고려대학교
KOREA UNIVERSITY

KU-The Future

Cloud computing – Distributed Systems

Heonchang Yu

Distributed and Cloud Computing Lab.

Introduction

- Distributed system
  - For a set of processes to coordinate their actions or to agree on one or more values
    - ✓ In the case of a spaceship, it is essential that the computers controlling it **agree** on such conditions as whether the spaceship's mission is **proceeding** or has been **aborted**
  - No fixed master-slave relationship
    - ✓ To avoid single points of failure, such as fixed masters
  - In an asynchronous system : no timing assumptions
  - In a synchronous system : bounds on the maximum message transmission delay, on the time to execute each step of a process, and on clock drift rates / use timeouts to detect process crashes

Introduction

- Need to coordinate the actions of the independent processes
  - **Failure detection**: how do I know in an asynchronous network whether my peer is dead or alive
  - **Mutual exclusion**: no two process will ever get access to a shared resource in a critical section at the same time
  - **Election**: in master-slave systems, how will the system elect a master (either at boot up time or when the master fails)
  - **Consensus** in the presence of faults (byzantine problems)
    - ✓ how to know whether acknowledgement was received over an unreliable communication medium
    - ✓ how to know whether peer process knows about one's own intentions in the presence of a non-confidential communication channel

Introduction

– Failure assumptions and failure detectors
  - Reliable communication protocol - a reliable channel eventually delivers a message
  - **Failure detector**: a service that processes queries about whether a particular process has failed
  - **Unreliable failure detector**
    - ✓ Unsuspected: the detector has recently received evidence suggesting that the process has not failed
      - ❖ A message was recently received from it.
    - ✓ Suspected: the failure detector has some indication that the process may have failed
      - ❖ No message from the process has been received for more than a nominal maximum length of silence
  - **Reliable failure detector**
    - ✓ Unsuspected: can only be a hint
    - ✓ Failed: determined that the process has crashed

Introduction

- Failure assumptions and failure detectors
  - Implementation of unreliable failure detector
    - ✓ Each process *p* sends a '*p* is here' message to every other process, and it does this every T seconds.
    - ✓ D: the maximum message transmission time
    - ✓ If the local failure detector at process *q* does not receive a '*p* is here' message within T+D seconds of the last one, then it reports to *q* that *p* is *Suspected*.
    - ✓ If it subsequently receives a '*p* is here' message, then it reports to *q* that *p* is *OK*.
    - ✓ There are limits on message transmission times.
      - ❖ If we choose small values for T and D, then the failure detector is likely to suspect non-crashed processes many times, and much bandwidth will be taken up with '*p* is here' message.
      - ❖ If we choose a large total timeout value then crashed processes will often be reported as *Unsuspected*.
    - ✓ A practical solution to this problem is to use timeout values

Distributed mutual exclusion

- Distributed processes often need to coordinate their activities.
  - If a collection of processes share a resource, then mutual exclusion is required to prevent interference and ensure consistency.
  - Critical section problem
- Algorithms for mutual exclusion
  - Assumptions
    - ✓ The system is asynchronous.
    - ✓ Processes do not fail.
    - ✓ Message delivery is reliable.
  - The application-level protocol for executing a critical section
    - ✓ *enter*() // enter critical section
    - ✓ *resourceAccesses*() // access shared resources in critical section
    - ✓ *exit*() // leave critical section

Distributed mutual exclusion

– Algorithms for mutual exclusion

- Requirements for mutual exclusion
  - ✓ ME1 - safety
    - ❖ At most one process may execute in the critical section(CS) at a time.
  - ✓ ME2 - liveness
    - ❖ Requests to enter and exit the critical section eventually succeed.
  - ✓ ME3 - ordering(→)
    - ❖ If one request to enter the CS happened before another, then entry to the CS is granted in that order.
- Criteria to evaluate algorithms for mutual exclusion
  - ✓ *Bandwidth* : is proportional to the number of messages sent in each entry and exit operation
  - ✓ *Client delay* : incurred by a process at each *entry* and *exit* operation
  - ✓ *Throughput* : the rate at which the collection of processes as a whole can access the critical section
    - ❖ Measuring the effect using the synchronization delay between one process exiting the critical section and the next process entering it

