

Coordination

Physical clocks

Problem

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.

Note

UTC is **broadcast** through short-wave radio and satellite. Satellites can give an accuracy of about ± 0.5 ms.

Clock synchronization

Precision

The goal is to keep the deviation **between two clocks on any two machines** within a specified bound, known as the **precision** π :

$$\forall t \forall p, q : |C_p(t) - C_q(t)| \leq \pi$$

with $C_p(t)$ the **computed** clock time of machine p at **UTC time** t .

Accuracy

In the case of **accuracy**, we aim to keep the clock bound to a value α :

$$\forall t \forall p : |C_p(t) - t| \leq \alpha$$

Synchronization

Internal synchronization: keep clocks **precise**

External synchronization: keep clocks **accurate**

Clock drift

Clock specifications

A clock comes specified with its **maximum clock drift rate** ρ .

$F(t)$ denotes oscillator frequency of the hardware clock at time t

F is the clock's ideal (constant) frequency \Rightarrow living up to specifications:

$$\forall t: (1 - \rho) \leq \frac{F(t)}{F} \leq (1 + \rho)$$

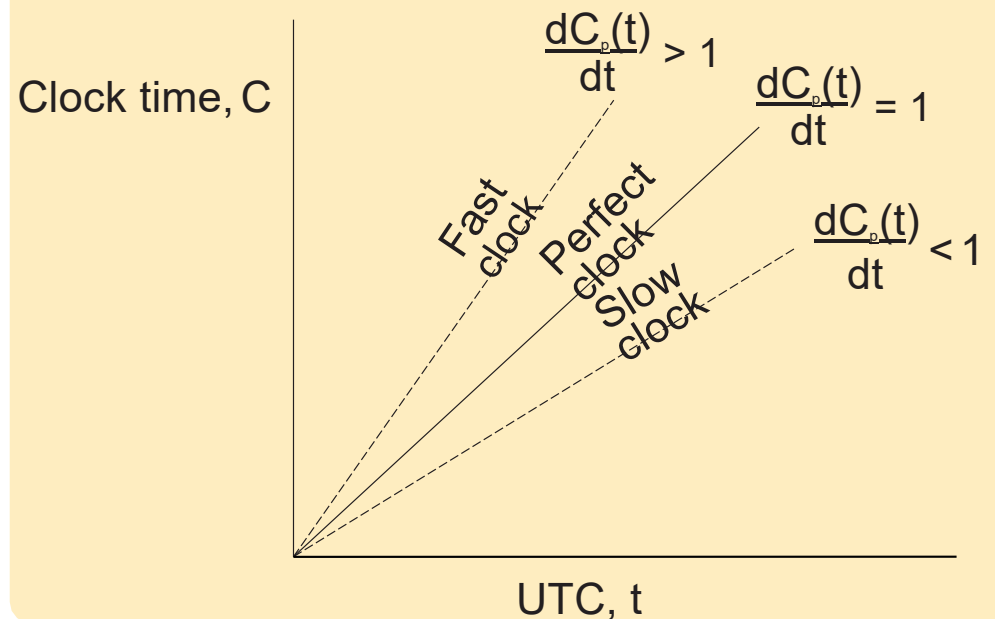
Observation

By using hardware interrupts we couple a software clock to the hardware clock, and thus also its clock drift rate:

$$C_p(t) = \frac{1}{F} \int_0^t F(t) dt \Rightarrow \frac{dC_p(t)}{dt} = \frac{F(t)}{F}$$

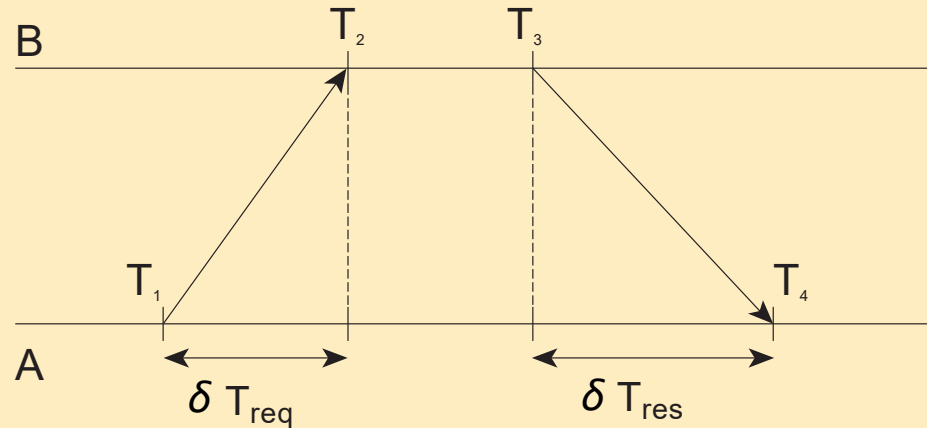
$$\Rightarrow \forall t: 1 - \rho \leq \frac{dC_p(t)}{dt} \leq 1 + \rho$$

Fast, perfect, slow clocks



Detecting and adjusting incorrect times

Getting the current time from a time server



Computing the relative offset ϑ and delay δ

Assumption: $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$

$$\vartheta = T_3 + ((T_2 - T_1) + (T_4 - T_3))/2 - T_4 = ((T_2 - T_1) + (T_3 - T_4))/2$$

$$\delta = ((T_4 - T_1) - (T_3 - T_2))/2$$

Network Time Protocol

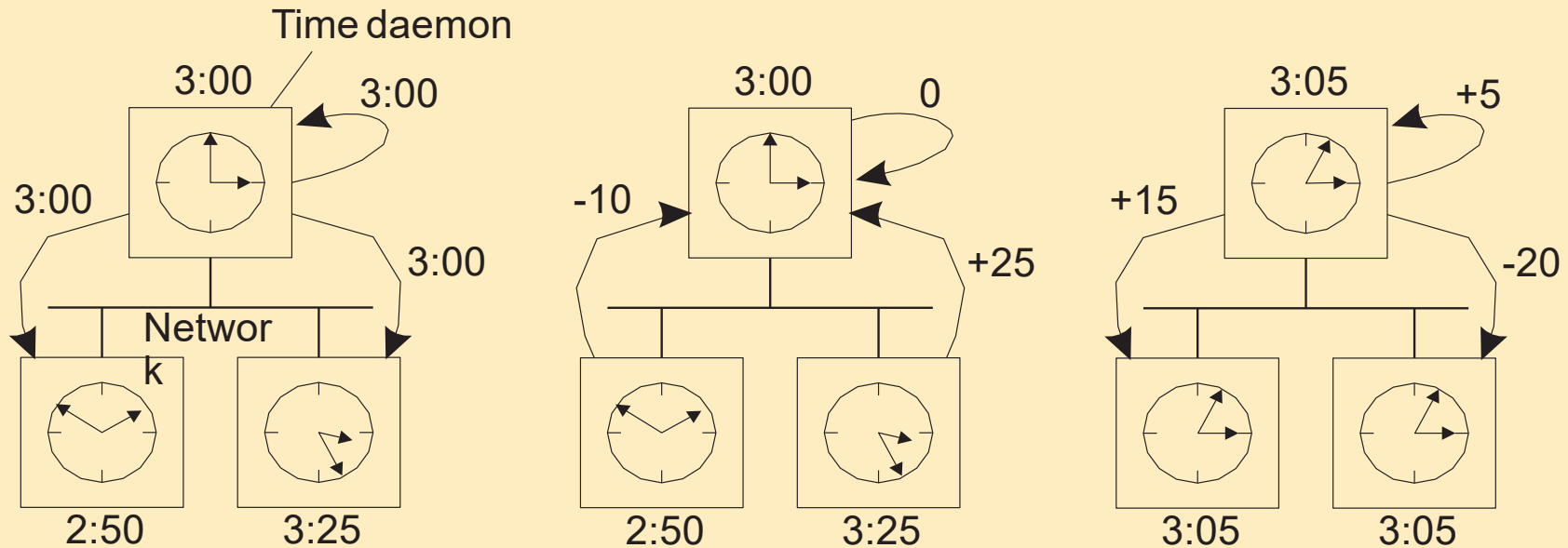
Collect eight (ϑ, δ) pairs and choose ϑ for which associated delay δ was minimal.

