Distributed Operating Systems

Synchronization – general purpose algorithms

dr inż. Adam Kozakiewicz

akozakie@ia.pw.edu.pl

Institute of Control and Information Engineering

Warsaw University of Technology

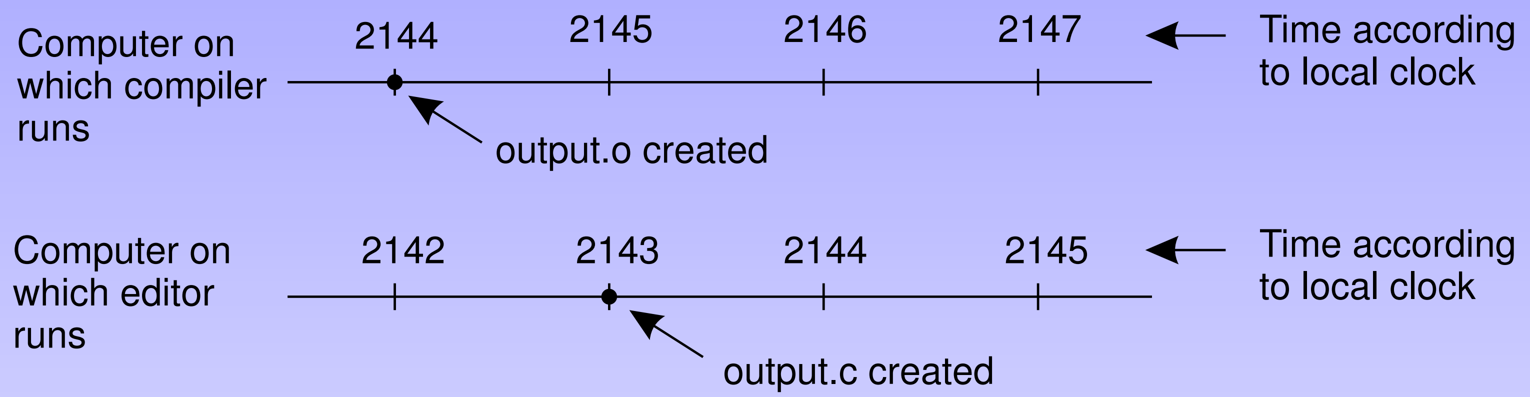
Synchronization

1. Clock synchronization
2. Logical clocks
3. Global state (distributed snapshot)
4. Election algorithms
5. Mutual exclusion

Synchronization

Setting the time order of a set of events caused by concurrent processes.

Clock Synchronization


Computer on which compiler runs
2144
2145
2146
2147
Time according to local clock
output.o created
Computer on which editor runs
2142
2143
2144
2145
Time according to local clock
output.c created

Problem: clocks are not perfect. When each machine has its own clock, an event that occured after another event may still get an earlier timestamp.

A simple example: calling `make` in a system, where compilation is actually done on a separate machine (compile farm).

Timers – Real Time Clocks

- √ timers work by counting oscillations in a crystal,
- √ whenever the counter reaches 0, a timer interrupt is generated, incrementing the holding register,
- √ frequency of the crystals is not a constant – different crystals can have different frequencies,
- √ the frequency of a crystal depends on external conditions (temperature, etc.),
- √ the differences cause an increasing **clock skew**.

Time Definitions

Solar time defined by observations of the Sun

- **Solar day** the time between transits of the Sun
- **Solar second** 1/86400 of the solar day

  - √ Earth's orbit is elliptical – this affects observations
  - √ Earth's rotation speed changes in time
    - ⋆ tidal forces from the moon slow it down in long term
    - ⋆ in short term the changes are not predictable (movement of tectonic plates, gravity of other celestial bodies, etc.)
  - √ Mean solar second, mean solar time – based on a mathematical construct eliminating seasonal changes of apparent solar time

sidereal time defined by observations of stars (including precession)

- √ Not influenced by the elliptic orbit of the Earth
- √ Only used in astronomy

Atomic time defined by atomic clocks (counting seconds)

Time Definitions – Atomic Time

- √ Atomic clocks – based on counting the natural oscillation of caesium atoms
- √ Theoretically almost perfect, according to the definition of a second in the SI system:

  A second is the duration of 9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium-133 atom
- √ Polish standard is the same as SI (rozporządzenie Rady Ministrów z dn. 30.11.2006 w sprawie legalnych jednostek miar, Dz.U. 2006 nr 225 poz. 1638, par. 3 ust. 3)
- √ In reality cesium atoms are in compounds, not at 0K, not in ground state (Earth's magnetic field!) and the measurement may not be perfect, relativistic time dilation further complicates matters
- √ Not perfect, but extremely accurate – 10^{-15}, about 0.1ns in a day

Time Standards

UT0 mean solar time at meridian 0 (Greenwich)

UT1 *Universal Time*, UT0 with a correction for the polar precession

UT1R, UT2, UT2R further smoothing of UT1, correcting for seasonal variations (UT2, UT2R) and tidal effects (UT1R, UT2R), rarely used

GMT *is not* a standard anymore, broken up into UT1 (direct successor) and UTC, the abbreviation now only identifies the Greenwich time zone, any other use is erronous

TAI atomic time, the average of the local atomic times of about 50 atomic clocks worldwide

UTC coordinated universal time, the most important in practice

UTC Standard

- √ The acronym UTC is not expandable – it is a compromise between CUT, *Coordinated Universal Time* and TUC, *Temps Universel Coordonné*
- √ TAI, synchronized with UT1
- √ differs from TAI by an integer number of seconds (currently 33)
- √ whenever the difference between UTC and UT1 reaches 0.9s, a *leap second* is introduced on one of two possible moments in a year:
  - ⋆ June 30, 23:59:60
  - ⋆ December 31, 23:59:60
  - ⋆ a negative leap second is also possible (never occured so far), on one of the two days 23:59:58 will be directly followed by 00:00:00
  - ⋆ if more than two leap seconds are required in a year, they can be scheduled at the end of March 31 or September 30 (never happened so far)
- √ can be obtained using radio or satellite (a short pulse at the start of each second), of course transmission lowers accuracy

Clock Synchronization

Assume we can synchronize directly with UTC. How often should we do it?

- √ denote the time according to the local clock by $C(t)$ (t – UTC time)
- √ for a perfectly accurate clock $Cp(t) = t$, so $dC/dt = 1$; real clocks can be fast or slow
- √ we can estimate an upper bound ρ on the clock skew in a given distributed system $(1 - \rho \leq dC/dt \leq 1 + \rho)$
- √ synchronizing with UTC every $\delta/(2\rho)$ seconds, guarantees that:
  - ★ the difference between the local time and UTC will not exceed $\delta/2$ seconds, or
  - ★ the difference between the local times on any two clocks in the system will not exceed δ seconds

  the latter goal can be achieved without access to UTC – any local time (or average of many) will do

Clock Synchronization – Two Approaches

Principle I Every machine asks a time server (preferably equipped with UTC receiver) for time at least every $\delta/(2\rho)$ seconds.

