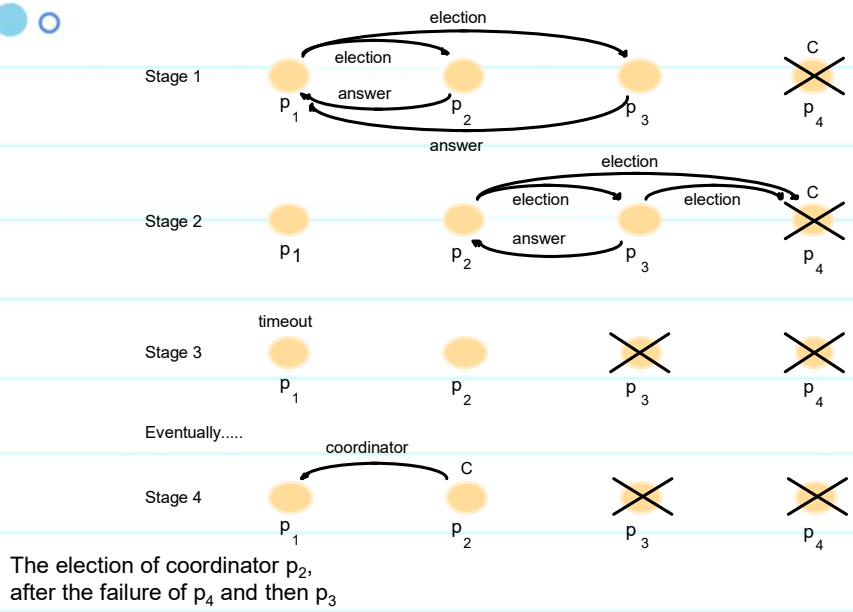


Bully Algorithm (2)

d) Process 6 tells 5 to stop
e) Process 6 wins and tells everyone



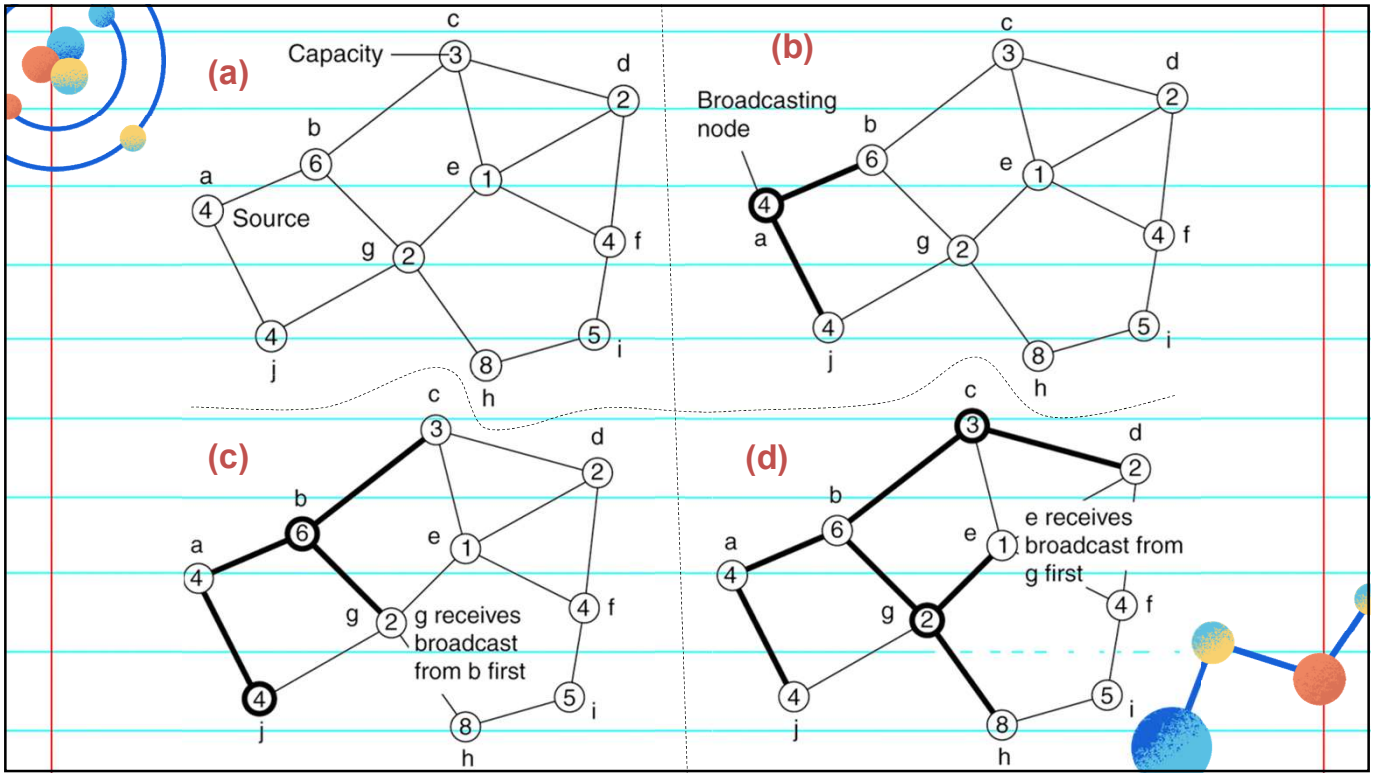
Bully Algorithm (3)