Keeping time without UTC

Principle

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**.

Using a time server



Fundamental

You'll have to take into account that setting the time back is **never** allowed \Rightarrow smooth adjustments (i.e., run faster or slower).

Reference broadcast synchronization

Essence

A node broadcasts a reference message $m \Rightarrow$ each receiving node p records the time $T_{p,m}$ that it received m .

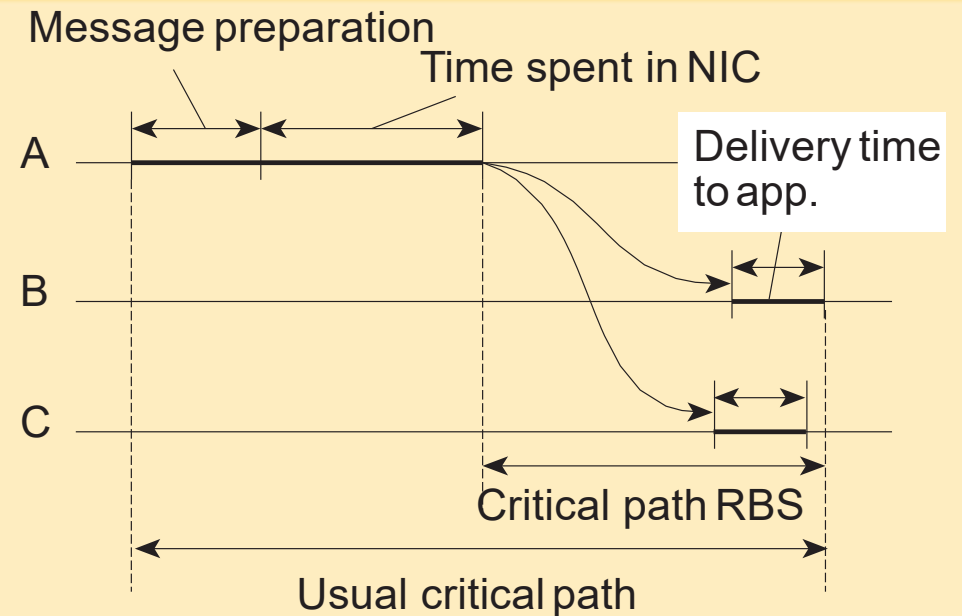
Note: $T_{p,m}$ is read from p 's local clock.

Problem: averaging will not capture drift \Rightarrow use linear regression

NO: $Offset[p, q](t) = \frac{\sum_{k=1}^M (T_{p,k} - T_{q,k})}{M}$

YES: $Offset[p, q](t) = \alpha t + \beta$

RBS minimizes critical path



The Happened-before relationship

Issue

What usually matters is not that all processes agree on exactly what time it is, but that they agree on the **order in which events occur**. Requires a notion of **ordering**.

The **happened-before** relation

TT.now(t)

TT.after(t)

TT.before(t)

Note

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


```

int TT_now(TT_interval* interval)
{
    struct ntptimeval tv;
    struct timeval delta;

    struct timeval* earliest_p = &(interval->earliest);
    struct timeval* latest_p = &(interval->latest);
    struct timeval* now_p = &(tv.time);
    struct timeval* delta_p = &delta;

    timerclear(&delta);
    timerclear(&interval->earliest);
    timerclear(&interval->latest);

    if(ntp_gettime(&tv) == 0) {
        tv.maxerror = tv.maxerror > 0 ? tv.maxerror : -(tv.maxerror);

        delta.tv_sec = delta.tv_sec + (tv.maxerror / 1000);
        delta.tv_usec = delta.tv_usec + ((tv.maxerror % 1000) * 1000);

        if(delta.tv_usec > 1000000) {
            delta.tv_usec -= 1000000;
            delta.tv_sec++;
        }

        timeradd(now_p, delta_p, latest_p);
        timersub(now_p, delta_p, earliest_p);
    } else {
        printf("error on ntp_gettime. %s\n", strerror(errno));
        return ERROR;
    }

    return SUCCESS;
}

```

```

typedef struct TT_interval {
    struct timeval earliest;
    struct timeval latest;
} TT_interval;

```

```
int TT_before(TT_interval* interval, bool* success)
{
    struct timeval* latest_p;
    struct timeval* earliest_p;
    TT_interval now;

    if(TT_now(&now) != SUCCESS) {
        return ERROR;
    }

    latest_p = &(interval->latest);
    earliest_p = &(now.earliest);

    if(timercmp(latest_p, earliest_p, <) != 0) {
        *success = true;
        return SUCCESS;
    } else {
        *success = false;
        return SUCCESS;
    }

    return ERROR;
}
```

```
typedef struct TT_interval {
    struct timeval earliest;
    struct timeval latest;
} TT_interval;
```

Logical clocks

Problem

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

Attach a timestamp $C(e)$ to each event e , satisfying the following properties:

