Concepts of Distributed Systems 2006/2007

Synchronization

Johan Lukkien

Contents

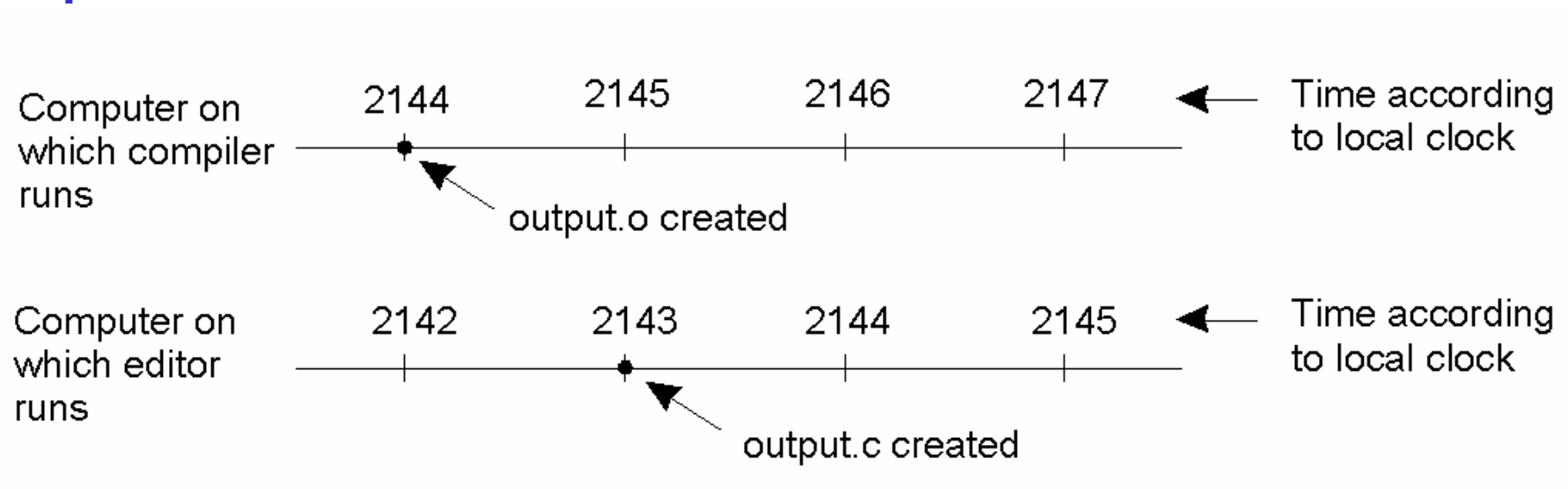
- Physical and logical clocks
- Distributed synchronization problems
  - global state
  - leader election
  - mutual exclusion
  - transactions (atomicity)

Synchronization and ordering

- Clock synchronization
  - ‘at the same time’, ‘in time’
  - real-time
  - ...physical clocks

- Event ordering
  - this before that
  - causality
  - ...logical clocks

Clock Synchronization



- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Clocks

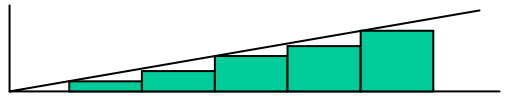
- *"What, then, is time? If no one asks me, I know what it is. If I wish to explain to him who asks me, I do not know"*

  (St.Augustine)

- Real-time is linear, transitive, irreflexive and dense ☺
  - though discrete in computer systems

- Measuring time:
  - access directly the environments' time ("share it") or
  - approximate it with an internal clock

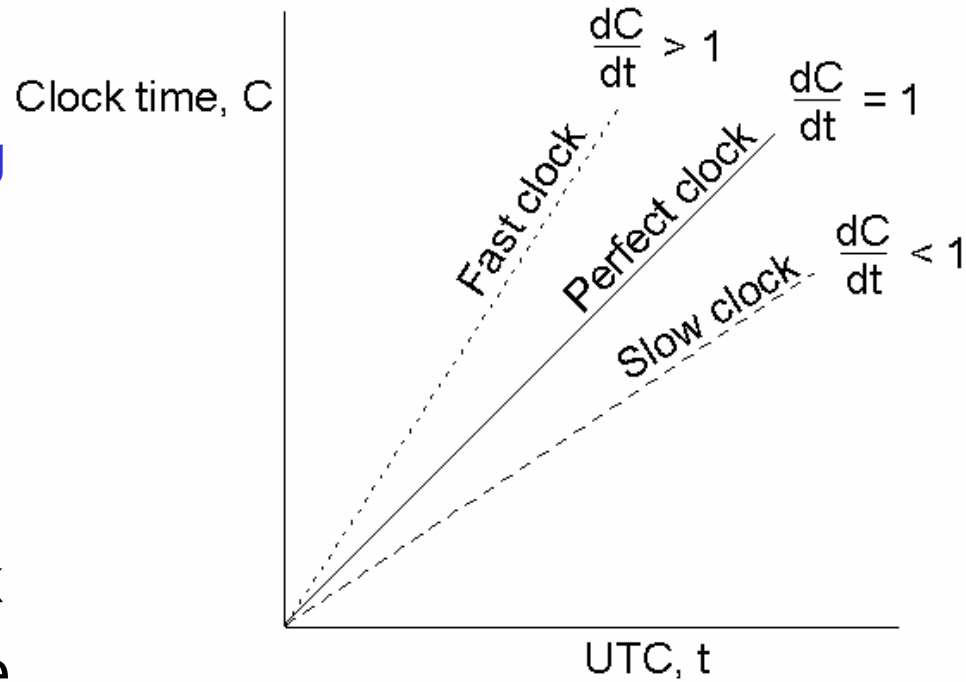
Clocks & time



- Physically
  - crystal + registers to control interrupt frequency
- Distributed clocks
  - will not run at the same speed: drift, skew
  - must synchronize with each other and with real time
- Real time
  - International Atomic Time (TAI)
    - average of +/- 50 Cesium clocks
      - not chronoscopic: needs leap seconds to map to solar time
  - Universal Time Coordinated (UTC)
    - TAI + leap seconds
    - broadcast by radio & satellites: accuracy of 0.5 ms

Clock model

- $C_p(t)$: local time at p at real time t;
  $c_p(T)$: inverse of C
  - real time correspondig to clock value
- **Drift**: $1-dC/dt$
- **Skew**: drift x time



- Correct clock: a clock behaving within these limits for given maximum |drift|

Property of a correct clock

- For $t0 \leq t1$ and $|\text{drift}| \leq r$
  - $(1-r)(t1-t0) \leq C_p(t1)-C_p(t0) \leq (1+r)(t1-t0)$

- ...since $r(t1-t0)$ is the maximum absolute skew in the period $t1-t0$.

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Computer Science, System Architecture and Networking

Problem statement

- For a given *fault model*, maintain *agreement* and *accuracy*:
  1. $\forall\ p,q,\ t :: |C_p(t) - C_q(t)| < \varepsilon$
  2. $\forall\ p,\ t :: |\ C_p(t) - t| < \varepsilon$

- **Note**: fault model can be extended to arbitrary behavior of clocks, communication failure etc.

