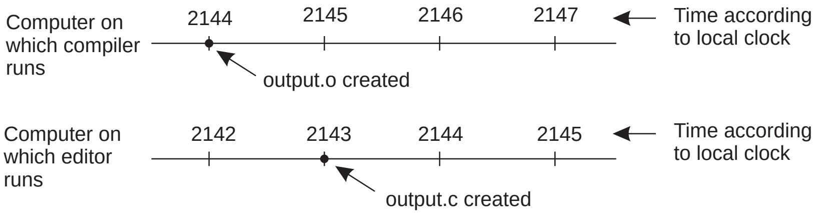
Distributed Systems
Synchronization (I)

[2] **Synchronization (I)**

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

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

[3] **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

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

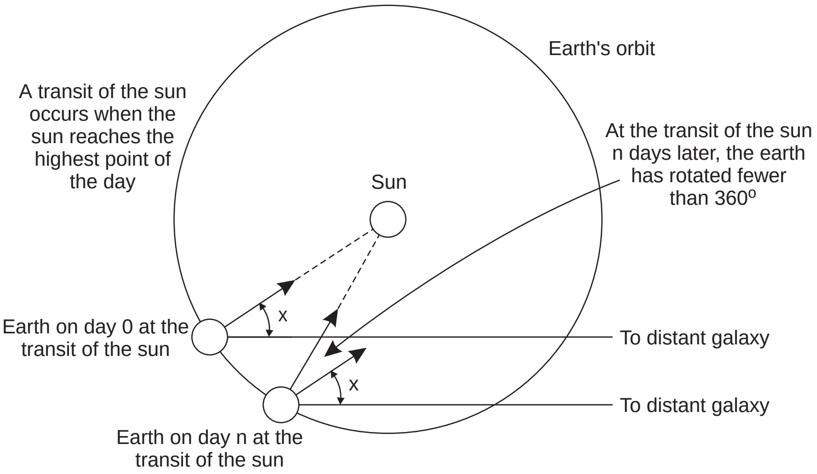
[4] **Timers**

- timer,
- registers associated with each crystal:
  - counter,

  - holding register;

- interrupt generated when counter gets 0,
- interrupt called every clock tick,
- impossible to guarantee two crystals run at exactly the same frequency,
- after getting out of sync, the difference in time values called **clock skew**.

[5] **The Mean Solar Day**


Earth's orbit
A transit of the sun occurs when the sun reaches the highest point of the day
At the transit of the sun n days later, the earth has rotated fewer than 360°
Sun
x
Earth on day 0 at the transit of the sun
To distant galaxy
x
To distant galaxy
Earth on day n at the transit of the sun

Computation of the mean solar day – the period of the earth's rotation is not constant.

[6] **Physical Clocks (1)**

Transit of the sun the event of the sun reaching its highest apparent point in the sky.

Solar day the interval between two consecutive transits of the sun.

Solar second 1/86400th of a solar day.

- mean solar second (300 million days ago a year has about 400 days),

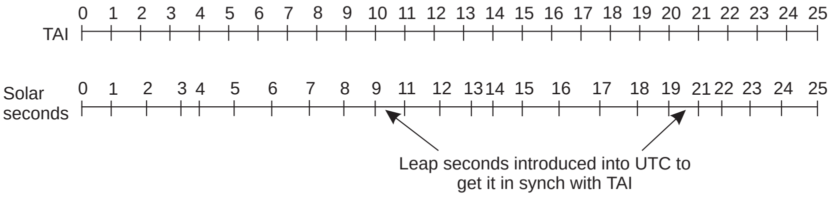
[7] **Physical Clocks (2)**

Sometimes we simply need the exact time, not just an ordering.
Solution: **Universal Coordinated Time (UTC)**:

- based on the number of transitions per second of the cesium 133 atom (pretty accurate),
- at present, the real time is taken as the average of some 50 cesium-clocks around the world,
- introduces a leap second from time to time to compensate that days are getting longer.

NIST operates a shortwave radio station with call letters WWV from Fort Collins in Colorado (a short pulse at the start of each UTC second). UTC is **broadcast** through short wave radio and satellite. Satellites can give an accuracy of about ±0.5 ms.
Does this solve all our problems? Don't we now have some global timing mechanism? This timing is still way too coarse for ordering every event.

