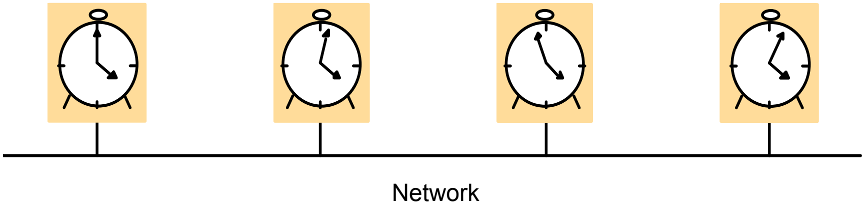
Time and Coordination in Distributed Systems

Operating Systems

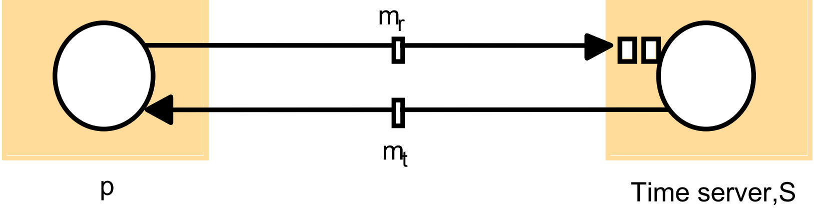
Clock Synchronization

- Physical clocks drift, therefore need for clock synchronization algorithms
  - Many algorithms depend upon clock synchronization
  - Often we need to know the order in which two events occurred on two different computers
  - Clock synch. Algorithms – Christian, NTP, Berkeley algorithm, etc.
- However, since we cannot perfectly synchronize clocks across computers, we cannot use physical time to order events

Skew between computer clocks in a distributed system



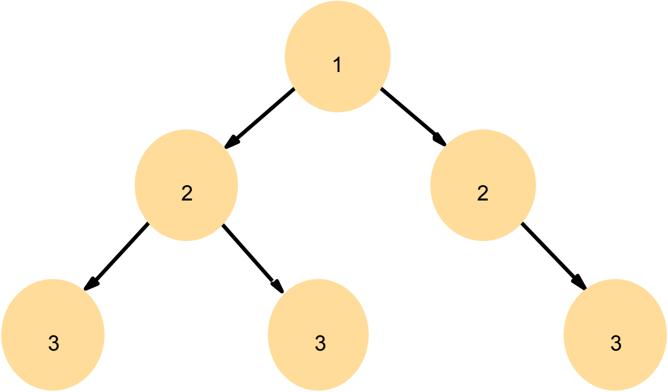
Clock synchronization using a time server



Clock synchronization algorithms

- Cristian’s algorithm
  - p should set its time to t + $T_{round}/2$
  - Earliest time at which S could have placed its time in m_t was min after p dispatched m_r
  - Latest point at which it could do so was min before m_t arrived at p
  - Time by S’s clock when message arrives at p is in range [t + min, t + T_{round} – min]
    - Accuracy $\pm(T_{round}/2\text{-min})$

An example synchronization subnet in an NTP implementation

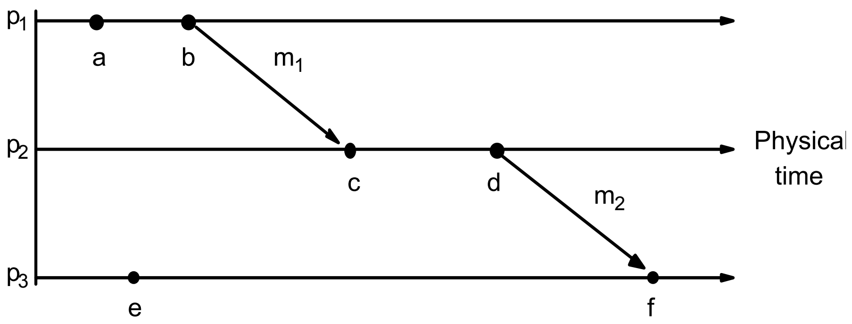


Note: Arrows denote synchronization control, numbers denote strata.