- Overview: Ramanathan
  - Parameswaran Ramanathan, Kang G. Shin, Ricky W. Butler, *Fault-Tolerant Clock Synchronization in Distributed Systems*, IEEE Computer 23(10), pp33-42, October 1990

Properties, requirements

- Centralized
  - Christian's algorithm
  - Berkeley
  - ....scaling, fault tolerance
- Decentralized
  - e.g. Network Time Protocol (NTP, Mills)
- Message control
  - Active: time is broadcast (push)
  - Passive: time is requested (pull)

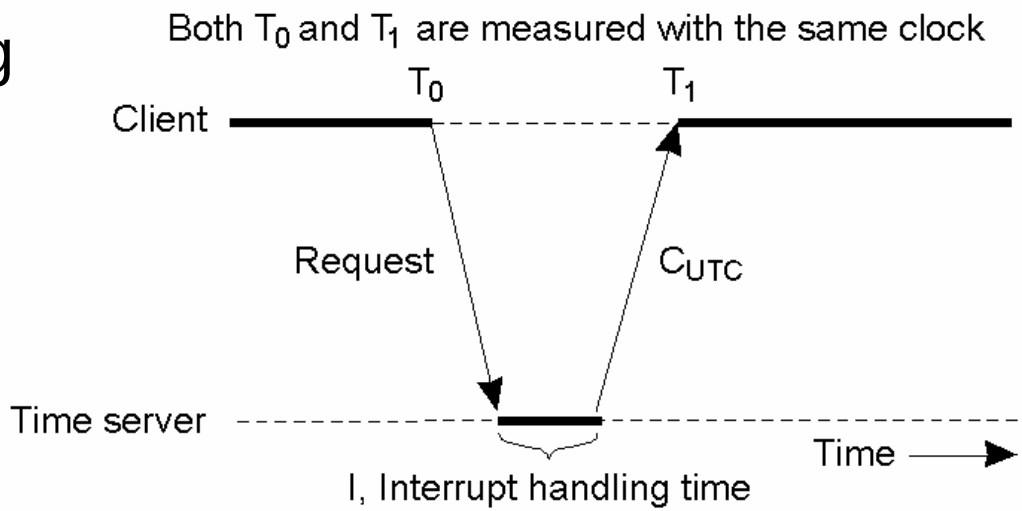
Properties, requirements

- Time does not run backwards
  - reduce drift gradually
  - make slow clocks faster

- No (large) discontinuities
  - maximal adjustment
    - absolute (state correction)
    - or in rate (rate correction)

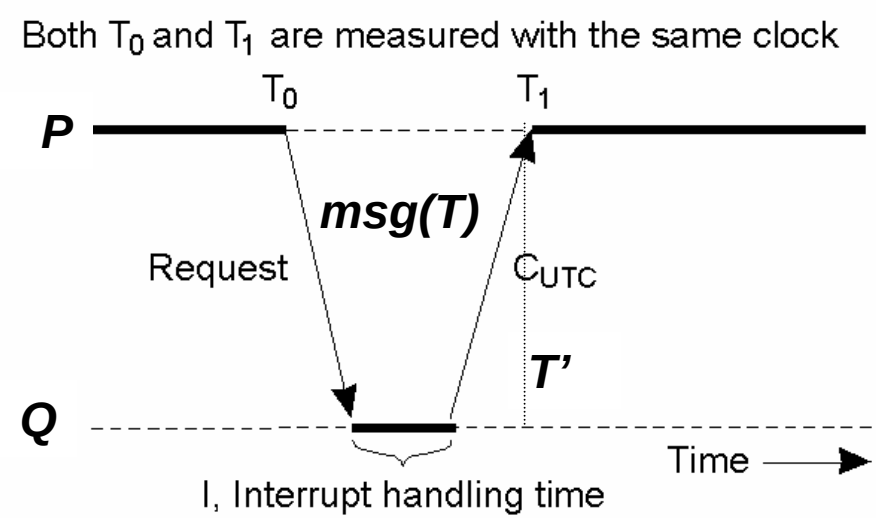
Cristian's Algorithm

- Ask time server frequent enough (passive)
  - roughly every *ε/(2*drift)* seconds
- Must take overhead times into account
  - e.g. average over several measurements
- Specifically good for synchronizing with UTC-server



Analysis

- Client: *P*, Server: *Q*
  - both |drift| ≤ *r*
- Minimum message transmission time: *m*
- Real-time round-trip delay: *d*
- Value of C_Q sent to *P* : *T*
- Value of C_Q *at* $c_P(T_1)$: *T'*



Boundaries

- P is interested in T' $(= C_Q(c_P(T_1))$)
- Properties
  - $c_P(T_1) - c_Q(T) \geq m$ (at least m seconds to transmit reply)
    - hence $T' - T \geq m(1-r)$
  - $c_P(T_1) - c_Q(T) \leq d-m$ (at least m seconds to transmit request)
    - hence $T'-T \leq (d-m)(1+r)$
  - $d = c_P(T_1) - c_P(T_0) \leq (T_1-T_0)/(1-r)$

    - a. $(T_1-T_0)/(1-r) \approx (T_1-T_0)(1+r)$
      - hence $T'-T \leq ((T_1-T_0)(1+r)-m)(1+r)$
      - $\approx (T_1-T_0)(1+2r) - m(1+r)$ (ignoring quadratic terms for r)

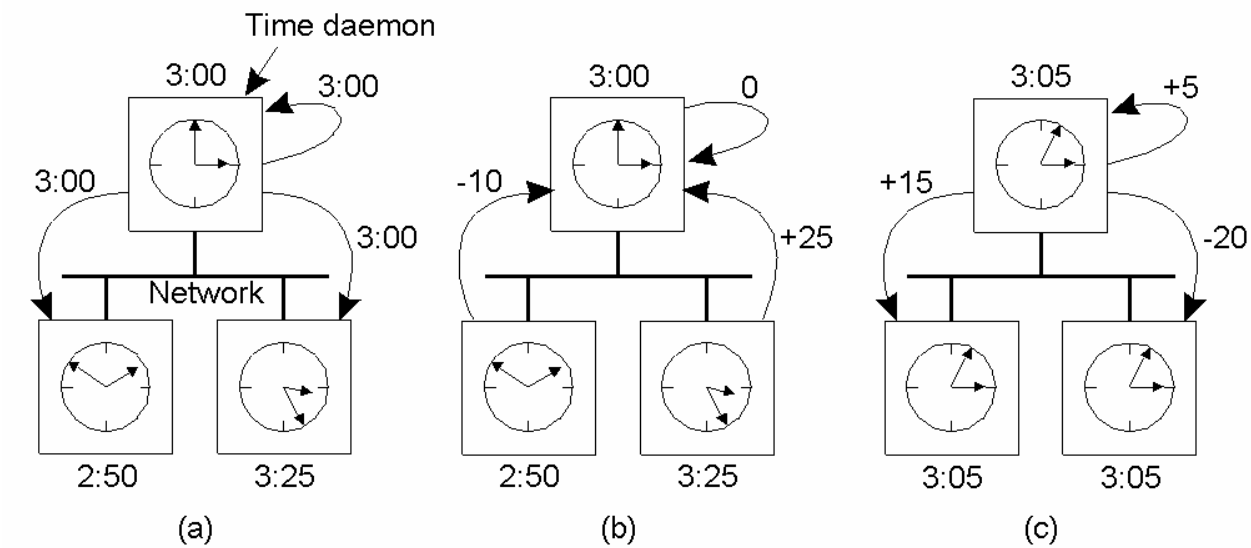
    - b. $T'-T \leq (T_1-T_0)(1+r)/(1-r) - m(1+r)$

Determine adjustments

- Approaches
  - take some value within the interval as estimate for the difference
    - e.g. centre: $T' = (T_1 - T_0)(1/2 + r) - m + T$
    - choose a value that takes a bias for the moment of recording the time
  - try to estimate *m*
  - repeat to obtain averages

The Berkeley Algorithm

- Actively determine network average
  - note: example does not obey the requirements!



