COMP4057
Distributed and Cloud Computing
COMP7940
Cloud Computing

Chapter 04

Technical Issues in
Distributed Systems (I)

Learning Outcomes

- Be able to understand the following technical issues in distributed systems and provide basic solutions
  - Time synchronization (part I)
  - Coordination and agreement (part I)
  - Transactions and concurrency control (part II)

Why is Timing Important?

- We often want to measure time accurately.
  - E.g., an e-Commerce transaction involves events at a merchant's computer and at a bank's computer. For auditing purposes, we need to timestamp the events accurately.

- Some basic time unit
  - 1 millisecond: 10^{-3} seconds
  - 1 microsecond: 10^{-6} seconds
  - 1 nanosecond: 10^{-9} seconds
  - 1 picosecond: 10^{-12} seconds

Clocks

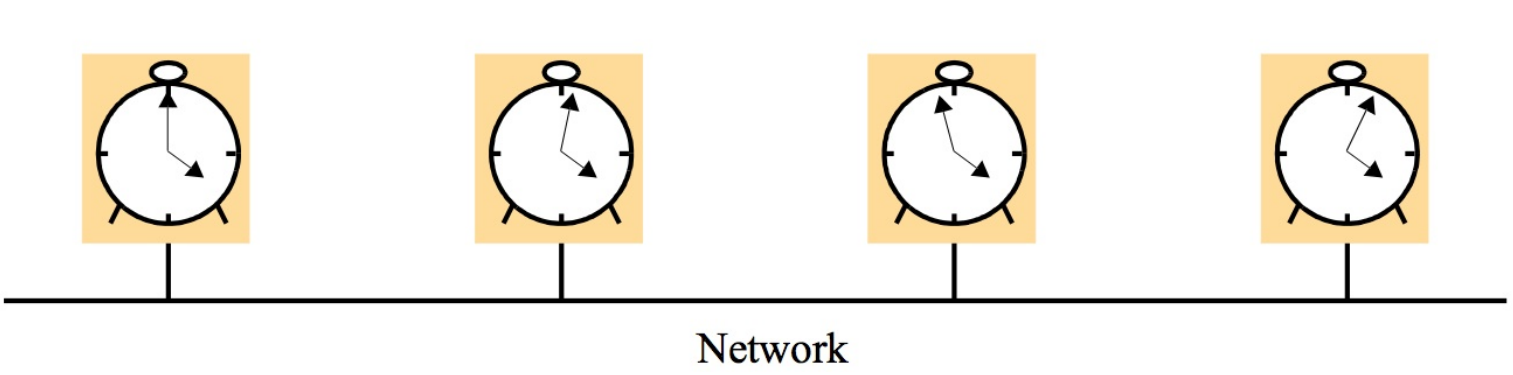
- How to timestamp the events?
  - Assign a date and time of day to each event.

- Computers each have their own physical clocks.
  - Electronic devices that count oscillations occurring in a crystal at a frequency, and divide this count and store the result in a counter register.

- Operating system reads the hardware clock value $H(t)$, scales it and adds an offset to produce a software clock $C(t) = \alpha H(t) + \beta$.
  - t is the actual amount of time elapsed since some reference point $(t = 0)$, e.g., $00{:}00{:}00$, Jan 1, 1970.
  - $C(t)$ is an approximation of the real time t.
    - Granularity of $C(t)$ is at least up to nanoseconds.
  - α and β are used to compensate for clock drift. (See below.)

Clock Skew

- Computer clocks tend not to be in perfect agreement.
- **Clock skew**: the instantaneous difference between the readings of any two clocks



Clock Drift

- Clock drift: different crystal-based clocks count time at different rates.
  - Two crystals of the same material have physical variations, so that their frequencies of oscillation differ slightly.
  - Even the same clock's frequency varies with temperature.

- Drift rate: the change in the offset between the clock and a nominal perfect reference clock per unit of time, measured by the reference clock
  - For ordinary clocks based on a quartz crystal, the drift rate is about 10^{-6} seconds/second, i.e., a difference of 1 second every 1,000,000 seconds (i.e., 11.6 days), or half a minute per year.
  - For high-precision quartz clocks, the drift rate can be 10^{-7} or 10^{-8}.
  - Drift rate can suddenly jump if the clock battery is very low.

Coordinated Universal Time

- We may synchronize computer clocks to some external sources with highly accurate time.
  - E.g., atomic clocks use atomic oscillators to achieve drift rate of about 10^{-13}.