Distributed mutual exclusion

- Algorithms for mutual exclusion
  - The central server algorithm
    - ✓ The simplest way to achieve mutual exclusion
    - ✓ A server that grants permission to enter the critical section
      - ❖ To enter a critical section, a process sends a request message to the server and awaits a reply from it
      - ❖ If no other process has the token at the time of the request, then the server replies immediately, granting the token.
      - ❖ If the token is currently held by another process, then the server does not reply but queue the request.
      - ❖ On exiting the critical section, a message is sent to the server, giving it back the token.
      - ❖ If the queue of waiting processes is not empty, then the server choose the oldest entry in the queue, removes it and replies to the corresponding process.
    - ✓ Satisfy properties safety and liveness, but not for ordering
    - ✓ Entering the critical section takes two messages – request & grant
    - ✓ Exiting the critical section takes one release message

Distributed mutual exclusion

- The central server algorithm

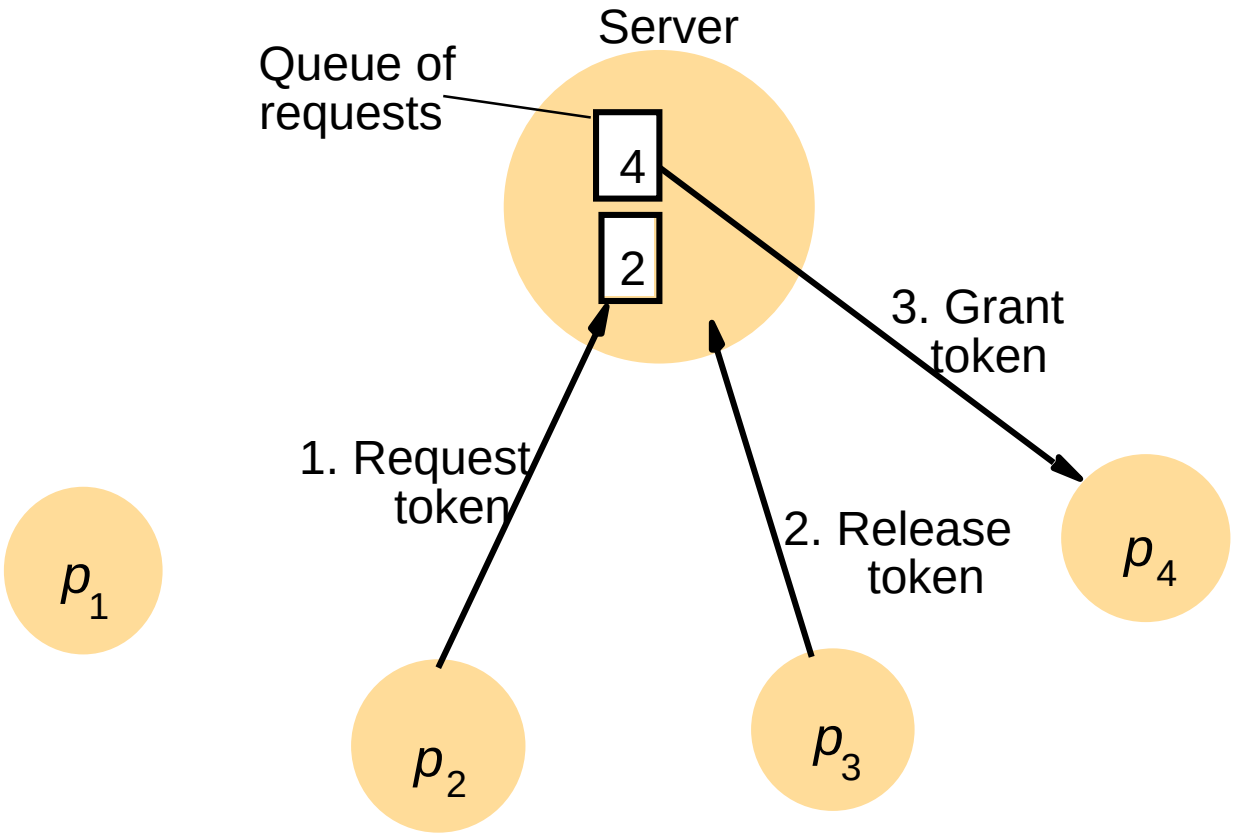


Figure 15.2 Server managing a mutual exclusion token for a set of processes

Distributed mutual exclusion

- Algorithms for mutual exclusion
  - A ring-based algorithm
    - ✓ To arrange N processes in a logical ring
    - ✓ Each process p_i has a communication channel to the next process in the ring, $p_{(i+1) \bmod N}$.
    - ✓ Exclusion is conferred by obtaining a token in the form of a message passed from process to process in a single direction.
    - ✓ If a process does not require to enter the critical section when it receives the token, then it immediately forwards the token to its neighbour.
    - ✓ To exit the critical section, the process sends the token on to its neighbour.
    - ✓ Satisfy properties safety and liveness, but not for ordering

Distributed mutual exclusion

- A ring-based algorithm