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

- or just the rates

Contents

- Physical and logical clocks
- Distributed synchronization problems
  - global state
  - leader election
  - mutual exclusion
  - transactions (atomicity)

Event ordering

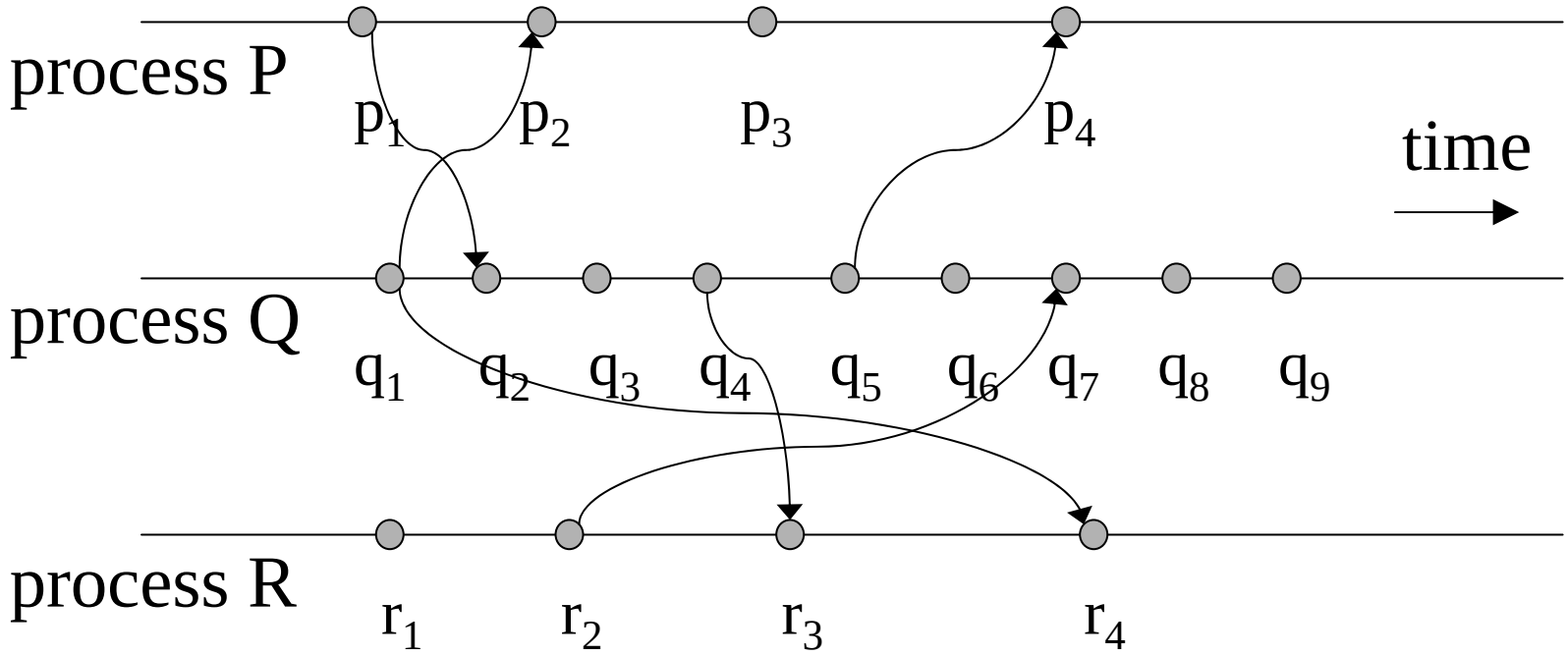
- Two orders in events
  - **temporal**: the time of occurrence in real-time
    - can be made a total order by using the process id.
    - induces a ‘happens-before’ relation
  - **causal**: cause-effect relationship; implies temporal order

- Develop a model time (logical time) that
  - respects temporal order
  - does not need real-time clocks
    - problems of synchronization etc.

Defining 'Happens Before': "$\rightarrow$"

- If *a* and *b* are events in the same process and *a* comes in time before *b*, then $a \rightarrow b$
- If *a* is the sending of a message and *b* is the receipt of the same message by another process, then $a \rightarrow b$
- If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$ (transitivity)
- $\neg(a \rightarrow a)$ (irreflexive)
- $\rightarrow$ is the smallest relation satisfying these
- **Concurrent** actions: $\neg(a \rightarrow b)$ and $\neg(b \rightarrow a)$

Example



happens before: $p_1 \rightarrow q_2$, $r_2 \rightarrow r_3$, $q_4 \rightarrow r_4$, $r_1 \rightarrow q_9$, …

concurrent: p_1 and q_1 ; p_3 and q_3 ,q_4, q_5 ; q_5 and r_3 , r_4 ; …

Timestamps

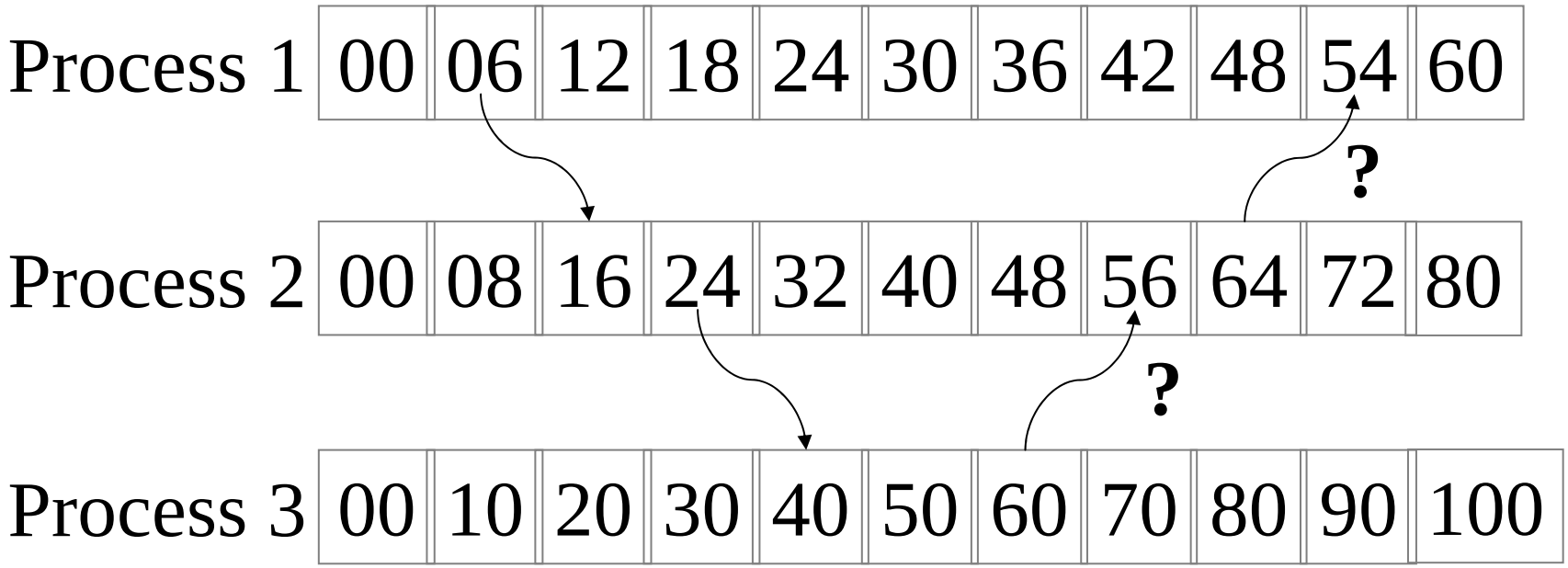
- Assign a *timestamp* (a number) *ts(a)* to each event *a* such that
  - $a \rightarrow b$ *implies* $ts(a) < ts(b)$
    - Question: not the other way around??