Logical time & clocks

- Lamport proposed using logical clocks based upon the "happened before" relation
  - If two events occur at the same process, then they occurred in the order observed
  - Whenever a message is sent between processes, the event of sending occurred before the event of receiving
  - X happened before Y denoted by X→Y

Events occurring at three processes

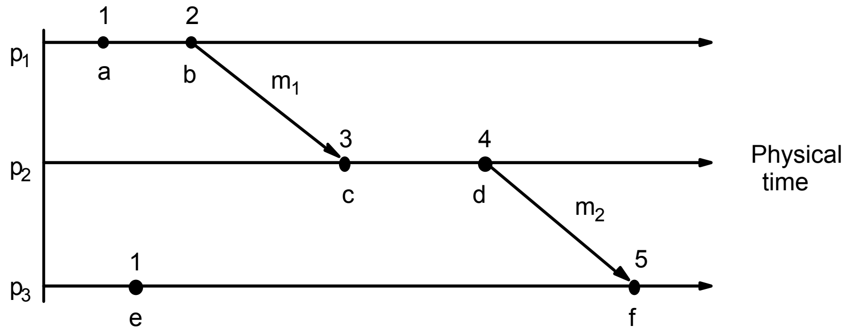


Lamport's algorithm

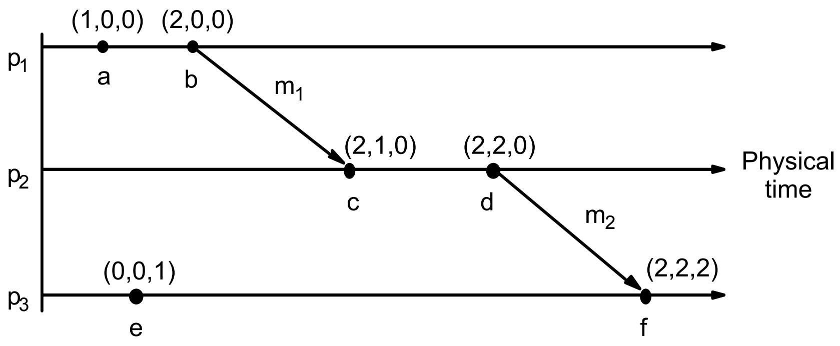
- Each process has its own logical clock
- LC1: C_p is incremented before each event at process p
- LC2:
  1. When process p sends a message it piggybacks on it the value C_p
  2. On receiving a message (m,t) a process q computes $C_q = \max(C_q, t)$ and then applies LC1 before timestamping the receive event

---

Lamport timestamps for the events



Vector timestamps for the events



Totally ordered logical clocks

- Logical clocks only impose partial ordering
- For total order, use (T_a, P_a) where P_a is processor id
- $(T_a, P_a) < (T_b, P_b)$ if and only if either $T_a < T_b$ or $(T_a = T_b$ and $P_a < P_b)$

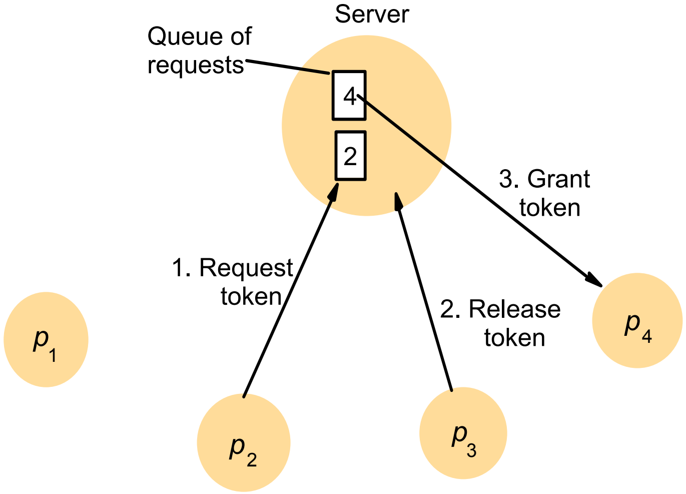
Distributed mutual exclusion

- Central server algorithm
- Ricart and Agrawal algorithm
  - A distributed algorithm that uses logical clocks
- Ring-based algorithms

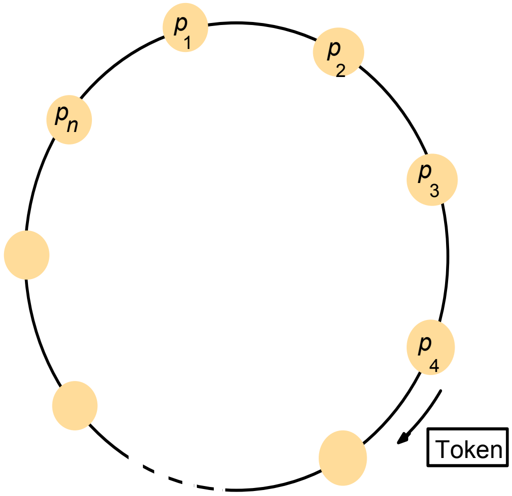
NOTE: the above algorithms are not fault-tolerant and not very practical. However, they illustrate issues in the design of distributed algorithms

- Several other mutual exclusion algorithms have been proposed
  - Quorum consensus algorithms – Maekawa's algorithm

