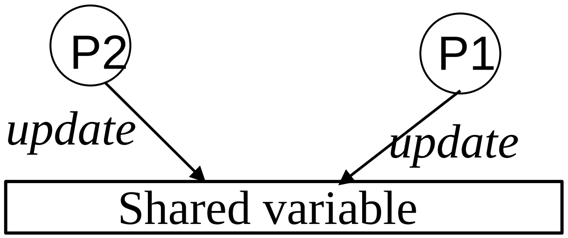
Distributed Coordination and Agreement

Coordination and Agreement

- What is coordination?
  - Coordinate the activities/decision of different processes (process synchronization)
- Coordination among two types of processes
  - Coordinating processes belonging to the same parent process
    - Distributed processes may need to coordinate with each other to complete a task
    - i.e., When to start and when to end
  - Independent processes from different applications
    - Coordination in accessing common resources, i.e., global data (data synchronization)
- What is an agreement (consensus)
  - All processes make the same decision
  - i.e., all processes /threads agree to commit or abort

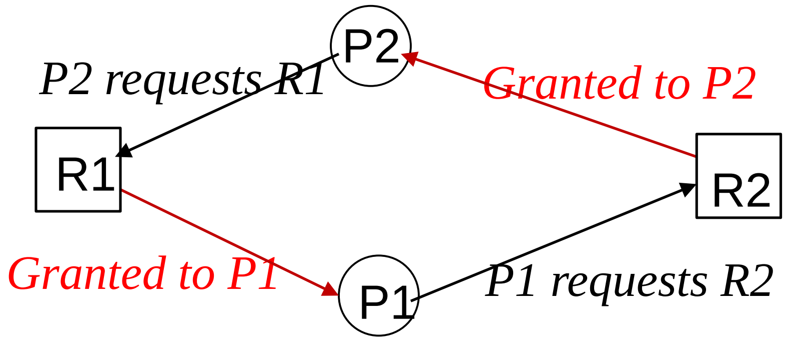
Mutual Exclusion and Critical Section

- Coordination is required to access shared resources to ensure data consistency (data synchronization)
  - Mutual exclusion prevents inconsistency among concurrent processes
- Mutual exclusion requirements:
  - Only one process is allowed to access the shared resource at a time
  - When a process is using a resource (i.e., shared global variables), other processes requesting the resource have to wait
- How to achieve mutual exclusion?
  - Mutual exclusion can be achieved by defining critical sections



Distributed Mutual Exclusion

- Distributed mutual exclusion problem:
  - Multiple processes in different locations need to access a shared resource
  - A global resource may be replicated at multiple locations and managed by multiple servers
  - Distributed critical sections



The deadlock problem

Operations for Critical Sections

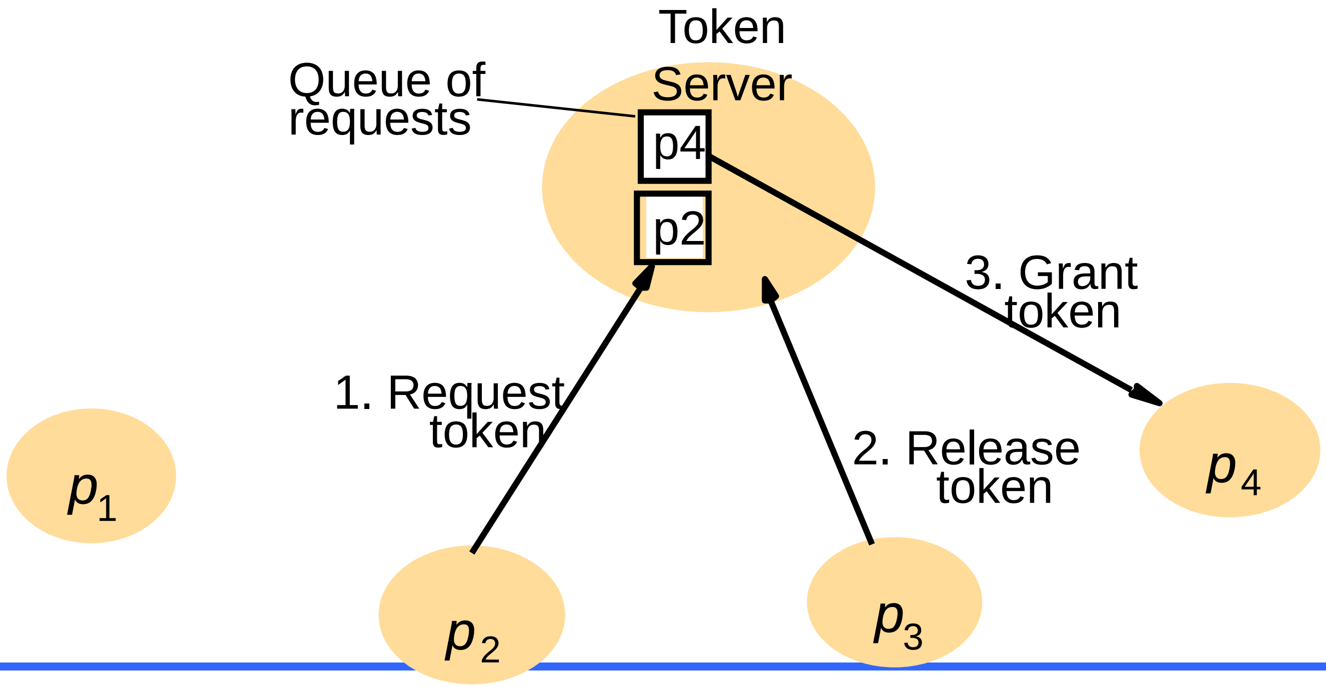
- Operations for accessing a shared resource:
  - Enter(): enter a critical section; otherwise it is blocked
  - ResourceAccess(): access shared resources in critical section
  - Exit(): leave a critical section