  - ✓ The delay experienced by a process requesting entry to the CS is between 0 messages (when it has just received the token) and N messages (when it has just passed on the token).
  - ✓ To exit the CS requires only one message.
  - ✓ The synchronization delay between one process's exit from the CS and the next process's entry is anywhere from 1 to N message transmission.

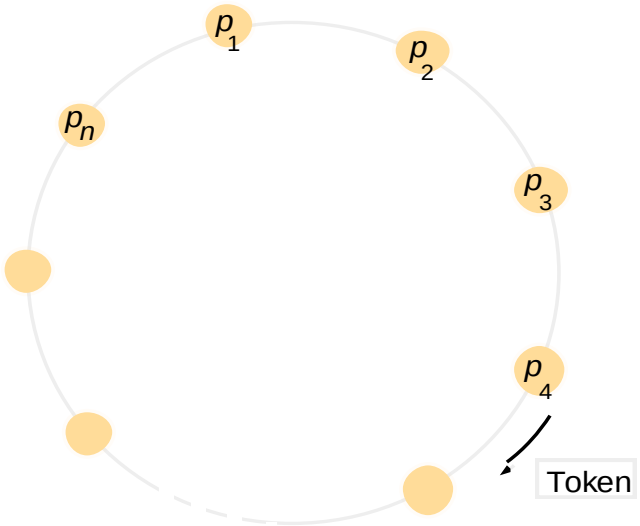


Figure 15.3 A ring of processes transferring a mutual exclusion token

Distributed mutual exclusion

– Algorithms for mutual exclusion

- An algorithm using multicast and logical clocks
  - ✓ Algorithm by Ricart and Agrawala
  - ✓ Processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message.
  - ✓ Assumptions
    - ❖ The processes p_1,..., p_N bear distinct numeric identifiers.
    - ❖ Possess communication channels to one another, and each process p_i keeps a Lamport clock
    - ❖ Messages requesting entry are of the form $<T_i, p_i>$, where T is the sender's timestamp and p_i is the sender's identifier.
  - ✓ States of each process
    - ❖ RELEASED - being outside the critical section
    - ❖ WANTED - wanting entry
    - ❖ HELD - being in the critical section

Distributed mutual exclusion

– Algorithms for mutual exclusion

- An algorithm using multicast and logical clocks

✓ The protocol in Figure 15.4

❖ If a process requests entry and the state of all other processes is RELEASED, then all processes will reply immediately to the request and the requester will obtain entry.

❖ If some process is in state HELD, then that process will not reply to requests until it has finished with the critical section, and so the requester cannot gain entry in the meantime.

❖ If two or more processes request entry at the same time, then whichever process's request bears the lowest timestamp will be the first to collect N-1 replies, granting it entry next.

❖ If the requests bear equal Lamport timestamps, the requests are ordered according to the processes' corresponding identifiers.