- UTC: Coordinated Universal Time
  - (UTC comes from the French abbreviation.)
  - An international standard for timekeeping
  - Based on atomic time, but a "leap second" is occasionally inserted or deleted
    - to maintain astronomical time definition of 1 year = 1 earth rotation about the sun and 1 day = 1 rotation of earth on its axis
  - UTC signals are synchronized and broadcast regularly from land-based radio stations and satellites (including GPS)

- GPS receivers can have a time accuracy (i.e., clock skew with UTC time) of about 1 microsecond.
- Land-based receivers have an accuracy of 0.1 to 10 milliseconds.

Synchronizing Physical Clocks

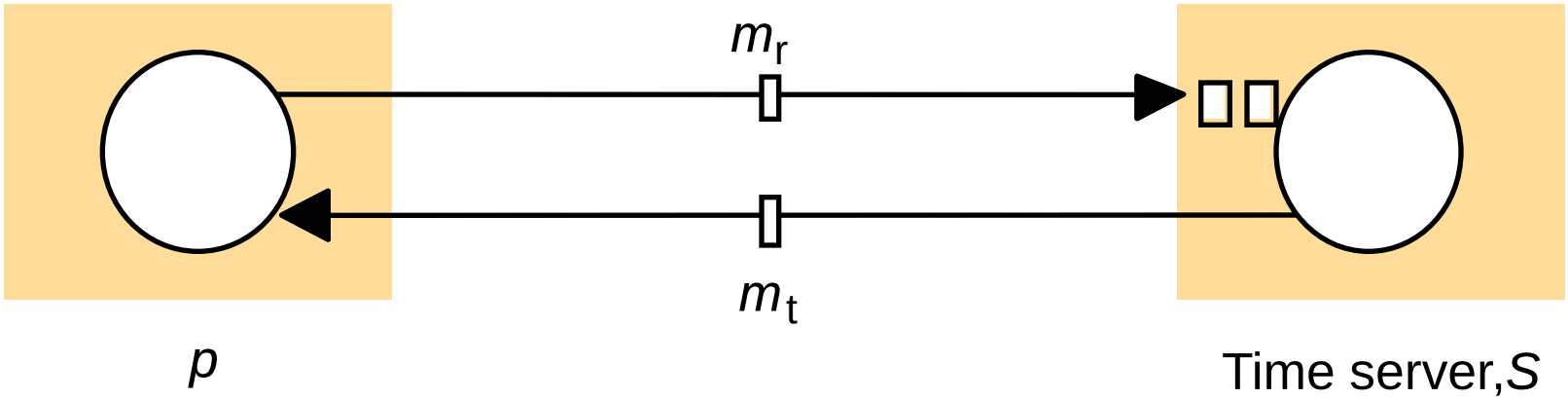
- External synchronization:
  - Synchronize a group of clocks $C_i, i = 1, \ldots, N$ with an authoritative external source S of UTC time.
  - Given a synchronization bound $D > 0$, synchronize $C_1, \ldots, C_N$ so that $|S(t) - C_i(t)| < D$ for all $i = 1, \ldots, N$ and t.
    - D is a very small number, e.g., $10^{-6}, 10^{-7}$, etc.
    - The clocks C_i are very close to S to within the small value D.
    - t lies within a "small real time interval" I, e.g., between 00:00:00 Dec 1, 2016 and 00:00:00 Dec 2, 2016.

Synchronizing Physical Clocks

- Internal synchronization:
  - Synchronize a group of clocks $C_i, i = 1, \dots, N$ with one another, without any external source of time.
  - Given a synchronization bound $D > 0$, synchronize $C_1, \dots, C_N$ so that $\left|C_i(t) - C_j(t)\right| < D$ for all $i, j = 1, \dots, N$ and t.
    - The clocks C_i agree with one another within the bound D.
  - Clocks that are internally synchronized may not be externally synchronized.
- If a distributed system is externally synchronized with a bound D, the system is internally synchronized with a bound $2D$.

External Synchronization by Cristian's Method

- A time server is connected to a device that receives signals from a source of UTC.
  1. A client process p requests the time in a message m_r.
  2. The time server returns the time t in a message m_t.
  3. Process p records the total round-trip time T_{round}.
  4. Process p sets its local time to $t + \frac{T_{round}}{2}$.