- √ Round trip delays must be well estimated!
- √ Very accurate with a UTC receiver; otherwise the global time depends entirely on the time server's clock.

Principle II The local times of all machines are averaged and all machines correct their time towards the average.

- √ A time server is optional.
- √ All clocks in the system are synchronized (probably).
- √ Using many clocks should increase the accuracy – important when UTC is not available.

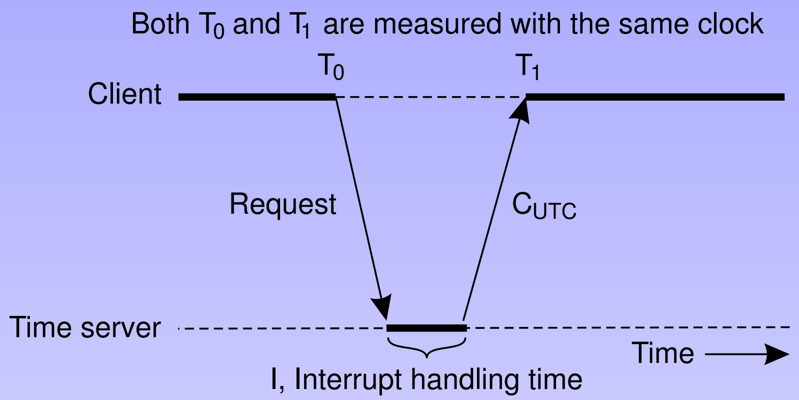
√ Time should never be set back! Only large corrections are introduced like that, normally smooth adjustments are used (slowing down or accelerating the local clock).

Clock Synchronization Algorithms

Clock synchronization algorithms:

- √ Cristian's algorithm
- √ Berkeley algorithm
- √ Averaging algorithms

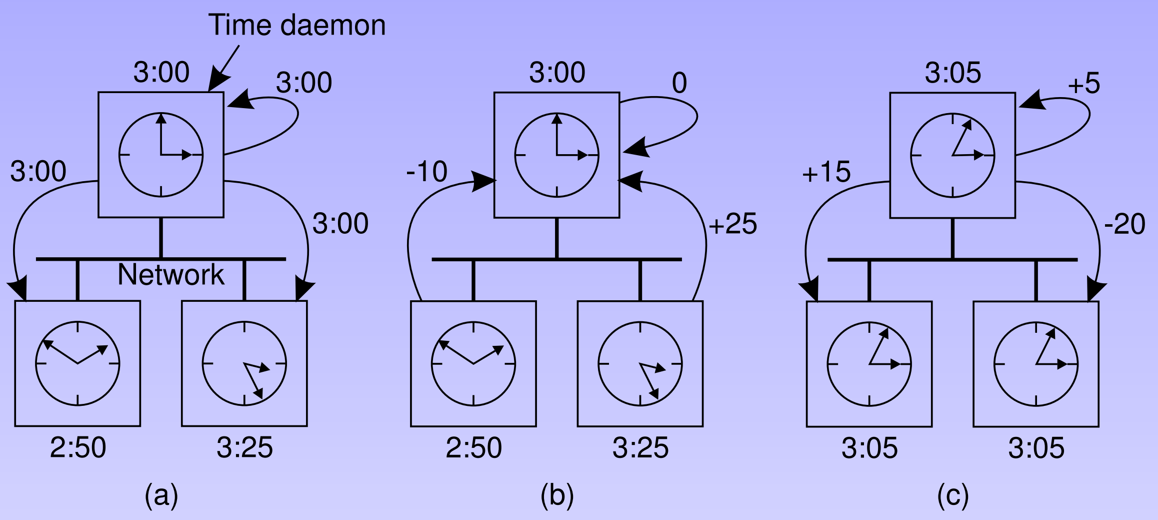
Cristian's Algorithm


Both T0 and T1 are measured with the same clock
T0
T1
Client
Request
CUTC
Time server
I, Interrupt handling time
Time

Get the time from the tiime server (several times)

- √ received time corresponds to $(T1 - T0)/2$ local time (approximately)
- √ messages with $T1 - T0$ above threshold are discarded as victims of network congestion
- √ the fastest response (lowest $T1 - T0$) is the most accurate and is used to correct local time

Berkeley Algorithm



1. the time server asks all machines for current local time or gives its own local time and asks for the difference
2. the time server collects the information and computes the average time
3. the time server sends corrections to all machines

Averaging algorithms

- √ fully distributed (previous methods were highly centralized)
  - ⋆ fixed-length resynchronization intervals: $T0 + (i+1)R$, where R is a system parameter
  - ⋆ machines broadcast their own local time, collect information sent by the others, compute the average and correct their clocks
  - ⋆ variant – add correction for propagation time for each message
- √ Internet uses the Network Time Protocol (**NTP**), accuracy in the range of 1-50ms.

Logical clocks

- √ UTC is not always essential,
- √ the important thing for a distributed system is usually internal consistency, not being close to the real time,
- √ time (in the calendar/wall clock sense) is very often completely unnecessary, the point is to agree on the order of events in the system – this is the task for **logical clocks**,
- √ **Lamport's algorithm** is used to synchronize logical clocks,
- √ **vector timestamps** are an extension of the Lamport's algorithm ensuring causality of the ordering.

The Happened-Before Relation

The **happened-before** relation in a set of events in a distributed system is the smallest relation satisfying:

- √ if a and b are two events in the same process and a comes before b, then $a \rightarrow b$.
- √ if a is the sending of a message and b is the receipt of that message, $a \rightarrow b$.
- √ the relation is transitive, that is if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

This relation is a *partial ordering* of events in a system with concurrently operating processes.

Concurrent events

This relation says nothing about which of two unrelated events happened first.

Logical Clocks (1)

How can we maintain a global system of event ordering consistent with the happened-before relation?

Solution: attach a time-stamp $C(e)$ to each event e, so that the following properties hold:

P1 If a and b are two events in the same process and $a \rightarrow b$, then $C(a) < C(b)$.

P2 If a is the sending of a message m and b is the receipt of message m, then $C(a) < C(b)$.

A perfectly accurate global clock generates time-stamps constistent with these properties, but we know it's impossible to keep perfect accuracy. How to create good time-stamps without it?

Solution: maintain a consistent set of logical clocks, one per process, and instead of trying to keep them in sync, focus on enforcing the happened-before relation.

