HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Chapter 3: Distributed Systems: Synchronization

Fall 2012

Sini Ruohomaa

Slides joint work with Jussi Kangasharju et al.

Chapter Outline

- Clocks and time
- Global state
- Mutual exclusion
- Election algorithms

Time and Clocks

What we need?	How to solve?
Real time (17:30:21)	Universal time (Network time)
Interval length (3 ms)	Computer clock
Order of events (1.,2.)	Network time (Universal time)

NOTE: ***Time*** is ***monotonous***

Measuring Time

- Traditionally time measured astronomically
  - Transit of the sun (highest point in the sky)
  - Solar day and solar second
- Problem: Earth's rotation is slowing down
  - Days get longer and longer – time units change..?
  - 300 million years ago there were 400 days in the year ;-)
- Modern way to measure time is the atomic clock
  - Based on transitions in Cesium-133 atom
  - Still need to correct for Earth's rotation
- Result: Universal Coordinated Time (UTC)
  - UTC available via radio signal, telephone line, satellite (GPS)

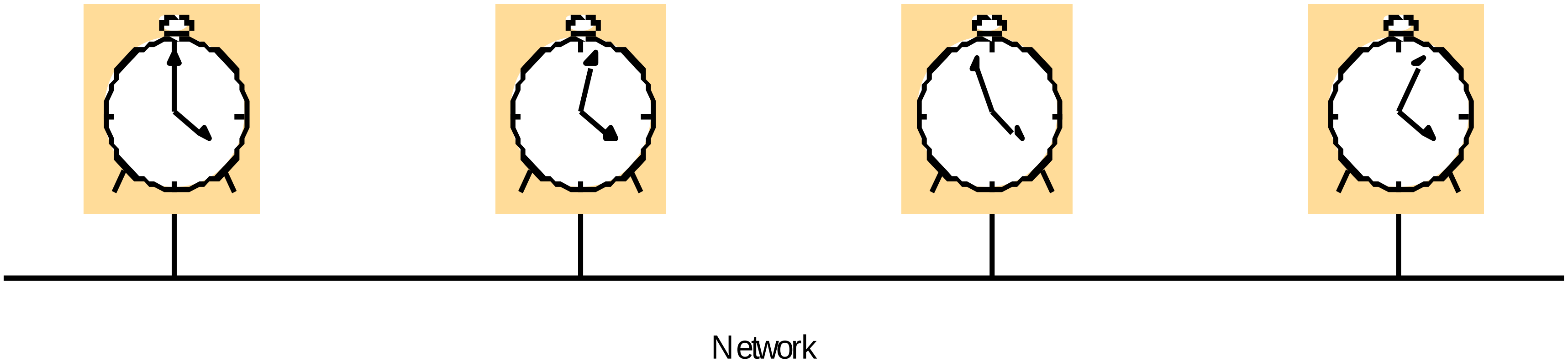
Hardware/Software Clocks

- Physical clocks in computers are realized as crystal oscillation counters at the hardware level
  - Correspond to counter register $H(t)$
  - Used to generate interrupts
- Usually scaled to approximate physical time t, yielding software clock $C(t)$, $C(t) = \alpha H(t) + \beta$
  - $C(t)$ measures time relative to some reference event, e.g., 64 bit counter for # of nanoseconds since last boot
  - Simplification: $C(t)$ carries an approximation of real time
  - Ideally, $C(t) = t$ (never 100% achieved)
  - Note: Values given by two consecutive clock queries will differ only if clock resolution is sufficiently smaller than processor cycle time

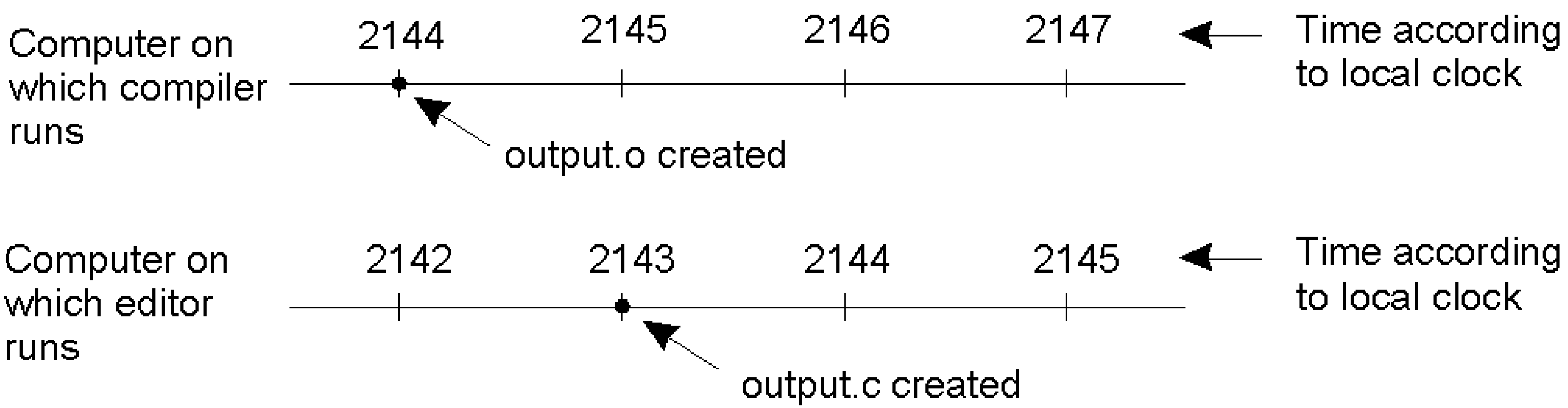
Problems with Hardware/Software Clocks

- Skew: Disagreement in the reading of two clocks
- Drift: Difference in the rate at which two clocks count the time
  - Due to physical differences in crystals, plus heat, humidity, voltage, etc.
  - Accumulated drift can lead to significant skew
- Clock drift rate: Difference in precision between a prefect reference clock and a physical clock,
  - Usually, 10^{-6} sec/sec, 10^{-7} to 10^{-8} for high precision clocks

Skew between computer clocks in a distributed system

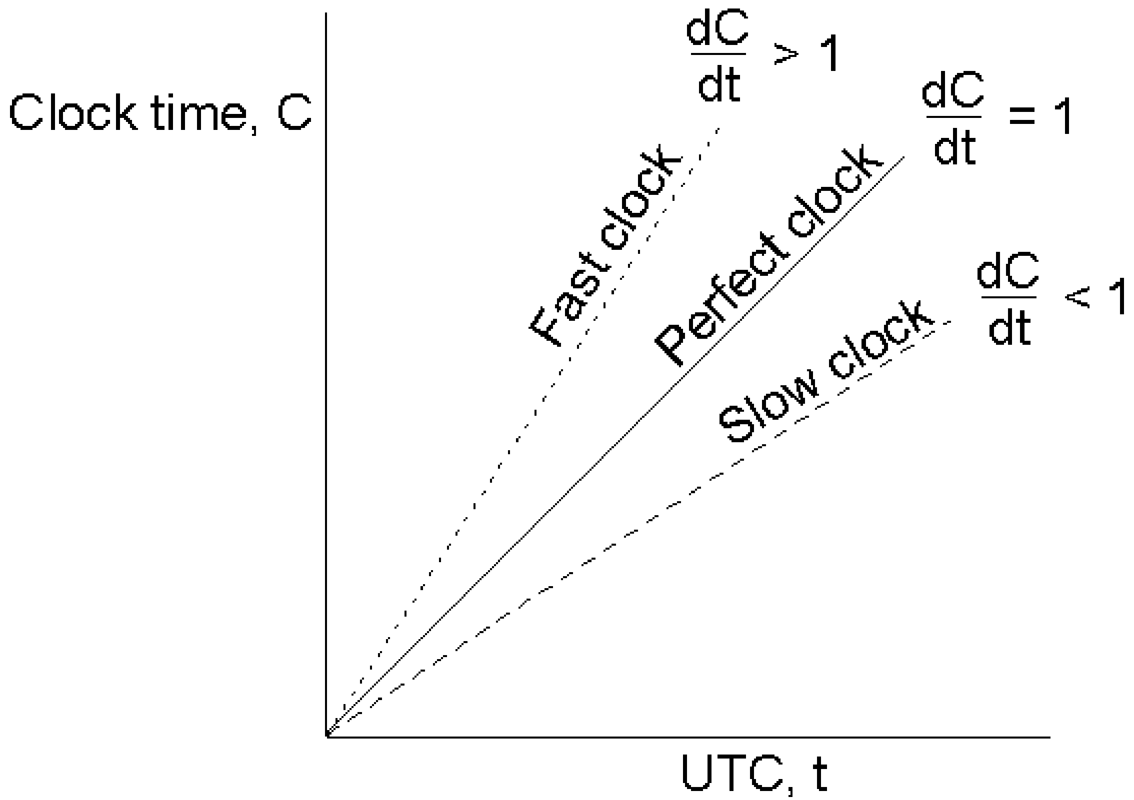


Clock Synchronization



When each machine has its own clock, an event that occurred after another event may end up with an earlier timestamp.

Clock Synchronization Problem



At drift rate of 10^{-6}

1 ms ~ 17 min

1 s ~ 11.6 days

UTC: coordinated universal time

accuracy:

radio 0.1 – 10 ms,

GPS 1 us

The relation between clock time and UTC when clocks tick at different rates.

Synchronizing Clocks

- External synchronization
  - Synchronize process's clock with an authoritative external reference clock S(t) by limiting skew to a delay bound D > 0
    - $|S(t) - C_i(t)| < D$ for all t
  - For example, synchronization with a UTC source
- Internal synchronization
  - Synchronize the local clocks within a distributed system to disagree by not more than a delay bound D > 0, without necessarily achieving external synchronization
    - $|C_i(t) - C_j(t)| < D$ for all i, j, t
- Corollary:
  - For a system with external synchronization bound of D, the internal synchronization is bounded by 2D

Clock Correctness

- When is a clock correct?

