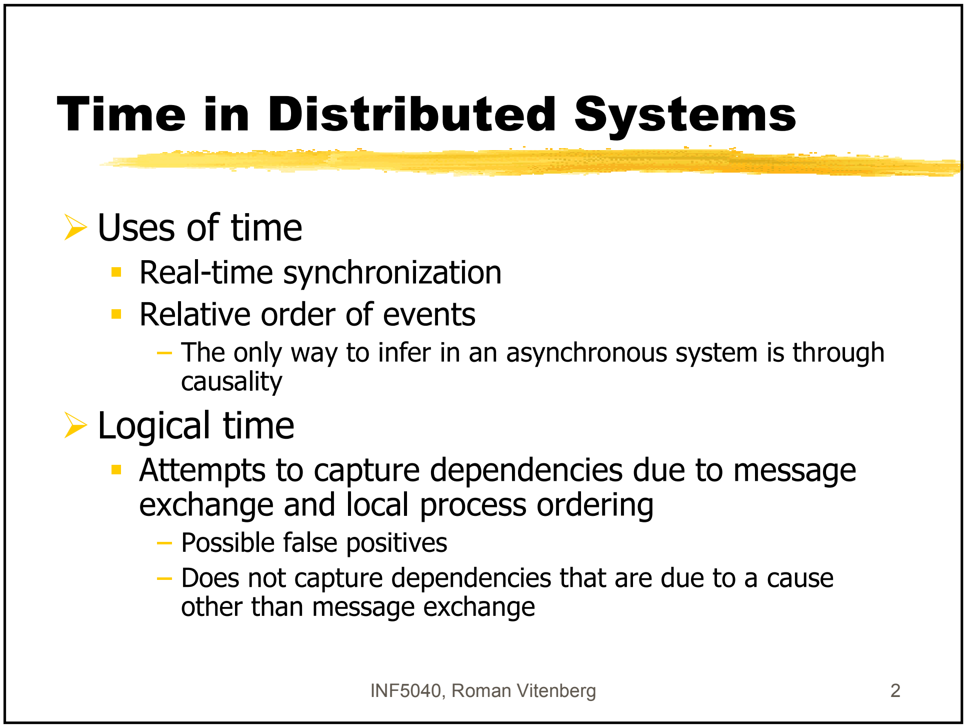
# Time and Coordination

## INF 5040 autumn 2007

**lecturer: Roman Vitenberg**



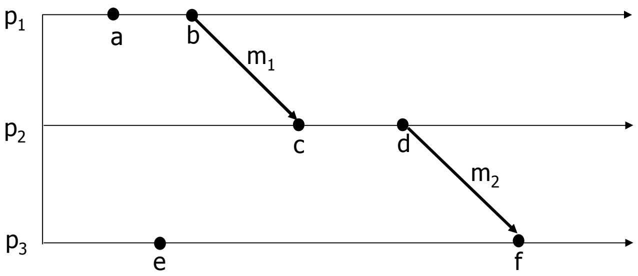
# Time in Distributed Systems

- Uses of time
  - Real-time synchronization
  - Relative order of events
    - The only way to infer in an asynchronous system is through causality
- Logical time
  - Attempts to capture dependencies due to message exchange and local process ordering
    - Possible false positives
    - Does not capture dependencies that are due to a cause other than message exchange

# "Happened-before" relation

- Notation
  - $x \rightarrow^{p} y$: $x$ happened before $y$ at process $p$
  - $x \rightarrow y$: $x$ happened before $y$
- Condition 1
  - If $\exists$ process $p$ : $x \rightarrow^{p} y$, then $x \rightarrow y$
- Condition 2
  - For each process $m$ : $send(m) \rightarrow rcv(m)$
- Condition 3
  - If $x$, $y$, and $z$ are events such that $x \rightarrow y$ and $y \rightarrow z$, then $x \rightarrow z$