[8] **Physical Clocks (3)**



TAI seconds are of constant length, unlike solar seconds. Leap seconds are introduced when necessary to keep in phase with the sun.

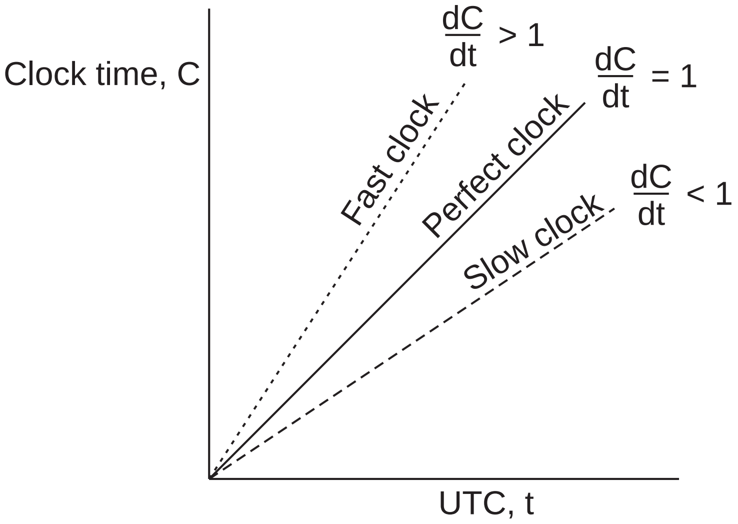
- TAI – International Atomic Time,
- 86400 TAI seconds is about 3 msec less than a mean solar day,
- UTC – TAI with leap seconds whenever the discrepancy between TAI and solar time grows to 800 msec.

[9] **Physical Clocks (4)**

Assumption: a distributed system with an UTC-receiver somewhere in it.
Basic principle:

- every machine has a timer that generates an interrupt H times per second,
- there is a clock in machine p that ticks on each timer interrupt. Denote the value of that clock by $Ci_p(t)$, where t is UTC time.
- ideally, we have that for each machine p, $Cp(t) = t$, or, in other words, $dC/dt = 1$
- Ideally: $dC/dt = 1$, in practice: $1 - \rho \leq dC/dt \leq 1 + \rho$
- in order to protect against difference bigger than δ time units ⇒ synchronize at least every $\delta/(2\rho)$ seconds.

[10] **Clock Synchronization Algorithms**



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

[11] **Clock Synchronization Principles**

Principle I Every machine asks a time server for the accurate time at least once every $\delta/(2\rho)$ seconds.

- needs an accurate measure of round trip delay, including interrupt handling and processing incoming messages.

Principle II Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time relative to its present time.

- probably gets every machine in sync.

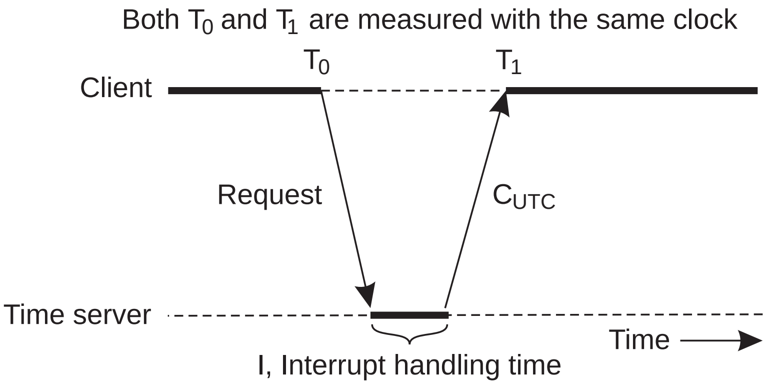
- setting the time back is never allowed, therefore smooth adjustments.