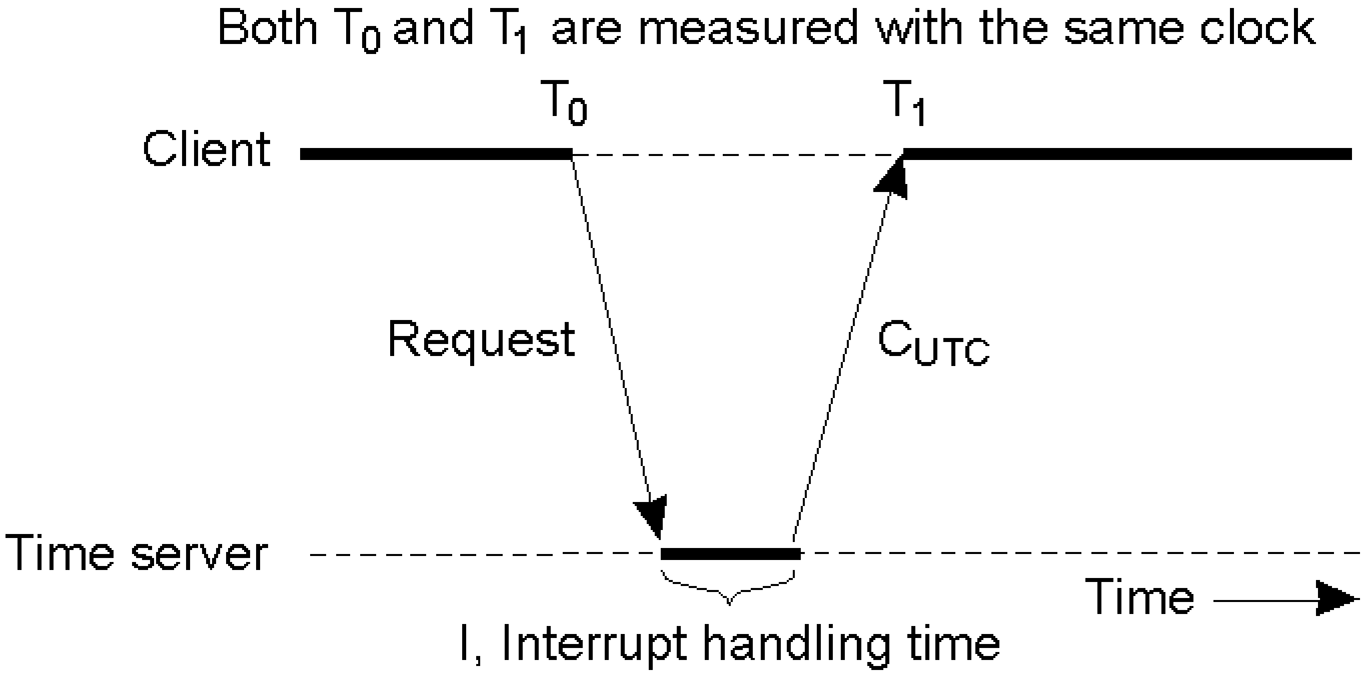
1. If drift rate falls within a bound $r > 0$, then for any t and t' with $t' > t$ the following error bound in measuring t and t' holds:
   - $(1-r)(t'-t) \leq H(t') - H(t) \leq (1+r)(t'-t)$
   - Consequence: No jumps in hardware clocks allowed
2. Sometimes monotonically increasing clock is enough:
   - $t' > t \Rightarrow C(t') > C(t)$
3. Frequently used condition:
   - Monotonically increasing
   - Drift rate bounded between synchronization points
   - Clock may jump ahead at synchronization points

Synchronization of Clocks: Software-Based Solutions

- Techniques:
  - time stamps of real-time clocks
  - message passing
  - round-trip time (local measurement)
- Cristian's algorithm – ask centralized clock
- Berkeley algorithm – synchronized within a group
- NTP: Network time protocol (Internet)

Christian's Algorithm (1/3)

- Observations
  - Round trip times between processes are often reasonably short in practice, yet theoretically unbounded
  - Practical estimate possible if round-trip times are sufficiently short in comparison to required accuracy
- Principle
  - Use UTC-synchronized time server S
  - Process P sends requests to S
  - Measures round-trip time T_{round}
    - In LAN, T_{round} should be around 1-10 ms
    - During this time, a clock with a 10^{-6} sec/sec drift rate varies by at most 10^{-8} sec
    - Hence the estimate of T_{round} is reasonably accurate
  - Naive estimate: Set clock to $t + ½T_{round}$



Current time from a time server: UTC from radio/satellite etc
Problems:
- time must never run backward
- variable delays in message passing / delivery

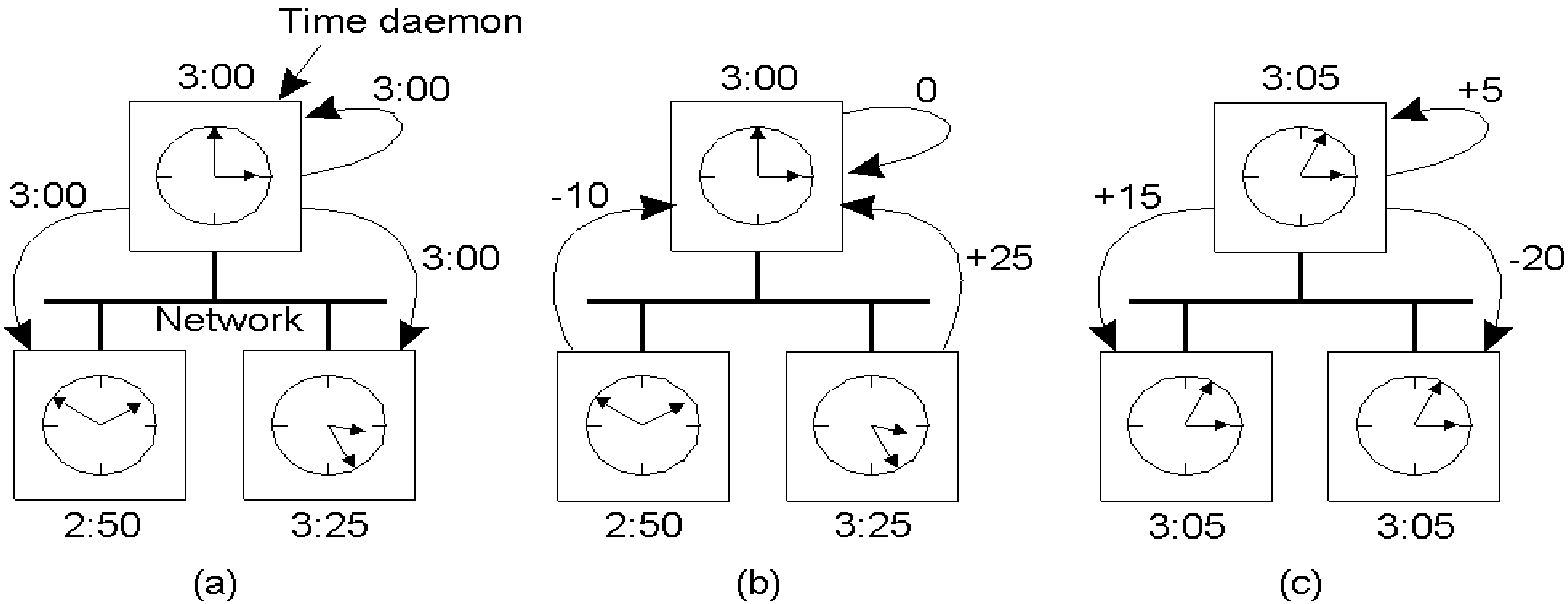
Christian's Algorithm: Analysis (3/3)

- Accuracy of estimate?
- Assumptions:
  - requests and replies via same net
  - *min* delay is either known or can be estimated conservatively
- Calculation:
  - Earliest time that S can have sent reply: $t_0 + min$
  - Latest time that S can have sent reply: $t_0 + T_{round} – min$
  - Total time range for answer: $T_{round} - 2 * min$
  - Accuracy is $\pm (\frac{1}{2}T_{round} - min)$
- Discussion
  - Really only suitable for LAN environment or Intranet
  - Problem of failure of S

Alternative Algorithm: Berkeley algorithm (1/2)

- Berkeley algorithm (Gusella&Zatti ‘89)
  - No external synchronization, but one master server
  - Master polls slaves periodically about their clock readings
  - Estimate of local clock times using round trip estimation
  - Averages the values obtained from a group of processes
    - Cancels out individual clock’s tendencies to run fast
  - Tells slave processes by which amount of time to adjust local clock
  - Master failure: Master election algorithm (see later)
- Experiment
  - 15 computers, local drift rate < 2×10^{-5}, max round-trip 10 ms
  - Clocks were synchronized to within 20-25 ms
- Note: Neither algorithm is really suitable for Internet

The Berkeley Algorithm (2/2)



a) The **time daemon asks** all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clock

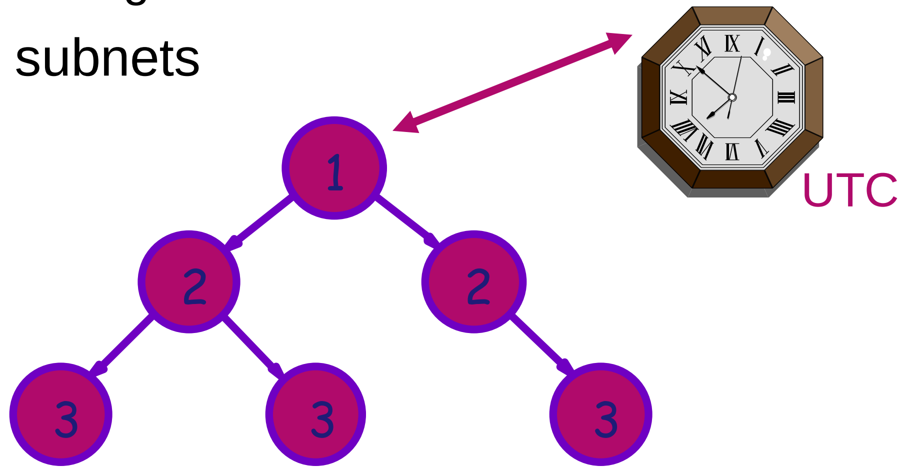
Clock Synchronization: NTP (1/6)

- Goals
  - ability to externally synchronize clients via Internet to UTC
  - provide reliable service tolerating lengthy losses of connectivity
  - enable clients to resynchronize sufficiently frequently to offset typical HW drift rates
  - provide protection against interference
- Synchronization subnets

strata 1

strata 2

strata 3
(user workstations)



NTP Basic Idea

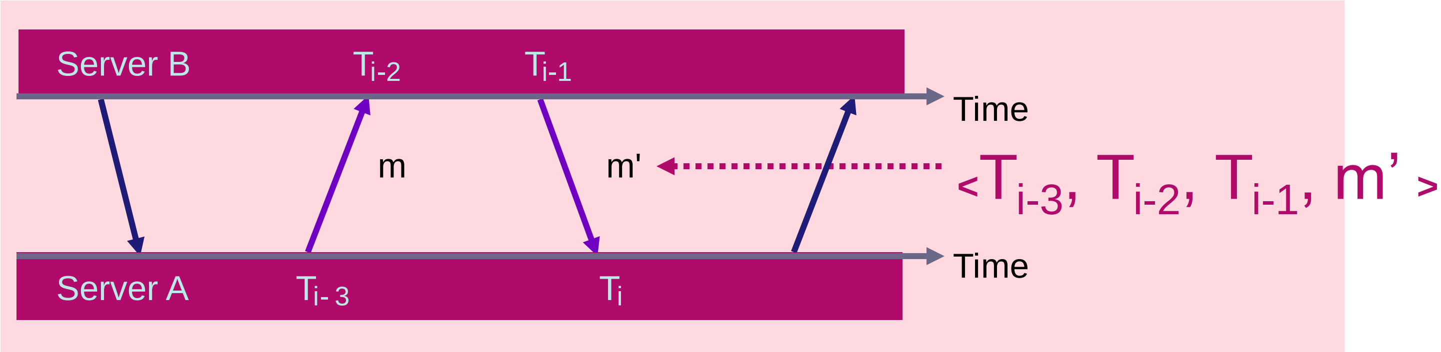
- Layered client-server architecture, based on UDP message passing
- Synchronization at clients with higher strata number less accurate due to increased latency to strata 1 time server
- Failure robustness: if a strata 1 server fails, it may become a strata 2 server that is being synchronized though another strata 1 server

NTP Modes

- Multicast:
  - One computer periodically multicasts time info to all other computers on network
  - These adjust clock assuming a very small transmission delay
  - Only suitable for high speed LANs; yields low but usually acceptable sync.
- Procedure-call: similar to Christian's protocol
  - Server accepts requests from clients
  - Applicable where higher accuracy is needed, or where multicast is not supported by the network's hard- and software
- Symmetric:
  - Used where high accuracy is needed

Procedure-Call and Symmetric Modes

- All messages carry timing history information
  - local timestamps of send and receive of the previous NTP message
  - local timestamp of send of this message