External Synchronization by Cristian's Method

- Accuracy of method (optional)
  - Let min be the minimum transmission time from p to S.
  - The earliest point at which S could have placed the time in m_t was min after p dispatches m_r.
  - The latest point at which S could have placed the time in m_t was min before m_t arrived at p.
  - The time by S's clock when the reply message arrives is therefore in the range $[t + min, t + T_{round} - min]$.
  - The width of this range is $(t + T_{round} - min) - (t + min) = T_{round} - 2\,min$.
  - Hence the accuracy is $\pm(\frac{T_{round}}{2} - min)$.
  - The accuracy is good, i.e., small, if $T_{round} \approx 2\,min$.

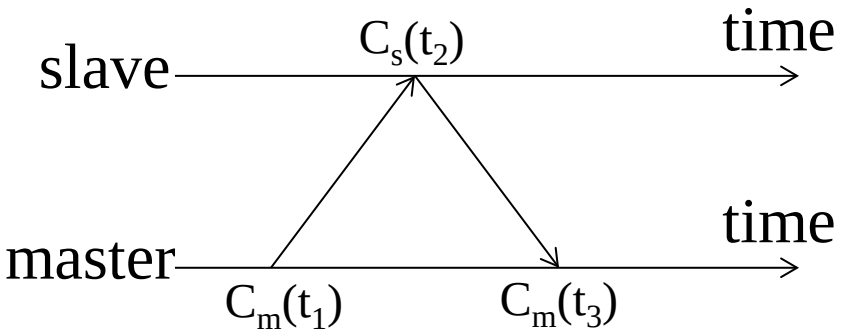
Internal Synchronization by Berkeley Algorithm

- In a group of computers, a coordinator computer is chosen to be the **master**. Other computers are **slaves.**
  - The master will maintain a "**network time**" agreed by all.
- The master periodically polls the slaves, and the slaves send back their clock values.
- The master estimates the differences between its own clock and the slaves' clocks by observing the round-trip times, and averages the differences.
  - Any values far outside the values of the others will be ignored.
  - This average cancels out the individual clocks' tendencies to run fast or slow.
- The master sets the "network time" as its local time adjusted by the average difference.
- The master adjusts its local time, and informs each slave about its amount of adjustment.
- The slaves adjust their clocks accordingly.

Illustrative Example of Berkeley Algorithm

- At beginning:
  - Master: 3:05 (*)
  - Slave 1: 2:55
  - Slave 2: 3:00
- By polling, master detects that
  - The time difference with Slave 1 is -10 seconds.
  - The time difference with Slave 2 is -5 seconds.
- The master calculates the average difference as:

  $\frac{(0-10-5)}{3} = -5$ seconds
- Master sets the "network time" to 3:05 – 0:05 = 3:00.
- Master informs Slave 1 to adjust its clock by $+5$ seconds.
- Master informs Slave 2 to adjust its clock by 0 second.



- Meaning of (*)
  - At time $C_m(t_1)$, masters sends a poll to slave.
  - The poll arrives at slave at time $C_s(t_2)$.
  - The slave sends its time $C_s(t_2)$ to master.
  - The reply arrives at master at time $C_m(t_3)$.
- Master uses the average time $\frac{(C_m(t_1)+C_m(t_3))}{2}$ as its time.
- The time difference between master and slave is $\frac{(C_m(t_1)+C_m(t_3))}{2} - C_s(t_2)$.
- In (*), for simplicity, we assume that the master time used for polling slaves 1 and 2 are both 3:05.

Process States

- Assume that a distributed system consists of N processes, $d_i, i = 1, \ldots, N$.
- A process p_i's state s_i includes the values of all its variables.
- A process can take two types of actions:
  - Send/receive operation
  - An operation that transforms its state.
- An event e is defined as the occurrence of a single action carried out by a process.
- $e \rightarrow_i e'$: event e occurs before e' at p_i.
- The history of process p_i : the series of events that take place within p_i, ordered by $\rightarrow_i$:
  - History(p_i)= $h_i =< {e_i}^0, {e_i}^1, {e_i}^2, \ldots, >$

Distributed Mutual Exclusion

- If a collection of processes share a resource (or collection of resources), one often uses **mutual exclusion** to prevent interference and ensure consistency when accessing the resources.
  - E.g., online ticket booking
- **Critical section**: a part of a multi-process program that may not be concurrently executed by more than one processes.
- Distributed mutual exclusion: in distributed system, there is **no shared variable** and the solution of mutual exclusion is based solely on message passing.