Logical Clocks (2)

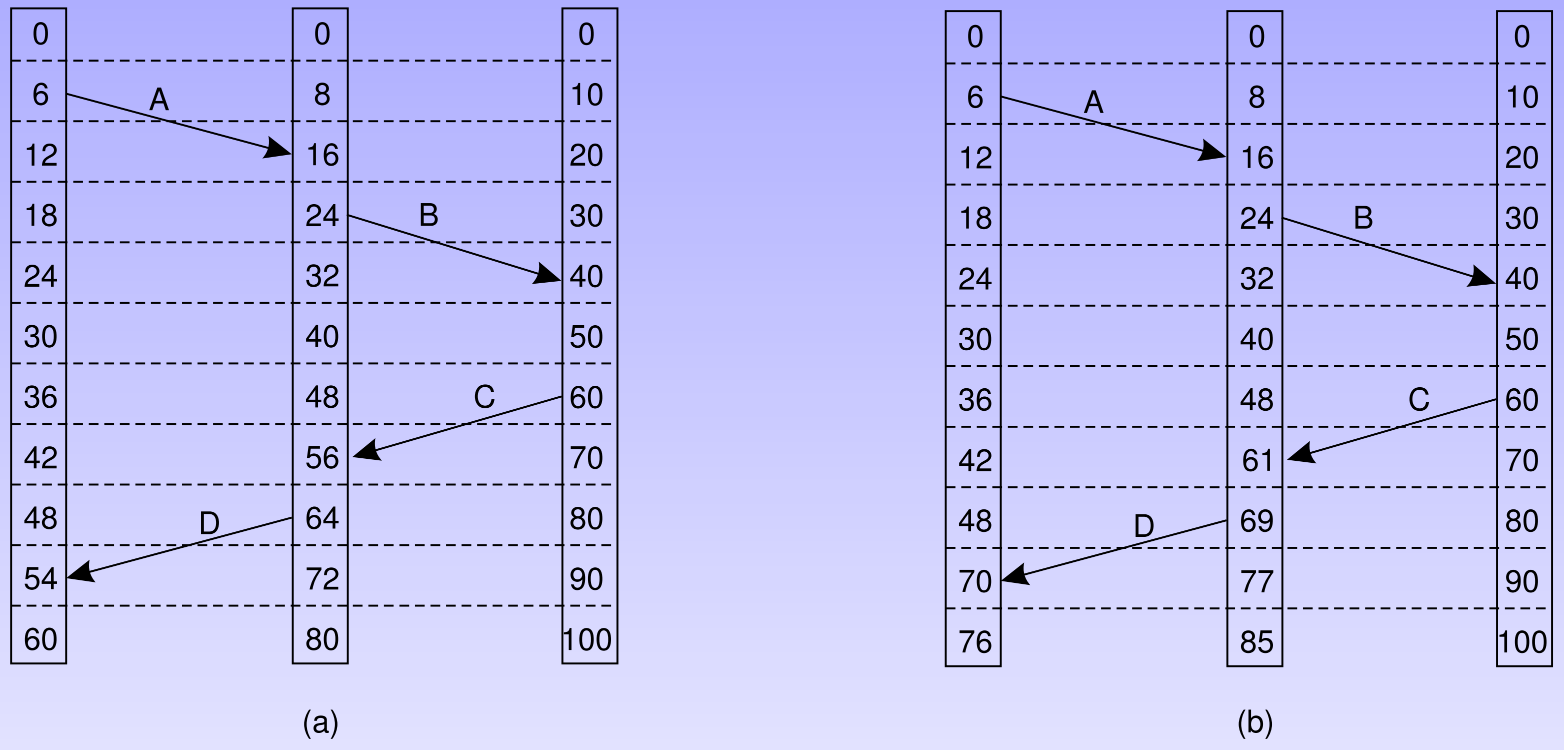
Every process P_i maintains a **local counter** C_i and adjusts it according to the following rules:

1. For every successive event within P_i increment the counter by 1. (Definition of event?)
2. Add a time-stamp $T_m = C_i$ to every message m sent by process P_i. Note: sending of a message is an event, so time-stamp should be attached after incrementation!
3. Whenever a message m with time-stamp T_m is received by process P_j, P_j increments its local counter C_j as follows:

$$C_j := max\{C_j + 1, T_m + 1\}.$$

- √ property **P1** is satisfied by rule 1,
- √ property **P2** is satisfied by rules 2 and 3.

Logical Clocks (3)



Lamport's algorithm example

Total Ordering of Events Using Lamport's Time

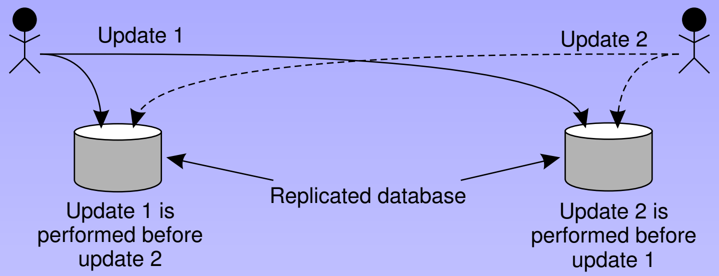
Lamport's algorithm allows two events to happen at the same time (two different messages sent by different processes can have the same time-stamp). Total ordering is possible by attaching the sender's process number to the time-stamp.

If in process P_i the time-stamp of event e is $C_i(e).i$

Then $C_i(a).i$ is before $C_j(b).j$ if and only if:

- √ $C_i(a) < C_j(a)$, **or**
- √ $C_i(a) = C_j(b)$ and $i < j$.

Example: Totally-Ordered Multicasting


Update 1
Update 2
Replicated database
Update 1 is performed before update 2
Update 2 is performed before update 1

- √ Total order is implemented using (extended) Lamport's algorithm.
- √ Every message is timestamped with the sender's local logical time.
- √ Received message is put into a local queue, ordered by time-stamps, receiver multicasts an acknowledgement to others.
- √ A message leaves the queue for processing only if it's at the head of the queue and has been acknowledged by all processes.

Vector Timestamps (1)

- √ Lamport timestamps do **not** guarantee that if $C(a) < C(b)$, then a indeed happened before b. This requires **vector timestamps**.
  - ⋆ every process P_i maintains an array of counters $V_i[1 \ldots n]$, where $V_i[j]$ denotes the number of events in process P_j that the process P_i already knows about.
  - ⋆ whenever P_i sends a message m, it increments its own counter, $V_i[i]$, by 1 and sends the entire array V_i with the message m as the vector timestamp $vt(m)$.
- √ timestamp vt of m tells the receiver how many events in other processes have preceded m, in the sense that the sender knew about them and they may have causally influenced the message m – therefore the receiver needs to know about those events to have a proper context to understand m.

Vector Timestamps (2)

- √ when a process P_j receives a message m from P_i with a timestamp $vt(m)$, it:
  - ⋆ updates all counters $V_j[k]$ to $\max\{V_j[k], vt(m)[k]\}$,
  - ⋆ increments its own counter $V_j[j]$ by 1.

Vector Timestamps (3)

- √ Causal delivery of broadcasted messages can be assured with vector timestamps only incremented for the events of sending messages and with delivery queues. Message m is delivered to process P_j only if:
  - ★ $vt(m)[i] = V_j[i] + 1$ **and**
  - ★ $vt(m)[k] \leq V_j[k]$ for all $k \neq i$,

  that is when P_j already received all earlier messages from P_i, and all messages from other processes that process P_i received before sending m. P_j may already have received some new messages from other processes – that is not important, as it doesn't affect the interpretation of m.

Example:
$\mathbf{V3} = [0, 2, 2]$, $\mathbf{vt(m)} = [1, 3, 0]$ – what does P_3 know? If it receives m from P_1, what will it do?