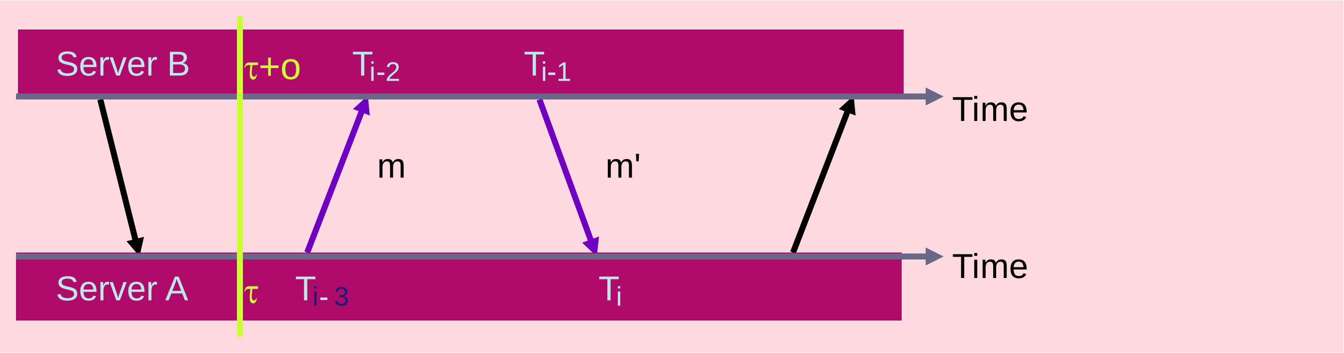


  - For each pair i of messages (m, m') exchanged between two servers the following values are being computed

    (based on 3 values carried w/ msg and 4th value obtained via local timestamp):

    - offset o_i: estimate for the actual offset between two clocks
    - delay d_i: true total transmission time for the pair of messages

NTP: Delay and Offset



- Let o the true offset of B’s clock relative to A’s clock, and let t and t’ the true transmission times of m and m’ (T_i, T_{i-1} ... are not true time)
- Delay

  $T_{i-2} = T_{i-3} + t + o$ (1) and $T_i = T_{i-1} + t' - o$ (2) which leads to

  $d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$ (clock errors zeroed out → (almost) true d)
- Offset

  $o_i = ½ (T_{i-2} – T_{i-3} + T_{i-1} – T_i)$ (only an estimate)

NTP Implementation

- Statistical algorithms based on 8 most recent <o_i, d_i> pairs: → determine quality of estimates
- The value of o_i that corresponds to the minimum d_i is chosen as an estimate for o
- Time server communicates with multiple peers, eliminates peers with unreliable data, favors peers with higher strata number (e.g., for primary synchronization partner selection)
- NTP phase lock loop model: modify local clock in accordance with observed drift rate
- Experiments achieve synchronization accuracies of 10 msecs over Internet, and 1 msec on LAN using NTP

Clocks and Synchronization

Requirements:

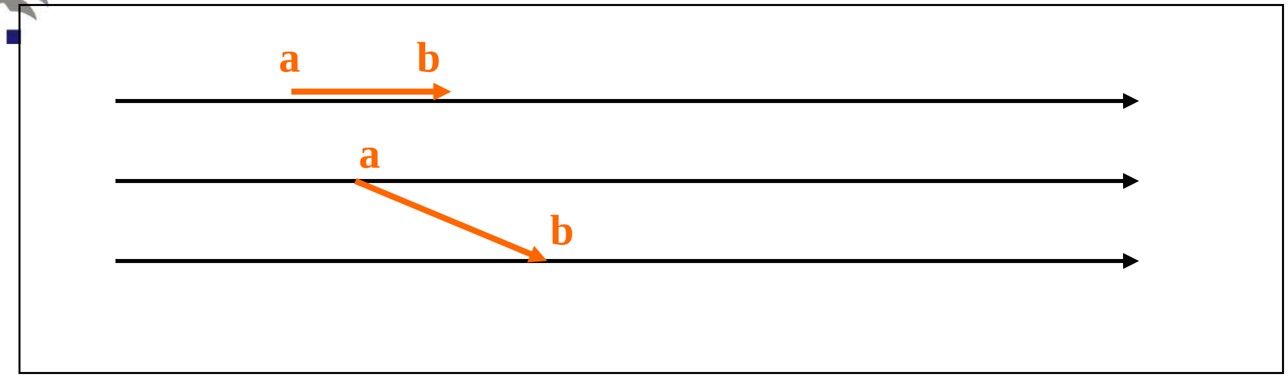
- ”*causality*”: real-time order ~ timestamp order (”behavioral correctness” – seen by the user)
- *groups* / *replicates*: all members see the events in the same order
- ”*multiple-copy-updates*”: order of updates, consistency conflicts?
- *serializability of transactions*: bases on a common understanding of transaction order

A perfect physical clock is sufficient!

A perfect physical clock is impossible to implement!

Above requirements met with much lighter solutions!

Happened-Before Relation ”a -> b”



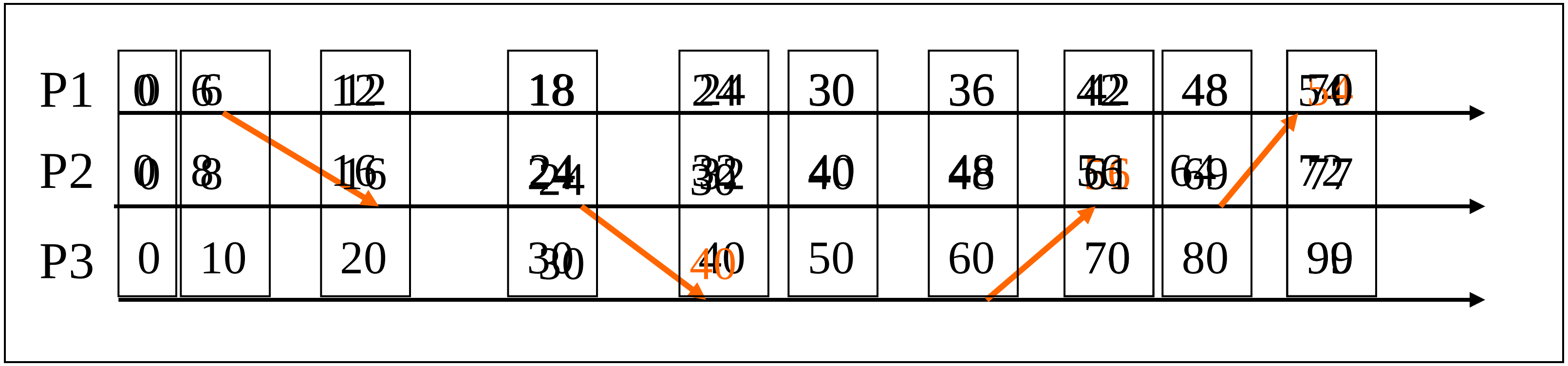
- if a, b are *events in the same process*, and a occurs before b, then a -> b

- if a is the event of a *message being sent*, and b is the event of the *message being received*, then a -> b

- a || b if neither a -> b nor b -> a (a and b are *concurrent*)

Note: if a -> b and b -> c then a -> c

Logical Clocks: Lamport Timestamps



process p_i , event e , clock L_i , timestamp $L_i(e)$

- ***at*** $\boldsymbol{p_i}$: before each event $L_i = L_i + 1$
- when p_i sends a ***message*** m to p_j

1. p_i: ($L_i = L_i + 1$); $t = L_i$; message = (m, t) ;
2. p_j: $L_j = \max(L_j, t)$; $L_j = L_j + 1$;
3. L_j(receive event) = L_j ;

Lamport Clocks: Problems

1. Timestamps do not specify the order of events
   - e -> e' => L(e) < L(e')

   BUT

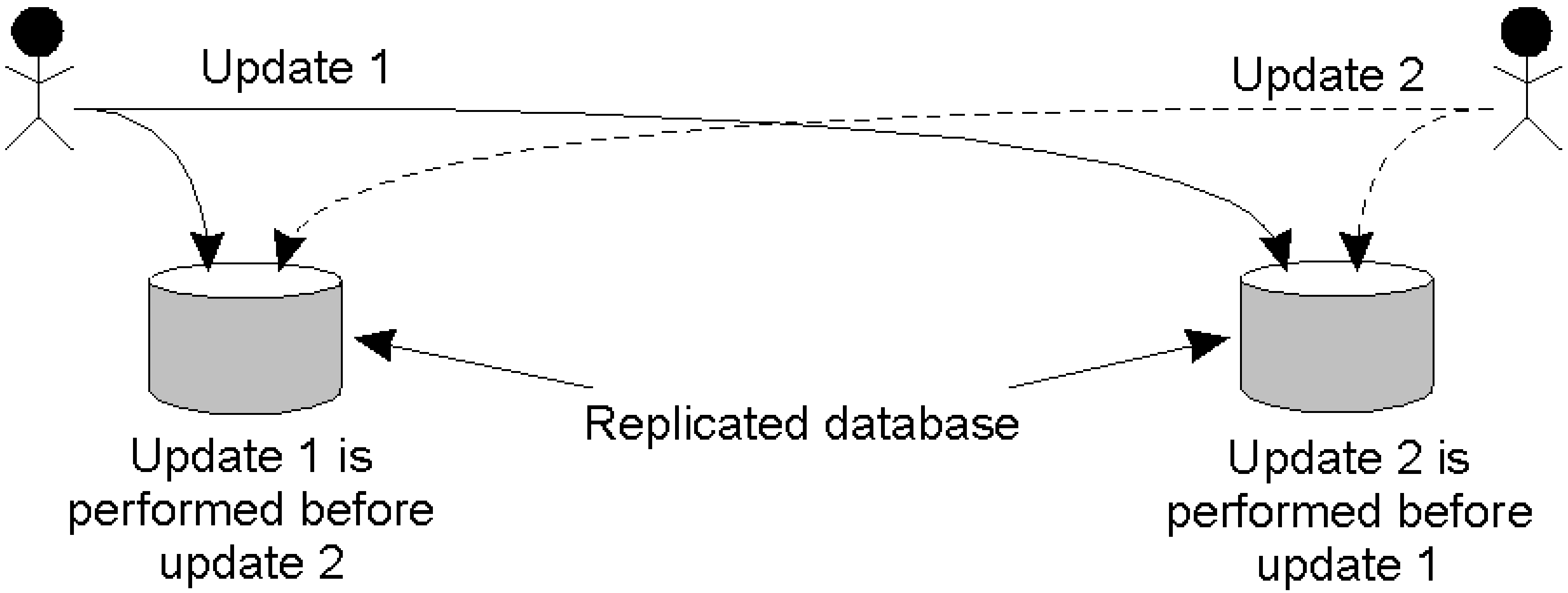
   - L(e) < L(e') does not imply that e -> e'
2. Total ordering
   - problem: define order of e, e' when L(e) = L(e')
   - solution: extended timestamp (T_i, i), where T_i is L_i(e)
   - definition: (T_i, i) < (T_j, j)

     if and only if

     either $T_i < T_j$

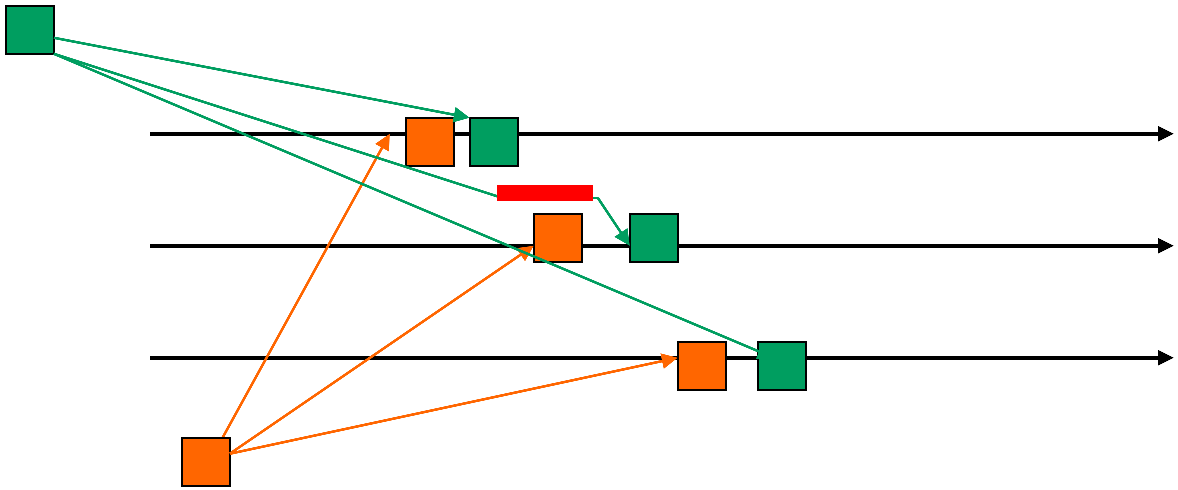
     or $T_i = T_j$ and i < j

Example: Totally-Ordered Multicasting (1)



Updating a replicated database and leaving it in an inconsistent state.