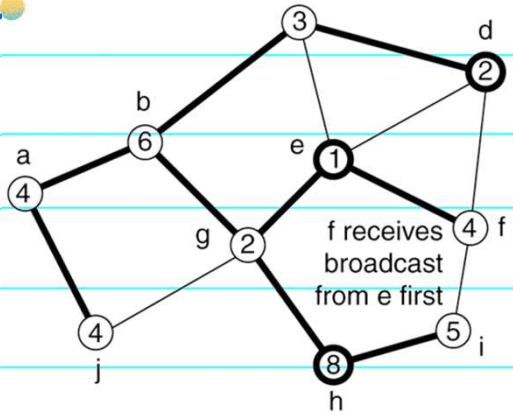
The election of coordinator p_2, after the failure of p_4 and then p_3

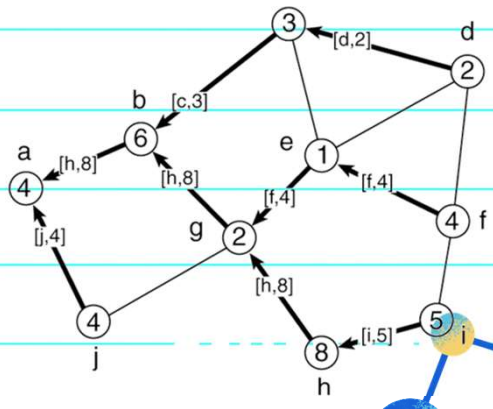
Wireless Environments

- Unreliable, and nodes may move.
- Algorithm:
  1. Any node starts by sending out an ELECTION message to neighbors.
  2. When a node receives an ELECTION message for the first time, it forwards to neighbors, and designates the sender as its parent.
  3. It then waits for responses from its neighbors.
     - Responses may carry resource information.
  4. When a node receives an ELECTION message for the second time, it just OKs it.



The build-tree phase.





Reporting of best node to source.