Causal Delivery of Messages Example – Setting

Assumptions:

- √ multicast – all messages are sent to everyone except the sender,
- √ messages from one process are received in the same order by all processes,
- √ reliable sending mechanism,
- √ order of messages from different senders in not enforced.

Sender:

1. Multicasts the message.

Receiver:

1. Receives the message (communication layer).
2. Processes the message (actual delivery to the process from the communication layer).

Causal Delivery of Messages Example – Rules

Let:

vt_m – vector timestamp of message m,

V_P – current vector of process P.

Rules

Every message m sent by process P includes a vector timestamp $vt(m)$ defined as follows:

1. $vt_m[P] = V_P[P] + 1$,
2. $vt_m[X] = V_P[X]$ for every $X \neq P$.

Message m received from process P is queued. It is delivered to process Q when:

1. $vt_m[P] = V_Q[P] + 1$
2. $vt_m[X] \leq V_Q[X]$ for every $X \neq P$.

After message m is delivered, Q updates its vector as follows:

1. $V_Q[X] = max\{V_Q[X], vt_m[X]\}$

Causal Delivery of Messages Example – Problem

Three processes, A, B i C, with initial vectors $V_A = V_B = V_C = (0, 0, 0)$.

Scenario:

1. A multicasts $m1$,
2. B multicasts $m2$ in response to $m1$,
3. C is an observer.

Goal:
Message $m2$ should be delivered to C **after** $m1$ is delivered. If $m2$ arrives at C first, it must be queued until $m1$ arrives.

Causal Delivery of Messages Example (1)

A sends $m1(0+1,0,0) = m1(1,0,0)$ and increments $V_A[A]$, so $V_A = (1,0,0)$,
B receives $m1(1,0,0)$ from A,

> $V_B = (0,0,0)$, $vt_{m1} = (1,0,0)$,
> $m1$ will be immediatly delivered, because:
>
> > $vt_{m1}[A] = V_B[A] + 1$,
> > $vt_{m1}[X] <= V_B[X]$ for every $X \neq A$.
>
> after delivery of $m1$ the new value of V_B is $V_B = (1,0,0)$.

B replies $m2(1,0+1,0) = m2(1,1,0)$ and increments $V_B[B]$, so $V_B = (1,1,0)$,
A receives $m2(1,1,0)$ from B,

> $V_A = (1,0,0)$, $vt_{m2} = (1,1,0)$,
> $m2$ will be delivered immediately, because:
>
> > $vt_{m2}[B] = V_A[B] + 1$,
> > $vt_{m2}[X] <= V_A[X]$ for every $X \neq B$.
>
> after delivery of $m2$ the new value of V_A is $V_A = (1,1,0)$.

Causal Delivery of Messages Example (2)

C receives $m2(1,1,0)$ from B,

> $V_C = (0,0,0)$, $vt_{m2} = (1,1,0)$,
> delivery of $m2$ will be **postponed**, because:
>
> > $vt_{m2}[A] > V_C[A]$ and $A \neq B$.

Comment:
The vector timestamp suggests, that at the moment of sending $m2$ B knew about one message from A. C hasn't received anything from A so far.

The missing message from A may be important for correct interpretation of $m2$ (indeed it is so, $m2$ is a reply and may need context – but C cannot be sure). Therefore $m2$ cannot be delivered at the moment – C must wait for $m1$.

Causal Delivery of Messages Example (3)

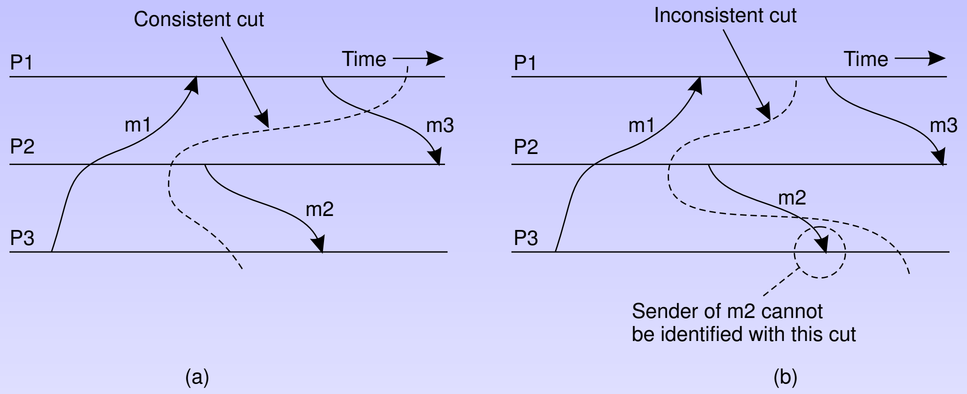
C receives $m1(1, 0, 0)$ from A,

> $V_C = (0, 0, 0)$, $vt_{m1} = (1, 0, 0)$,
> $m1$ will be delivered immediately, because:
>
> > $vt_{m1}[A] = V_C[A] + 1$,
> > $vt_{m1}[X] <= V_C[X]$ for every $X \neq A$.
>
> after delivery of $m1$ the new value of V_C is $V_C = (1, 0, 0)$.
> Since the input queue of C is not empty, we check its state;
> $m2$ can be delivered now, because:
>
> > $V_C = (1, 0, 0)$, $vt_{m2} = (1, 1, 0)$,
> > $vt_{m2}[C] = V_C[C] + 1$,
> > $vt_{m2}[X] \leq V_C[X]$ for every $X \neq C$.
>
> after delivery of $m2$ the new value of V_C is $V_C = (1, 1, 0)$.

After two multicast messages $A \rightarrow BC$ and $B \rightarrow AC$, the time vectors in all processes are: $V_A = V_B = V_C = (1, 1, 0)$

Global System State (1)

Collecting a global system state (**distributed snapshot**) is important in most distributed systems – in case of a malfunction it allows us to restart the system from a given moment, instead of starting it from the beginning . The distributed snapshot consist of a complete set of local snapshots of individual processes and a complete set of messages currently in transit.



A distributed snapshot must reflect a **consistent state**. In figure (b) the snapshot of the third process is done after receiving message $m2$, which hasn't been sent yet when the sender made its snapshot – restarting from the snapshot will probably result in duplicating that message.