Example: Totally-Ordered Multicasting (2)



Total ordering:

all receivers (applications) see all messages in the same order (which is not necessarily the original sending order)

Example: multicast operations, group-update operations

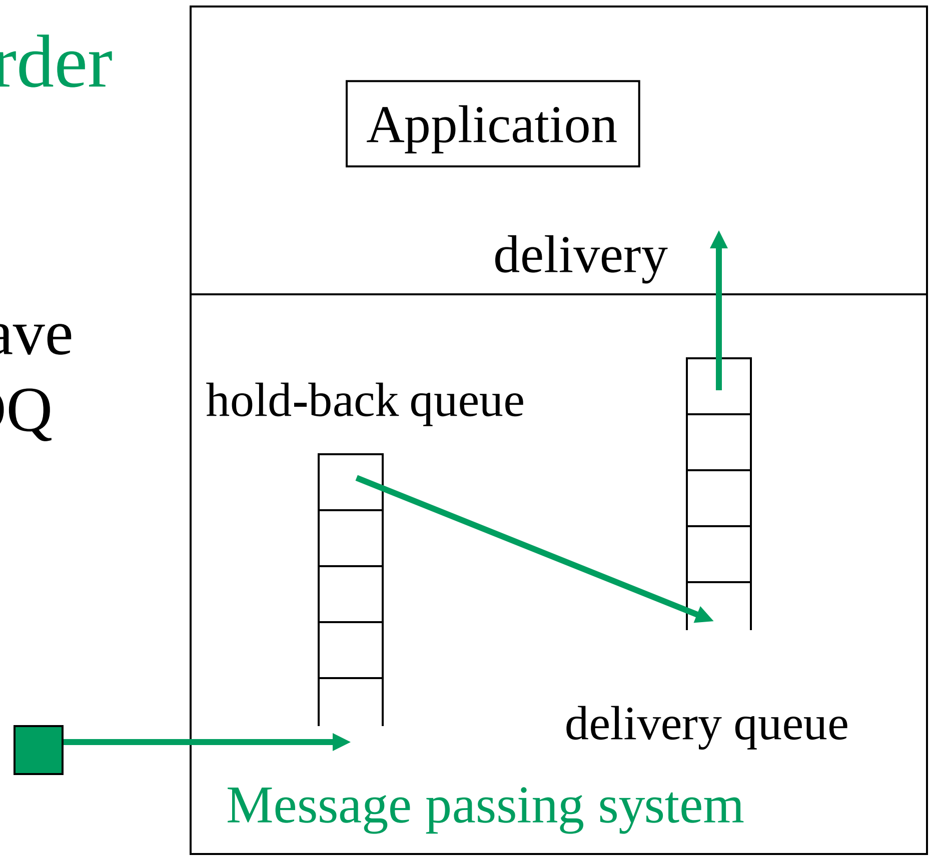
Example: Totally-Ordered Multicasting (3)

Guaranteed delivery order

- *new* message => HBQ

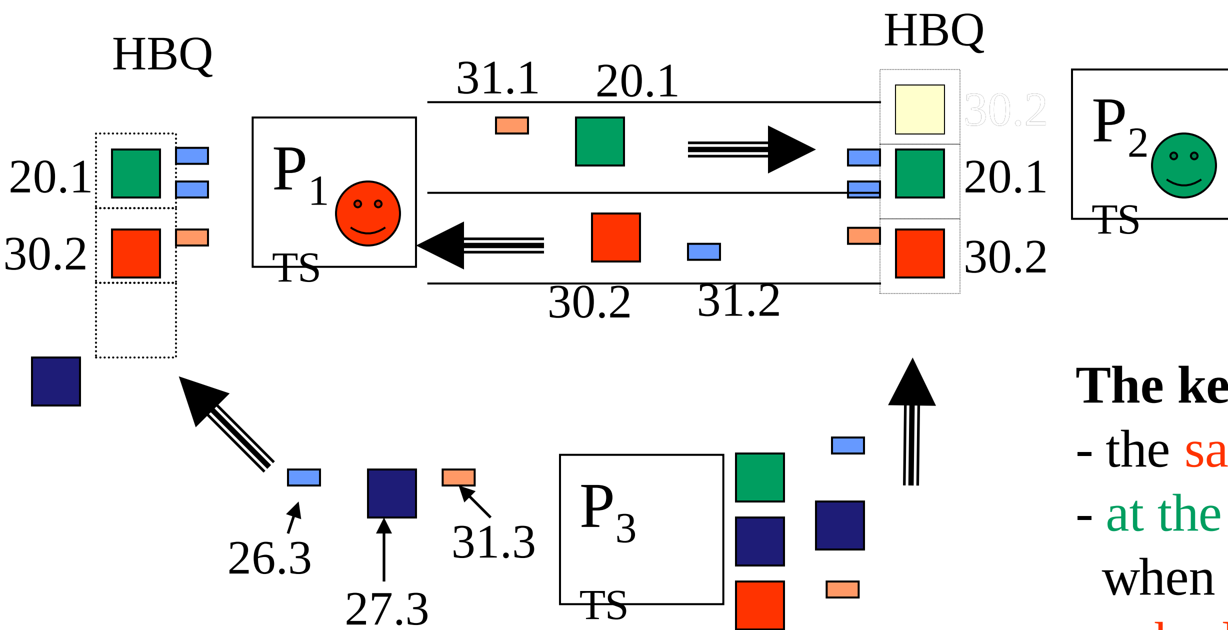
- when *all predecessors* have arrived: message => DQ

- when *at the head of DQ*: message => application (application: *receive* …)



Algorithms:
see. Defago et al ACM CS, Dec. 2004

Example: Totally-Ordered Multicasting (4)



Original timestamps

P_1 19
P_2 29
P_3 25

The key idea

- the same order in all queues
- at the head of HBQ:
  when all ack's have arrived
  nobody can pass you

Multicast:

- everybody receives the message (incl. the sender!)
- messages from one sender are received in the sending order
- no messages are lost

Various Orderings

- Total ordering
- Causal ordering
- FIFO (First In First Out)

  (wrt an individual communication channel)

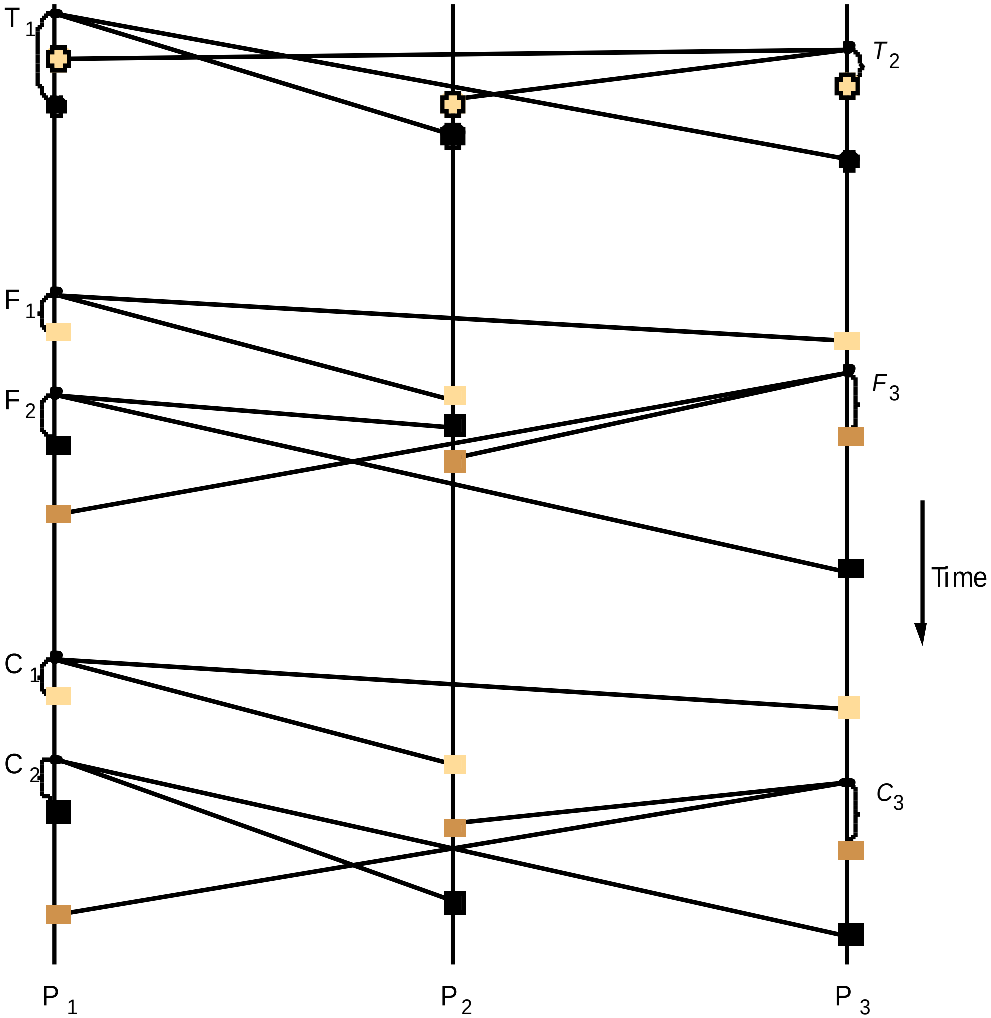
  Total and causal ordering are independent:

  neither induces the other;

  Causal ordering induces FIFO

Total, FIFO and Causal Ordering of Multicast Messages

Notice the consistent ordering of totally ordered messages T_1 and T_2, the FIFO-related messages F_1 and F_2 and the causally related messages C_1 and C_3 – and the otherwise arbitrary delivery ordering of messages.