Algorithms for Mutual Exclusion

- Performance metrics
  - Communication cost (number of messages generated)
  - Access delay: waiting time to access to a resource
  - Synchronization time: minimum time from the release of a resource to the next assignment of the resource (someone is waiting)
- Other performance concerns
  - Deadlock
  - Starvation: infinite postponement (waiting forever)
  - Fairness: fair for all processes waiting for entering a critical section (i.e., following the arrival order of the requests)

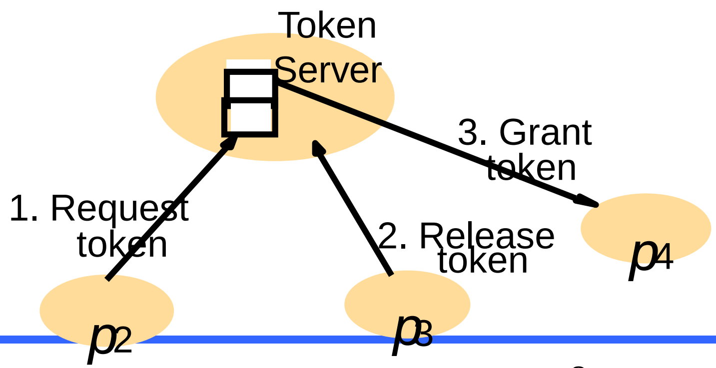
A Central Server Algorithm

- Algorithm with a Central Token-server
  - A centralized server is responsible for granting token to access the resource
  - To access to a shared resource, send a request to the server with the token
  - Note: The server only manages the token, but not the resource (2-in-1 is ok)



Performance of Central Server Algorithm

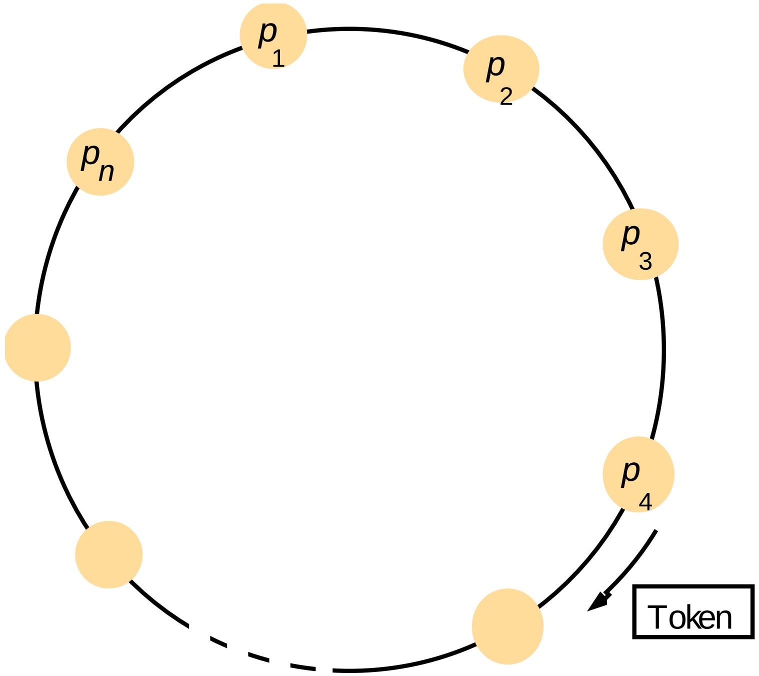
- What is the queuing/scheduling method for waiting requests?
  - FIFO, priority-based, etc.
- What is the performance of the method?
  - Entering the critical section costs: 2 messages
  - Exiting the critical section needs: 1 message
  - The minimum delay for access to the resource is one round-trip delay
  - The synchronization delay is also a round-trip delay:
    - a release message to the server followed by a grant message
- Other issues:
  - The central server becomes the bottleneck
  - Not scalable



A Distributed Method: Token-Ring Algorithm

- All processes are organized in a logical ring. A token is circulated from one process to the next along the ring.
- Only the process that holds the token is allowed to access the resource.
  - The process passes the token to the next once it finishes the access.
- A process will pass the token immediately to the next if does not need to access the resource.