- Timestamps are of a global nature; they are derived from a local variable C_p called *a logical clock*

Logical clocks algorithm (Lamport)

- By construction of the ‘before’ relation
- Increment C_p upon each event *a*
  - $ts(a) := C_p ; C_p := C_p + 1$
- Assign C_p as the timestamp of a message that is sent.
  - $ts(m) := C_p ; send(m); ts(send(m)) := C_p ; C_p := C_p + 1$
- Upon receipt of message by process *q*
  - $receive(m); C_q := \mathbf{max}(C_q, ts(m)+1);$
    $ts(receive(m)) := C_q ; C_q := C_q + 1$

Execution Example



Corrected Execution

Process 1	00	06	12	18	24	30	36	42	48	70	76
Process 2	00	08	16	24	32	40	48	61	69	77	85
Process 3	00	10	20	30	40	50	60	70	80	90	100

Extension to Total Order

- Define some total order on processes
  - e.g. a numbering
- Extend timestamps with process number
  - to break ties in timestamps
- Now even concurrent events are ordered

Applications of logical clocks

- Totally ordered multicast, e.g. replicated database
  - each received message acknowledged by multicast
- Extend to causality relation
  - *'relation-to-define' implies* $a \rightarrow b$
  - use vector clocks... (check the errata of the book)
- Global-state algorithms

- Barbara Liskov, *Practical uses of synchronized clocks in distributed systems*, Distributed Computing, vol. 6, pp. 211-219, 1993

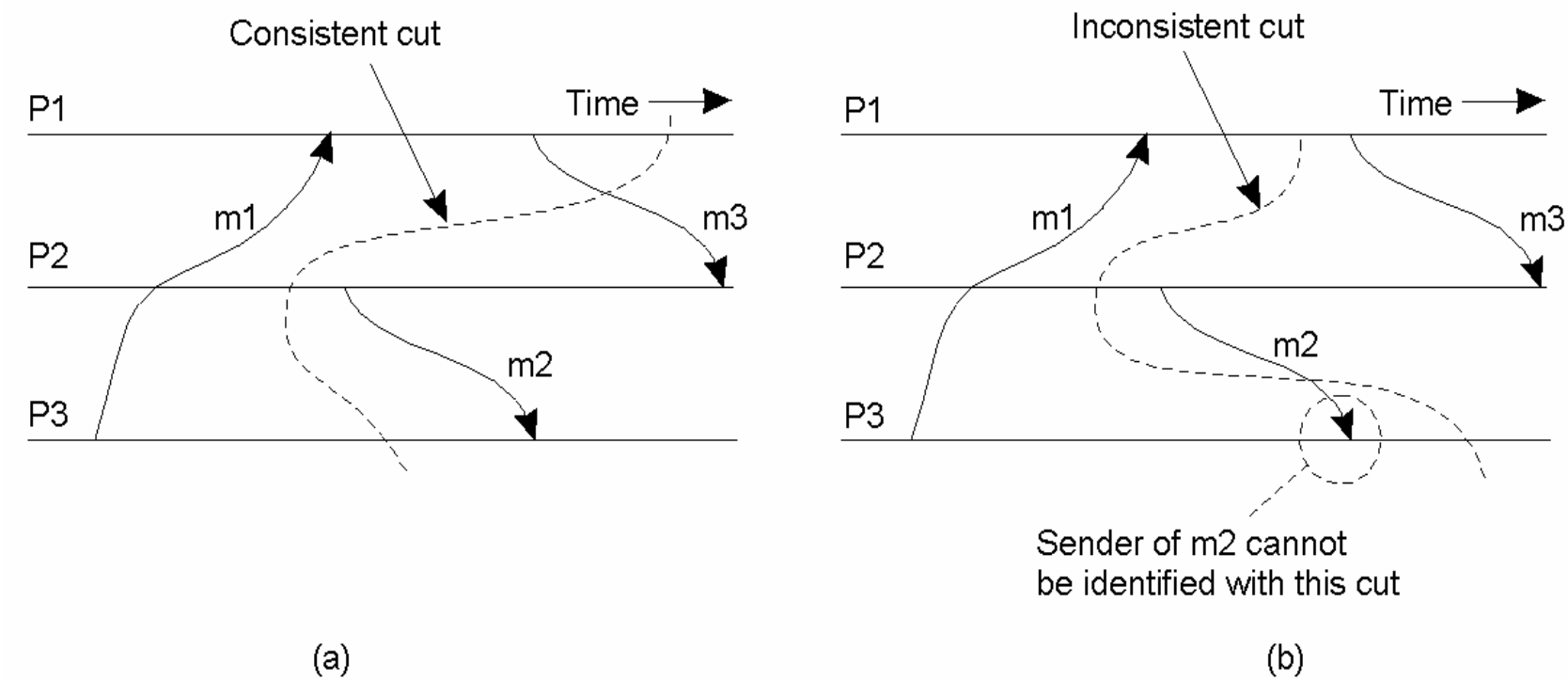
Contents

- Physical and logical clocks
- Distributed synchronization problems
  - global state
  - leader election
  - mutual exclusion
  - transactions (atomicity)

Distributed algorithms

- Global State (distributed snapshot)
  - consistent cut ('picture') of the system
  - local state + messages in transit
- Leader election
  - determine and agree upon a node with a special role
- Mutual exclusion
  - exclusive access to some resource
- Distributed transactions
  - a series of actions that should have an atomic behavior
    - all or nothing

Snapshot



a) A consistent cut (a *possible* distributed state)

b) An inconsistent cut

Consistent cut

- Cut
  - a record of process states and message queues observed locally at a certain point in time
- Consistent cut
  - for each message in the cut the sending of that message is recorded as well
- ...is a *possible* distributed system state
  - not necessarily one that occurred at any time
  - but one that could have occurred if we vary just processing speeds

Snapshot algorithm

- Assumptions
  - messages are not lost
  - buffered message passing using a queue per *channel* (= point-to-point communication link)
  - message order between any pair of communicating processes is maintained, i.e., channels preserve message order

Snapshot algorithm

- Start at *Q*
  - save local state
  - send *marker(Q)* on all outgoing channels

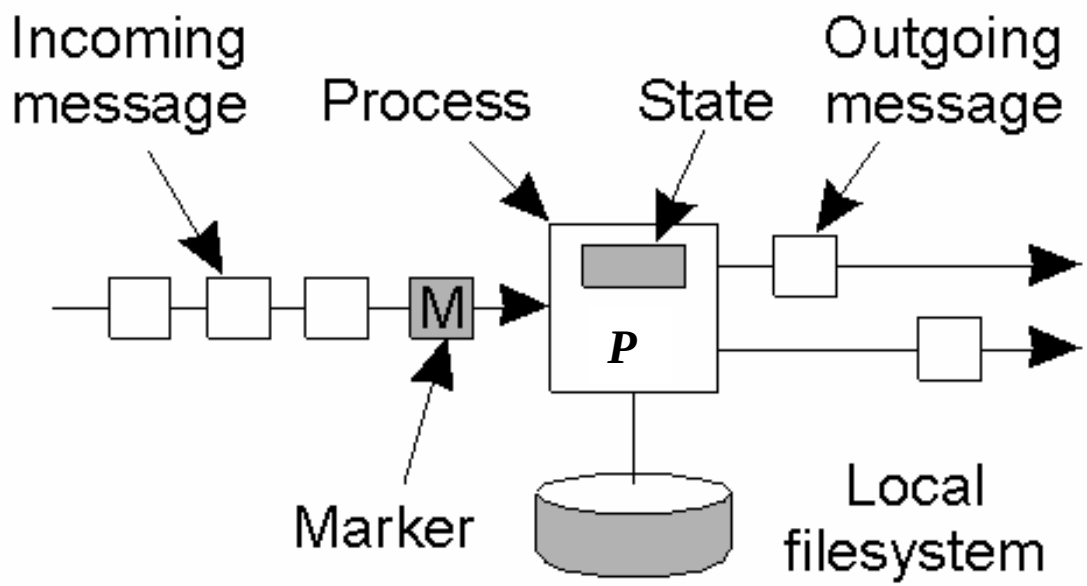
- Upon receipt of *marker(Q)* along channel *C*
  - if not started, perform start actions (see above)
  - otherwise: record channel state
    - received messages between last state recording and this received marker

Termination

- Each channel carries the marker precisely once
  - (at most) no process forwards the marker more than once
  - (at least) if channel *C, P→R*, does not carry the marker then *P* does not receive it. Assume that *P* is closest to *Q* with this property. Then no neighbors of *P* (of which one must be closer to *Q*) did send the marker to *P,* a contradiction