Vector Timestamps

Goal:

timestamps should reflect *causal ordering*

L(e) < L(e') => " e happened before e' "

=>

Vector clock

each process P_i maintains a vector V_i :

1. $V_i[i]$ is the number of events that have occurred at P_i

   (the current local time at P_i)

2. if $V_i[j] = k$ then P_i knows about (the first) k events that have occurred at P_j

   (the local time at P_j was k, as P_j sent the last message that P_i has received from it)

Order of Vector Timestamps

Order of timestamps

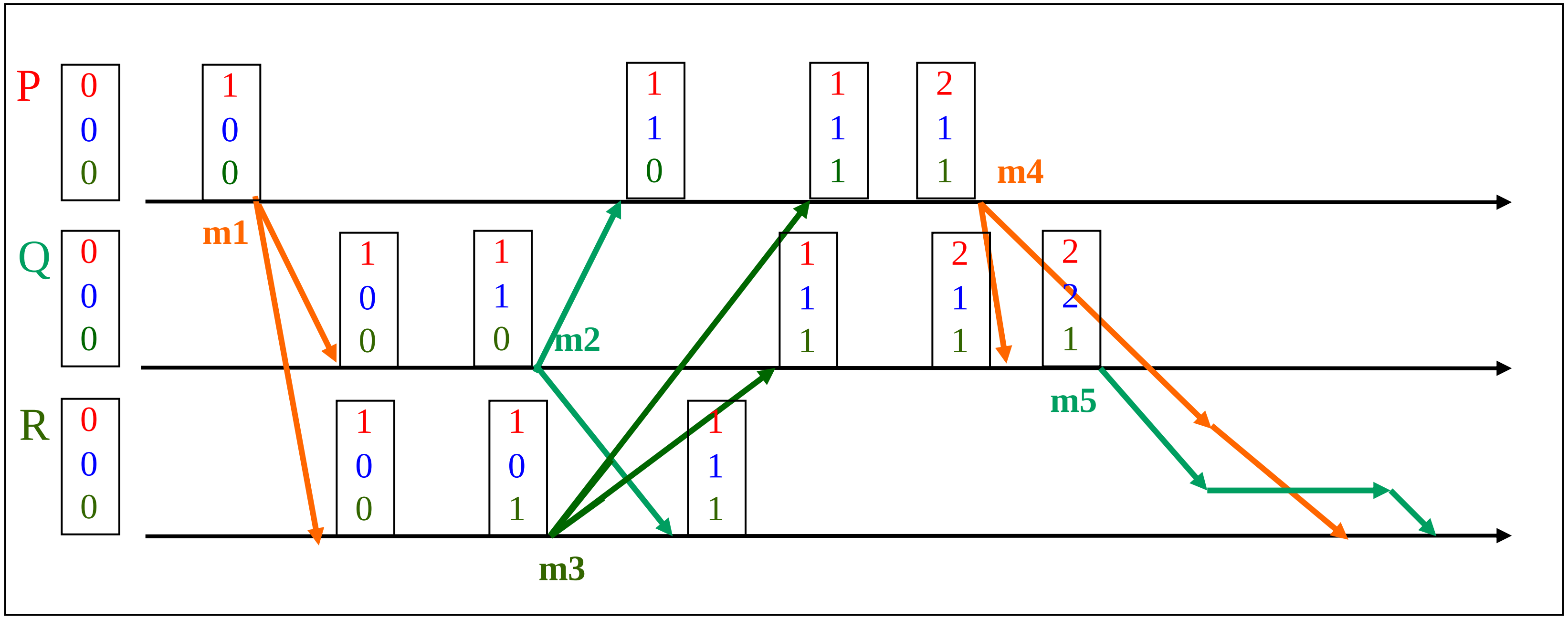
- $V = V'$ iff $V[j] = V'[j]$ for all j
- $V \le V'$ iff $V[j] \le V'[j]$ for all j
- $V < V'$ iff $V \le V'$ and $V \ne V'$

Order of events *(causal order)*

- e -> e' => V(e) < V(e')
- V(e) < V(e') => e -> e'
- concurrency:

e || e' if **not** $V(e) \le V(e')$
and **not** $V(e') \le V(e)$

Causal Ordering of Multicasts (1)



Event:
message sent

Timestamp [i,j,k] :

i messages sent from P

j messages sent form Q

k messages sent from R

R: m1 [100]   m4 [211]
    m2 [110]   m5 [221]
    m3 [101]

m5 [221] vs. 111

Causal Ordering of Multicasts (2)

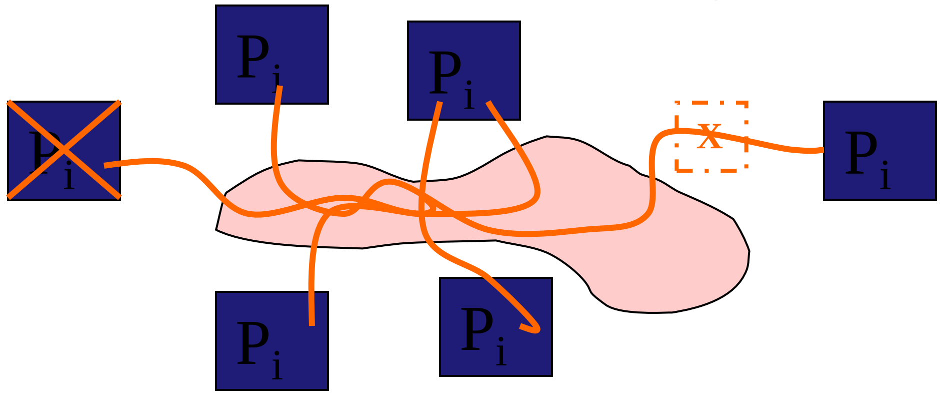
Use of timestamps in causal multicasting

1) P_s multicast: $V_s[s] = V_s[s] + 1$
2) Message: include $vt = V_s[*]$
3) Each receiving P_r : the message **can be delivered when**
   - **$vt[s] = V_r[s] + 1$** *(all previous messages from P_s have arrived)*
   - for each component **k ($k \neq s$): $V_r[k] \geq vt[k]$**
     (P_r has now seen all the messages that P_s had seen when the message was sent)
4) When the message from P_s becomes deliverable at P_r the message is inserted into the delivery queue

(note: the delivery queue preserves causal ordering)

5) **At delivery**: $V_r[s] = V_r[s] + 1$

Coordination and Agreement



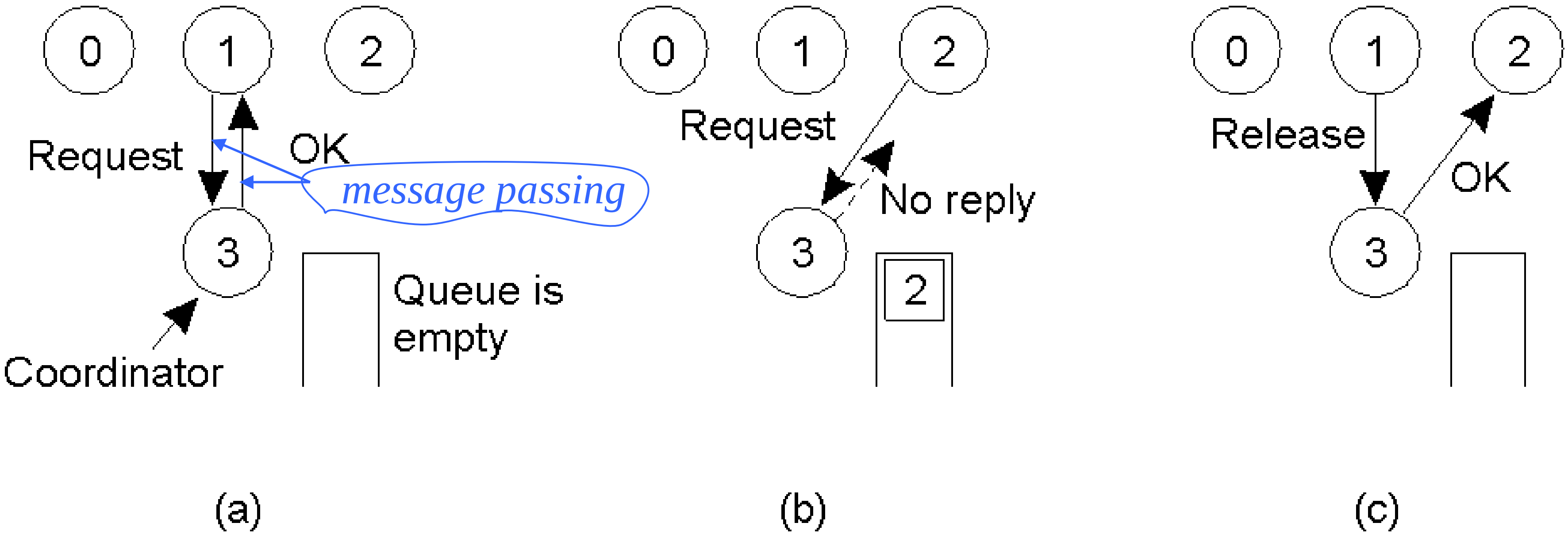
Coordination of functionality

- reservation of resources *(distributed mutual exclusion)*
- elections (coordinator, initiator)
- multicasting
- distributed transactions

Decision Making

- Centralized: one coordinator (decision maker)
  - algorithms are simple
  - no fault tolerance *(if the coordinator fails)*
- Distributed decision making
  - algorithms tend to become complex
  - may be extremely fault tolerant
  - behaviour, correctness ?
  - assumptions about failure behaviour of the platform !
- Centralized role, changing “population of the role”
  - easy: one decision maker at a time
  - challenge: management of the “role population”

Mutual Exclusion: A Centralized Algorithm (1)

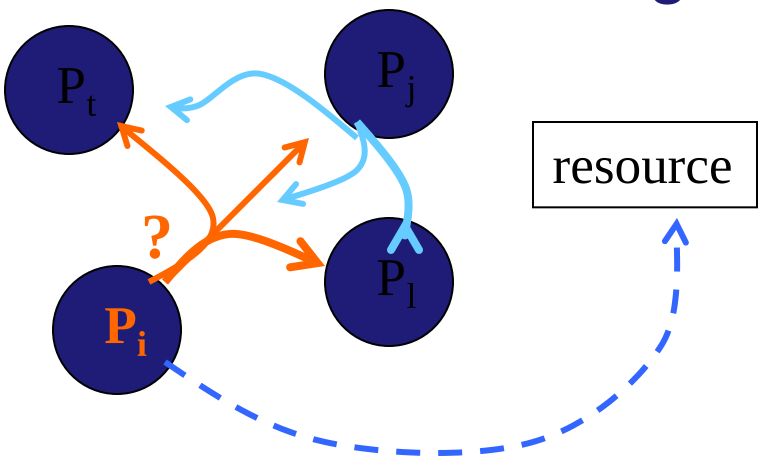


a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted
b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.
c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2

Mutual Exclusion: A Centralized Algorithm (2)

- **Examples** of usage
  - a stateless server (e.g., Network File Server)
  - a separate lock server
- General **requirements** for mutual exclusion
  1. **safety**: at most one process may execute in the critical section at a time
  2. **liveness**: requests (enter, exit) eventually succeed *(no deadlock, no starvation)*
  3. **fairness** (ordering): if the request A *happens before* the request B then A is honored before B
- **Problems**: fault tolerance, performance

A Distributed Algorithm (1)



Ricart – Agrawala

- The general idea:
  - ask everybody
  - wait for permission from everybody

The problem:
- several simultaneous requests (e.g., P_i and P_j)
- all members have to agree (*everybody*: “first P_i then P_j”)

- Assumes total order of messages and up-to-date list of nodes

A Distributed Algorithm (2)

- *On initialization*
  state := RELEASED;

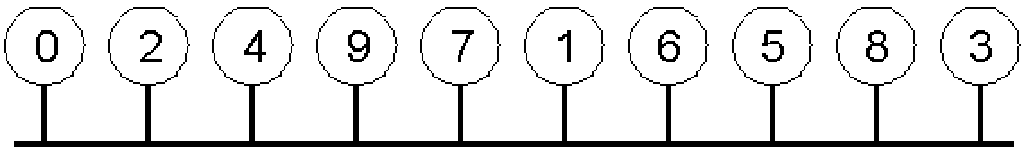
  To enter the section
  state := WANTED;
  T := request's timestamp;
  Multicast *request* to all processes;
  Wait until (number of replies received = (N-1));
  state := HELD;

  request processing deferred here

  On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
  if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
  then
  queue *request* from p_i without replying;
  else
  reply immediately to p_i;
  end if;

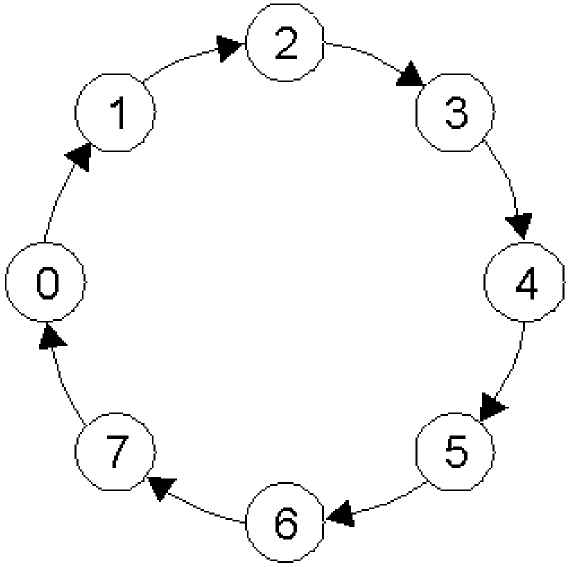
  To exit the critical section
  state := RELEASED;
  reply to all queued requests;

A Token Ring Algorithm



An unordered group of processes on a network.