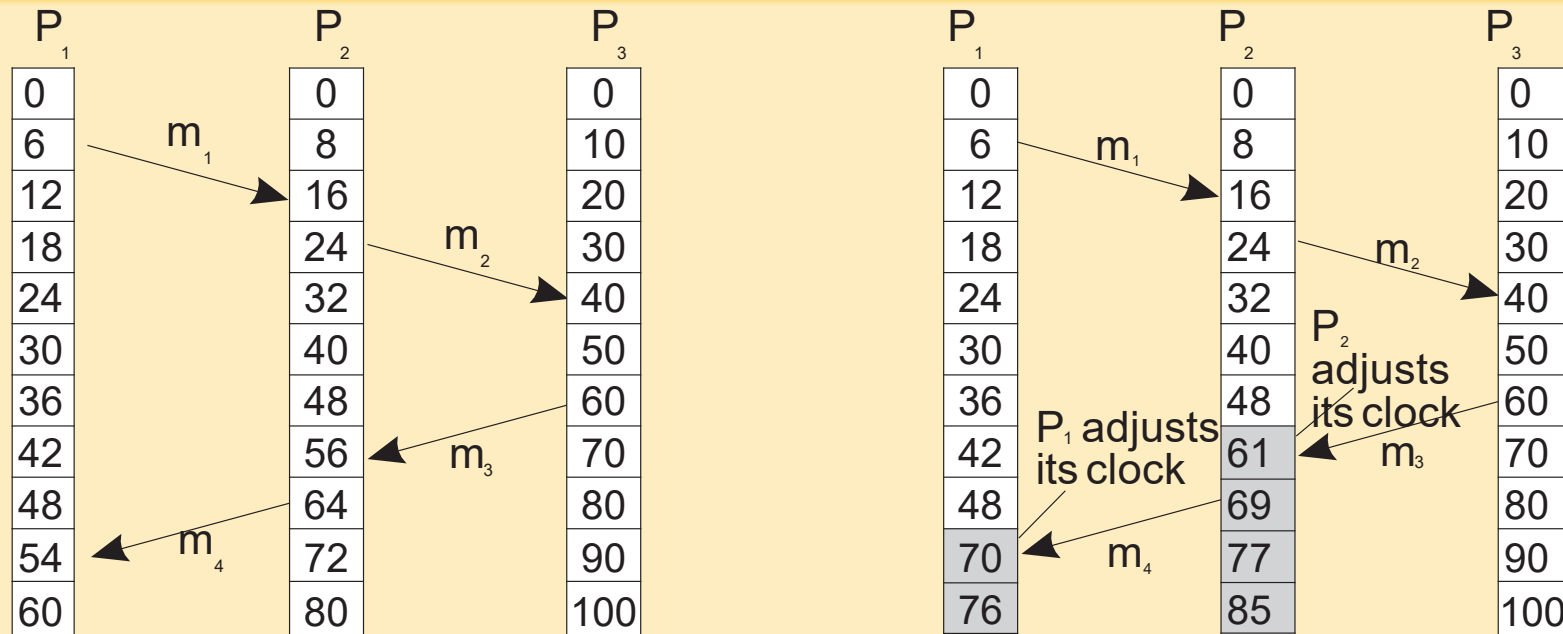
- P1** If a and b are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.
- P2** If a corresponds to sending a message m , and b to the receipt of that message, then also $C(a) < C(b)$.

Problem

How to attach a timestamp to an event when there's no global clock \Rightarrow maintain a **consistent** set of logical clocks, one per process.

Logical clocks: example

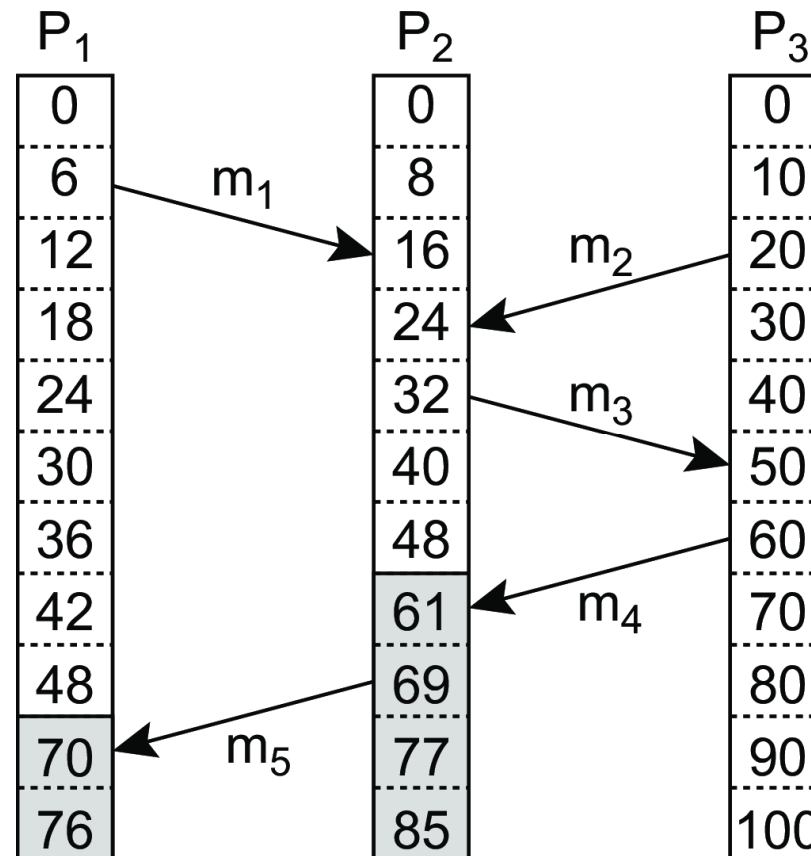
Consider three processes with **event counters** operating at different rates



Logical clocks: example

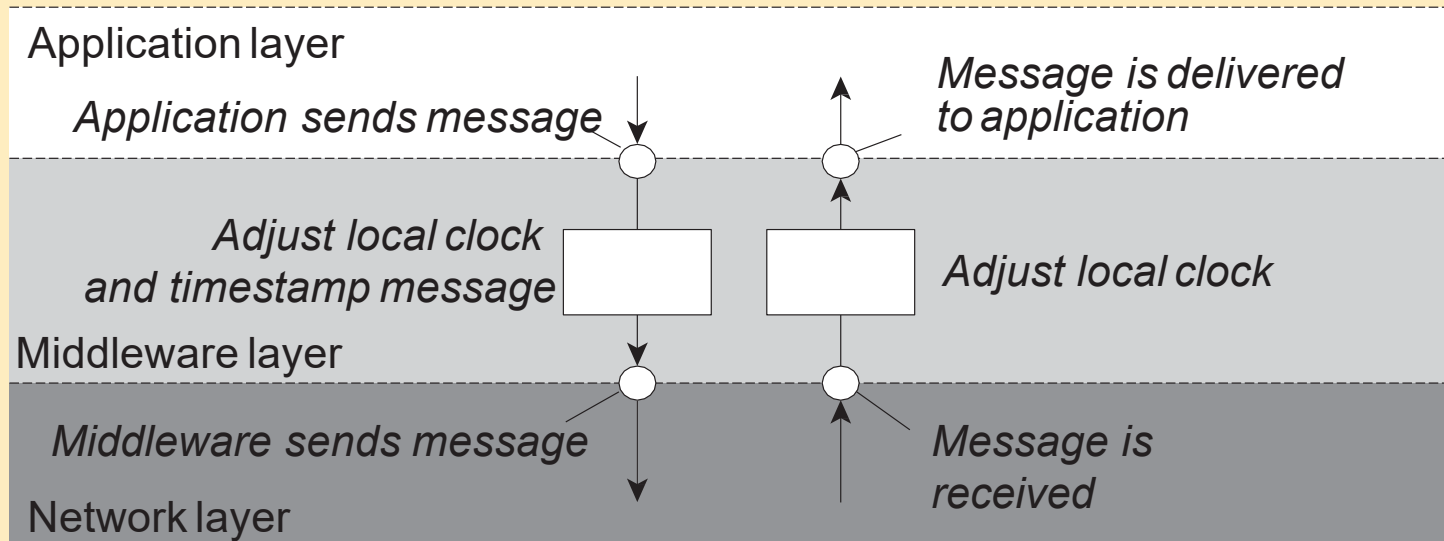
Observation

Lamport's clocks do not guarantee that if $C(a) < C(b)$ then a **causally preceded** b .