[12] **Clock Synchronization Algorithms**

Clock synchronization algorithms:

- Cristian's Algorithm
- The Berkeley Algorithm
- Averaging Algorithms

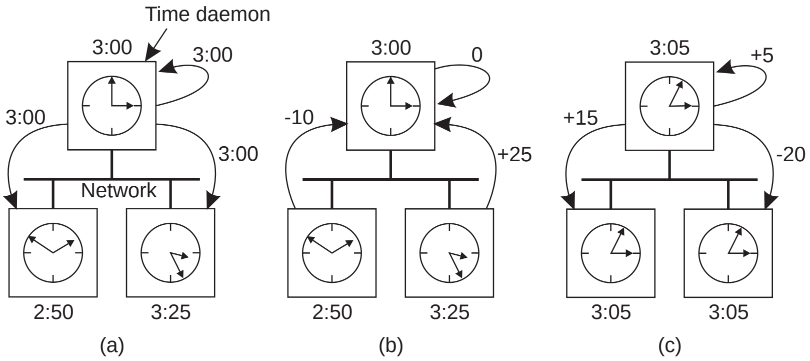
[13] **Cristian's Algorithm**


Both T_0 and T_1 are measured with the same clock
T_0
T_1
Client
Request
C_{UTC}
Time server
Time
I, Interrupt handling time

Getting the current time from a time server.

- $(T1 - T0)/2$,
- messages with $T1 - T0$ above some threshold discarded as being victims of network congestion,
- the message that came back fastest is the most accurate one.

[14] **The Berkeley Algorithm**



1. The time daemon asks all the other machines for their clock values.
2. The machines answer.
3. The time daemon tells everyone how to adjust their clock.

[15] **Averaging Algorithms**

- previous methods highly centralized,
- decentralized algorithms:
  - dividing time into fixed-length resynchronization intervals,
  - $T0 + (i + 1)R$, where R is a system parameter,
  - machines broadcast the current time according to their clocks,
  - another variation: correcting each message by considering propagation time from the source,

- Internet: the Network Time Protocol (**NTP**), accuracy in the range of 1-50 msec.

[16] **Logical Clocks**

- often if it is sufficient that all machines agree on the same time,
- internal consistency only matters, not whether they are particularly close to the real time,
- what usually matters is not that all processes agree on what time is, but rather that they agree on the order in which events occur,
- **Lamport's algorithm**, which synchronizes logical clocks,
- an extension to Lamport's approach, called **vector timestamps**.

[17] **The Happened-Before Relationship**

The **happened-before** relation on the 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, then $a \rightarrow b$.
- if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

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

Concurrent events
Nothing can be said about when the events happened or which event happened first.

[18] **Logical Clocks (1)**

How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

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

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

P2 If a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$.

How to attach a time-stamp to an event when there's no global clock?

Solution: maintain a consistent set of logical clocks, one per process.

[19] **Logical Clocks (2)**

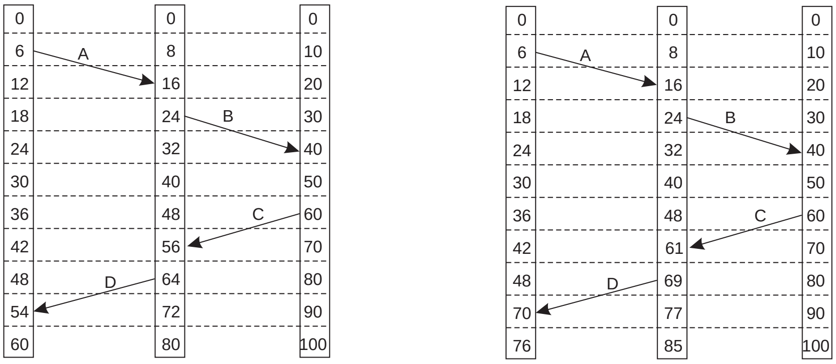
Each process Pi maintains a **local counter** C_i and adjusts this counter according to the following rules:

1. For any two successive events that take place within P_i, C_i is incremented by 1.
2. Each time a message m is sent by process P_i, the message receives a time-stamp $T_m = C_i$.
3. Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j:

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

- property **P1** satisfied by 1.,
- property **P2** satisfied by 2. and 3.