(a)



(b)

A logical ring constructed in software.

Algorithm:

- token passing: straightforward
- lost token: 1) detection? 2) recovery?

Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

A comparison of three mutual exclusion algorithms.

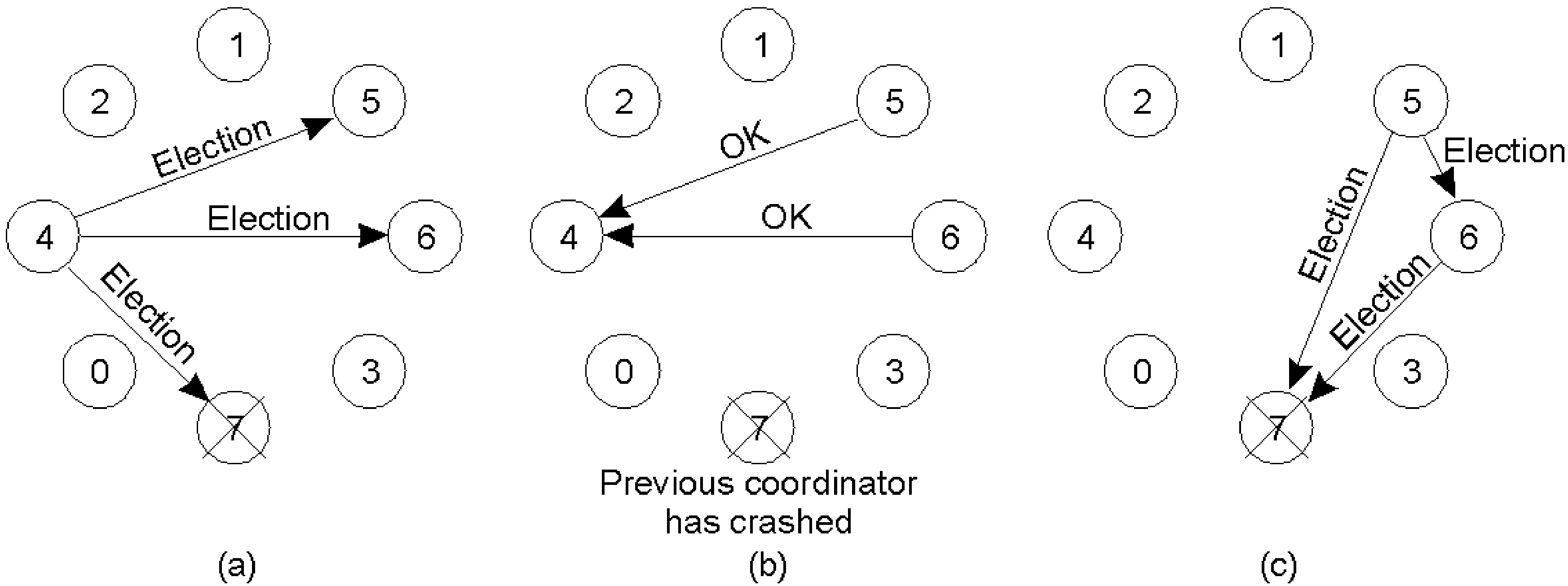
Election Algorithms

- Need:
  - computation: a group of concurrent actors
  - algorithms based on the activity of a special role (coordinator, initiator)
  - election of a coordinator: initially / after some special event (e.g., the previous coordinator has disappeared)
- Premises:
  - each member of the group {Pi}
    - knows the identities of all other members
    - does not know who is up and who is down
  - all electors use the same algorithm
  - election rule: the member with the highest Pi
- Several algorithms exist

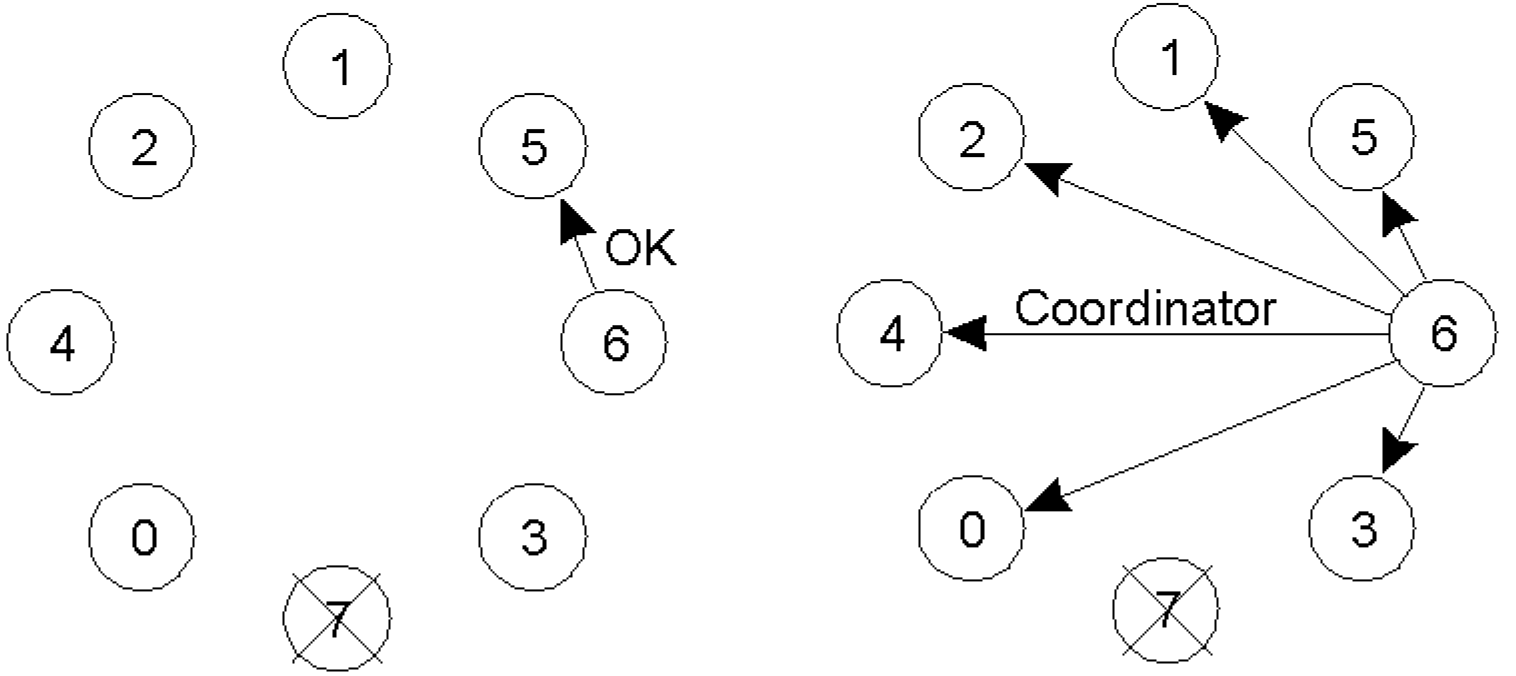
The Bully Algorithm (1)

- P_i notices: coordinator lost
    1. P_i to {all P_j st P_j>P_i}: ELECTION!
    2. if no one responds => P_i is the coordinator
    3. some P_j responds => P_j takes over, P_i's job is done
- P_i gets an ELECTION! message:
    1. reply OK to the sender
    2. if P_i does not yet participate in an ongoing election: hold an election
- The new coordinator P_k to everybody: " P_k COORDINATOR"
- P_i: ongoing election & no "P_k COORDINATOR": hold an election
- P_j recovers: hold an election
- *("OK" means "stand down, I'll take it from here")*

The Bully Algorithm (2)



The bully election algorithm
a) Process 4 holds an election
b) Process 5 and 6 respond, telling 4 to stop
c) Now 5 and 6 each hold an election



(d)

(e)

d) Process 6 tells 5 to stop

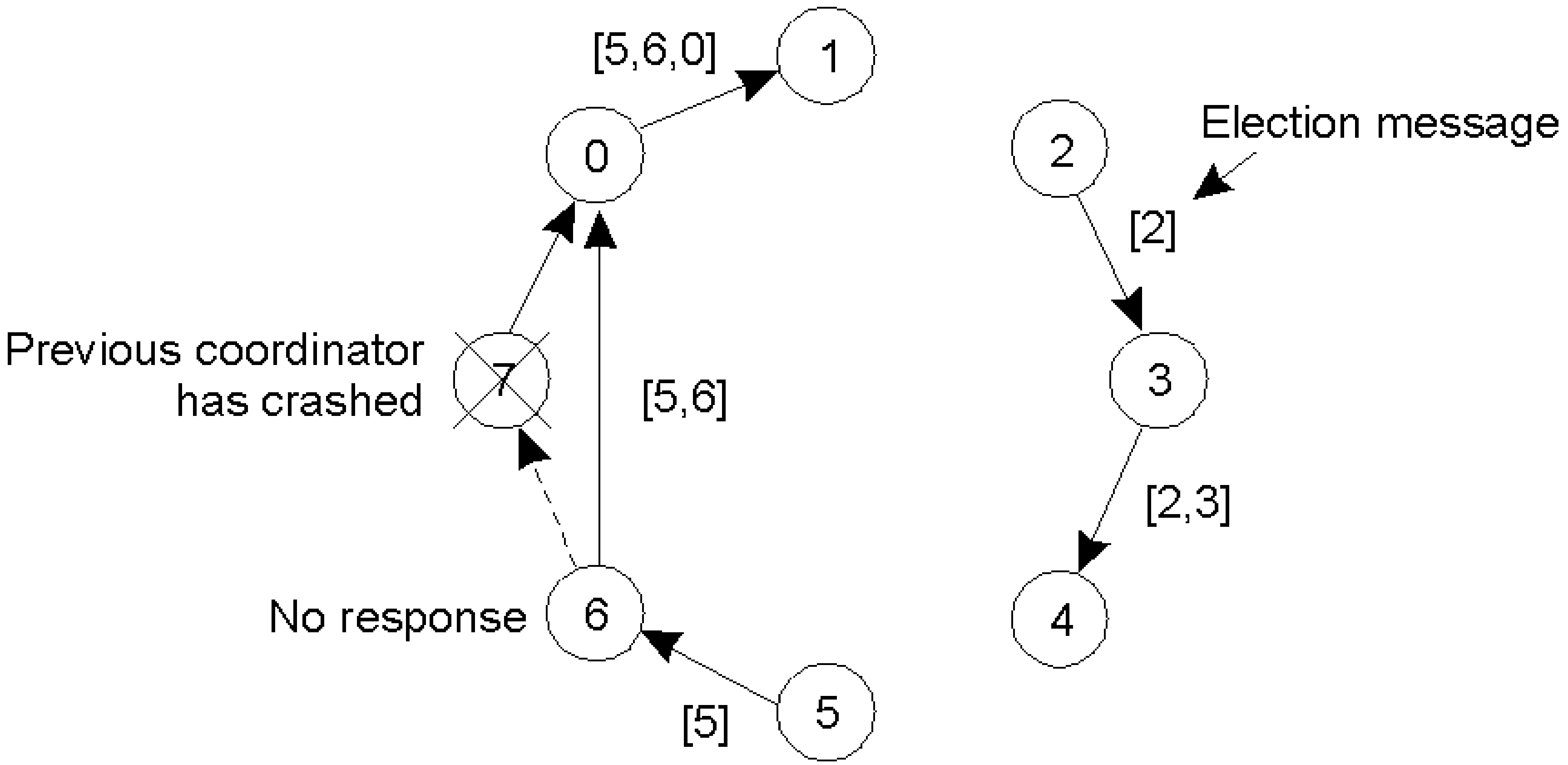
e) Process 6 wins and tells everyone

A Ring Algorithm (1)