- Ready
  - after receipt of marker*(Q)* on all channels

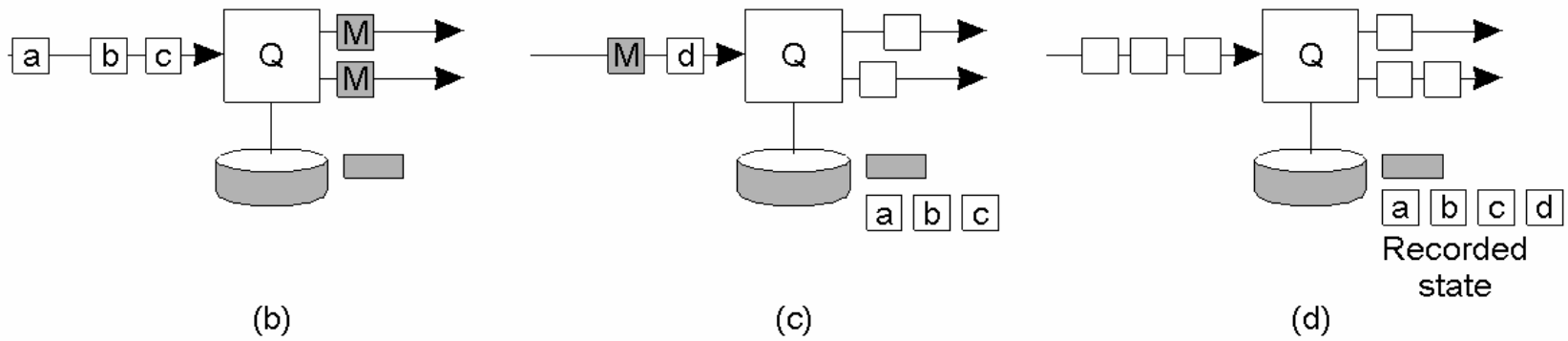
Snapshot algorithm



(a)

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

Snapshot algorithm



b) Process Q sends 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

Property of recorded state, *S**

- *S** is reachable from the distributed state S_Q that was assumed at the moment *Q* started the snapshot

- Any state reachable from *S** is reachable from S_Q as well

Contents

- Physical and logical clocks
- Distributed synchronization problems
  - global state
  - leader election
  - mutual exclusion
  - transactions (atomicity)

Leader election

- Select a partner in the group to have a special role
  - run a specific service
  - start or stop a distributed algorithm
    - e.g. in token rings
    - distributed garbage collection
  - store specific data
  - ...
- (distributed) agreement on this leader
- Election starts upon loss of leadership

Context, assumptions

- There is a way to distinguish processes
  - e.g. an identifier
- The set of possible processes is known through their identifiers
  - processes may or may not be ‘on’
  - this can change dynamically: crash/recovery of nodes
- Message can be sent between every pair of processes

Bully algorithm

- Process *p,* variables *holds_election, leader*

- Upon finding loss of leadership, or (re-)starting:
  - *holds_election := true*
  - send ELECTION to higher numbered processes

- Upon receiving an ELECTION from *q:*
  - if *q<p* send then
    - *holds_election := true*
    - send OK to *q*
    - send ELECTION to higher numbered processes
  - else *holds_election := false*

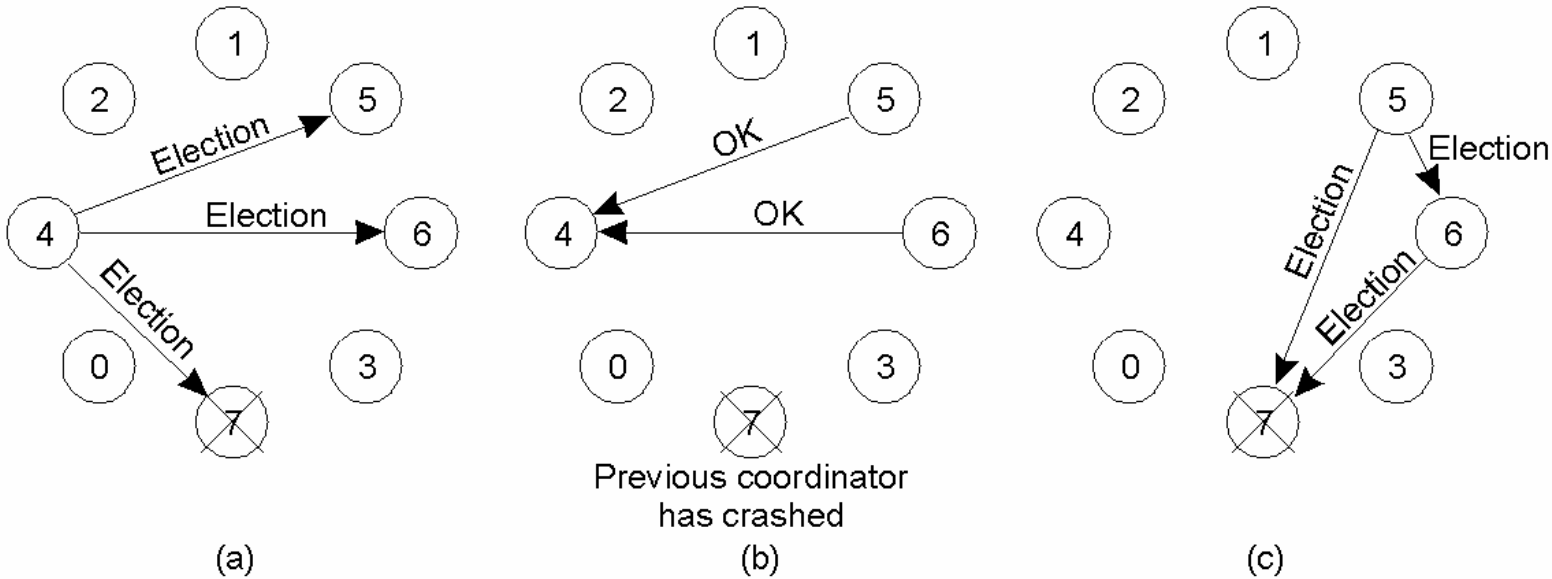
- Upon receiving an OK:
  - *holds_election := false*

Bully algorithm (cnt'd)

- Upon timeout after sending ELECTION:
  - if *holds_election then*
    - *leader := p*; *holds_election := false*
    - send *(*COORDINATOR*, p)* to all other processes