# "Happened-before" illustrated

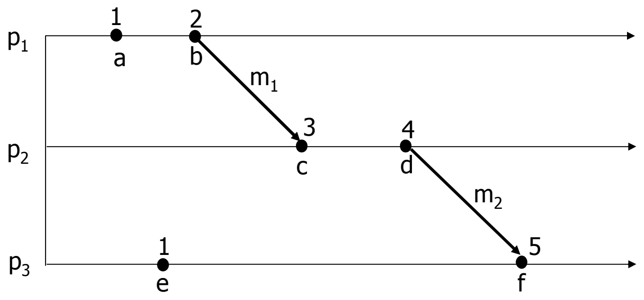


- Events that are not related by the "happened-before" relation are called concurrent: $a \; || \; e$

# Logical clock

- Each process `p` maintains its own logical clock $C_p$
  - Monotonically increasing counter
  - Used to timestamp events
- $C_p(a)$ : the timestamp of event `a` at process `p`
- Rules for logical clock
  - LC1:
    - $C_p$ is incremented by 1 before each event is issued at process `p`
  - LC2:
    - When a process `p` sends a message `m`, it piggybacks $C_p$ on `m`
    - When `(m,t)` is received by `q`, `q` computes $C_q := \max(C_q, t)$ and applies LC1 before timestamping the event `rcv(m)`.

# Example for Logical Clocks

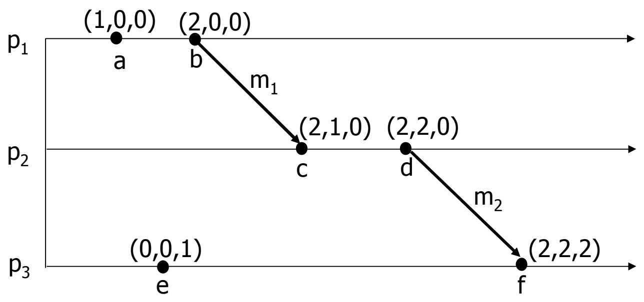


$x \rightarrow y \Rightarrow C(x) < C(y)$ (not equivalent!!)

# Vector clocks

- Each process `p` maintains its vector clock $V_p$ of size N
- $V_p(a)$ : the timestamp of event `a` at process `p`
- Rules for vector clock
  - VC1: $V_p[j]$ is initially 0 for all j
  - VC2:
    - `p` sets $V_p[p]:=V_p[p]+1$ before timestamping each event
  - VC3:
    - When `p` sends a message `m`, it piggybacks $V_p$ on `m`
  - VC4: When `(m,t)` is received by `q`, `q` computes $V_q[j]:=\max(V_q[j],t[j])$ for all j

# Example for Vector Clocks



$x \rightarrow y \Leftrightarrow V(x) < V(y)$ (equivalent)

# Local events and states

The history ($h$) of a process is modelled as a sequence of events and corresponding states:

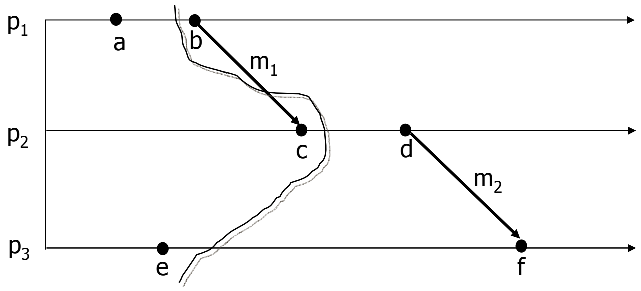
$$h_i = s_i^0, e_i^1 \leftrightarrow s_i^1, e_i^2 \leftrightarrow s_i^2, e_i^3 \leftrightarrow s_i^3, \cdots$$

Sometimes it is assumed that sending a message does not alter the local state

Sometimes we are only interested in the events:

$$h_i = e_i^1, e_i^2, e_i^3, \cdots$$

# Global histories and cuts



- Cut: union of prefixes of process histories
  - May be consistent or not

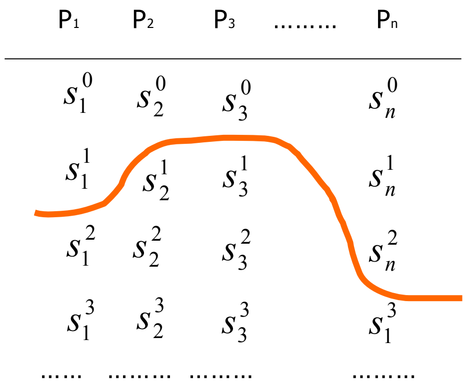
# Consistent cuts

Cut $C$ is consistent if

$$e \in C \wedge (f \rightarrow e) \Rightarrow f \in C$$

We are only interested in consistent cuts!