Logical clocks: where implemented

Adjustments implemented in middleware



Logical clocks: solution

Each process P_i maintains a **local** counter C_i and adjusts this counter

- 1 For each new event that takes 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 timestamp $ts(m) = C_i$.
- 3 Whenever a message m is **received** by a process P_j , P_j adjusts its local counter C_j to **$\max\{C_j, ts(m)\}$** ; then executes step 1 before passing m to the application.

Notes

Property **P1** is satisfied by (1); Property **P2** by (2) and (3).

It can still occur that two events happen at the same time. Avoid this by **breaking ties through process IDs**.

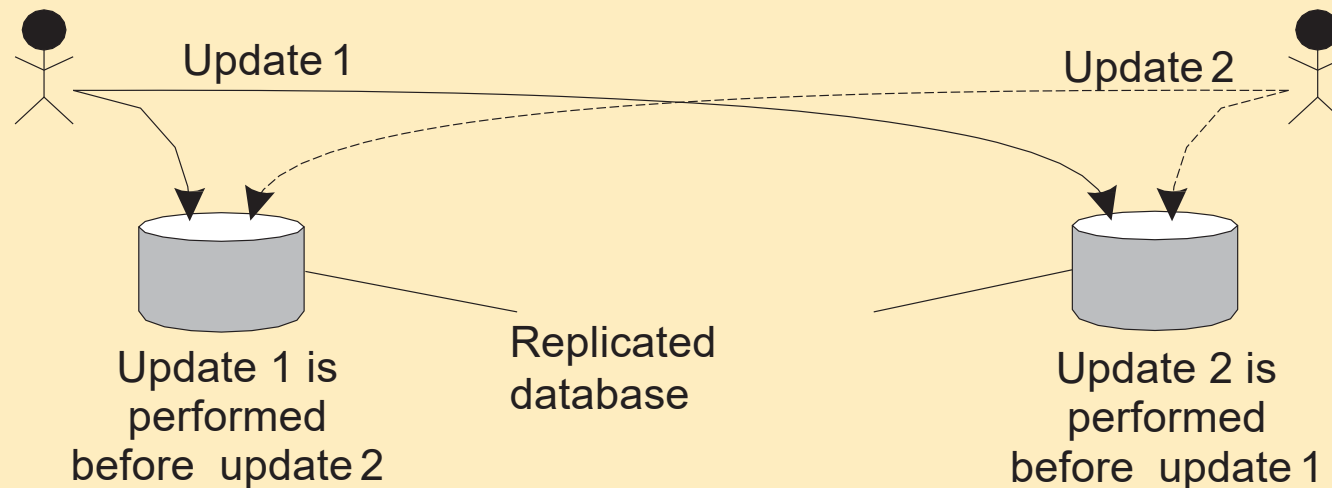
Example: Total-ordered multicast

Concurrent updates on a replicated database are seen in the same order everywhere

P_1 adds \$100 to an account (initial value: \$1000)

P_2 increments account by 1%

There are two replicas



Result

In absence of proper synchronization:
replica #1 \leftarrow \$1111, while replica #2 \leftarrow \$1110.

Example: Total-ordered multicast

Solution

Process P_i sends **timestamped message** m_i to all others. The message itself is put in a local queue $queue_i$.

Any incoming message at P_j is queued in $queue_j$, **according to its timestamp**, and **acknowledged** to every other process.

P_j passes a message m_i to its application if:

- (1) m_i is at the head of $queue_j$
- (2) m_i has been acknowledged by each process P_k OR
for each process P_k , there is a message m_k in $queue_j$ with a larger timestamp.

Note

We are assuming that communication is **reliable** and **FIFO ordered**.

Lamport's clocks for mutual exclusion

Analogy with total-ordered multicast

With total-ordered multicast, all processes build identical queues, delivering messages in the same order

Mutual exclusion is about agreeing in which order processes are allowed to enter a critical section

Lamport's clocks for mutual exclusion

```
1 class Process:
2     def init (self, chan):
3         self.queue      = []                # The request queue
4         self.clock      = 0                # The current logical clock
5
6     def requestToEnter(self):
7         self.clock = self.clock + 1        # Increment clock value
8         self.queue.append((self.clock, self.procID, ENTER)) # Append request to q
9         self.cleanupQ()                    # Sort the queue
10        self.chan.sendTo(self.otherProcs, (self.clock, self.procID, ENTER)) # Send request
11
12    def allowToEnter(self, requester):
13        self.clock = self.clock + 1        # Increment clock value
14        self.chan.sendTo([requester], (self.clock, self.procID, ALLOW)) # Permit other
15
16    def release(self):
17        tmp = [r for r in self.queue[1:] if r[2] == ENTER] # Remove all ALLows
18        self.queue = tmp                                   # and copy to new queue
19        self.clock = self.clock + 1                    # Increment clock value
20        self.chan.sendTo(self.otherProcs, (self.clock, self.procID, RELEASE)) # Release
21
22    def allowedToEnter(self):
23        commProcs = set([req[1] for req in self.queue[1:]]) # See who has sent a message
24        return (self.queue[0][1] == self.procID and len(self.otherProcs) == len(commProcs))
```

Lamport's clocks for mutual exclusion

```
1  def receive(self):
2      msg = self.chan.recvFrom(self.otherProcs) [1]
3      self.clock = max(self.clock, msg[0])
4      self.clock = self.clock + 1
5      if msg[2] == ENTER:
6          self.queue.append(msg)
7          self.allowToEnter(msg[1])
8      elif msg[2] == ALLOW:
9          self.queue.append(msg)
10     elif msg[2] == RELEASE:
11         del(self.queue[0])
12     self.cleanupQ()
```