Assume: no process or network failure



Performance of Token-Ring Algorithm

- Performance issues:
  - It costs network messages constantly (regardless whether there is any process requesting the resource)
  - The access delay is from 0 message to N messages depending on the location of the token at the time when it generates the request
    - Best case: when the token is passed to P1 and P1 just wants to access the resource (0 waiting)
    - Worst case: right after P1 releases the token, P1 wants to access to the resource (N-hops waiting)
  - The synchronization delay is also from 0 message to N messages

Ricart and Agrawala's Algorithm: Using Multicast and Timestamp

- When a process wishes to access a resource, it multicasts a request message to all other processes in the group
  - Assumption: network is interconnected and the communication is reliable
- Any process that receives the request replies "ok"
- The process can access the resource only when it receives replies from all other processes
- No two processes will receive permissions from all others at the same time (mutual exclusion is guaranteed)
- How about two processes multicast requests simultaneously?

Resolve Concurrent Requests using Timestamps

- When a process multicasts a request, it attaches its own ID and its current timestamp $<T_i, p_i>$ to the message
- In case there are multiple concurrent requests, when a process receives a request:
  - It replies a request only if itself is not waiting to access OR itself is waiting but the timestamp of the request is smaller than its own request-time
    - Earlier request has a higher priority (FCFS)
  - All requests are serialized according to their timestamps
  - Assume all clocks of processes are strictly synchronized

Algorithm of using Multicast and Timestamp

- A process is in one of the following states
  - RELEASED: outside the critical section
  - HOLD: inside the critical section (hold the resource)
  - WAIT: waiting to enter the critical section
- When p_i wishes to enter a critical section:
  - It sets its status to WAIT and multicasts $<T_i, p_i>$ to all other processes
  - It blocks until receiving all replies before entering the critical section
- Upon receiving a request message, if this process is in:
  - RELEASED, it replies to p_i immediately
  - HOLD, it delays the reply to p_i until it exits from the critical section
  - WAIT, it compares its timestamp with the one in the message. If its own timestamp is greater, it replies to p_i immediately; otherwise, it delays the reply until it exits from the critical section

Ricart and Agrawala's Algorithm: Using Multicast and Timestamp

On initialization

    state := RELEASED;

On wishing to enter the critical section

    state := WAIT; T := request's timestamp;

    Multicast *request* to all processes;

On receiving a reply from p_i

    if the number of replies received == (N – 1)

        state := HOLD;

        enter the critical section (access the resource);

    end if

On p_j *receiving a request* $<T_i, p_i>$ *from* p_i $(i \neq j)$

    if (*state* = HOLD or (*state* = WAIT and ($T_j < T_i$)))

        queue *request* from p_i without replying;

    else

        reply p_i immediately;

    end if

On exiting the critical section

    state := RELEASED;

    reply to all queued requests; // unblock all others

Performance of Ricart and Agrawala's Algorithm

- Performance issues:
  - Gaining entry into the critical section requires 2 (N – 1) messages
  - Minimum delay in accessing: one round-trip time (2d)
  - Synchronization delay: d (once a process leaves a critical section, it replies to all waiting processes)
  - ? Deadlock
  - ? Starvation

Motivation of Maekawa's Voting Algorithm

- Problems of the multicast algorithm
  - Large number of synchronization messages (1 multicast and N replies)
  - Have to get the permission from all the member processes
  - To enter a critical section, there is no need to get permission from all peers
- Improvement by Maekawa's Algorithm
  - A process only needs to obtain permissions from a subset of its peers (votes), so long as any two subsets always have overlap
  - Why? A process can give permission to only one process. The overlapping member would prevent two processes from entering at the same time
- How to determine the size of the subset (called *quorum*):
  - Any two subsets must have at least one common member (process)
  - A process must obtain sufficient votes (quorum) to enter the critical section

Prof. Maekawa

Theory of Maekawa's Voting Algorithm

- Each p_i is associated with a voting set V_i (p_i needs to obtain permissions from all processes in V_i), $p_i \in V_i$
- $V_i \cap V_j \neq \Phi$ for all $i, j = 1, 2, \ldots,$ N
  - The overlapping element prevents two processes from entering a critical section at the same time
  - e.g., V1: {p1, p2} and V3: {p2, p3}. p2 will not grant permission to p3 if it has already voted for p1
  - Majority voting (an easy example)
- $|V_i|$ = K: all processes require the same number of votes (quorum)
  - Minimizing K can improve the performance as the number of messages for synchronization is reduced
  - The minimal K for mutual exclusion: K ~ $\sqrt{N}$ – why?

Maekawa's Voting Algorithm

- When process p_{i} wishes to enter a critical section:
  - p_i multicasts a *request* message to all members in V_i (including itself)
  - p_i is blocked until it receives all replies from the members in V_i
- When p_j in V_i receives p_i's request: if it is HOLD or it has already replied to (voted for) another process (including itself), it queues p_i's request; otherwise it replies p_i immediately
- When p_i exits the critical section, it sends a *release* message to all members in V_i
- When p_j receives a *release* message, it removes the head of the queued requests and replies to it
  - Serving the requests one by one

Maekawa's Algorithm

On initialization

state := RELEASED;
voted := FALSE;

For p_i to enter the critical section

state := WAIT;
voted := TRUE; // vote for itself
Multicast *request* to all processes in V_i;
Wait until (No. of replies received = K);
state := HOLD;

On receipt of a request from p_i at p_j

if (*state* = HOLD *or* *voted* = TRUE)
    queue *request* from p_i (no reply);
else
    send *reply* to p_i;
    voted := TRUE;
end if

For p_i to exit the critical section

state := RELEASED;
Multicast *release* to all processes in V_i;

On receipt of a release from p_i at p_j

if (queue of requests is non-empty)
    remove head of queue, say p_k;
    send *reply* to p_k;
    voted := TRUE;
else
    voted := FALSE;
end if

An Example of Maekawa's Voting Algorithm

V1: {P1, P2, P4}

V2: {P2, P3, P4}

V3: {P1, P2, P3}

V4: {P2, P3, P4}

1) If P1 wishes to enter a critical section, it multicasts a *request* to itself, P2 and P4. When it receives replies from all of them, it enters the critical section

2) In the meantime, if P3 wishes to enter, it multicasts a *request* to itself, P1 and P2. It will be blocked by P1 and P2.