# Global state



| $P_1$ | $P_2$ | $P_3$ | ……… | $P_n$ |
|---|---|---|---|---|
| $s_1^0$ | $s_2^0$ | $s_3^0$ | | $s_n^0$ |
| $s_1^1$ | $s_2^1$ | $s_3^1$ | | $s_n^1$ |
| $s_1^2$ | $s_2^2$ | $s_3^2$ | | $s_n^2$ |
| $s_1^3$ | $s_2^3$ | $s_3^3$ | | $s_1^3$ |
| …… | ……… | ……… | | ……… |

If local states do not include message sends, we additionally need to capture messages in transition

…consistent states correspond to consistent cuts...

## Linearization and properties

- Linearization is a full ordering of all events in a global history that preserves →
- State S′ is reachable from state S if there is a linearization that starts in S and ends in S′
- Property: a global state predicate
  - Stable property: if true in S, true in every state reachable from S
  - Safety property: true in every state reachable from $S_0$
  - Liveness property: in every linearization, there is a state reachable from $S_0$ in which it is true

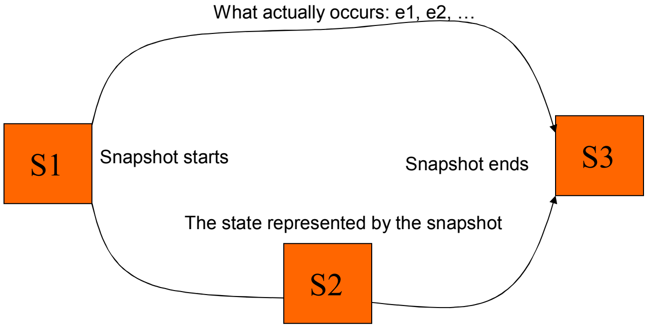
---

## Total ordering of events

- Logical clocks give a partial ordering
  - Events issued by different processes may have identical timestamps
- Extension to total ordering
  - Each process has a unique identifier
  - Process identifiers are totally ordered
- Global timestamp
  - $a$ is an event at $p_a$ with local timestamp $T_a$
  - Global logical timestamp for $a$ are $(T_a, p_a)$
- $(T_a, p_a) < (T_b, p_b)$ if $T_a < T_b$, or $T_a = T_b$ and $p_a < p_b$

# The snapshot problem

- Finds a consistent global state that may have occurred



What actually occurs: e1, e2, …

S1

Snapshot starts

Snapshot ends

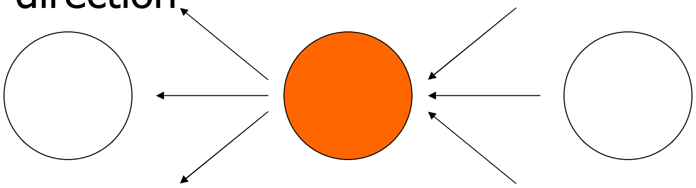
S3

The state represented by the snapshot

S2

# Assumptions for the snapshot algorithm

- No process or network link fails
- Network links preserve FIFO
- Full network: each pair of processes connected by two network links, one in each direction



# Responsibility of the processes

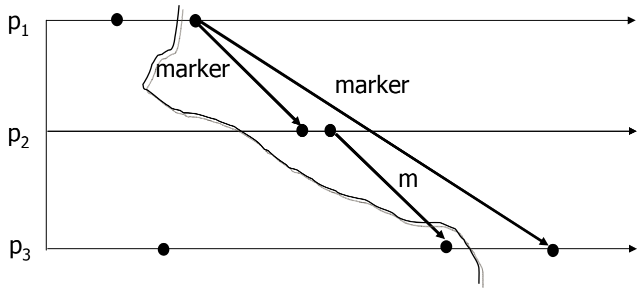
- Every process can initiate a snapshot
  - A process takes initiative to log its own state and sends a marker message on all output channels.
- Each process has responsibility for
  - Logging its own state,
  - Logging the incoming messages on input channels,
  - Sending or forwarding the marker.
- Upon termination, the collection of local states of processes and recorded states of channels should give us a consistent global state