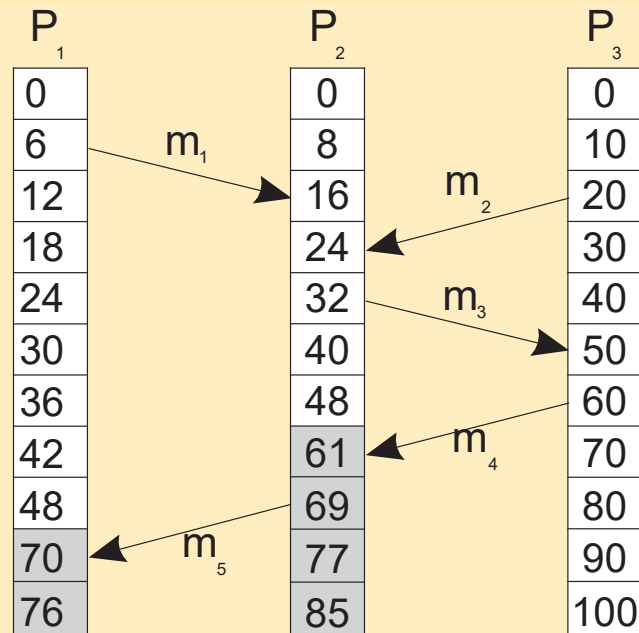
Pick up any message
Adjust clock value...
...and increment
Append an ENTER request
and unconditionally allow
Append an ALLOW
Just remove first message
And sort and cleanup

Vector clocks

Observation

Lamport's clocks do not guarantee that if $C(a) < C(b)$ then a **causally preceded** b .

Concurrent message transmission using logical clocks



Observation

Event a : m_1 is received at $T = 16$;
Event b : m_2 is sent at $T = 20$.

Note

We **cannot** conclude that a causally precedes b .

Capturing causality

Solution: each P_i maintains a vector VC_i

$VC_i[i]$ is the local logical clock at process P_i .

If $VC_i[j] = k$ then P_i knows that k events have occurred at P_j .

Maintaining vector clocks

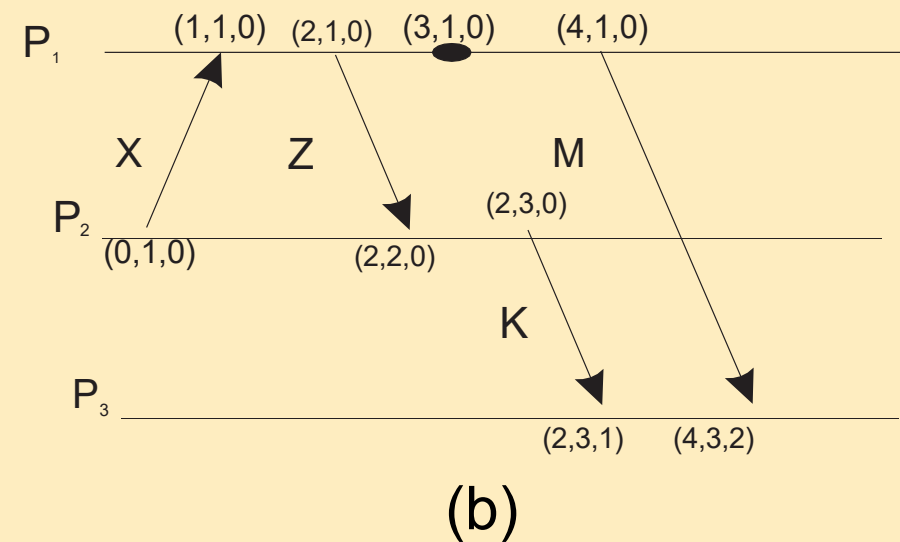
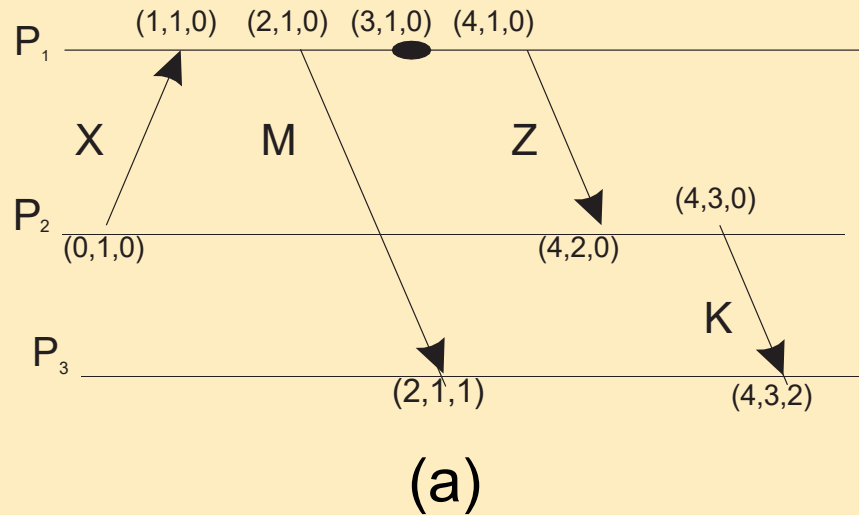
Before executing an event (step2) P_i executes $VC_i[i] \leftarrow VC_i[i] + 1$ (step1).

When process P_i sends (step2) a message m to P_j , it sets m 's (vector) timestamp $ts(m)$ equal to VC_i after having executed the increment (step1).

Upon the receipt of a message m , process P_j sets $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each k , after which it executes the local Increment (step1) and then delivers the message to the application (step2).

Vector clocks: Example

Capturing potential causality when exchanging messages



Analysis

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

Causal dependency

Definition

We say that b may causally depend on a if $ts(a) < ts(b)$, with:

for all k , $ts(a)[k] \leq ts(b)[k]$ and

there exists at least one index k for which $ts(a)[k] < ts(b)[k]$

Precedence vs. dependency

We say that a causally precedes b .

b **may** causally depend on a , as there may be information from a that is propagated into b .

Causally ordered multicasting

Observation

We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.

Adjustment

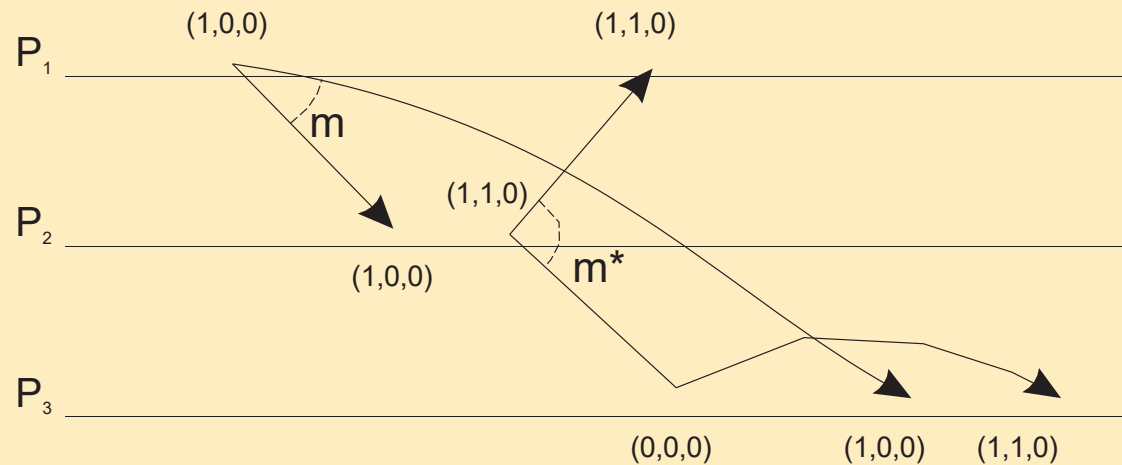
P_i increments $VC_i[i]$ only when sending a message, and P_j “adjusts” VC_j when receiving a message (i.e., effectively does not change $VC_j[j]$).