Maekawa's Voting Algorithm

- Performance Issues
  - $O(\sqrt{N})$ messages per entry into the critical section – why?
  - $O(\sqrt{N})$ messages per exit from a critical section
  - Synchronization delay is round-trip (2d)

- The algorithm is deadlock prone - why?
  - How to solve it?

King's Poisoned Wine Problem

- A King prepared 100 barrels of wine to host a big banquet with his international guests
- Before the banquet starts, the King got the information that one barrel of the wine was poisoned
- Design a method to use the minimum number of prisoners to sample the wine and find out the poisoned barrel of wine

Election Algorithms

- Many distributed applications require one process act as a leader, such as the central server algorithm for mutual exclusion. How to elect the leader?
- Assume that each process has a unique ID (used as the priority). The process with the highest ID will be elected
- The goal of election algorithms: when election completes, all participating processes agree on who the new leader is
- Any process can initiate the election at any time
  - e.g., when a process detects the failure of the current leader, or feels there is a need to elect a new leader

Requirements of Election Algorithms

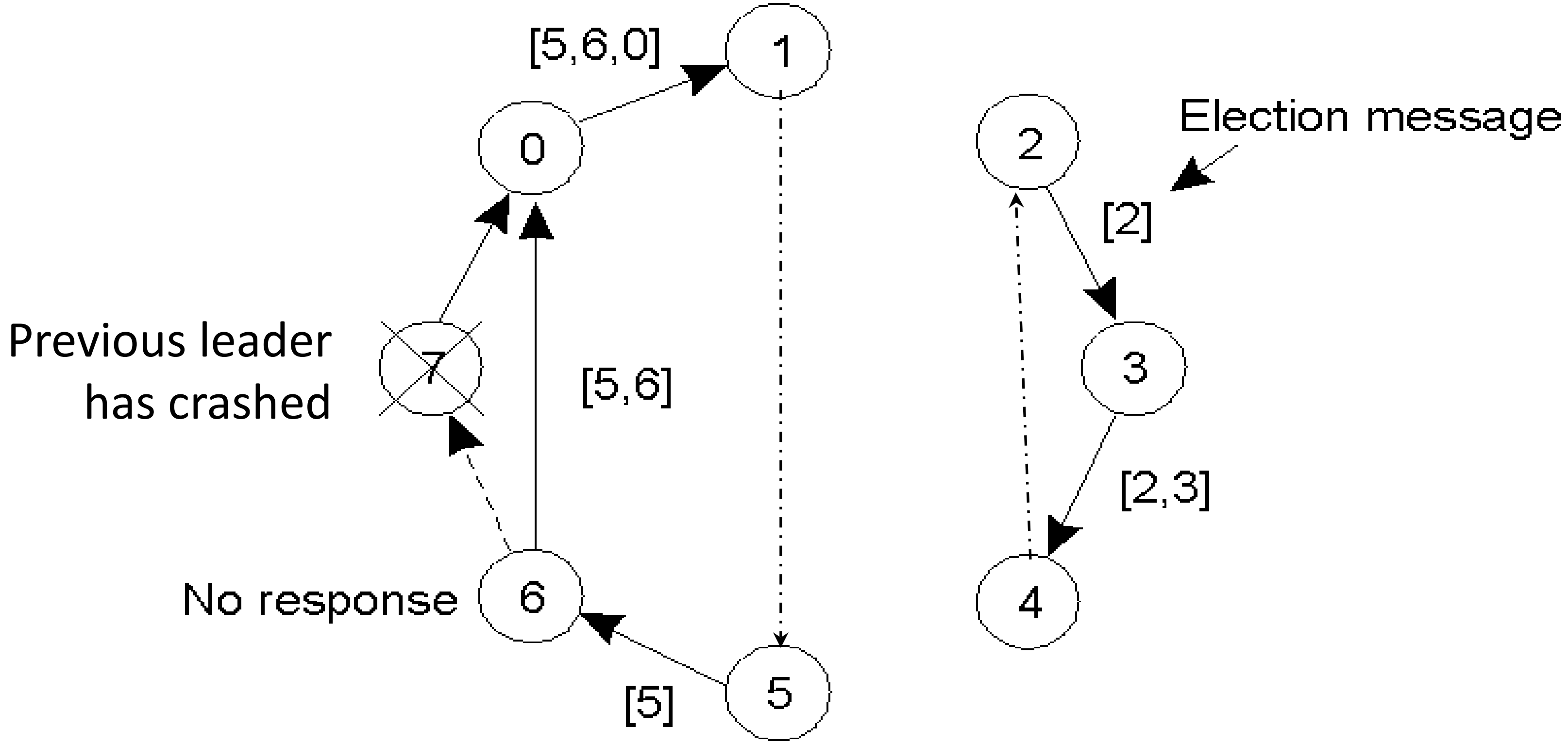
- Each process p_i (i = 1, 2, …, N) has a variable $elected_i$ that is the ID of the elected leader.
  - $elected_i$ is initially set to Φ (*EMPTY*) when p_i participates in an election, i = 1, 2, …, N
- Requirements
  - E1 (safety): A participating process p_i has either $elected_i = \Phi$ or $elected_i = P$ (P is the elected leader)
  - E2 (liveness): All participating processes, say p_i, eventually set $elected_i \neq \Phi$ or crash
- Performance:
  - Number of messages for election
  - Turnaround time for election (No. of rounds of message exchanges)