Global System State (2)

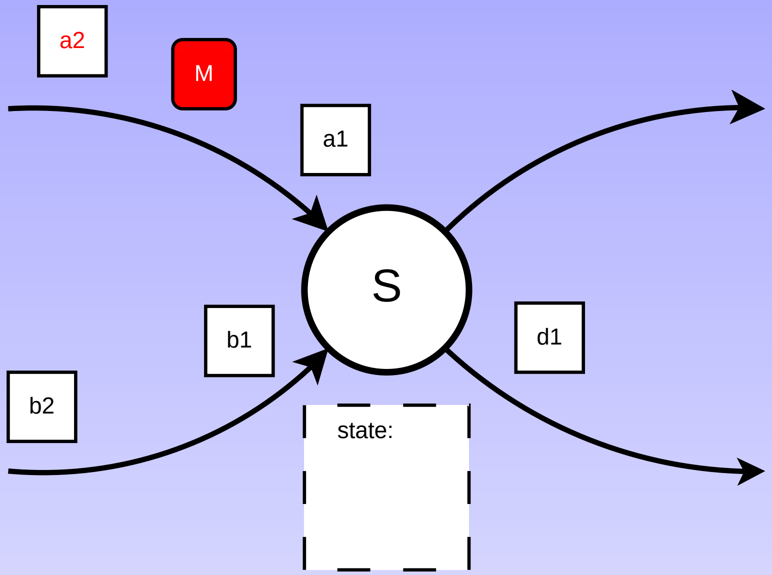
- √ the system is a collection of distributed processes connected through unidirectional point-to-point communication channels
  - ⋆ the less channels we have, the better, but even bidirectional channels fit into this model as pairs of channels
- √ a distributed snapshot can be initiated by any process P
  - ⋆ if P is not predefined, the markers used in the algorithm should contain its identifier, to avoid errors caused by collection being started concurrently by two processes

Global System State (3)

1. P starts by recording its own local state,
2. P sends a marker through each of its outgoing channels,
3. when Q receives a marker from channel C, its action depends on whether it has already recorded its local state:
   - √ *not yet recorded:* Q records its local state and sends the marker through all outgoing channels,
   - √ *already recorded:* a marker on channel C means that the sender just recorded its state, so Q records all messages received from channel C since its own local state was recorded,
4. the operation ends for a process (including P) when it has received markers from all its incoming channels.

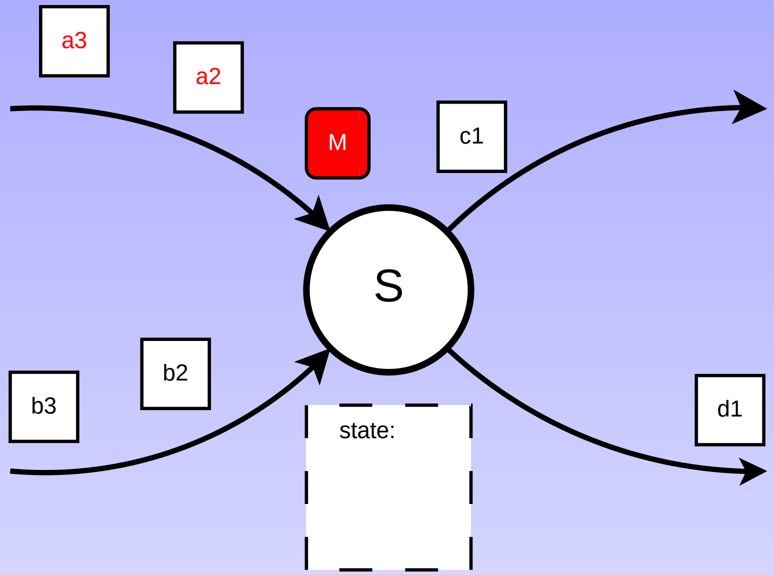
Global System State – Example

Distributed snapshot, including channel state:



Global System State – Example

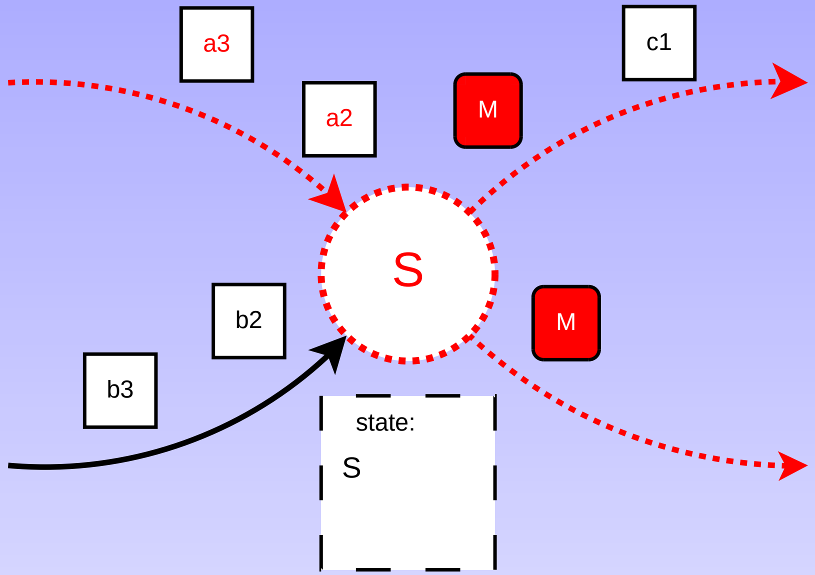
Distributed snapshot, including channel state:



1. Process S recieves its first marker – stops processing to make a local snapshot.

Global System State – Example

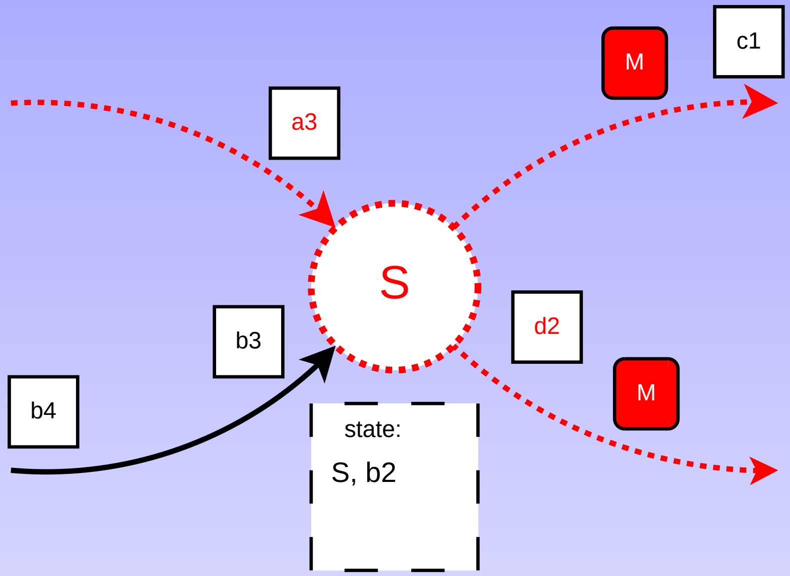
Distributed snapshot, including channel state:



1. Process S made a local snapshot, now it sends markers through all outgoing channels and resumes processing.

Global System State – Example

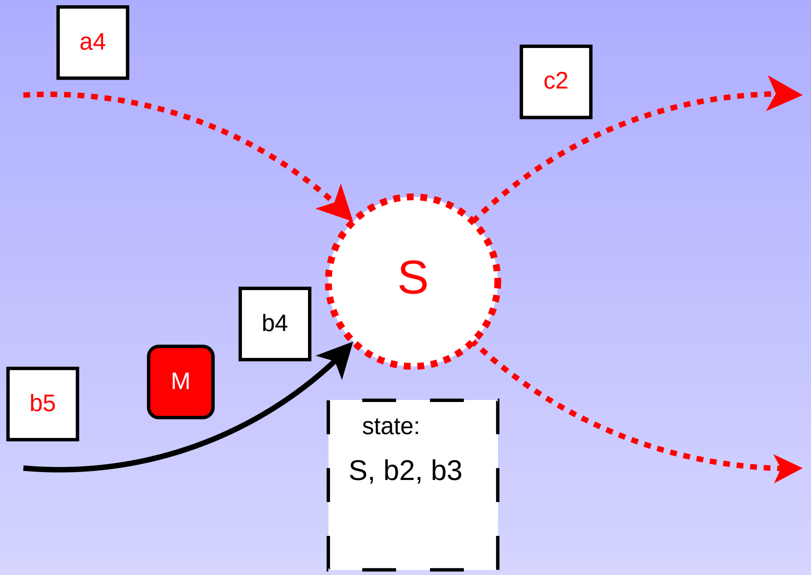
Distributed snapshot, including channel state:



1. *Process S recieves its first marker and makes a local snapshot.*
2. S records all messages from incoming channels on which no marker has been received (b). Messages following a marker (channel a) are ignored.