[20] **Logical Clocks (3)**



(a)

(b)

Lamport's algorithm example

[21] **Total Ordering with Logical Clocks**

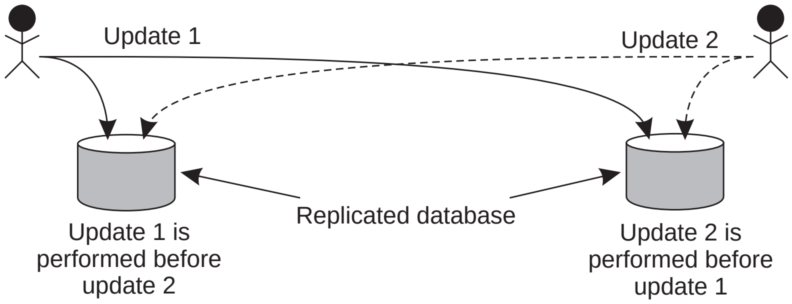
Still can occur: two events happen at the same time. May be avoided by attaching a process number to an event:

If: P_i time-stamps event e with $C_i(e).i$

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

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

[22] **Example: Totally-Ordered Multicasting**


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

- this situation requires totally-ordered multicasting - to be implemented with Lamport timestamps,
- each message is always timestamped with the current logical time of the sender,
- received message put into a local queue, ordered according to its timestamp, receiver multicasts an acknowledgement to others,
- a process can deliver a queued message to the application it is running only when that message is at the head of the queue and has been acknowledged by each other process.

[23] **Vector Timestamps (1)**

- Lamport timestamps do not guarantee that if $C(a) < C(b)$ that a indeed happened before b. **Vector timestamps** are required for that.
  - each process P_i has an array $V_i[1 \dots n]$, where $V_i[j]$ denotes the number of events that process P_i knows have taken place at process P_j,
  - when P_i sends a message m, it adds 1 to $V_i[i]$, and sends V_i along with m as vector timestamp $vt(m)$. Upon arrival, each other process knows P_i's timestamp.
- timestamp vt of m tells the receiver how many events in other processes have preceded m, and on which m may causally depend.

[24] **Vector Timestamps (2)**

- when a process P_j receives m from P_i with $vt(m)$, it:
  - updates each $V_j[k]$ to $max\{V_j[k], V(m)[k]\}$,
  - increments $V_j[j]$ by 1.
- to support causal delivery of messages, assume you increment your own component only when sending a message. Then, P_j postpones delivery of m until:
  - $vt(m)[i] = V_j[i] + 1$ **and**
  - $vt(m)[k] \leq V_j[k]$ for $k \neq i$.

Example
Given $\mathbf{V3} = [0, 2, 2]$, $\mathbf{vt(m)} = [1, 3, 0]$:
What information does P_3 have, and what will it do after receiving m (from P_1)?

[25] **An example of Causal Delivery of Messages (1)**

Assumptions:

- messages multicasted by the processes to all other participating in communication,
- all messages sent by one process received in the same order by each other process,

- reliable message sending mechanism,
- order of messages from different processes not forced.

Actions on the sender side:

1. Sending (multicasting) of the message.

Actions on the receiver side:

1. Receiving of the message by the communication layer.
2. Delivering of the message to the target process.

[26] An example of Causal Delivery of Messages (2)

Let

vt_m - vector timestamp of message m,

V_P - current vector of process P.

Rules
When message m sent by process P, sent together with vector timestamp vt_m built up in the following way:

1. $vt_m[P] = V_P[P] + 1$,
2. $vt_m[X] = V_P[X]$ for all X different to P.

Received message m from P delivered into the process Q only if the following conditions are met:

1. $vt_m[P] = V_Q[P] + 1$
2. $vt_m[X] \leq V_Q[X]$ for all X different to P.

When message m delivered to the process Q:

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

[27] An example of Causal Delivery of Messages (3)

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

General scenario:

1. Process A multicasts request m1
2. Process B multicasts reply m2 as a result of obtaining request in message m1.

Goal:
All processes should have delivered message m2 only after delivering message m1. If the message m2 is received by the transport layer of some process as the first one, delivery of the m2 must be postponed until m1 is received and delivered before.

[28] **An example of Causal Delivery of Messages (4)**

A sends $m1(0+1,0,0) = m1(1,0,0)$,
B receives $m1(1,0,0)$ from A,

> $V_B = (0,0,0)$, $vt_{m1} = (1,0,0)$,
> m1 delivered at once because:
>
> > $vt_{m1}[A] = V_B[A] + 1$,
> > $vt_{m1}[X] <= V_B[X]$ for all X different to A.
>
> after m1 delivery new value of V_B set to $V_B = (1,0,0)$.