✓ Safety [ME1] property is satisfied. – It is impossible for two processes p_i and p_j ($i \neq j$) to enter the CS at the same time, since the pairs $< T_i, p_i >$ are totally ordered.

✓ Liveness [ME2] and ordering [ME3] properties are satisfied.

Distributed mutual exclusion

- ✓ Gaining entry takes 2(N-1) messages in the algorithm: N-1 to multicast the request, followed by N-1 replies.
- ✓ Synchronization delay is only one message transmission time.

```
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 <Ti, pi> at pj (i≠ j)
    if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
    then
                queue request from pi without replying;
    else
                reply immediately to pi;
    end if

To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```

Figure 15.4 Ricart and Agrawala's algorithm

Distributed mutual exclusion

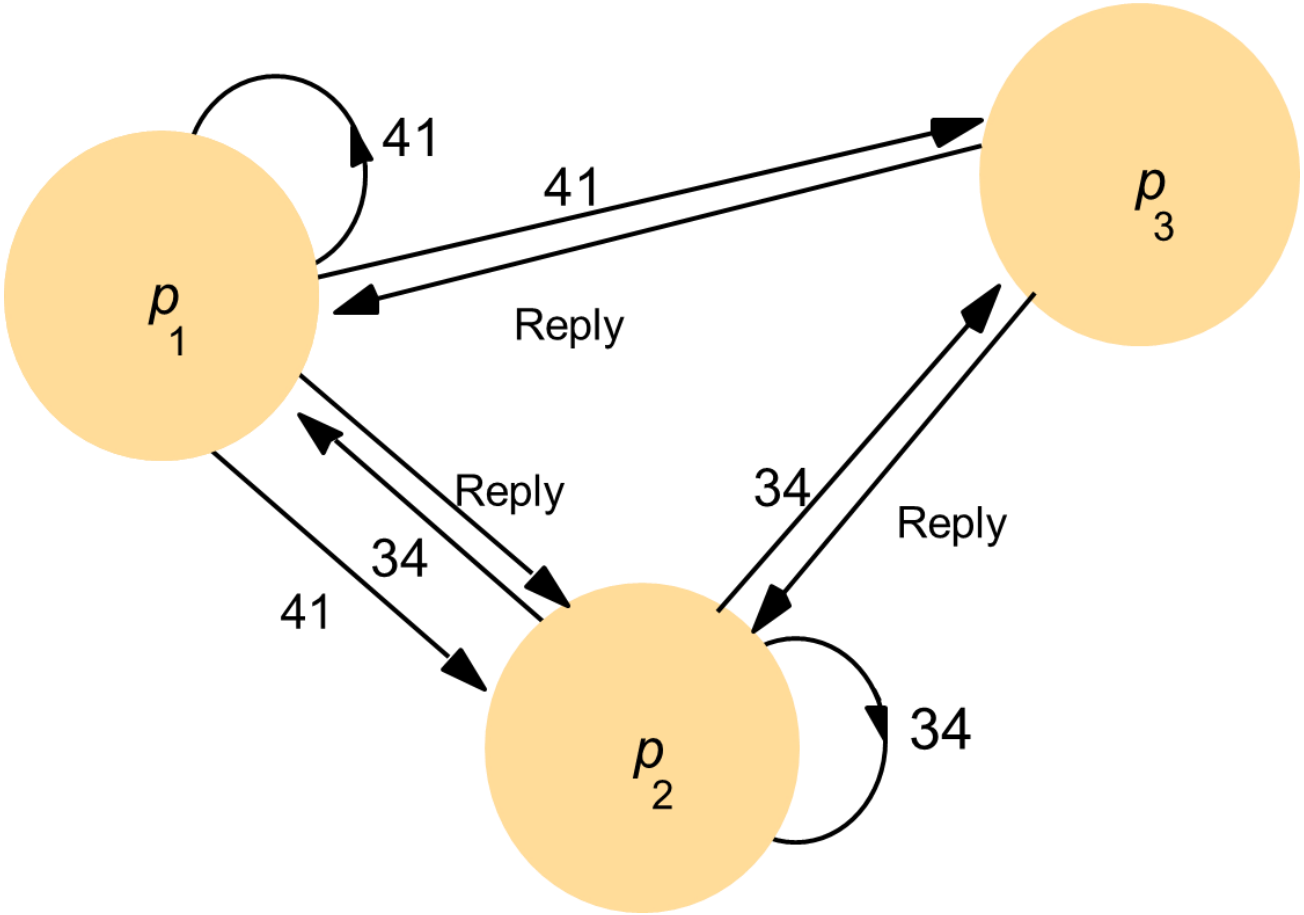


Figure 15.5 Multicast synchronization

Distributed mutual exclusion

– Algorithms for mutual exclusion

- Maekawa's voting algorithm
  - ✓ Processes need only obtain permission to enter from subsets of their peers, as long as the subsets used by any two processes overlap.
  - ✓ Voting set V_i with each process p_i (i=1,...,N), where $V_i \subseteq \{p_1, ..., p_N\}$
    - ❖ $p_i \in V_i$
    - ❖ $V_i \cap V_j \neq \emptyset$ - there is at least one common member of any two voting sets
    - ❖ $|V_i| = K$ - to be fair, each process has a voting set of the same size
    - ❖ Each process p_j is contained in M of the voting sets V_i.
  - ✓ The optimal solution, which minimizes K and allows the processes to achieve mutual exclusion, has $K \sim \sqrt{N}$ and M=K.

Distributed mutual exclusion

– Algorithms for mutual exclusion

- Maekawa's voting algorithm

✓ The protocol in Figure 15.6

❖ To obtain entry to the critical section, a process p_i sends *request* messages to all K members of V_i (including itself).

❖ p_i cannot enter the critical section until it has received all K *reply* messages.

❖ When a process p_j in V_i receives p_i's request message, it sends a *reply* message immediately, unless either its state is HELD or it has already replied ('voted') since it last received a *release* message.