Election Algorithm: Ring-based Algorithm

- All processes are organized in a logical ring (similar to the token-ring)
- When a process notices that the leader is not functioning, it initiates the election by sending an *election* message containing its ID to its successor along the ring. If the successor is down, skips over it and goes to the next one, or the next after that, until a live process is located
- When a process receives an election message, if its own ID is greater than the one in the message:
  1) replaces the ID in the message by its own (or adds its ID to the list)
  2) forwards the message to its successor
- If a process receives the same election msg again and its ID is the greatest:
  1) sets its status to be the leader and $elected_i \leftarrow$ its ID
  2) informs all processes by circulating $elected_i$ message along the ring
- The election is complete when the $elected_i$ msg reaches the original sender

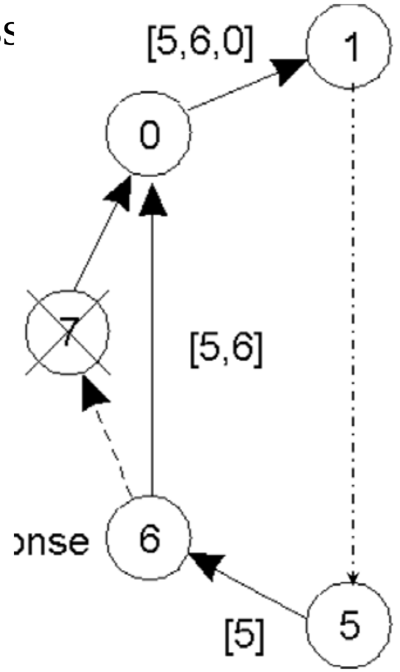
The Ring-based Algorithm: an example

Election algorithm using a ring with process failures



Performance of Ring-based Algorithm

- Performance Issues
  - N – 1 messages for an initiator to reach the process with the highest ID in worst case
    - When the process with highest ID is next to the initiator in anti-clock wise (eg, P0), it takes N – 1 msgs to reach this highest ID process
  - N messages to circulate the highest ID
    - The process with highest ID knows it wins the election by now
  - N messages to inform all members
    - For announcing the result
  - Totally 3N – 1 messages and delays (worst case)