P_j postpones delivery of m until:

- 1 $ts(m)[i] = VC_j[i] + 1$
- 2 $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$

Causally ordered multicasting

Enforcing causal communication



Example

Take $VC_3 = [0, 2, 2]$, $ts(m) = [1, 3, 0]$ from P_1 . What information does P_3 have, and what will it do when receiving m (from P_1)?

Mutual exclusion

Problem

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

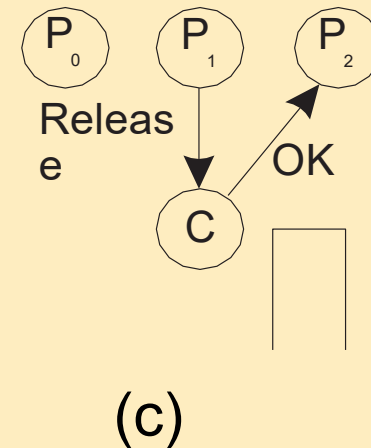
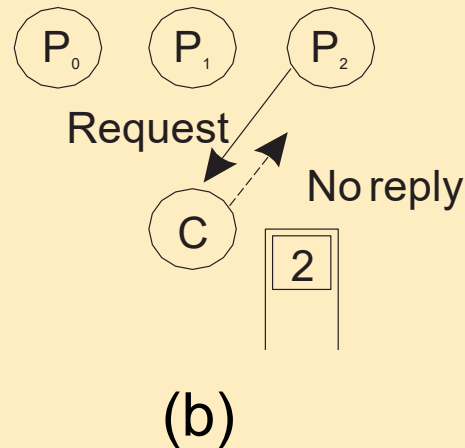
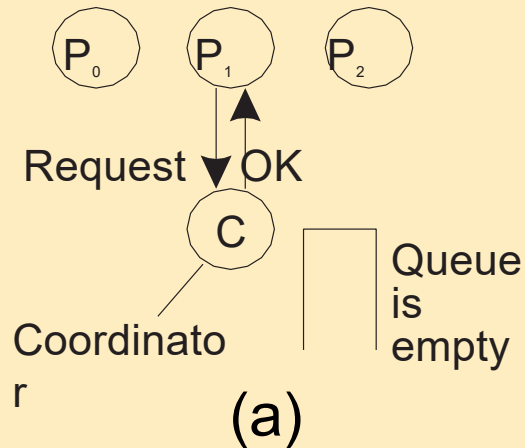
Basic solutions

Permission-based: A process wanting to enter its critical section, or access a resource, needs permission from other processes.

Token-based: A token is passed between processes. The one who has the token may proceed in its critical section, or pass it on when not interested.

Permission-based, centralized

Simply use a coordinator



- (a) Process P_1 asks the coordinator for permission to access a shared resource. Permission is granted.
- (b) Process P_2 then asks permission to access the same resource. The coordinator does not reply.
- (c) When P_1 releases the resource, it tells the coordinator, which then replies to P_2 .

Mutual exclusion Ricart & Agrawala

The same as Lamport except that acknowledgments are not sent

Return a response to a request only when:

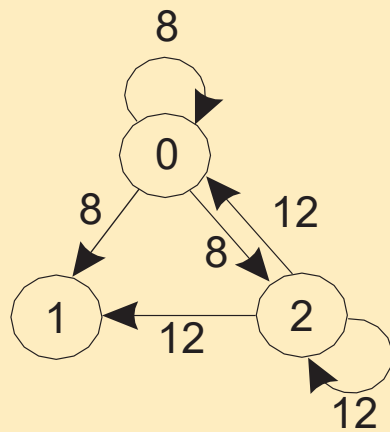
- The receiving process has no interest in the shared resource; or

- The receiving process is waiting for the resource, but has lower priority (known through comparison of timestamps).

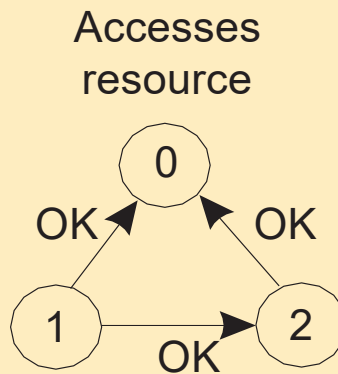
In all other cases, reply is **deferred**, implying some more local administration.

Mutual exclusion Ricart & Agrawala

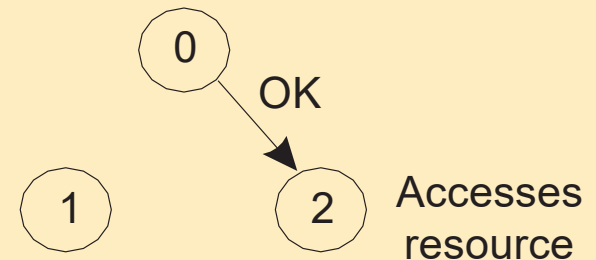
Example with three processes



(a)



(b)



(c)

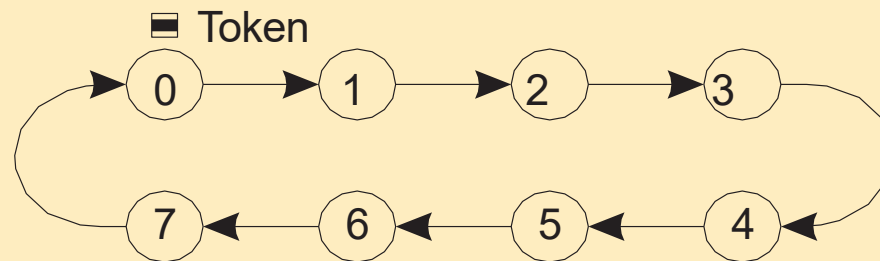
- (a) Two processes want to access a shared resource at the same moment.
- (b) P_0 has the lowest timestamp, so it wins.
- (c) When process P_0 is done, it sends an *OK* also, so P_2 can now go ahead.

Mutual exclusion: Token ring algorithm

Essence

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

An overlay network constructed as a logical ring with a circulating token



Decentralized mutual exclusion

Principle

Assume every resource is replicated N times, with each replica having its own coordinator \Rightarrow access requires a **majority vote** from $m > N/2$ coordinators. A coordinator always responds immediately to a request.