- Upon receiving (COORDINATOR,*q*)
  - leader := q // Bully

Bully Algorithm - diagrams

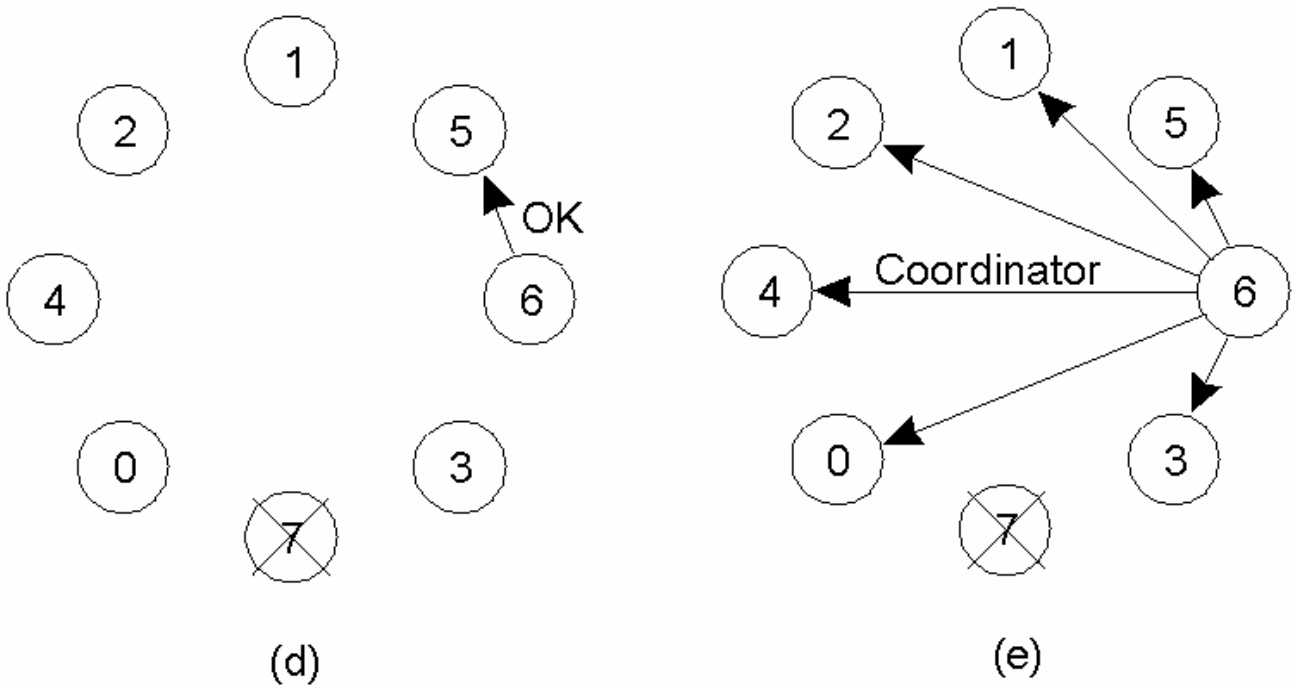


- Process 4 holds an election
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

Bully Algorithm - diagram

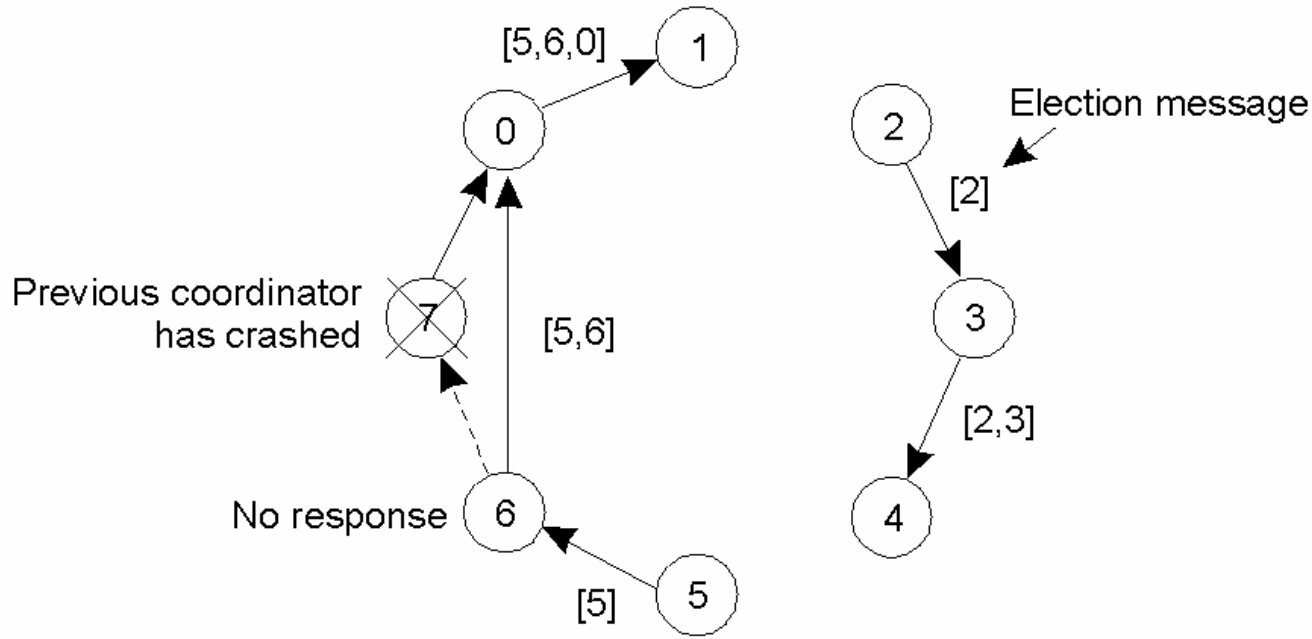
d) Process 6 tells 5 to stop

e) Process 6 wins and tells everyone



A Ring Algorithm

- 2 rounds: election, coordinator
  - coordinator: max. election round



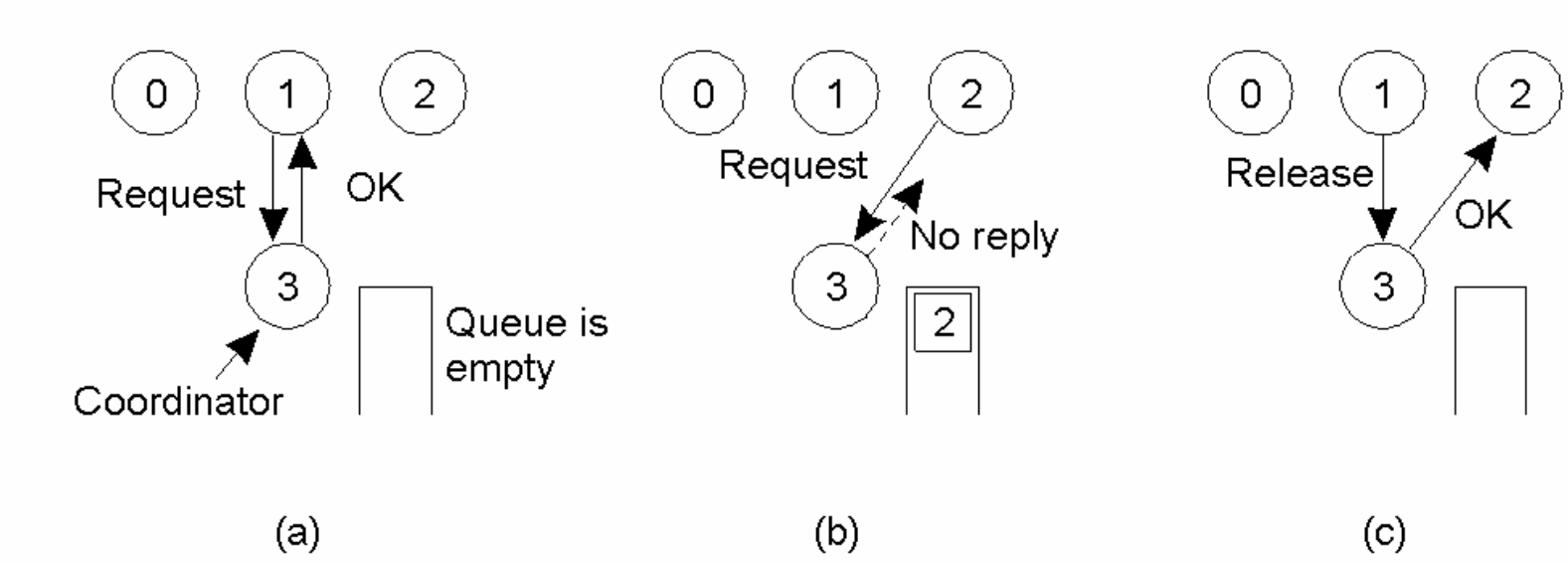
Contents

- Physical and logical clocks
- Distributed synchronization problems
  - global state
  - leader election
  - mutual exclusion
  - transactions (atomicity)

Mutual exclusion

- Critical sections – shared resources
- Notes
  - rather centralized problem
  - often combined with leader election