Problem Definition

- Consider a DS of N processes $p_i, i = 1, \ldots, N$ that do not share variables. They can access common resources in a critical section as follows:
  1. enter( ); // enter critical section – block if necessary
  2. resourceAccess( ); // access shared resources
  3. exit( ); // leave critical section, other processes may now enter
- Basic assumptions:
  - The system is asynchronous.
  - The processes do not fail.
  - Message delivery is reliable; any message sent is eventually delivered intact, exactly once.
- Essential requirements for mutual exclusion are:
  - ME1: **safety** – at most one process may execute in the critical section at a time.
  - ME2: **liveness** – requests to enter and exit the critical section eventually succeed, which implies freedom from both deadlock and starvation.
- Another possible requirement is:
  - ME3: **ordering** – if one request to enter the critical section happened-before another, then entry to the critical section is granted in that order.

Performance Concerns

- Algorithms for distributed mutual exclusion can be evaluated by the following criteria:
  - Consumed **bandwidth**: the number of messages sent in each enter() and exit() operation
  - **Client delay**: the incurred delay by a process at each enter() and exit() operation
  - System **throughput**: the rate at which the collection of processes as a whole can access the critical section. This can be measured by **synchronization delay** between one process exiting the critical section and the next process entering it.
    - The throughput is greater when the synchronization delay is shorter.