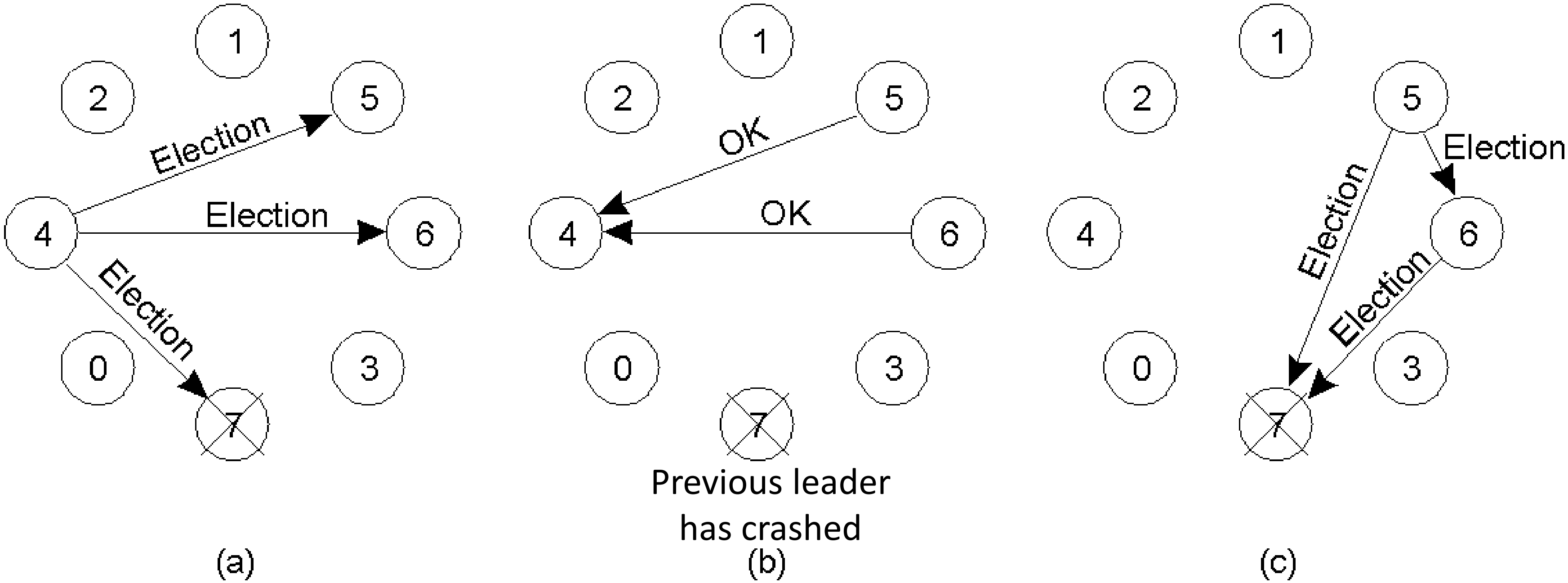


Election: The Bully Algorithm

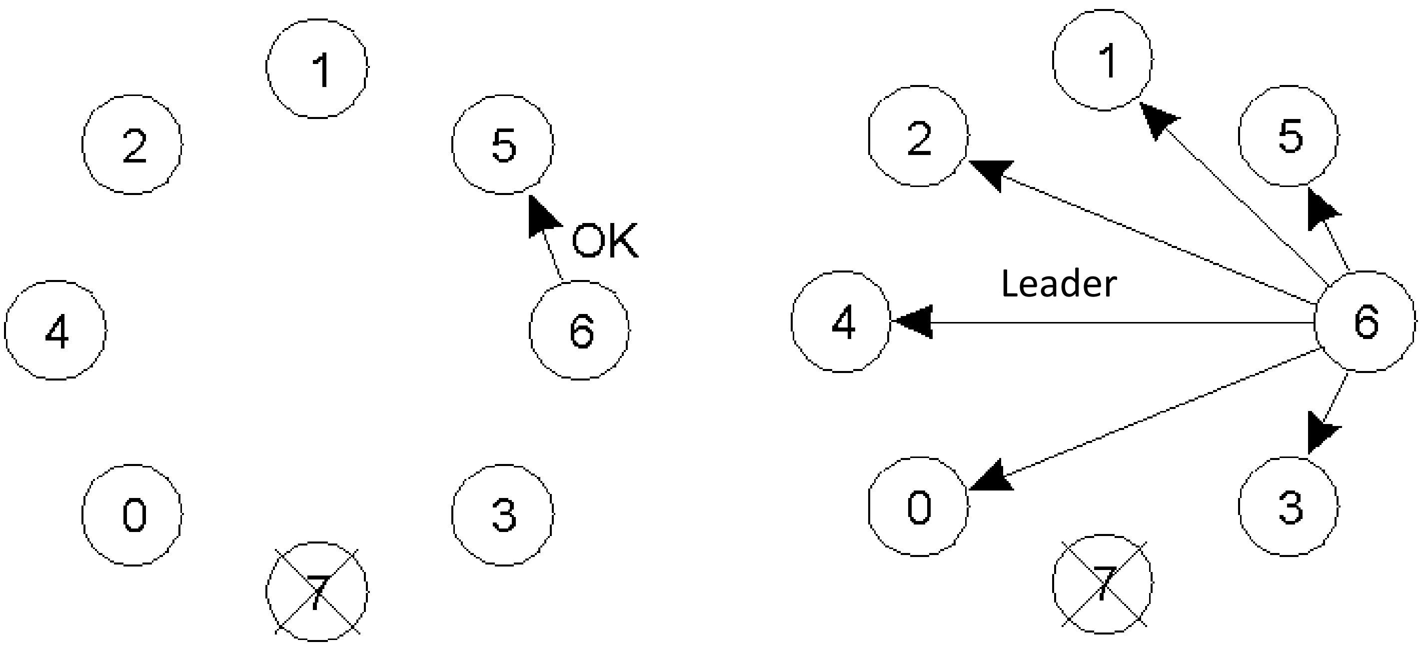
Assume: every process knows the process IDs of others

- When process p_i discovers the failure of the leader, it initiates an election by sending an *election* message to all processes that have higher IDs than itself
- When process p_j receives the message from a lower-ID process, it replies "*I'm alive*" (OK) message to the sender and it takes over the election by sending out an election message to higher-ID processes (the same as p_i)
- If p_i receives no reply, it wins the election and becomes the leader:
  - The new leader announces its victory by sending all processes a message telling them that it is the new leader

The Bully Algorithm: an example



The Bully Algorithm: example (cont'd)