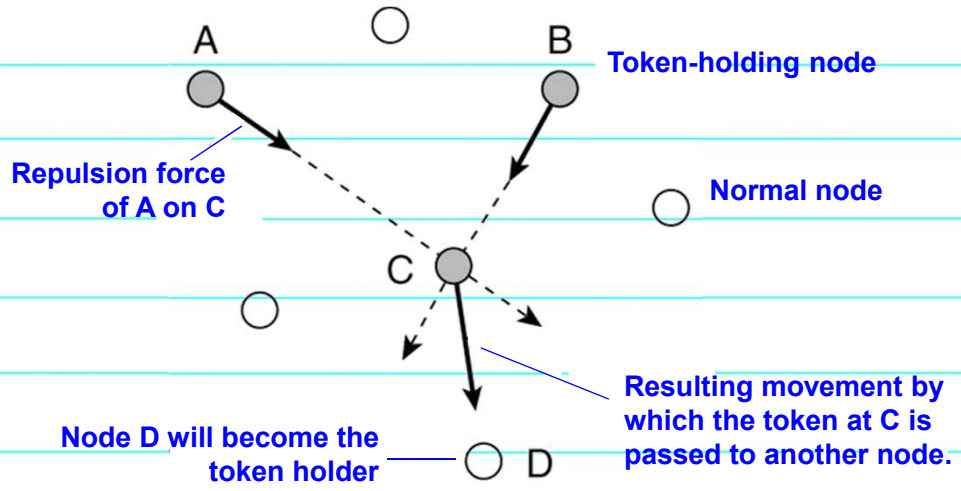
Elections in Large-Scale Systems (1)

- In large-scale systems, sometimes need to select more than one.
  - For example, sometimes need to select multiple superpeers.
- Requirements for superpeer selection:
  1. Normal nodes should have low-latency access to superpeers.
  2. Superpeers should be evenly distributed across the overlay network.
  3. There should be a predefined portion of superpeers relative to the total number of nodes in the overlay network.
  4. Each superpeer should not need to serve more than a fixed number of normal nodes.

Elections in Large-Scale Systems (2)

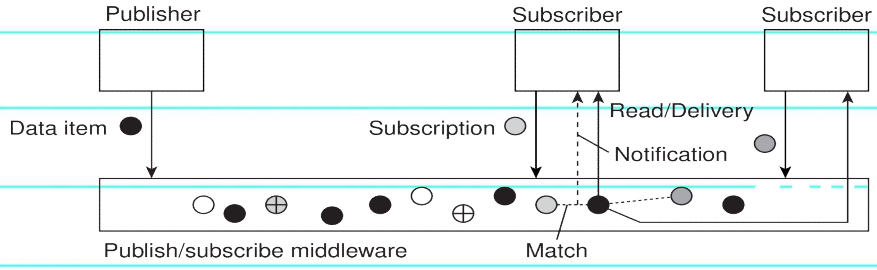
- One approach, if a DHT is being used:
  - Reserve part of the ID space to identify superpeers.
  - For example, save the top k bits, and let superpeers have zeros everwhere else.
  - A node routes a message to the superpeer by sending to p AND 11100000, for key p, for example.
    - What if this node doesn't exist?
    - How can a node know if it is a superpeer?
- Another approach, if a geometric overlay is being used.
  - Assume N tokens are distributed among N nodes.
  - Assume the tokens repel each other. This will cause tokens to move away from each other.
  - When a token is held for given, specified length of time, it will promote itself to superpeer.

- Nodes can learn about other nodes through gossiping.
- If a node discovers other nodes are nearby, it will move the node (and relinquish the superpeer status).



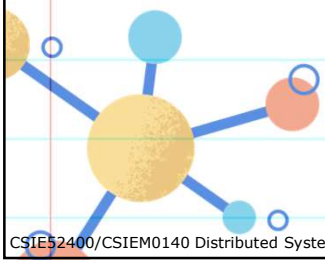
Distributed Event Matching

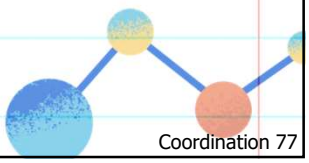
- Event matching (notification filtering) is at the heart of **publish-subscribe** systems.
- A process specifies a subscription S for events of interest.
- When a process publishes a notification N for an event, the system determines if S matches N.
- If matched, send N (and associated data) to the subscriber.