Central Server Algorithm

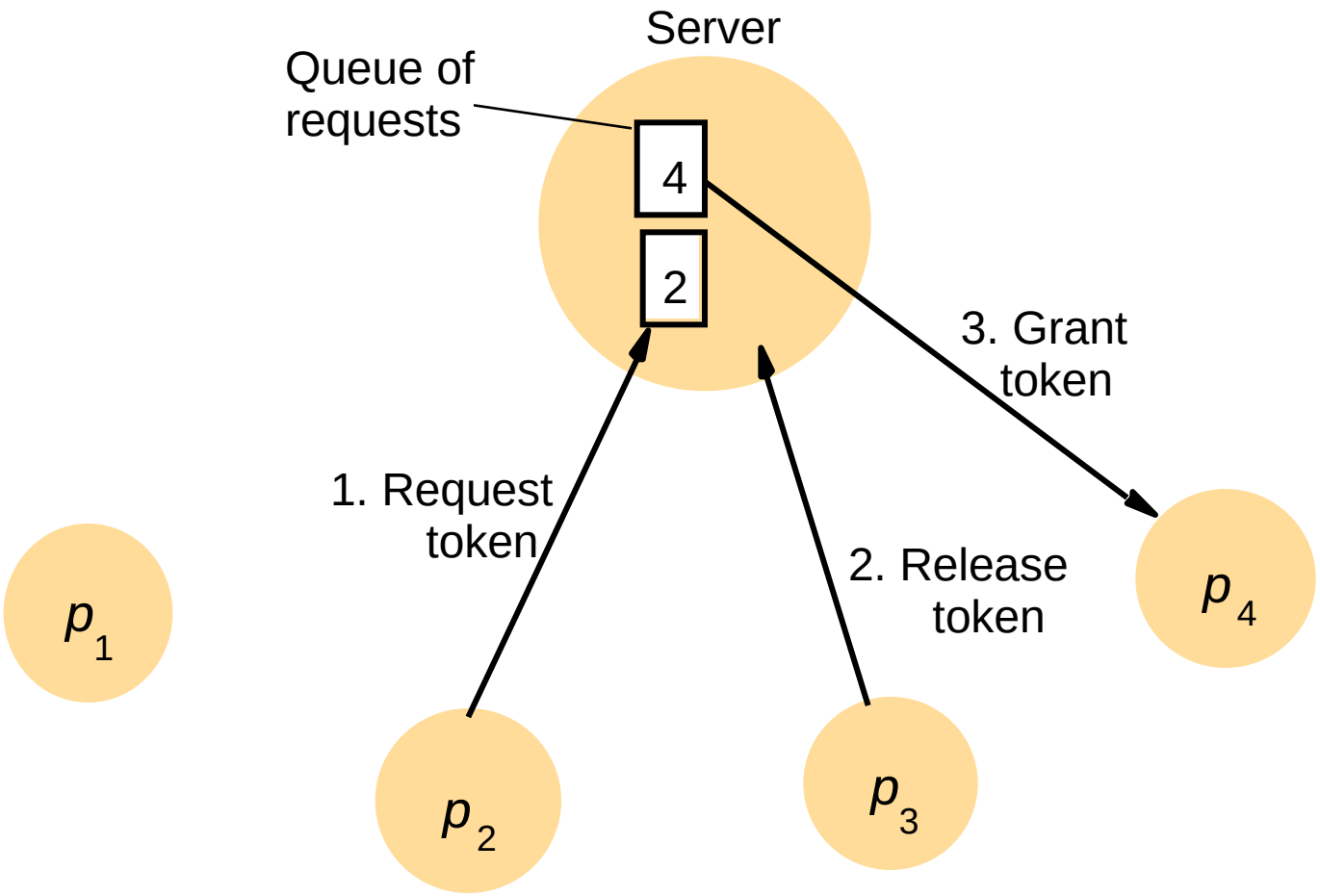


Fig. 15.2 Server managing a mutual exclusion token for a set of processes

Central Server Algorithm

- Idea: employ a server that grants permission to enter the critical section
- A client process's actions:
  - To enter a critical section, a process sends a request message to the server and waits for a reply.
  - The reply constitutes a token granting permission to enter the critical section.
  - If a client process receives the token from the server, it can enter the critical section.
  - If a process exits the critical section, it sends a message to the server, giving back the token to the server.
- The server's actions:
  - Upon receiving a token request:
    - If no other process has the token, then replies immediately, granting the token.
    - If the token is currently held by another process, the server does not reply, but queues the request.
  - Upon receiving a token release message:
    - If the queue is not empty, removes the oldest entry from the queue, and replies to the corresponding process, granting the token.

Performance Analysis

- Bandwidth cost:
  - It requires two messages to enter the critical section: a request message and a grant message.
  - It requires one message to exit the critical section: a release message.
- Client delay:
  - A round-trip delay for entering the critical section
  - One message to exit
- Synchronization delay: two messages -- the time for a release message to the server and a grant message to the next process
- ME1 (safety) and ME2 (liveness) are met, but not ME3 (ordering).
  - A client may make a request for entering the critical section early.
  - However, due to network delay, the request may arrive at the server at an unpredictable time.
- Server may become a performance bottleneck.