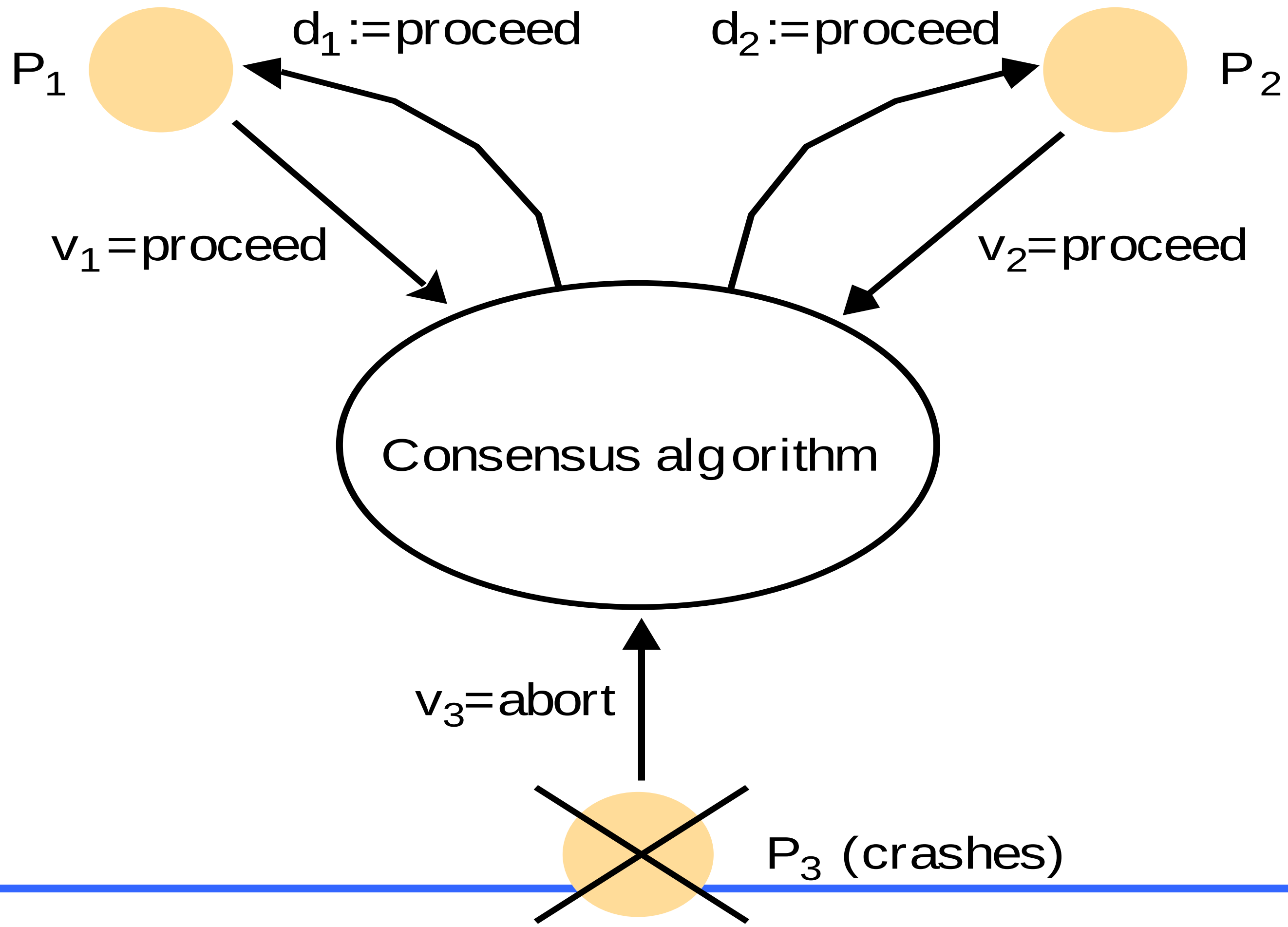


(d) (e)

Consensus Problem and Requirements

- Consensus: a group of processes agree on a value/decision after one or more of them propose the values/decisions
- Consensus problem:
  - Each process p_i begins with the *undecided* state and proposes a value v_i drawn from a set D (i = 1, 2, …, N)
  - All processes exchange their values with each other
  - Each process then decides the final value d_i based on the values proposed by others
- Requirements:
  - Agreement: all correct processes eventually have the same decision value
  - Integrity: if the correct processes all proposed the same value, any process in the decided state must choose this value

Consensus of Three Processes



A Simple Consensus Algorithm: No process failure

1) In a group of N processes, each process multicasts its proposed value to all other members in the group
2) Each process waits until it has collected all N values including its own
3) It then evaluates a function, i.e., *majority*(v_1, v_2, …, v_n), *max* or *min*, *etc.*, to arrive a final value for the decision

- Termination is guaranteed by reliable communication
- Agreement and integrity are guaranteed (all processes receive the same set of values and follow the same *majority* function)
- But, if processes can fail, …

Consensus in Synchronous System: processes can (crash) fail

Assume: up to f out of N processes can suffer "crash" failure

1) Initial round: each process p_i includes its own value in V_i^1 and multicasts V_i^1 to all group members
2) The r^{th} round: each p_i collects values multicast from other group members, adds them into V_i^{r+1}, and multicasts $V_i^{r+1} - V_i^r$.
3) Repeat step 2) until $r > f+1$. Each p_i returns $d_i = Min\{V_i^{f+1}\}$

Note:

- "multicast" is not atomic: a process can fail in the middle of a multicast, leaving some receive but others not receive the msg
- At the end of $f+1$ round, every alive process receives the same set of values (some values could be proposed by failed processes)

Algorithm of Consensus in Synchronous System of up to *f* (crash) failures

Algorithm for process $p_i \in G$;

On initialization

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\,\}$;

In round r $(1 \le r \le f+1)$

multicast$(G,\ Values_i^r\ -\ Values_i^{r-1})$; //send only values that are not sent before