# The first attempt

```
P sends a marker over all outgoing links…
P waits until it receives a marker on all input channels
P logs its own state

When another process Q receives a marker
        Q logs its own state
        Q sends the marker back to P…
```

# Is the protocol correct?



- The captured state may be inconsistent
- It does not capture messages in transit

# Correct snapshot protocol

- [Chandy,Lamport 1985]
- The procedure to start the snapshot

```
P logs its own state.
P sends a marker over all outgoing links.
P starts to log imcoming messages on all
        input channels
```

# The procedure upon marker reception

```
When P receives a marker over channel c
  IF P has not recorded its state
     P records its state.
     P forwards the marker over all output channels
     P initialize the state of c to the empty set
     P starts to record incoming messages
        on all other input channels
  ELSE
     P records the state of c:
             all the messages that have been
             received on c since P recorded
             its own state, which are said to be
             in transition over the channel
  END
```

# Proof of protocol correctness

- The recorded state is consistent.
  - If x → y, and y occurred at p before p recorded its state, then x must have been received at q before q recorded its state.
- State S2 must be reachable from S1.
- State S3 must be reachable from S2.
-

# Distributed consensus

- N processes out of which at most f can be faulty
- Two possible input values, 0 or 1
- Agreement (also called correctness)
  - No two non-faulty processes decide on different values
- Termination
  - If there are non-faulty processes, at least one of them decides
- Integrity (or validity or non-triviality)
  - if all non-faulty processes start with the same initial value v, then v is the only possible decision value for a non-faulty process

# Other agreement problems

- Reliable multicast (also called terminating reliable broadcast)
- Group membership
- Leader election
- Distributed locking
- Mutual exclusion
- Atomic transactions
- Resource allocation

## Reliable broadcast

- One sender that sends a single message
- Termination: Every non-faulty process delivers a message (possibly ⊥)
- Agreement: No two non-faulty processes deliver different messages
- Validity: no spurious messages
- Integrity: If the sender is non-faulty, it delivers the message it sent

## Group membership

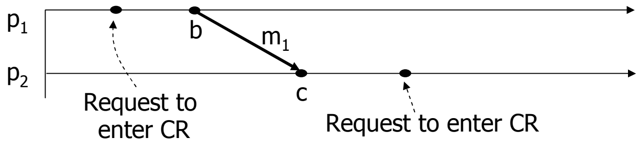
- Each process starts with a list of processes it considers correct
- Agreement on the list of participating processes
- Validity 1: If a process is in all input lists, then it will be in the decided list
- Validity 2: If a process is in no input list, then it will not be in the decided list

# Known impossibility results

- Impossible to solve if faulty processes are malicious and N ≤ 3f
  - Can be alleviated by using digital signatures
- Impossible to solve in asynchronous systems
  - Can be circumvented by masking faults
  - Or by designating the process that adds to asynchrony as faulty
  - Or by using randomization

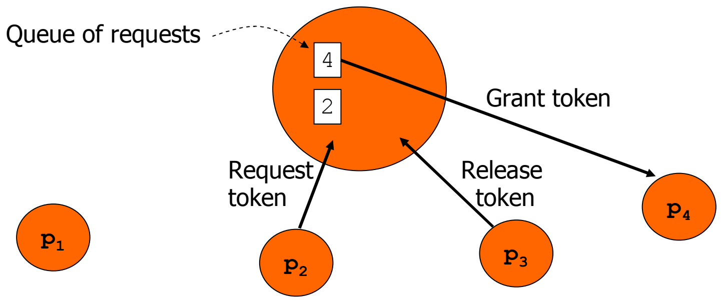
# Mutual exclusion problem

- Safety:
  - At most one process can be in a critical section at a time
- Liveness:
  - Each request to enter or exit the critical section eventually succeeds (as long as the process that executes in the critical section eventually requests to leave it)
- Ordering:
  - Entrance to the CS must observe the "happened-before" relation



# Central server algorithm

- Central server that grants entrance to the critical section
- protocol
  - enter() -- enter critical section - blocks if necessary
    exit() -- leaves critical section - other processes can now enter



# Evaluation of the
## central server algorithm

- Are safety and liveness satisfied?
- Is ordering satisfied?
  - How to ensure it?
- Shortcomings of the algorithm
  - Performance bottleneck
  - The server can fail
    - We can make one of the clients a new server
    - Requires distributed election
    - How to ensure that the old order preceding the failure is preserved?
- Client with the token may fail
  - How to ensure that the token becomes accessible again?