A Ring-based Algorithm

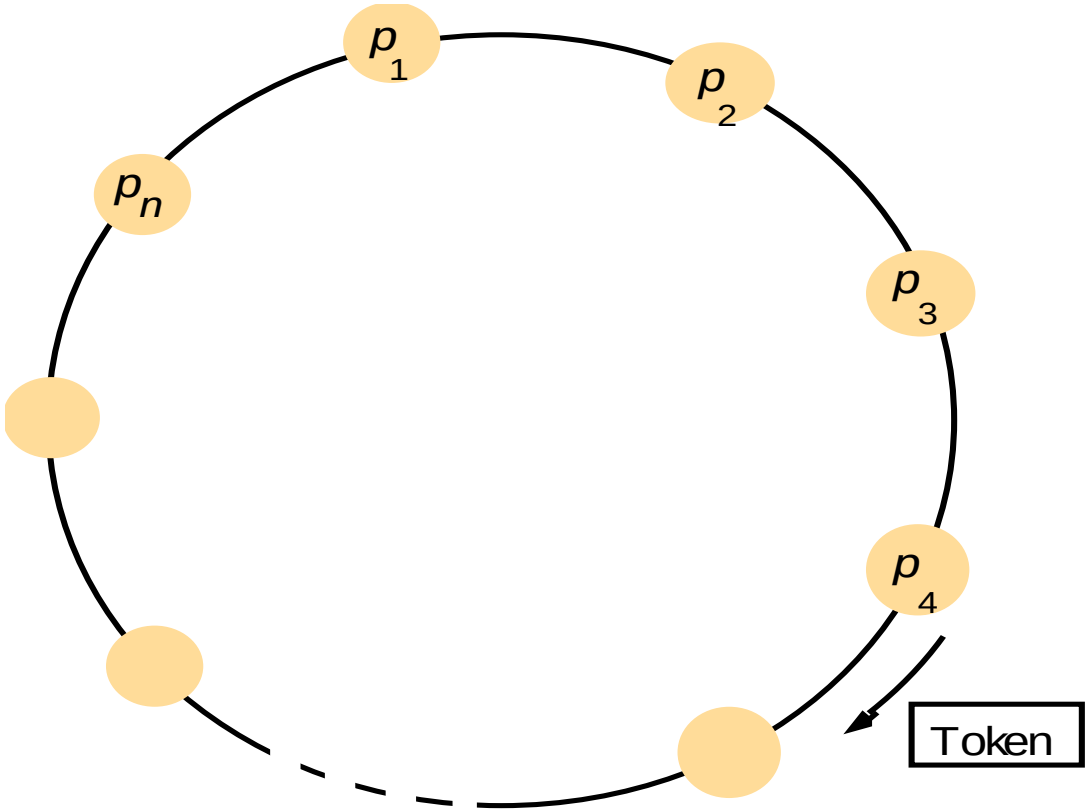


Fig. 15.3 A ring of processes transferring a mutual exclusion token

A Ring-based Algorithm

- The *N* processes are arranged in a logical ring:
  - Each process p_i has a communication channel to the next process in the ring, $p_{(i+1)\ mod\ N}$.
  - Mutual exclusion is conferred by obtaining a token in the form of a message passed from process to process in a single direction around the ring.
- If a process does not require to enter the critical section when it receives the token, it simply forwards the token to the next process in the ring.
- A process that requires the token waits until it receives the token, then enters the critical section, exits, and sends the token to the neighbor process.

Performance Analysis

- Bandwidth cost:
  - The algorithm continuously consumes network bandwidth, even when no process requires entry to the critical section.
- Client delay:
  - A delay of 0 to N message transmissions for entering the critical section
  - One message delay for exiting the critical section
- Synchronization delay:
  - The time for one client to release the token and another client to pick up the token.
  - Can be from 1 to $(N-1)$ message transmissions.
- ME1 (safety) and ME2 (liveness) are met, but not ME3 (ordering).
  - A far away client may make a request first.
  - An intermediate client may make a request at a later time, but intercepts the token earlier.

Elections

- An **election algorithm** is used for choosing a unique process from a collection of processes to play a particular role.
  - E.g., how to select the server in the server based mutual exclusion algorithm?
- A process **calls the election** if it initiates a particular run of the election algorithm.
  - An individual process calls at most one election at a time.
  - In principle, N processes could call N concurrent elections.
- A process is a **participant** if it is engaged in some run of the election algorithm, or a **non-participant** if it is not engaged in any election.

Problem Definition

- The elected process is unique, even if several processes call elections concurrently.
  - Consider the scenario that two processes detect the failure of a coordinator process at around the same time.
- Without loss of generality, we want to elect the process that has the largest identifier.
  - We assume that each process has a unique identifier.
- Each process $p_i, i = 1, \ldots, N$, uses the variable name $\boldsymbol{elected_i}$ to represent the identifier of the elected process.

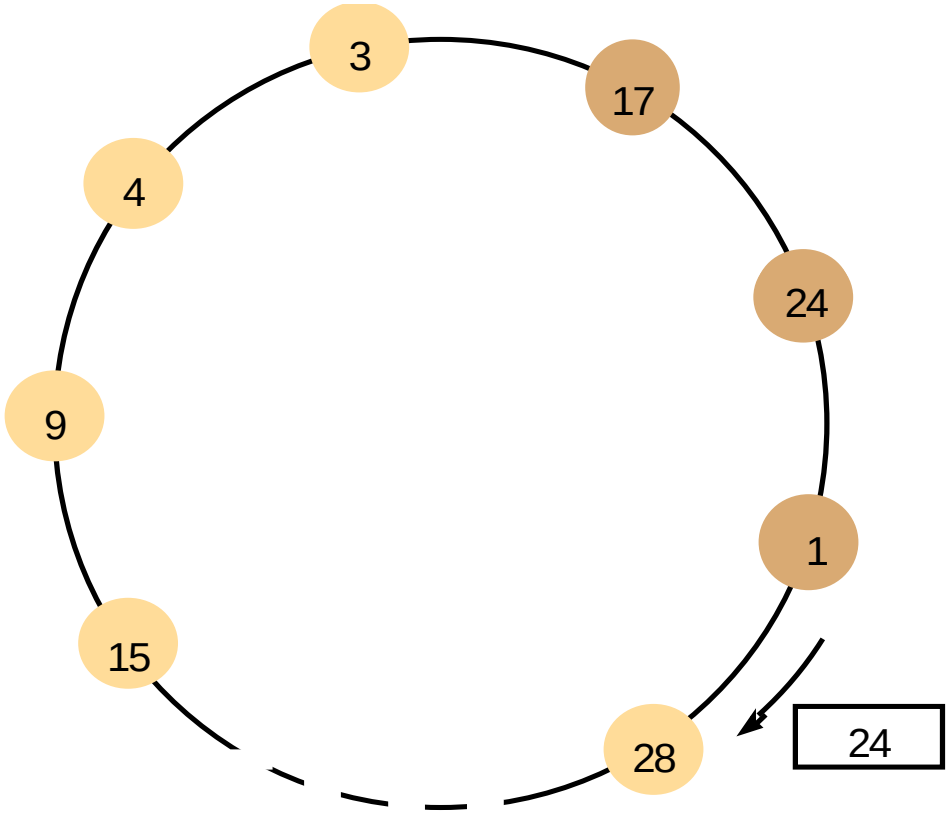
A Ring-based Election Algorithm (Cont.)