Assumption

When a coordinator crashes, it will recover quickly, but will have forgotten about permissions it had granted.

Decentralized mutual exclusion

How robust is this system?

Let $p = \Delta t / T$ be the probability that a coordinator resets during a time interval Δt , while having a lifetime of T .

The probability $P[k]$ that k out of m coordinators reset during the same interval is

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

f coordinators reset \Rightarrow correctness is violated when there is only a minority of nonfaulty coordinators: when $m - f \leq N/2$, or, $f \geq m - N/2$.

The probability of a violation is $\sum_{k=m-N/2}^m P[k]$.

Decentralized mutual exclusion

Violation probabilities for various parameter values

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

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

So....

What can we conclude?

Mutual exclusion: comparison

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

Election algorithms

Principle

An algorithm requires that some process acts as a coordinator. The question is how to select this special process **dynamically**.

Note

In many systems the coordinator is chosen by hand (e.g. file servers). This leads to centralized solutions \Rightarrow single point of failure.

Teasers

- 1 If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?
- 2 Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?

Basic assumptions

All processes have unique id's

All processes know id's of all processes in the system (but not if they are up or down)

Election means identifying the process with the highest id that is up

Election by bullying

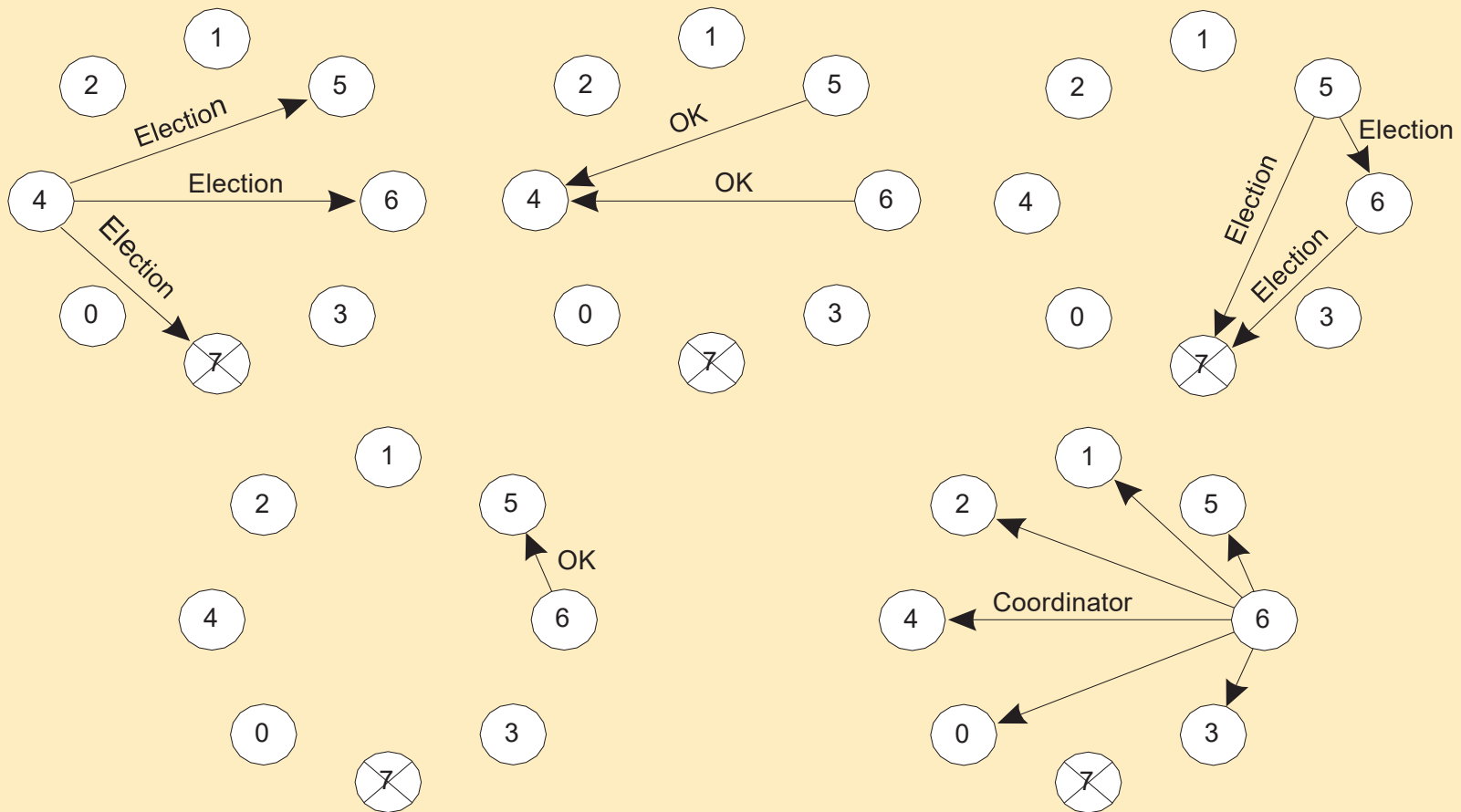
Principle

Consider N processes $\{P_0, \dots, P_{N-1}\}$ and let $id(P_k) = k$. When a process P_k notices that the coordinator is no longer responding to requests, it initiates an election:

- 1 P_k sends an *ELECTION* message to all processes with higher identifiers: $P_{k+1}, P_{k+2}, \dots, P_{N-1}$.
- 2 If no one responds, P_k wins the election and becomes coordinator. If
- 3 one of the higher-ups answers, it takes over and P_k 's job is done.

Election by bullying

The bully election algorithm



Election in a ring

Principle

Process priority is obtained by organizing processes into a (logical) ring. Process with the highest priority should be elected as coordinator.

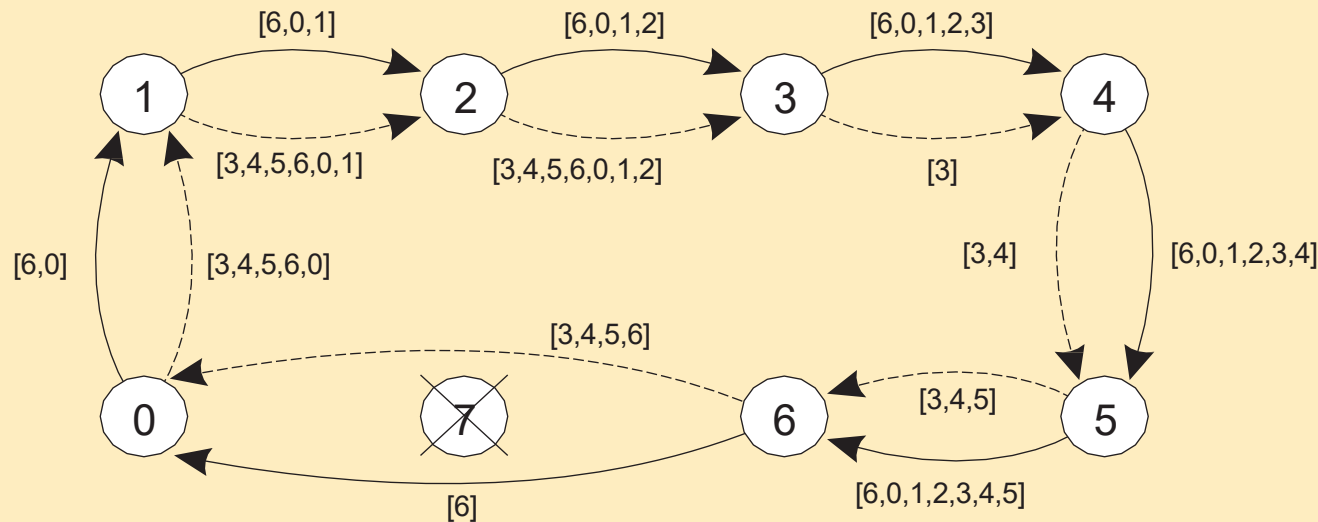
Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on to the next successor.