B sends $m2(1,0+1,0) = m2(1,1,0)$,
A receives $m2(1,1,0)$ from B,

> $V_A = (1,0,0)$, $vt_{m2} = (1,1,0)$,
> m2 delivered at once because:
>
> > $vt_{m2}[B] = V_A[B] + 1$,
> > $vt_{m2}[X] <= V_A[X]$ for all X different to B.
>
> after m2 delivery new value of V_A set to $V_A = (1,1,0)$.

[29] **An example of Causal Delivery of Messages (5)**

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

> $V_C = (0,0,0)$, $vt_{m2} = (1,1,0)$,
> m2 delivery **postponed** because:
>
> > $vt_{m2}[A] > V_C[A]$ and A is different to B.

Comment:
We should not deliver the message m2 sent by B to the process C now because at the time of sending that message by the process B it knew already some message received from process A about which we do not know yet.

Perhaps in that message, received before by B and not received by us yet, was something important what should be received by C before receiving m2. Firstly, C has to have delivered the previous message, already delivered to B before the moment of sending by B the message m2.

[30] **An example of Causal Delivery of Messages (6)**

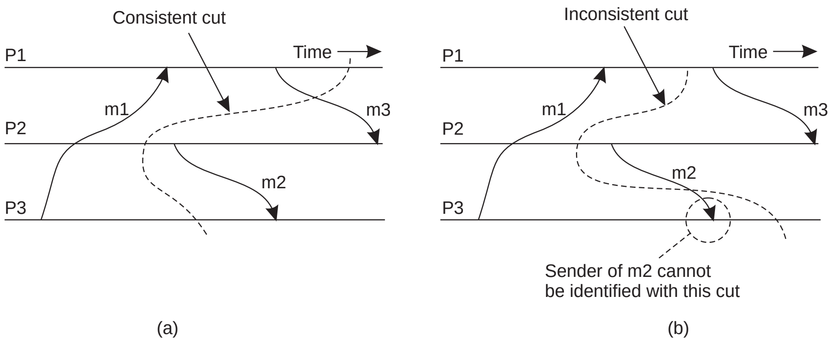
C receives $m1(1,0,0)$ from A

> $V_C = (0,0,0)$, $vt_{m1} = (1,0,0)$,
> m1 delivered at once because:
>
> > $vt_{m1}[A] = V_C[A] + 1$,
> > $vt_{m1}[X] <= V_C[X]$ for all X different to A.
>
> after m1 delivery new value of V_C set to $V_C = (1,0,0)$,
> now on C we check delivery queue,
> now m2 may be and is delivered 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 all X different to C.
>
> after m2 delivery new value of V_C set to $V_C = (1,1,0)$.

After two multicasts $A \rightarrow BC$ and $B \rightarrow AC$, current values of vector timestamps of processes are as follows: $V_A = V_B = V_C = (1,1,0)$

[31] **Global State (1)**

Sometimes one wants to collect the current state of a distributed computation, called **a distributed snapshot**.
It consists of: (1) all local states and (2) messages currently in transit.



A distributed snapshot should reflect **a consistent state**.

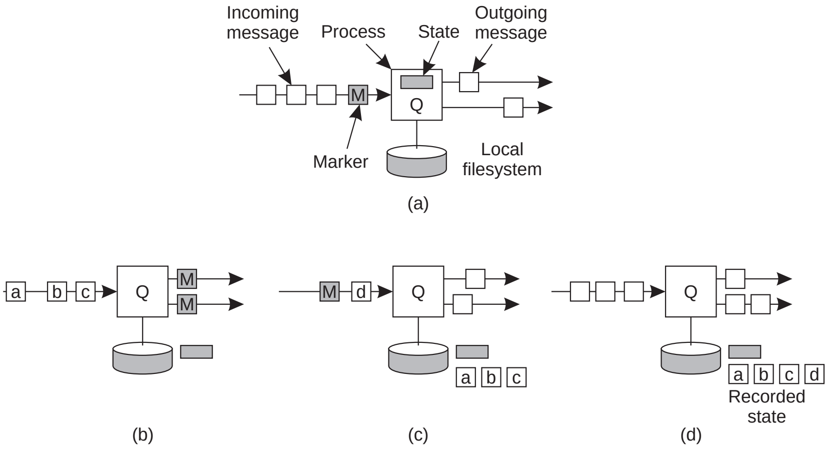
[32] **Global State (2)**

- collection of processes connected to each other through unidirectional point-to-point communication channels,
- any process P can initiate taking a distributed snapshot.