- Arrange a collection of N processes as a logical ring.
  - Each process p_i has a communication channel to the next process in the ring, $p_{(i+1)\ mod\ N}$.
  - All messages are sent clockwise around the ring.
  - Assuming that no failures occur, and the system is asynchronous.
- Initially, every process is marked as **non-participant**.
- Any process can begin an election by marking itself as a **participant**, placing its identifier in an **election message**, and sending the message to the process's clockwise neighbor.

A Ring-based Election Algorithm (Cont.)

- When a process receives an **election message**, it compares the identifier in the message with its own.
  - If the arrived identifier is greater, the process forwards the message to its neighbor, and also marks itself as a participant.
  - If the arrived identifier is smaller, two cases:
    - If the receiver is not a participant, it substitutes its own identifier in the message and forwards the message. The receiver also marks itself as a participant.
    - If the receiver is already a participant, it does nothing.
  - If the arrived identifier is that of the receiver itself, this receiver process's identifier must be the greatest. This process becomes the **coordinator**. The coordinator marks itself as a **non-participant** and sends an **elected message** to its neighbor.
- When a process receives an elected message, it marks itself as a non-participant, sets its variable $elected_i$ to the identifier in the message, and forwards the message to its neighbor unless it is the new coordinator.

A Ring-based Election Algorithm



Note: the election was started by process 17. The highest process identifier encountered so far is 24. Participant processes are shown in a dark tint.

Fig. 15.7: A ring-base election in progress

Performance Analysis

- Consider that only a single process starts an election.
- How many messages are required in the worst-case?
  - If the anti-clockwise neighbor of the initial process has the highest identifier, it takes $(3N - 1)$ messages:
    - $(N - 1)$ election messages for the initial message to reach the anti-clockwise neighbor (with highest identifier).
    - The highest identifier neighbor changes the election message's identifier and forwards the election message. This election message goes around the ring, taking up N more election messages.
    - Finally, the highest identifier neighbor becomes the coordinator, and announces the "winning" result with N elected messages.
- Turnaround time: since the messages are sent sequentially, the worst-case turnaround time is also $(3N - 1)$.

The Bully Algorithm

- The ring-based algorithm assumes no process failure.
- The bully algorithm allows failure and assumes that:
  - The system is synchronous and process failure can be detected by timeout.
    - Assume that the maximum message transmission delay is T_{trans} and the maximum message processing delay is $T_{process}$, then $T = 2T_{trans} + T_{process}$ can be used as a timeout bound to detect process failure.
    - Several processes may discover a failure concurrently.
  - Each process knows which processes have higher identifiers.
  - Message delivery between processes is reliable.
- Basic idea: a process tries to find out whether there is any other non-crashed process with higher identifier.

The Bully Algorithm (Cont.)

- There are three types of messages
  - Election message: to announce an election
  - Answer message: in response to an election message
  - Coordinator message: to announce the identity of the elected process (i.e., the new coordinator)
- If any process detects that the existing coordinator has failed (by timeout), it may start the election procedure.
  - Case 1: If this initiation process happens to have the highest identifier, it directly sends a **coordinator message** to all other processes.
  - Case 2: Otherwise, the process sends an **election message** to those processes with higher identifiers than itself, and awaits answer messages in response.

The Bully Algorithm (Cont.)

- In Case 2:
  - If no response comes back within time T, the initiation process considers itself the coordinator and sends a **coordinator message** to all processes with lower identifiers.
  - Otherwise, the initiation process waits a further period T' for a coordinator message from the new coordinator. If none arrives, the initiation process begins another round of election.
- If a process p_i receives a **coordinator message**, the process sets its variable $elected_i$ to the coordinator's identifier.
- If a process receives an **election message**, the process sends back an **answer message** and begins another election, unless the process has begun one already.