- Group $\{P_i\}$ "fully connected"; but organized as ring for election purposes
- P_i notices: coordinator lost
  - send ELECTION(P_i) to the next P
- P_j receives ELECTION(P_i)
  - send ELECTION(P_i, P_j) to successor
- . . .
- P_i receives ELECTION(..., P_i, ...)
  - active_list = {collect from the message}
  - New coordinator = max id from {active_list}
  - send COORDINATOR(active_list) to the next P (circulate through the ring)

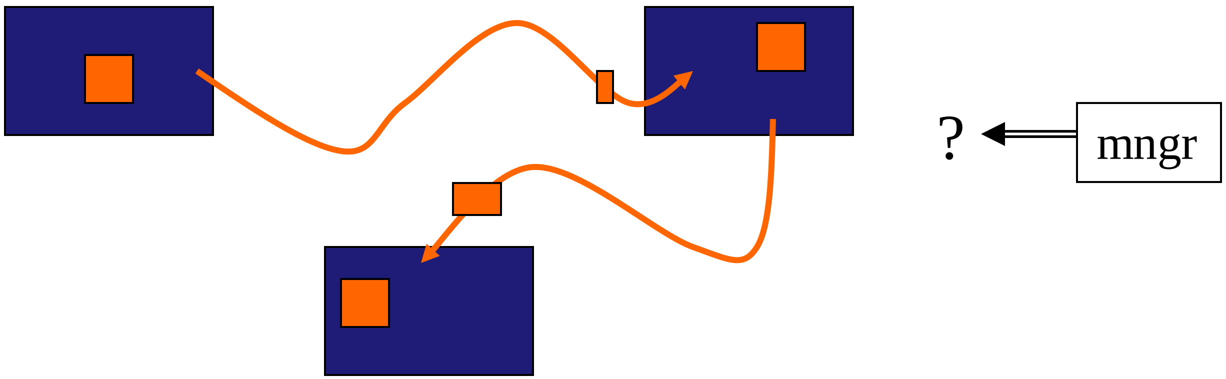
Note: Others can also find the new coordinator by picking the max id from the active list.

A Ring Algorithm (2)



Election algorithm using a ring.

Global State (1)



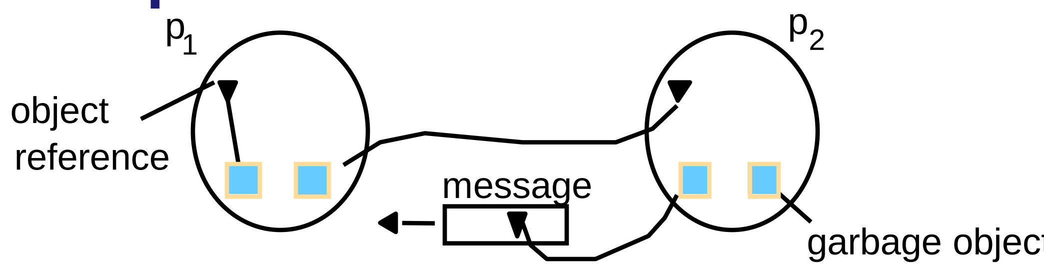
- Needed for: checkpointing, garbage collection, deadlock detection, termination, testing

- How to observe the state
  - states of processes
  - messages in transfer

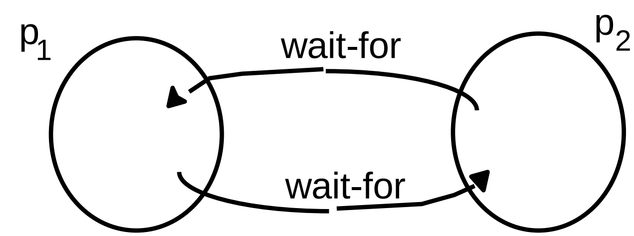
A **state**: application-dependent specification

Detecting Global Properties

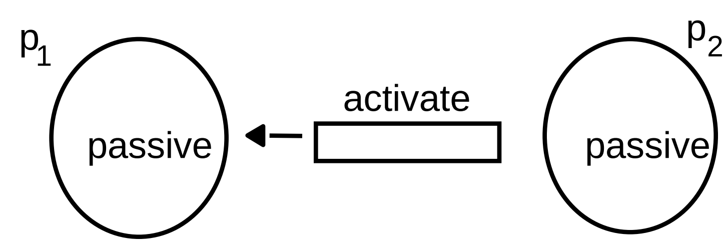
a. Garbage collection



b. Deadlock



c. Termination

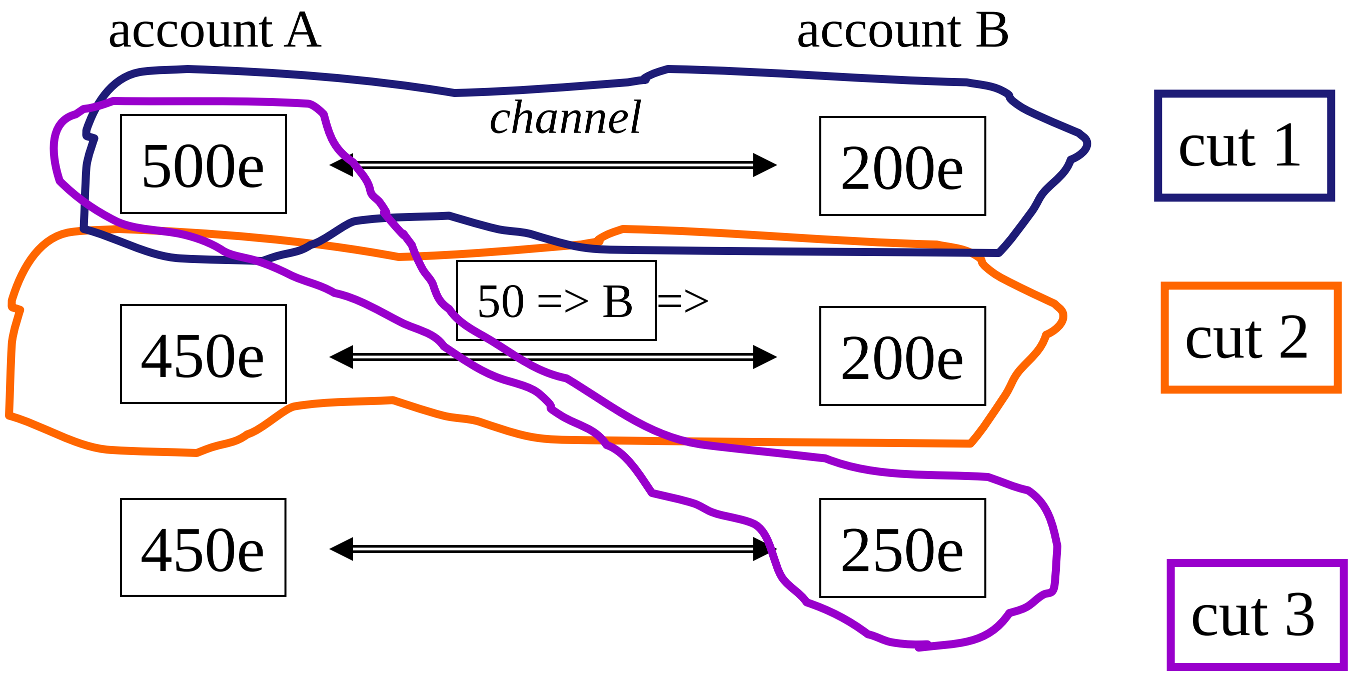


Distributed Snapshot

- Each node: history of important events
- Observer: at each node i
  - time: the local (logical) clock ” T_i ”
  - state S_i (history: {event, timestamp})

  => system state { S_i }
- A *cut:* the system state { S_i } ”at time T”
- Requirement:
  - {S_i} might have existed ⬄ consistent with respect to some criterion
  - one possibility: consistent wrt ” happened-before relation ”