Event Matching Requirements

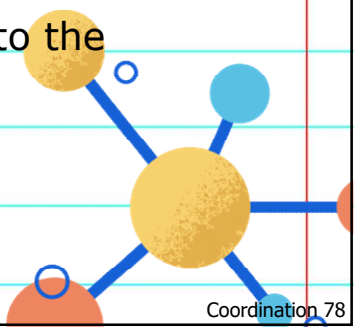
- Matching subscriptions against events.
- Notifying subscribers in case of a match.
- Assume the existence of a function match(S, N) which returns true when subscription S matches the notification N, and false otherwise.





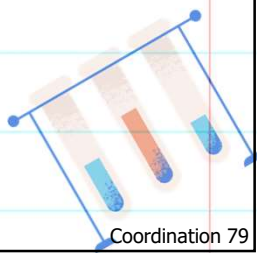
Centralized Event Matching

- A centralized server that handles all subscriptions and notifications.
- A subscriber submits a subscription, which is subsequently stored.
- When a publisher submits a notification, it is checked against every subscription.
- When a match is found, the notification is sent to the associated aubscriber.
- Not scalable but still feasible for many cases.



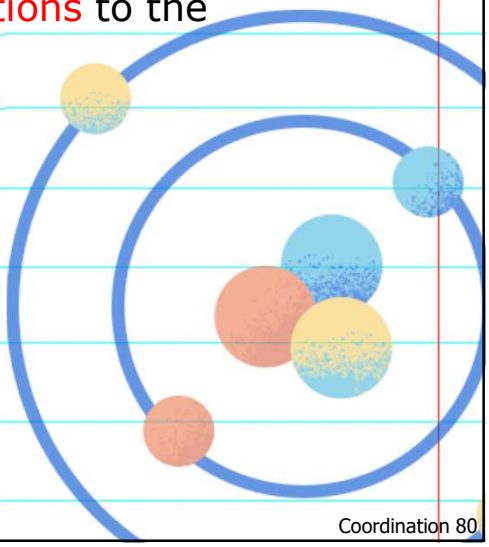
Multiple Servers Matching

- A simple scale up of the centralized approach.
- Deterministically divide the work across multiple servers.
- A function *sub2node*(S) which maps S to a nonempty subset (the rendezvous nodes for S) of servers.
- A function *not2node*(N) which maps N to a nonempty subset (the rendezvous nodes for N) of servers.
- For any S and matching N, make sure that *sub2node*(S) ∩ *not2node*(N) ≠ ∅



Notification Routing

- The servers (brokers) are organized into an overlay network.
- The issue becomes how to route notifications to the appropriate set of subscribers.
- Three classes of methods:
  - Flooding
  - Selective routing
  - Gossip-based dissemination



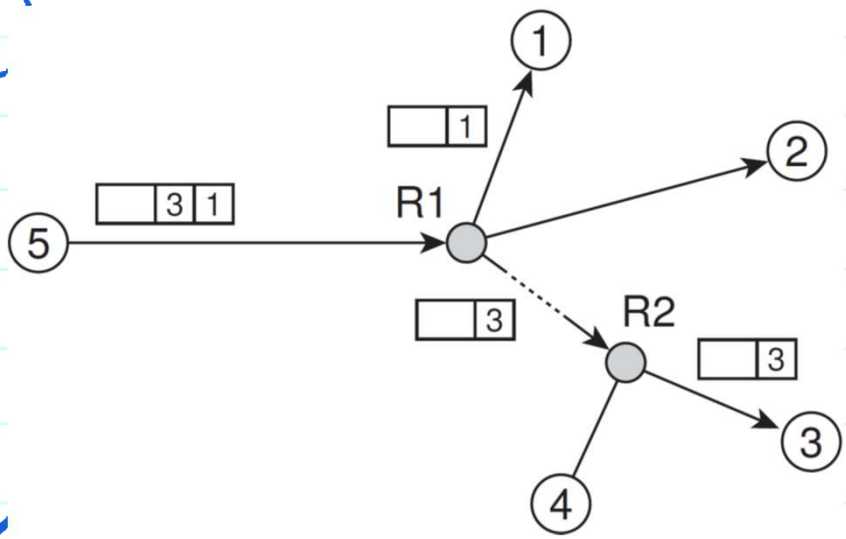
Notification Flooding

- Use broadcasting to make sure that notifications reach their subscribers.
- Two approaches:
  - Store each **subscription** at every broker while publishing notifications only a single broker. The later identifies the matching subscriptions and copy/forward the notification.
  - Store a **subscription** only at one broker while broadcasting notifications to all brokers. Matching is distributed across the brokers.