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.

Election in a ring

Election algorithm using a ring

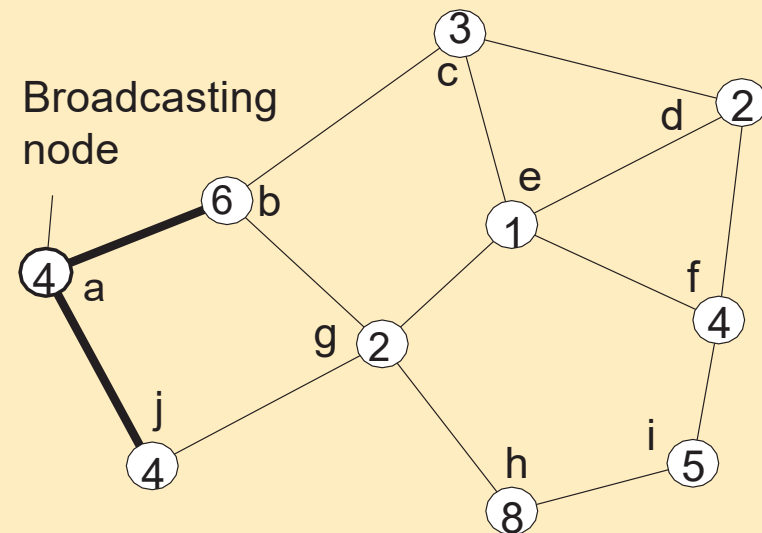
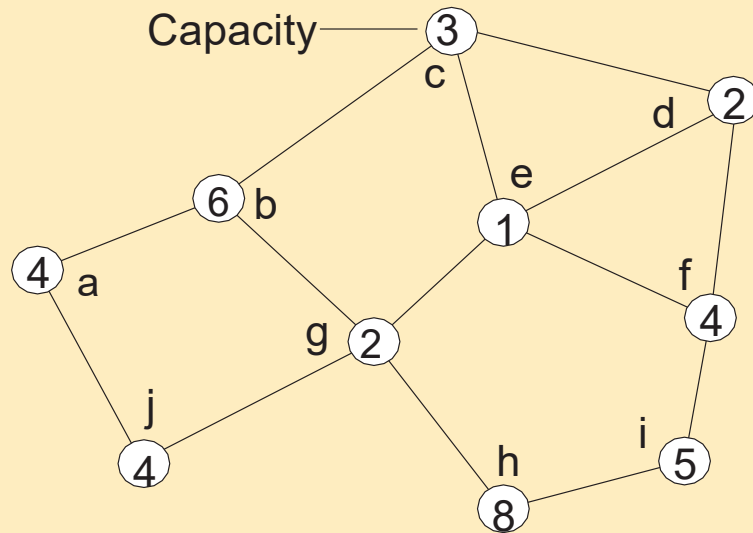


The solid line shows the election messages initiated by P_6

The dashed one the messages by P_3

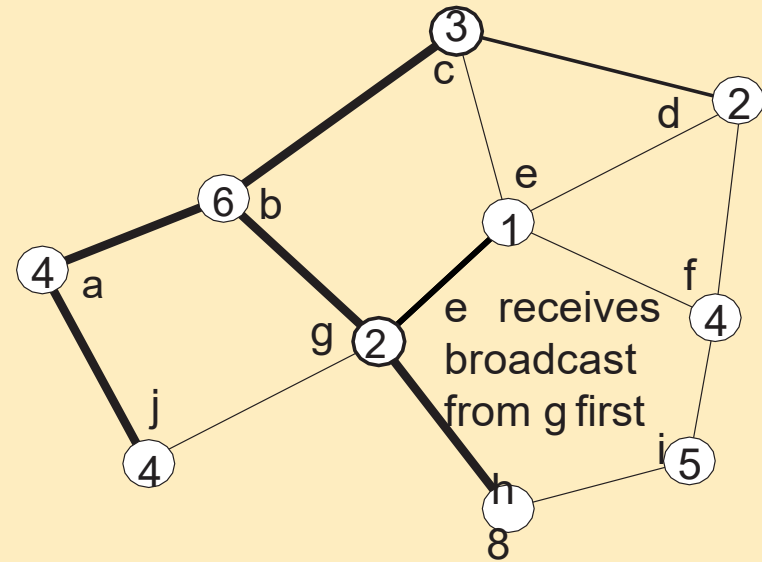
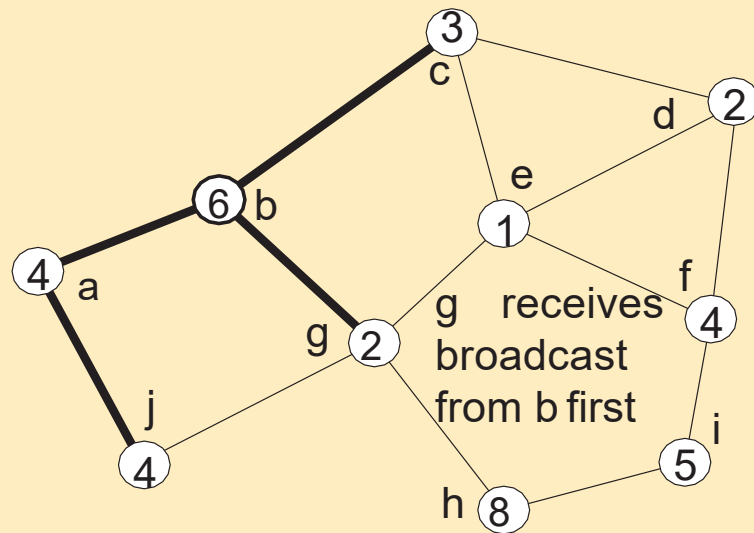
A solution for wireless networks

A sample network



A solution for wireless networks

A sample network



A solution for wireless networks

A sample network