# Ring-based algorithm

- A token rotating in one direction
- A process can enter the critical section when it has the token
- When a process that has not requested to enter receives a token, it passes the token on

# Evaluation of the ring-based algorithm

- Fault-tolerance
  - Problematic when a node crashes
    - Mend the ring
    - Ensure that the ring contains exactly one token
- No central bottleneck
  - Redundant messages are sent if no process attempts to enter the critical section
- Safety and liveness are trivially satisfied, but ordering requires an additional mechanism

# Distributed algorithm based on logical clocks

- Basic idea [Ricart & Agrawala, 1981]:
    - A process that wishes to enter a critical section, multicasts a message to all the processes
    - A process can enter a CS when it gets acks from all the processes
    - Rules wrt when to send an ack in order to ensure fulfillment of the requirements
- Assumptions
    - Processes know each other addresses
    - Every sent message will eventually be delivered
    - Each process maintains a logical clock
    - Timestamps include processId: `<T,p>` (i.e., total ordering)
    - Each process maintains its state wrt token pocession
        - RELEASED, WANTED, HELD

---

# Ricart & Agrawala algorithm

```
Upon initialization
    state := RELEASED;
To enter the critical section
    state:= WANTED;
    Multicast a request to all the processes
    T := the current timestamp;
    wait until ((n-1) acks are received);
    state := HELD;
Upon receiving a request with $<T_i, p_i>$ at $p_j$ ($i \neq j$)
    if (state=HELD or (state=WANTED and $(T,p_j) < (T_i,p_i)$ ) )
        queue the request from $p_i$ without replying
    else
        send an ack to $p_i$
    end if
Upon exiting from the critical section
    state := RELEASED
    reply to all queued messages
```

# Evaluating of the Ricart & Agrawala algorithm

- Are safety and liveness satisfied?
- Is ordering satisfied?
- Shortcomings
  - Many messages are sent in order to enter critical section
    - 2(n-1) messages without HW support for multicast
    - n messages with HW support for multicast
  - Not resilient to process crashes

# Summary of distributed mutual exclusion algorithms

- Little resilience to failures
  - Can be improved by additional mechanisms
  - But it will never be perfect in an asynchronous system
- Central server requires the lowest number of messages but can become a bottleneck

# Requirements for distributed leader election

- In many distributed algorithms, one of the participating processes will play the role of a central coordinator
  - Central server in the mutual exclusion algorithms
  - Coordinator of a distributed transaction
- If a coordinator fails, one of the remaining processes can be elected to take over the central role
  - Provide better fault-tolerance
- The main requirement
  - Only one leader may exist at a time

# The “Bully” algorithm

- [Silberschatz et al, 1993]
- Prerequisites
  - The processes know each other identities and addresses
  - Process identifiers are totally ordered
  - The algorithm selects the process with the biggest identifier
- Message types

| | |
|---|---|
| `election`: | announces an election |
| `answer`: | is sent as a reply to the election message |
| `coordinator`: | announces the identity of the new coordinator |