Selective Routing

- Brokers take routing decisions by considering the content of a notification.
- Each notification carries enough info to cut-off routes that do not lead to its subscribers.
- A naïve content-based routing:
  - Brokers are organized into a broadcast tree.
  - Every broker broadcasts its subscriptions to all other brokers to compile a list of (subject, destination) pairs.
  - A notification message N is prepended with the destination brokers.
  - The router uses the list to decide on the paths the message should follow. (next slide)

Content-based Routing



Gossip-based Routing

- Subscribers interested in the same notifications form their own overlay network (constructed through gossiping).
- Once a notification is published, it merely needs to be routed to the appropriate overlay.
- Subscriber overlay can also be built based on topics.
- The simplest overlay is a ring with shortcuts to facilitate efficient notification dissemination.

Gossip-based Coordination

- Gossiping is useful in many areas: aggregation, large-scale peer sampling, overlay construction,…
- An example in aggregation:
  - Every node Pi initially chooses an arbitrary number vi
  - When Pi contacts node Pj, they update their value:

    $v_i, v_j \leftarrow (v_i + v_j)/2$
  - Eventually all nodes will have the same value, the average of all initial values. (Why?)
  - Propagation speed is exponential. (Why?)
  - What if v1 = 1 and vj = 0, for all other j ?

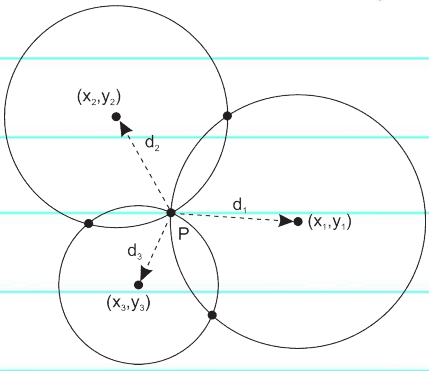
Positioning Nodes

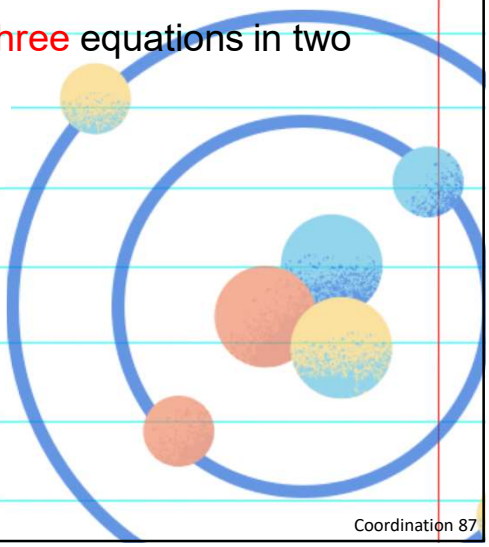
- Issue: In large-scale distributed systems in which nodes are dispersed across a wide-area network, we often need to take some notion of proximity or distance into account ⇒ it starts with determining a (relative) location of a node.

Computing Position

- Observation: A node P needs $d + 1$ landmarks to compute its own position in a d-dimensional space. Consider two-dimensional case.
- Computing a position in 2D: P needs to solve three equations in two unknowns (x_P, y_P):

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





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

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: $d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$

- Observation: 4 satellites $\Rightarrow$ 4 equations in 4 unknowns (with Δ_r as one of them)