Global System State – Example

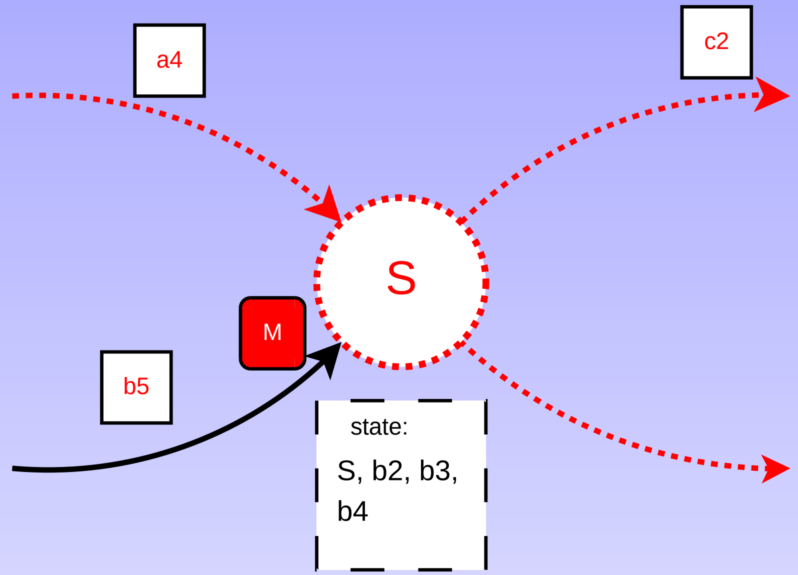
Distributed snapshot, including channel state:



1. *Process S recieves its first marker and makes a local snapshot.*
2. S records all messages from incoming channels on which no marker has been received (b). Messages following a marker (channel a) are ignored.

Global System State – Example

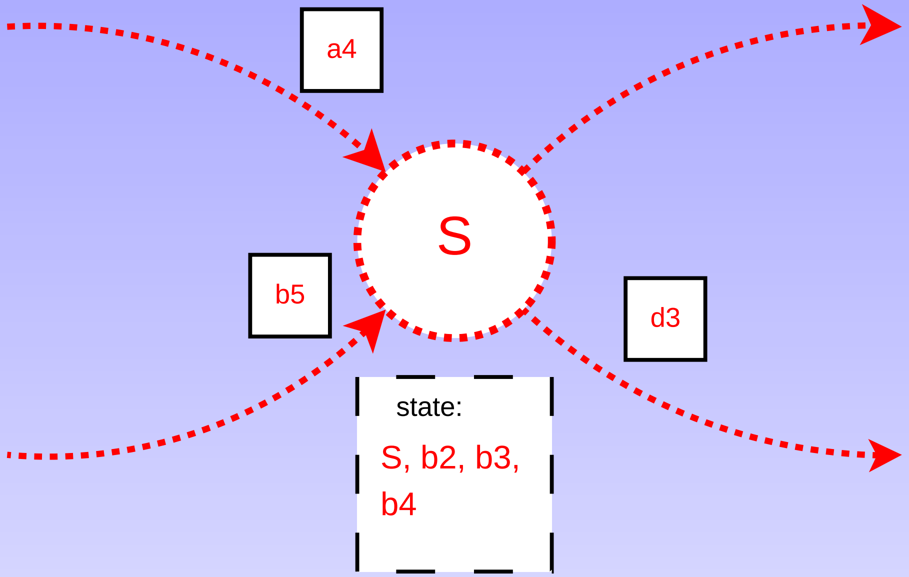
Distributed snapshot, including channel state:



1. *Process S recieves its first marker and makes a local snapshot.*
2. *S records all messages from incoming channels on which no marker has been received (b). Messages following a marker (channel a) are ignored.*
3. S receives a marker from the last incoming channel and saves all recorded messages with the local state - this is S's part of the global snapshot.

Global System State – Example

Distributed snapshot, including channel state:



1. *Process S recieves its first marker and makes a local snapshot.*
2. *S records all messages from incoming channels on which no marker has been received (b). Messages following a marker (channel a) are ignored.*
3. *S receives a marker from the last incoming channel and saves all recorded messages with the local state - this is S's part of the global snapshot.*

Election algorithms

Some distributed algorithms require a coordinator. How can we select one?

- √ in a centralized system the coordinator is selected manually (file servers, etc.) and is a single point of failure.
- √ just because there is a coordinator, doesn't mean that the system is centralized – if the coordinator can be selected dynamically.
- √ a fully distributed algorithm, without any coordinator, is not always the best choice – it tends to complicate the system and slow it down.

Some election algorithms:

- √ the bully algorithm,
- √ ring algorithms.

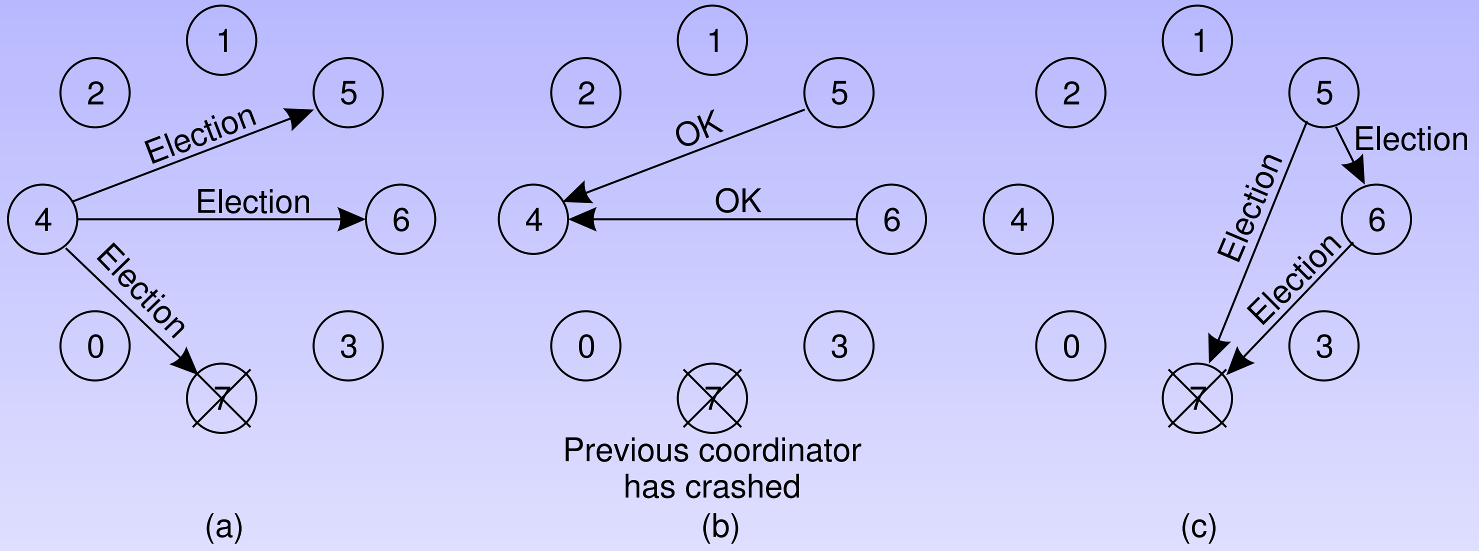
The Bully Election Algorithm (1)