The Bully Algorithm (Cont.)

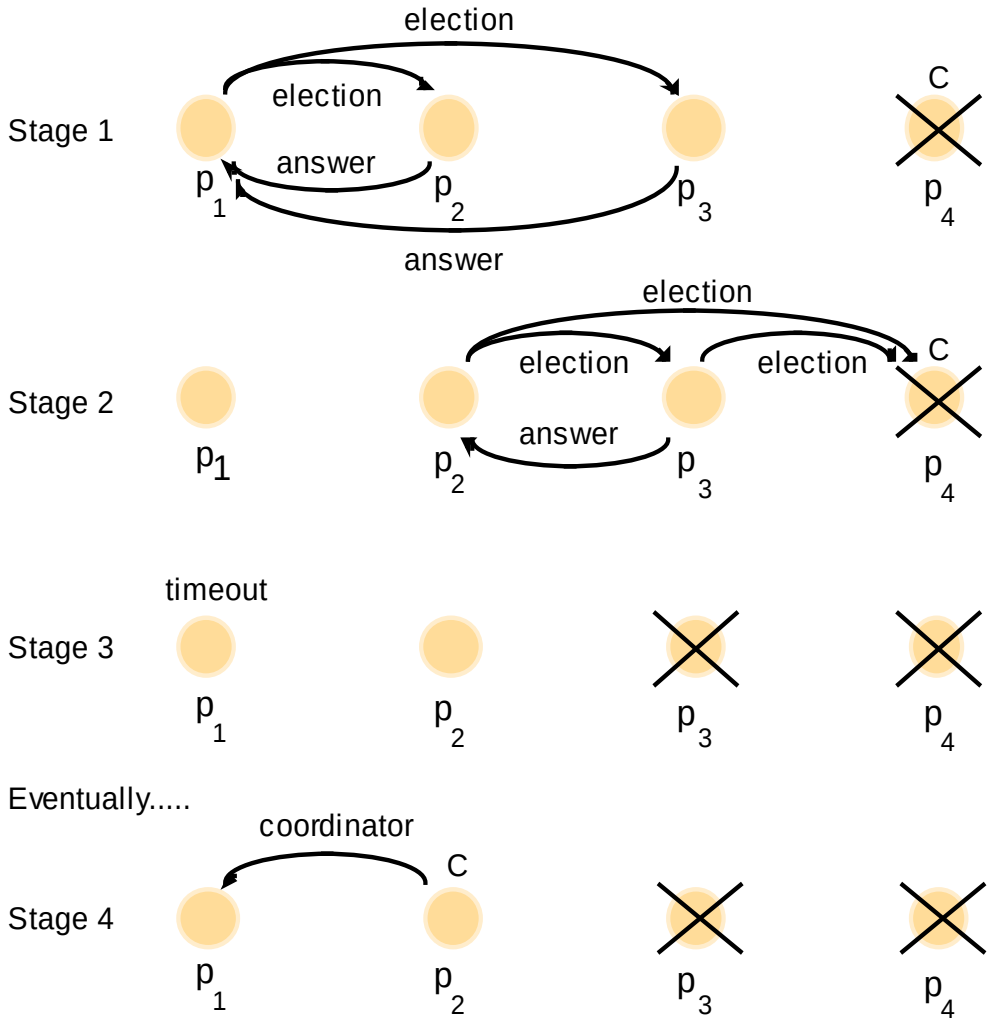


Fig. 15.8: The bully algorithm

Assumption: ID of p_4 > ID of p_3 > ID of p_2 > ID of p_1

Stage 1: p_1 detects the failure of coordinator p_4 and begins an election. p_2 and p_3 both respond to p_1 and begin their own elections.

Stage 2: p_2 gets an answer from p_3, but p_3 gets no response from p_4. So p_3 sets itself to coordinator. Unfortunately, it crashes before it informs others.

Stage 3: When p_1's timeout period T' expires, it begins a new round of election.

Stage 4: p_2 determines itself as coordinator, and sends a coordinator message to p_1.

Performance Analysis

- In the best case, the process with the second-highest identifier notices the coordinator's failure and begins the election.
  - Only $N - 2$ coordinator messages are sent.
  - The turnaround time is one message.
- In the worst case, the process with the lowest-identifier begins the election.
  - Altogether $N - 1$ processes will begin elections.
  - $O(N^2)$ messages are required.