A Centralized Algorithm



a) Process 1 asks coordinator permission to. Permission is granted

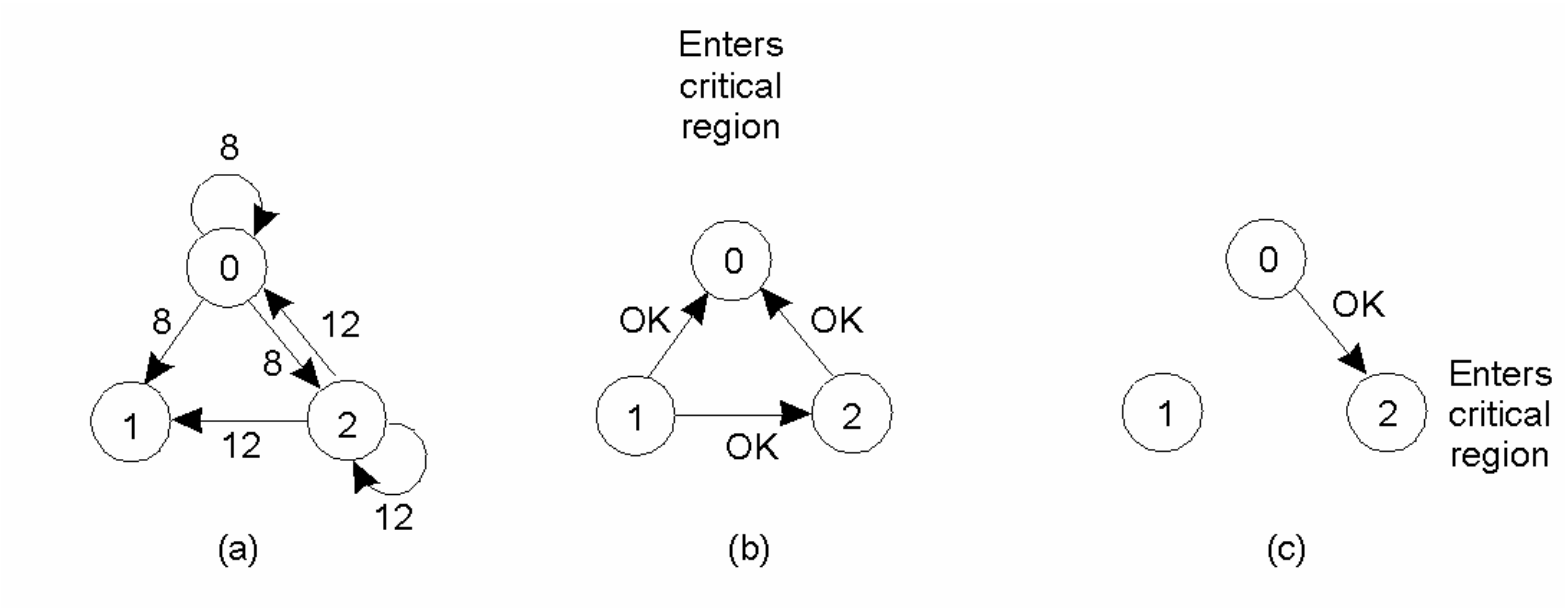
b) Process 2 asks permission to enter the same critical region. The coordinator does not reply.

c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2

Distributed algorithm

- Entering the critical section
  - send a request to all partners (must be known)
  - await acknowledge from all these
- Response to a request
  - ok, if not interested self or if request smaller than own request
  - queue, otherwise
- Ordering
  - total event order (e.g. based on Lamport's clocks)

A Distributed Algorithm



a) Two processes want to enter the same critical region at the same moment.

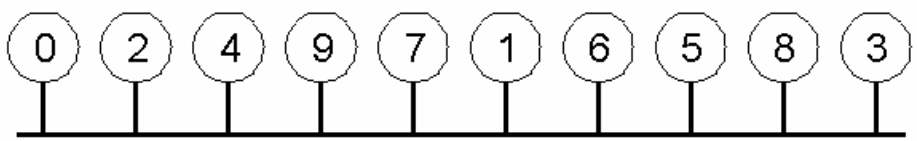
b) Process 0 has the lowest timestamp, so it wins.

c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

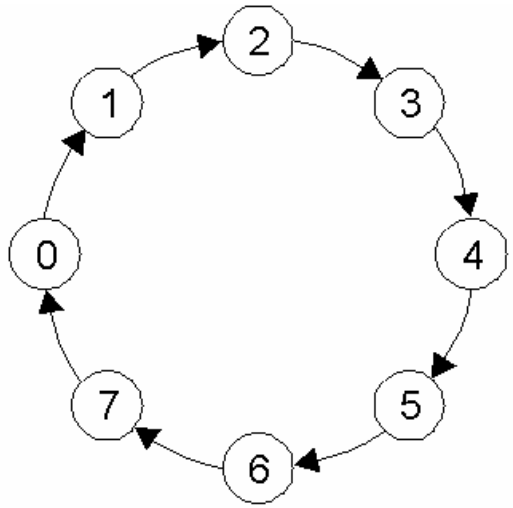
A Token Ring Algorithm

a) An unordered group of processes on a network.

b) A logical ring (overlay network)



(a)



(b)

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

Contents

- Physical and logical clocks
- Distributed synchronization problems
  - global state
  - leader election
  - mutual exclusion
  - transactions (atomicity)

Transactions

- **Transaction:** sequence of actions, combined into a single operation with the ACID properties
  - **atomic**, i.e., taken ('committed') or not taken, indivisible
    - reserving an airline seat
  - **consistent**, maintaining system invariants
    - e.g. no seats are lost
  - **isolated**, concurrent transactions appear serialized – transient states not observable
    - e.g. it becomes my seat or your seat
  - **durable**, a *committed* transaction is persistent
    - reservation indeed works

- **NOTE:** not completely independent notions

Example primitives

Primitive	Description
BEGIN_TRANSACTION	Make the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Example