1. P starts by recording its own local state,
2. P subsequently sends a marker along each of its outgoing channels,
3. when Q receives a marker through channel C, its action depends on whether it had already recorded its local state:
   - *not yet recorded:* it records its local state, and sends the marker along each of its outgoing channels,
   - *already recorded:* the marker on C indicates that the channel's state should be recorded: all messages received since the time Q recorded its own state and before that marker to be recorded as the channel's state,
4. Q is finished when it has received a marker along each of its incoming channels.

[33] **Global State (3)**

Distributed snapshot, channel state recording:



1. Process Q receives a marker for the first time and records its local state.

2. Q records all incoming message.

3. Q receives a marker for its incoming channel and finishes recording the state of the incoming channel.

[34] **Election Algorithms**

An algorithm requires that some process acts as a coordinator. How to select this special process dynamically?

- in many systems the coordinator chosen by hand (e.g. file servers). This leads to centralized solutions ⇒ *single point of failure*.

- if a coordinator chosen dynamically, to what extent one can speak about a centralized or distributed solution? Having a central coordinator does not necessarily make an algorithm non-distributed.

- is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution? Fully distributed solutions not necessarily better.

Example election algorithms:

- the bully algorithm,

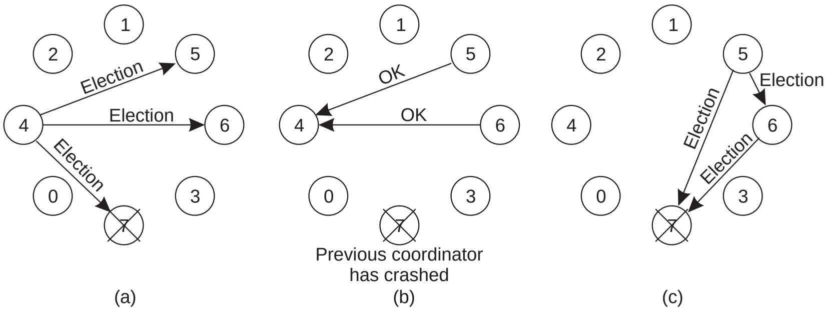
- a ring algorithm.

[35] **The Bully Election Algorithm (1)**

Each process has an associated priority (weight). The process with the highest priority should always be elected as the coordinator.
How to find the heaviest process?

- any process can just start an election by sending an election message to all other processes (assuming you don't know the weights of the others).
- if process P_{heavy} receives an election message from lighter process P_{light}, it sends a take-over message to P_{light}. P_{light} is out of the race.
- if a process doesn't get a take-over message back, it wins, and sends a victory message to all other processes.

[36] **The Bully Election Algorithm (2)**

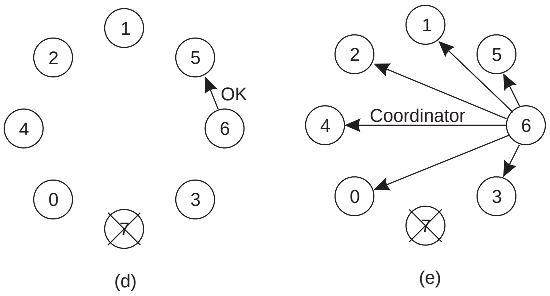


a. process 4 holds an election,

b. process 5 and 6 respond, telling 4 to stop,

c. now 5 and 6 each hold an election.

[37] **The Bully Election Algorithm (3)**



d. process 6 tells 5 to stop,

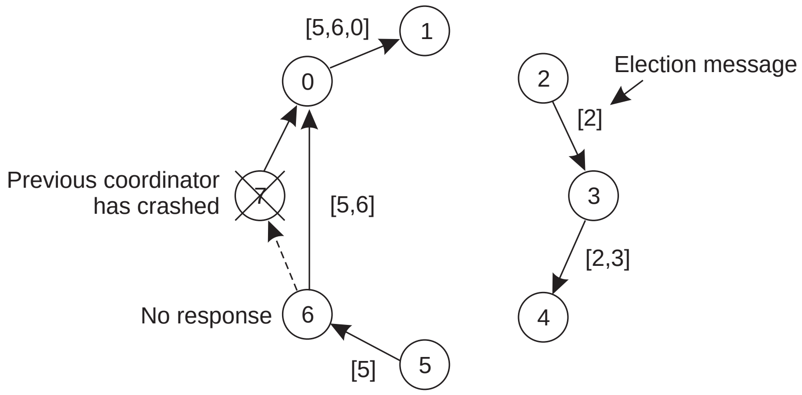
e. process 6 wins and tells everyone.