Server managing a mutual exclusion token for a set of processes



A ring of processes transferring a mutual exclusion token

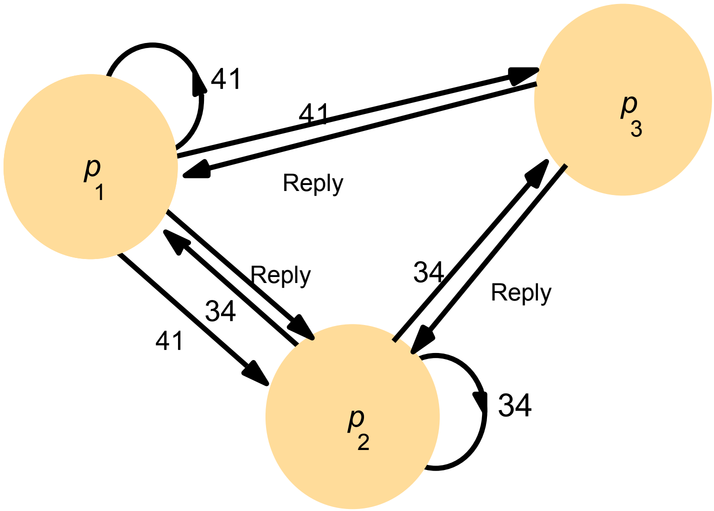


Ricart and Agrawala's algorithm

```
On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast request to all processes;      } request processing deferred here
    T := request's timestamp;
    Wait until (number of replies received = $(N – 1)$);
    state := HELD;

On receipt of a request $<T_i, p_i>$ at $p_j$ $(i \neq j)$
    if (state = HELD or (state = WANTED and $(T, p_j) < (T_i, p_i)$))
    then
        queue request from $p_i$ without replying;
    else
        reply immediately to $p_i$;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```

Multicast synchronization



Maekawa’s algorithm

- Every node needs permission from the other nodes in its quorum before it enters the critical section
- Quorums are constructed in such a way that no two nodes can be in their critical section at the same time
- The size of each nodes quorum is O(sqrt(N)), which can proved to be optimal

Construction of coteries

Consider a system with 9 nodes

1	2	3
4	5	6
7	8	9

The quorum for any node includes the other nodes in the same row and column

Node 1’s quorum = {1,2,3,4,7}
Node 2’s quorum = {1,2,3,5,8}
Node 6’s quorum = {4,5,6,3,9}

The quorum of any two nodes have a non-null intersection. This ensures that two nodes cannot get permission to enter their critical section at the same time

---

Maekawa’s algorithm

```
On initialization
    state := RELEASED;
    voted := FALSE;
For $p_i$ to enter the critical section
    state := WANTED;
    Multicast request to all processes in $V_i - \{p_i\}$;
    Wait until (number of replies received = $(K - 1)$);
    state := HELD;
On receipt of a request from $p_i$ at $p_j$ $(i \neq j)$
    if (state = HELD or voted = TRUE)
    then
        queue request from $p_i$ without replying;
    else
        send reply to $p_i$;
        voted := TRUE;
    end if
```

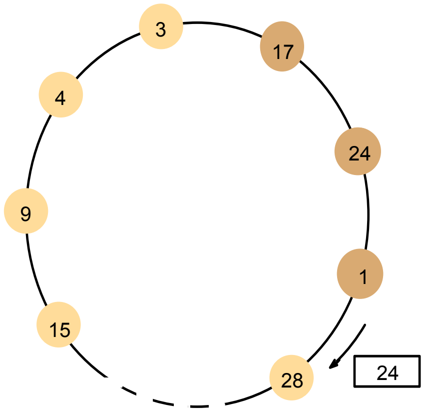
Maekawa's algorithm – cont'd

```
For $p_i$ to exit the critical section
    state := RELEASED;
    Multicast release to all processes in $V_i - \{p_i\}$;
On receipt of a release from $p_i$ at $p_j$ $(i \neq j)$
    if (queue of requests is non-empty)
    then
        remove head of queue – from $p_k$, say;
        send reply to $p_k$;
        voted := TRUE;
    else
        voted := FALSE;
    end if
```

Election Algorithms

- An election is a procedure carried out to chose a process from a group, for example to take over the role of a process that has failed
- Main requirement: elected process should be unique even if several processes start an election simultaneously
- Algorithms:
    - Bully algorithm: assumes all processes know the identities and addresses of all the other processes
    - Ring-based election: processes need to know only addresses of their immediate neighbors

A ring-based election in progress



Note: The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown darkened

The bully algorithm

The election of coordinator p_2,
after the failure of p_4 and then p_3