$Values_i^{r+1} := Values_i^r$;

While (in round *r*) { // collect values multicast by other processes in *r*th round

receive value V_j from p_j;

$Values_i^{r+1} := Values_i^{r+1}\ \cup\ V_j$;

}

After $(f+1)$ *rounds*

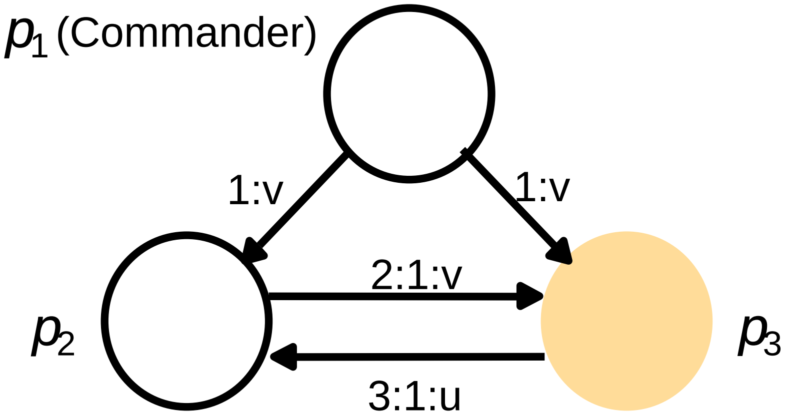
$\text{Assignd}_i = min\left(Values_i^{f+1}\right)$;

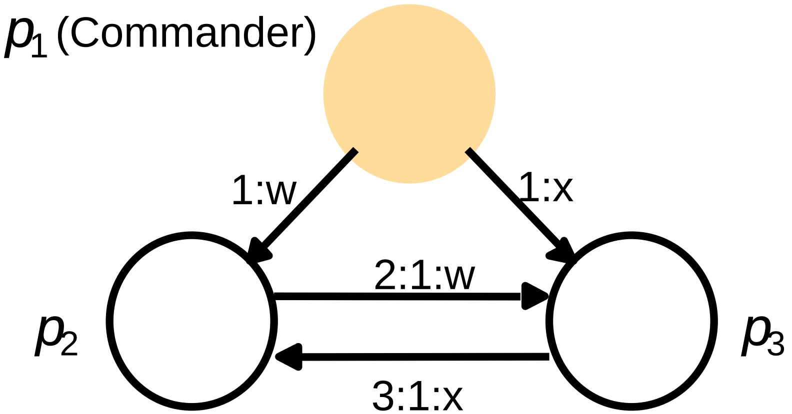
Consensus with Arbitrary Failures: The Byzantine Generals Problem

A command and generals exchange msgs to agree on attack or retreat

- The commander issues an order to his generals on attack or retreat
  - The commander could be faulty too
  - The generals exchange msgs among themselves about what they hear
- Arbitrary failure model: A faulty general (or the commander) can propose attack to some of the generals but retreat to others
- Agreement: The decision by all correct processes is the same
- Integrity: if the commander is correct, all correct processes decide on the value that the commander has proposed
- Conclusion: Byzantine generals problem has a solution iff $f < N/3$
  - Faulty processes must be less than $N/3$
  - Impossibility of 3 processes with 1 faulty

The Byzantine Generals Problem: impossibility of 3 processes





Faulty processes are shown coloured

Algorithm for Byzantine General Problem

Assumptions:

- A1: every message is delivery correctly
- A2: the receiver of a message knows who sent it
- A3: the absence of a message can be detected

Definition. majority function: majority(v_1, …, v_{n-1}) returns:

1) The majority value of among {v_1, …, v_{n-1} } if it exists; otherwise "retreat"
2) The median value of the ordered set {v_1, …, v_{n-1} }

Note: the default value is "retreat" if a process doesn't receive any value from another (or commander)

The algorithm

Suppose 1 commander, n – 1 generals and m traitors, algorithm BGP(m) is a recursive function:

<u>BGP(0)</u>

1) The commander sends his value to every general
2) Each general uses the value he receives, or “retreat” as default

<u>BGP(m)</u> // m is the number of traitors

1) The commander sends his value to every general
2) Each general p_i, receiving value v_i from commander, acts as the commander to call BGT(m – 1) sends v_i to all n – 2 other generals
3) Each p_i, receiving value v_j from p_j in step 2 (using BGT(m – 1)), uses value majority(v_1, …, v_{n-1})

Summary

- Distributed mutual exclusion
  - Central algorithm
  - Token-ring algorithm
  - Distributed Algorithm (Ricart and Agrawala's)
  - Voting Set Algorithm (Maekawa's)
- Elections
  - Ring-based Algorithm
  - Bully Algorithm
- Consensus
  - Distributed Consensus with crash failures
  - Byzantine Generals Problem

Exercise

1a) Explain why no two processes can enter critical section at the same time with Maekawa's Algorithm.

1b) Maekawa's algorithm can reduce the communication cost to $O(\sqrt{N})$ for a process to enter or exit from the critical section, where N is the total number of processes in the system. Why?

2) In the Consensus Algorithm that can tolerate at most f process failures (crashes), it requires (f +1) rounds of collecting & multicasting values before all correct processes reach consensus. Why?