Ad-hoc State Snaphots



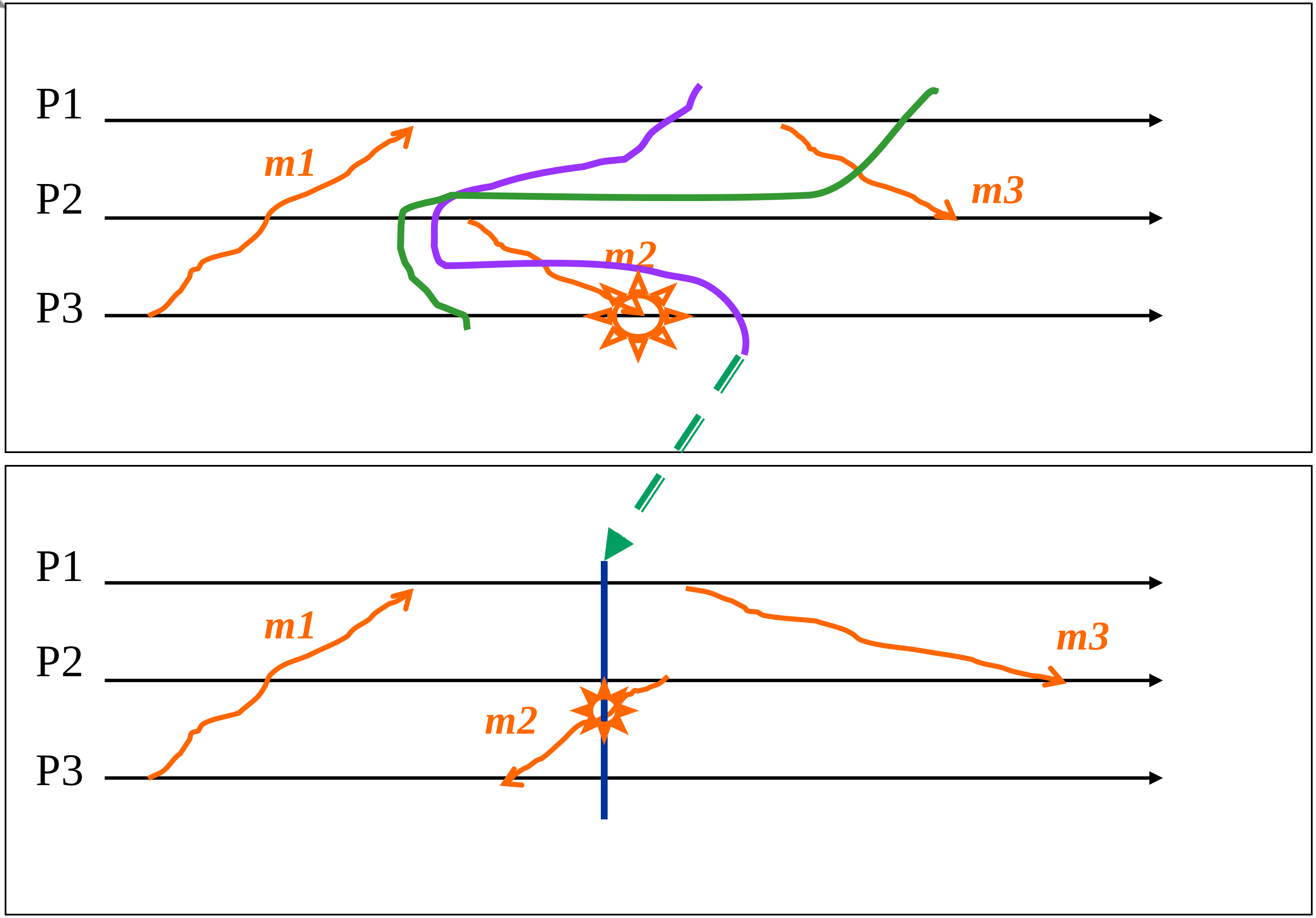
(inconsistent or)
weakly consistent

inconsistent

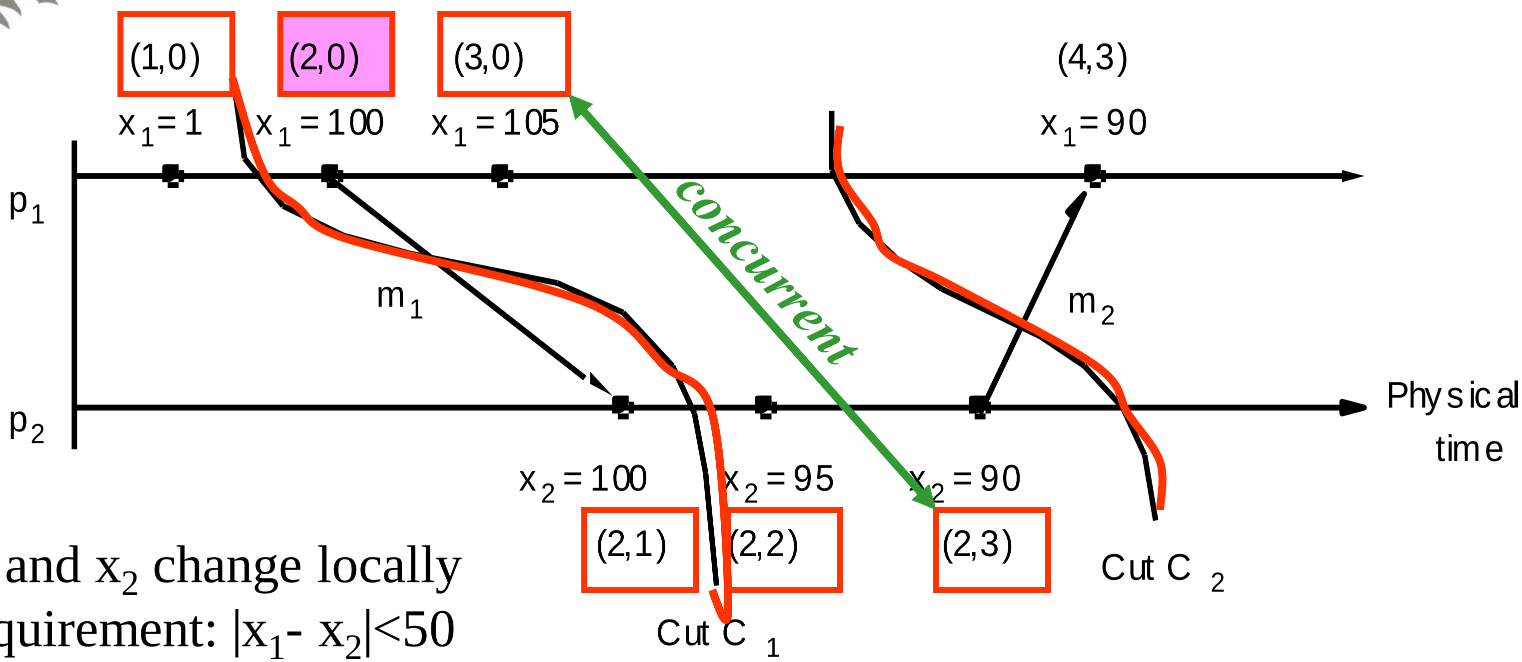
strongly consistent

state changes: money transfers A ⇔ B
invariant: A+B = 700

Consistent and Inconsistent Cuts



Cuts and Vector Timestamps



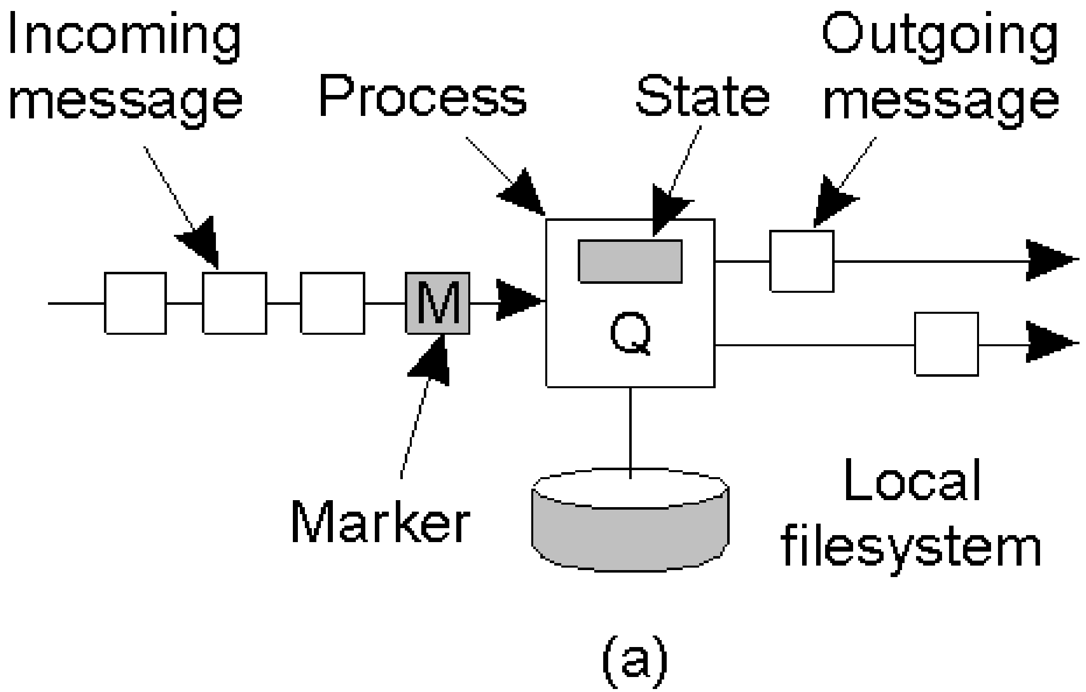
x_1 and x_2 change locally
requirement: $|x_1 - x_2| < 50$
a ”large” change (”>9”) =>
send the new value to the other process

event: a change of the local x
=> increase the vector clock

{S_i} system state history: all events
Cut: all events before the ”cut time”

A cut is consistent if, for each event, it also contains all the events that ”happened-before”.

Example: Chandy Lamport (1)

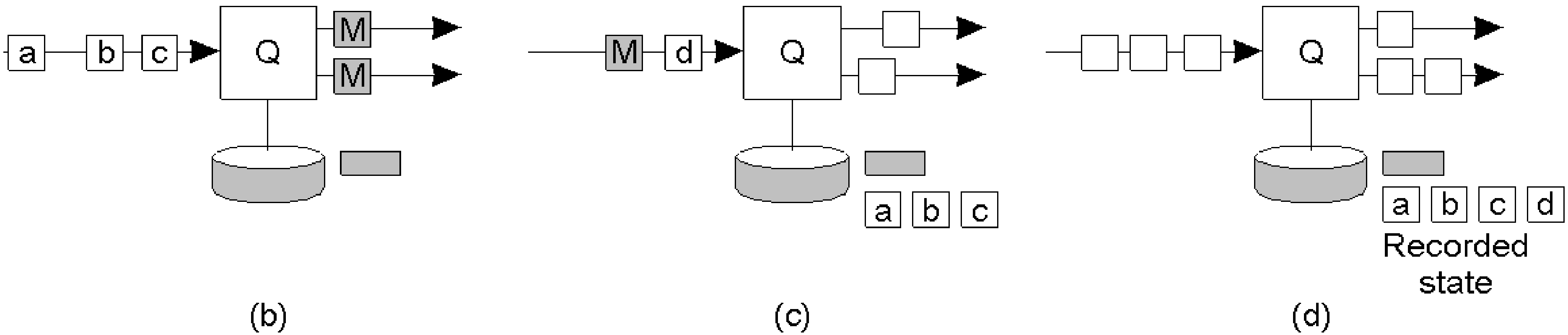


The snapshot algorithm of Chandy and Lamport

- Assumes point-to-point, order-preserving connections.

a) Process Q receives a marker message ("let's take a snapshot")

Chandy Lamport (2)



- b) Process Q receives a marker for the first time, so records its local state, and sends marker on every outgoing channel
- c) Q records all incoming messages
- d) Q receives a marker for its incoming channel a second time and finishes recording the state of this incoming channel

Chandy and Lamport's 'Snapshot' 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.

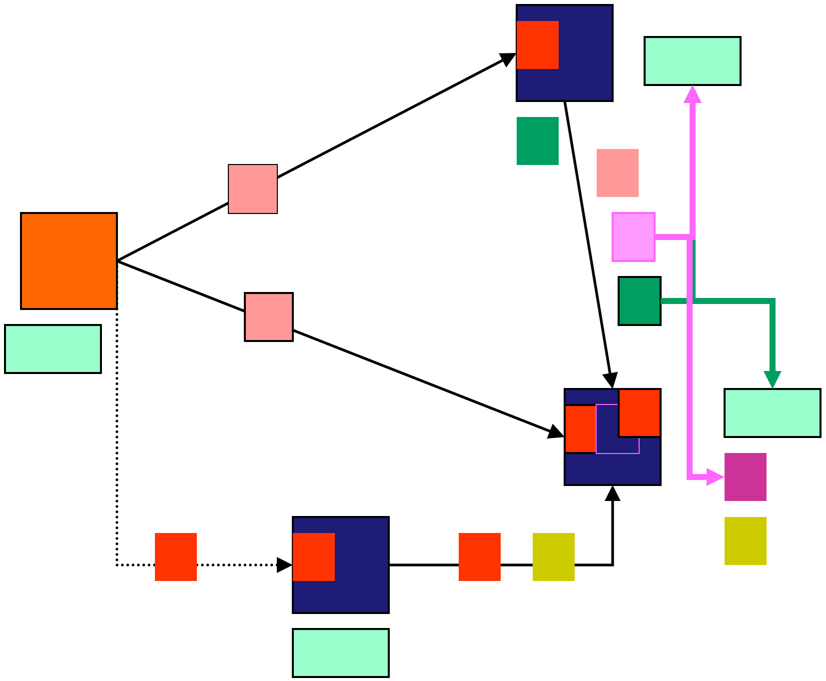
end if

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).

Implementation of Snapshot



Chandy, Lamport

point-to-point, order-preserving connections

Chapter Summary

- Synchronization
- Clocks
- Logical and vector clocks
- Coordination, elections