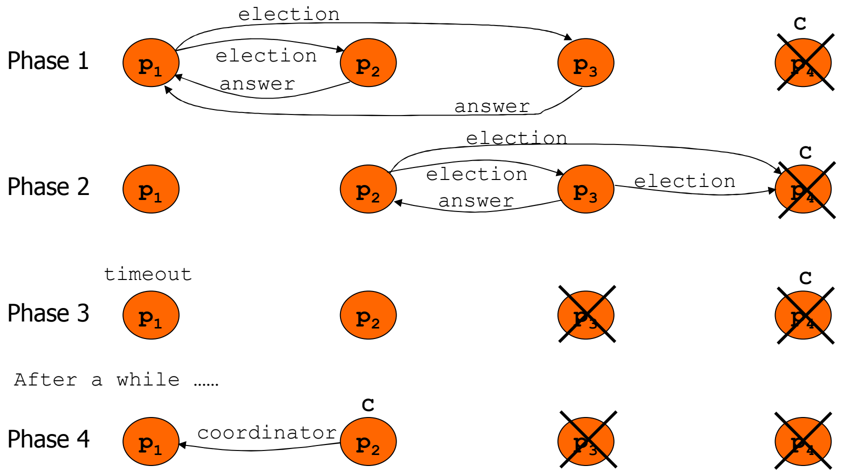
# The “Bully” algorithm II

- Election procedure
  - The process (that detects that the coordinator has failed) sends the `election` message to the processes that have a bigger identifier
  - It then waits for the `answer` message a limited amount of time
  - If no `answer` message is received, the process considers itself as a new coordinator and sends a `coordinator` message to all the processes with smaller identifiers
  - If an `answer` message is received, the process wait a limited time for a `coordinator` message. If none arrives, it starts a new election.

# The “Bully” algorithm III

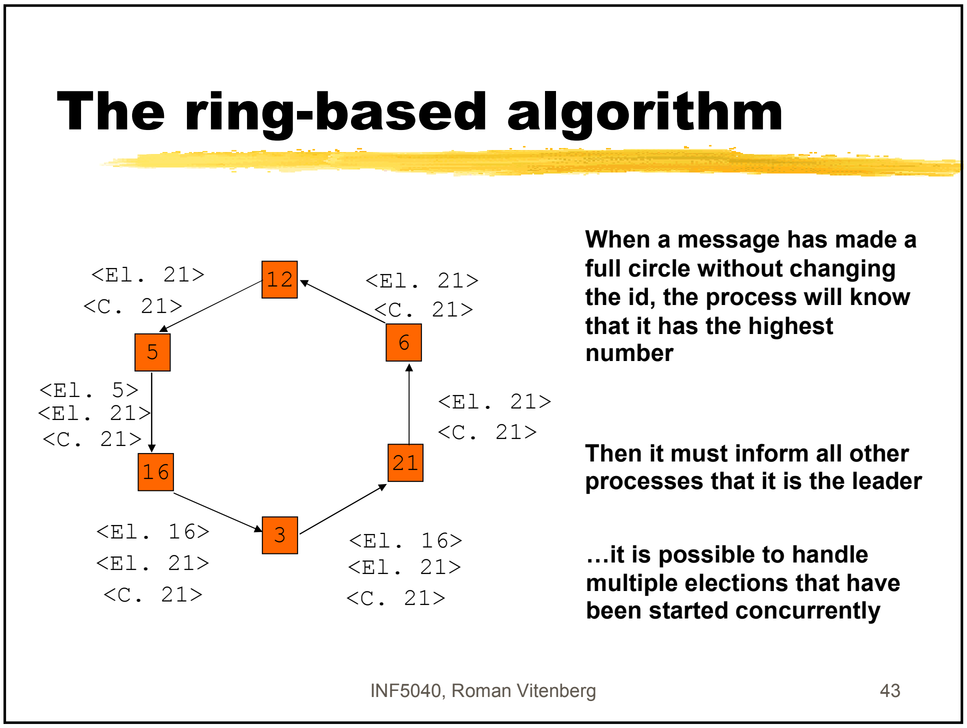
- Election procedure (continued)
  - If a process receives a `coordinator` message, it memorizes the identifier included in the message and considers the process as the new coordinator
  - If a process receives an `election` message, it sends back an `answer` message and starts a new election - unless the process has already started one
  - When a process recovers or joins the system, it starts a new election. If it has the biggest identifier, it makes itself a coordinator and announces it, even if there is another functioning coordinator

# Illustration for the “bully” algorithm



# Evaluation of the “bully” algorithm

- Best case: n-2 `coordinator` messages
  - Occurs when the process with the second highest id detects that the coordinator has failed
- Worst case: O($n^2$) messages
  - Occurs when the process with the lowest id detects that the coordinator has failed
  - => (n-1) processes start an election
- Ring-based algorithm is more efficient wrt the number of messages



# The ring-based algorithm

When a message has made a full circle without changing the id, the process will know that it has the highest number

Then it must inform all other processes that it is the leader

…it is possible to handle multiple elections that have been started concurrently