- Operational (not syntactical) view on an airline reservation transaction

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi;
END_TRANSACTION
```

(a)

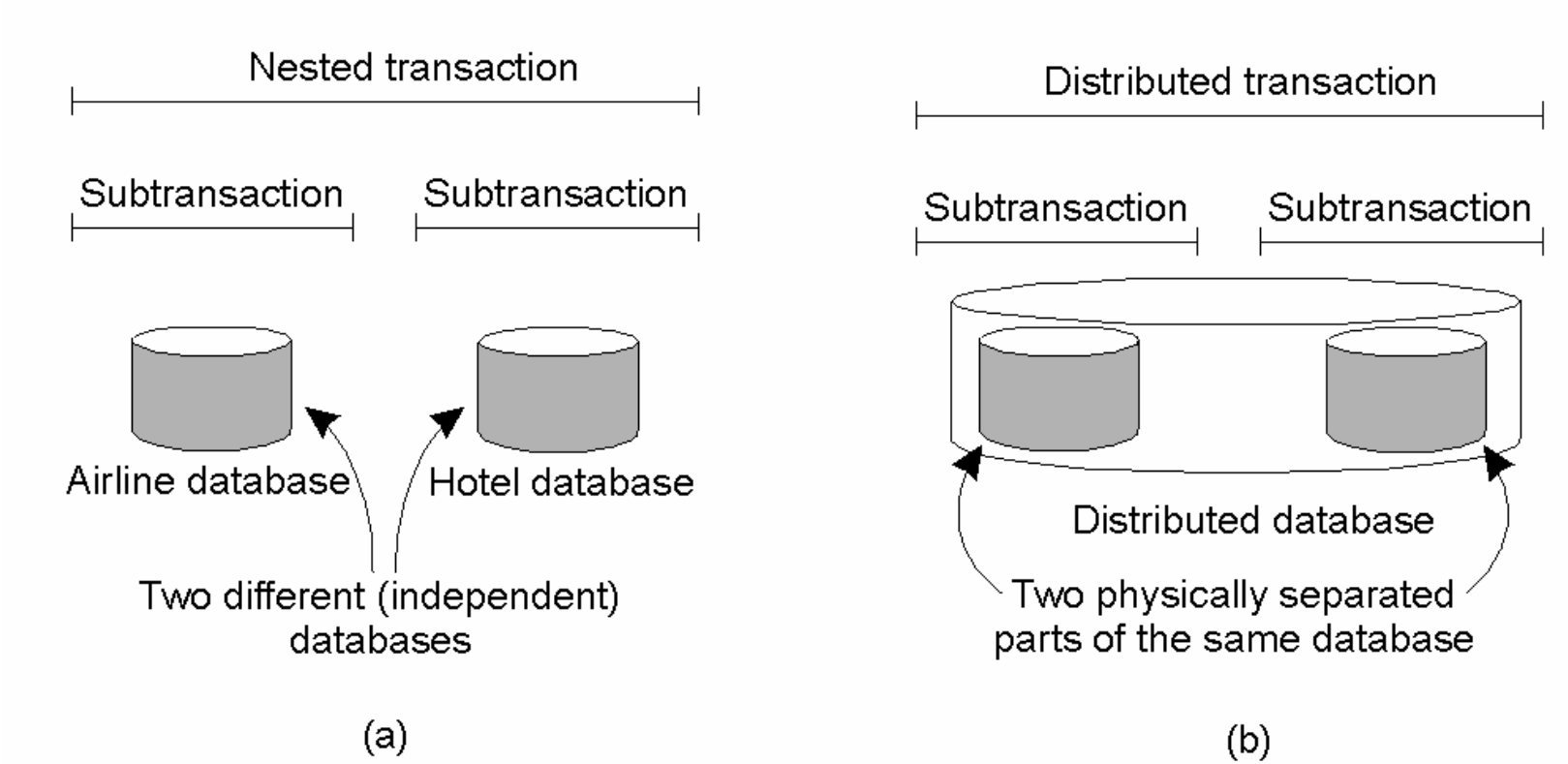
```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi full =>
ABORT_TRANSACTION
```

(b)

Transaction types

- Flat transactions
  - just ACID
  - drawbacks
    - commitment may take long
    - partial commitment may be meaningful
- Nested transactions
  - contains sub-transactions
    - logical work division
    - durability only for top-level
- Distributed transactions
  - (flat) transactions across multiple machines
    - physical (e.g. distributed data) work division

Nested and distributed



Implementation aspects

- Atomicity – roll-back possibility
  - local workspace
  - write-ahead log

- Atomicity and isolation: no interference
  - **serializability:** a trace or schedule consisting of actions of different transactions is serializable iff it is consistent with executing these transactions in some order
    - read/write and write/write conflicts
  - concurrency control
    - optimistic: upon finding occurrence of conflict, abort
    - pessimistic: always avoid conflicts

Serializable

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 2;
END_TRANSACTION
```

(b)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 3;
END_TRANSACTION
```

(c)

Schedule 1	x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3	Legal
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;	Legal
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;	Illegal

(d)

Private workspace, optimistic

- Workspace of process *P* doing the transaction
  - initialized upon transaction start
    - copy data, or through reference
  - upon read:
    - read local data when copied or written previously by *P*
    - follow reference otherwise
      - if modified since last read: abort transaction
  - upon write:
    - write into local copy
  - upon completion, perform as a critical section:
    - if global info corresponding to local info was modified since start of transaction: abort transaction
    - else: commit: write local info into global
  - ...need timestamps, for detection

Writeahead Log

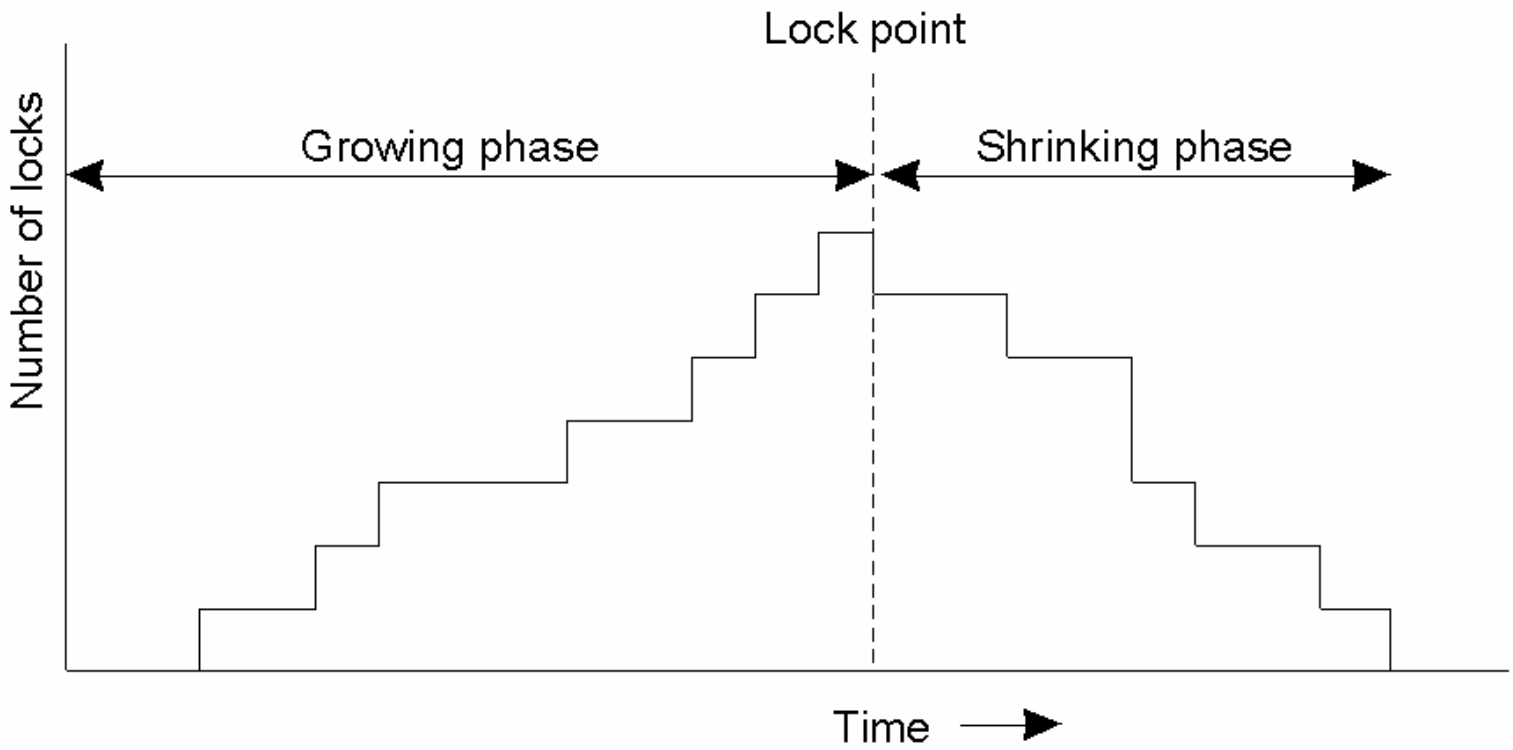
x = 0;	Log	Log	Log
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0 / 1]	[x = 0 / 1]	[x = 0 / 1]
y = y + 2		[y = 0/2]	[y = 0/2]
x = y * y;			[x = 1/4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

- Record previous and new value, as well as the name of the transaction (latter not shown)
- Admits
  - roll-back
  - detecting conflict

Pessimistic: locking

- read locks and write locks
  - no conflicting locks are allowed
- 2-phase locking protocol
  - a transaction locks an item before using it
    - conflicting locks lead to delay of the last in line
  - a transaction never locks any item after having released an item (or is not permitted to...)
- **Property**: any schedule resulting from this protocol is serializable
- Deadlock avoidance: avoid cycles in acquisition graph

Two-Phase Locking - picture



Timestamp Ordering