Every process has an associated priority (weight) – for example the process number. Excluding crashed/malfunctioning processes, the process with the highest priority should always be elected.

- √ any process may start an election (e.g. when it fails to establish a connection with the current coordinator) by sending an election message to all processes (if it has some knowlegde about the other processes' weights, it may ignore weaker processes).
- √ when process P_{heavy} receives an election message from P_{light} with a lower priority, it sends a take-over message to P_{light} and starts a new election.
- √ if a process receives no take-over messages, it has been elected and must communicate its victory to other processes.

The Bully Election Algorithm (2)

To make the figure more readable assume that processes know the weights of all others (actually a simpler algorithm is possible in this case).

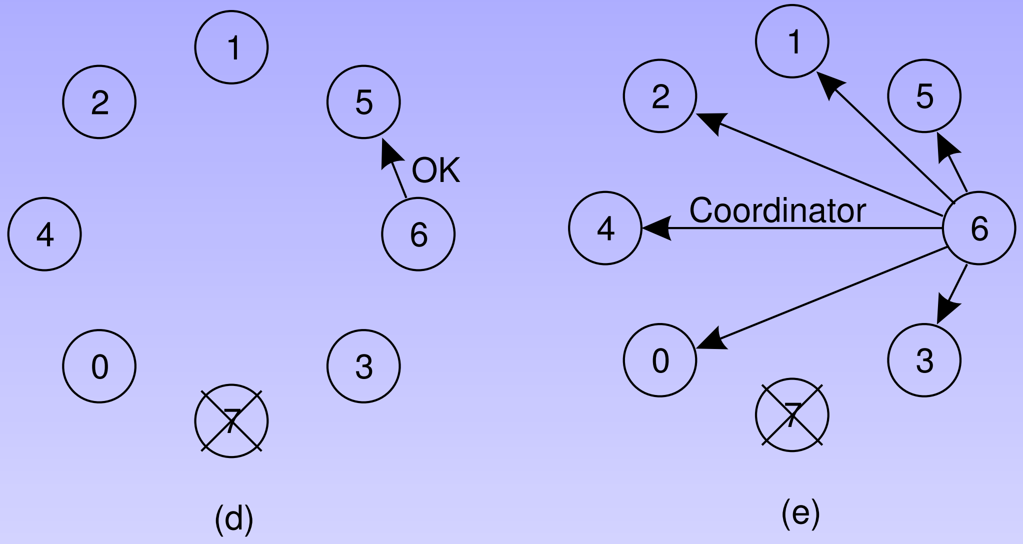


a. process 4 holds an election,

b. processes 5 and 6 respond, trampling the weakling 4,

c. 5 and 6 hold elections.

The Bully Election Algorithm (3)



d. process 6 crushes the hopes of process 5, sending a take-over message,

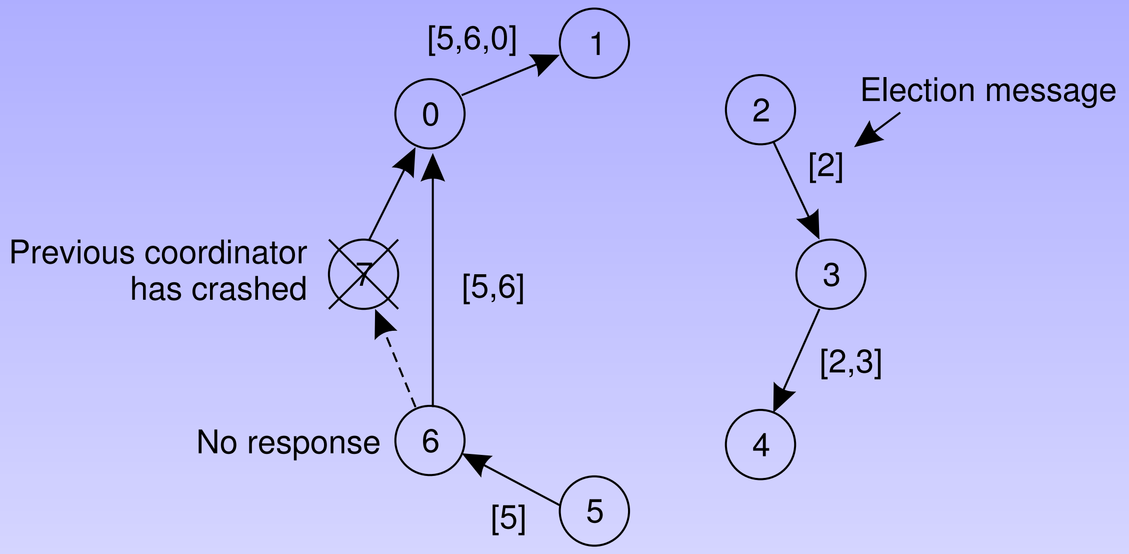
e. process 6 wins and broadcasts its joy.

Ring Algorithms (1)

The election is still based on priorities. Processes are organized into a logical ring.

- √ any process may start an election by sending to its successor an election message (a ballot). If the successor is down, the message is sent to the next one, and so on.
- √ every process forwards the ballot, adding its priority to it.
- √ when the ballot returns to the initiator, it contains a full list of the currently available processes and their priorities.
- √ the initiator sends the list or just the identifier of the heaviest process to all processes.

Ring Algorithms (2)



Ring Algorithms (3)

The previous algorithm transmits more data than strictly necessary. It can be simplified as follows:

- √ any process may start an election by sending a ballot with its identifier and priority to its successor. If the successor is down, the next process on the ring is contacted, and so on.
- √ receiver compares the priority on the ballot with its own priority:
  - ⋆ if the priority on the ballot is higher, the process forwards the ballot to the successor without changes (variant: the process may store the data from the ballot),
  - ⋆ if the priority on the ballot is lower, the ballot is discarded and the process sends to the successor a new ballot with its own identifier and priority.