[38] **A Ring Algorithm (1)**

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 successor.
- if a message is passed on, the sender adds itself to the list. When it gets back to the initiator, everyone had a chance to make its presence known.
- the initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority is elected as coordinator.

[39] **A Ring Algorithm (2)**

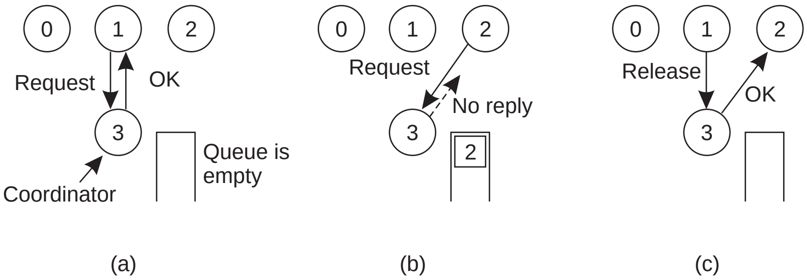


[40] **Mutual Exclusion**

A number of processes in a distributed system want exclusive access to some resource.
Standard solutions:

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

[41] **MutEx: A Centralized Algorithm**



1. Process 1 asks the coordinator for permission to enter a critical region. Permission is granted.

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

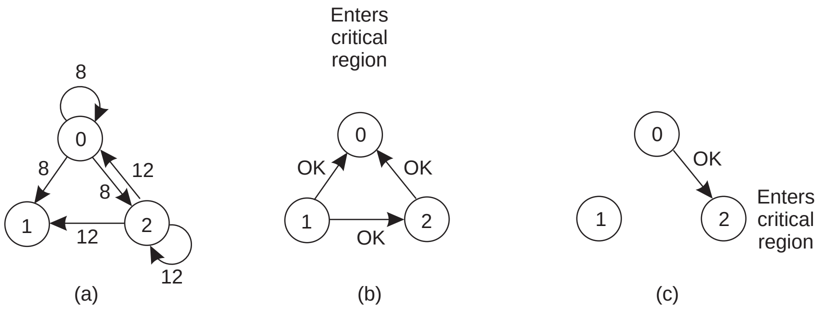
3. When process 1 exits the critical region, it tells the coordinator, when then replies to 2.

[42] **MutEx: Ricart & Agrawala Algorithm (1)**

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

- the same as Lamport except that acknowledgments aren't sent. Instead, replies (i.e. grants) are sent only when:
  - 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).
- in all other cases, reply is deferred, implying some more local administration.

[43] **MutEx: Ricart & Agrawala Algorithm (2)**

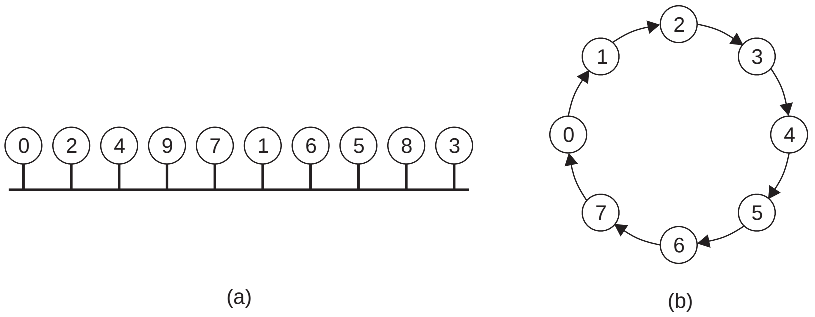


1. Two processes want to enter the same critical region at the same moment.

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.

[44] **MutEx: A Token Ring Algorithm**



1. An unordered group of processes on a network.
2. A logical ring constructed in software.

[45] **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

A comparison of three mutual exclusion algorithms.