Ring Algorithms (4)

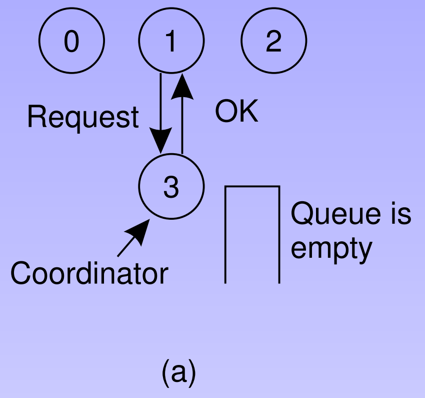
- √ a process that receives a ballot with its own identifier is the winner. The result can be communicated in two ways:
  - ⋆ the winner informs everybody, using broadcast or a „winner” message forwarded along the ring,
  - ⋆ if the processes stored the information from the received ballot, the winner doesn’t have to do anything, a timeout suffices – this version works, but is more fragile, a lost packet can change the outcome of the election.

Mutual Exclusion

A number of processes in a distributed system want exclusive access to some resource (they want to enter a **critical region/section**). A mutual exclusion mechanism is needed – a **lock** (also called a **mutex**).
Standard solutions:

- √ via a centralized server,
- √ completely distributed, with no topology imposed,
- √ completely distributed, making use of a logical ring.

MutEx: A Centralized Algorithm



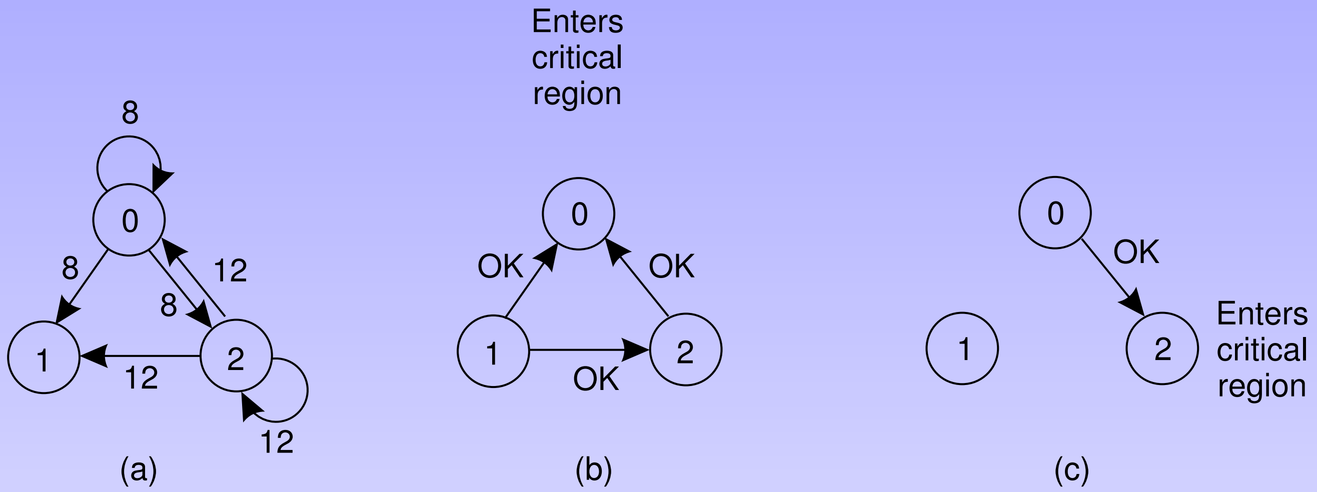
1. Process 1 asks the coordinator for permission to enter a critical region. The resource is fre, so permission is granted.
2. Process 2 then asks permission to enter the same critical region. The coordinator does not reply.
3. When process 1 exits the critical region, it releases the resource by informing the coordinator, who then replies to process 2, allowing it access.

MutEx: Ricart & Agrawala Algorithm (1)

Ricart & Agrawala algorithm – completely distributed, with no topology imposed.

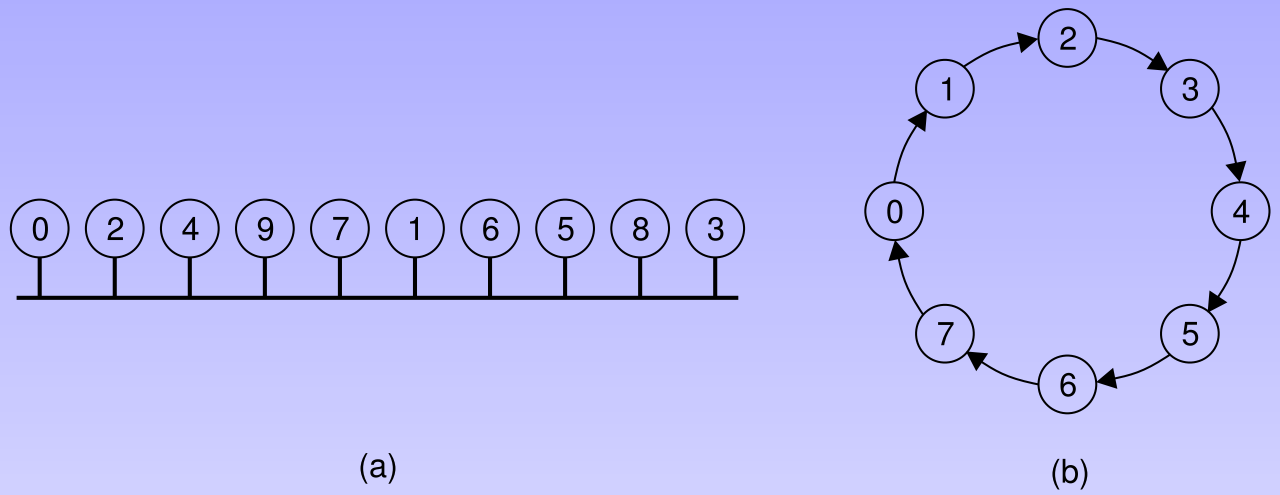
- √ the same as Lamport except that acknowledgments aren't sent.
- √ a process wishing to access a shared resource broadcasts a request to all other processes. The receiving process sends a reply (a grant), if and only if:
  - ⋆ the receiving process has no interest in the shared resource, or
  - ⋆ the receiving process is waiting for the resource, but has lower priority (known through comparison of time-stamps – earlier requests have priority).
- √ in all other cases (the process currently uses the resource or is waiting for it with higher priority than the newcomer), reply is deferred, implying some more local administration.

MutEx: Ricart & Agrawala Algorithm (2)



1. Two processes want to enter the same critical region at the same moment.
2. Process 0 has the lowest timestamp, so it wins.
3. When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

MutEx: A Token Ring Algorithm



1. Physically – an unordered group of processes on a network.
2. Logically – a ring constructed in software, around which a token is forwarded.

Mutual Exclusion - Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Potential 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

The token ring algorithm requires a second token to be circulated if the resource is being used for a very long time – this is an information that the token hasn't been lost. Alternatively exceedingly long critical sections can be banned.