❖ Otherwise, it queues the request message (in the order of its arrival) but does not yet reply.

❖ When a process receives a *release* message, it removes the head of its queue of outstanding requests and sends a *reply* message.

❖ To leave the critical section, p_i sends *release* messages to all the K members of V_i (including itself).

✓ Achieves the safety property ME1 - It is impossible for two processes p_i and p_j ($i \neq j$) to enter the CS at the same time since the algorithm allows a process to make at most one vote.

Distributed mutual exclusion

On initialization
  state := RELEASED;
  voted := FALSE;
For p_i to enter the critical section
  state := WANTED;
  Multicast *request* to all processes in V_i;
  Wait until (number of replies received = K);
  state := HELD;
On receipt of a request from p_i at p_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

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)
  then
    remove head of queue – from p_k, say;
    send *reply* to p_k;
    voted := TRUE;
  else
    voted := FALSE;
  end if

Figure 15.6 Maekawa's algorithm

Distributed mutual exclusion

– Algorithms for mutual exclusion

- Maekawa's voting algorithm
  - ✓ Deadlock-prone - Consider three processes p_1, p_2 and p_3 with $V_1=\{p_1, p_2\}$, $V_2=\{p_2, p_3\}$ and $V_3=\{p_3, p_1\}$.
    - ❖ If three processes concurrently request entry to the CS, then it is possible for p_1 to reply to itself and hold off p_2 ; for p_2 to reply to itself and hold off p_3 ; and for p_3 to reply to itself and hold off p_1. Each process has received one out of two replies, and none can proceed.
- Fault tolerance
  - ✓ None of the algorithms would tolerate the loss of messages, if the channels were unreliable.
  - ✓ The ring-based algorithm cannot tolerate a crash failure of any single process.
  - ✓ Maekawa's algorithm can tolerate some process crash failures: if a crashed process is not in a voting set that is required, then its failure will not affect the other processes.
  - ✓ Central server algorithm can tolerate the crash failure of a client process that neither holds nor has requested the token.

Elections

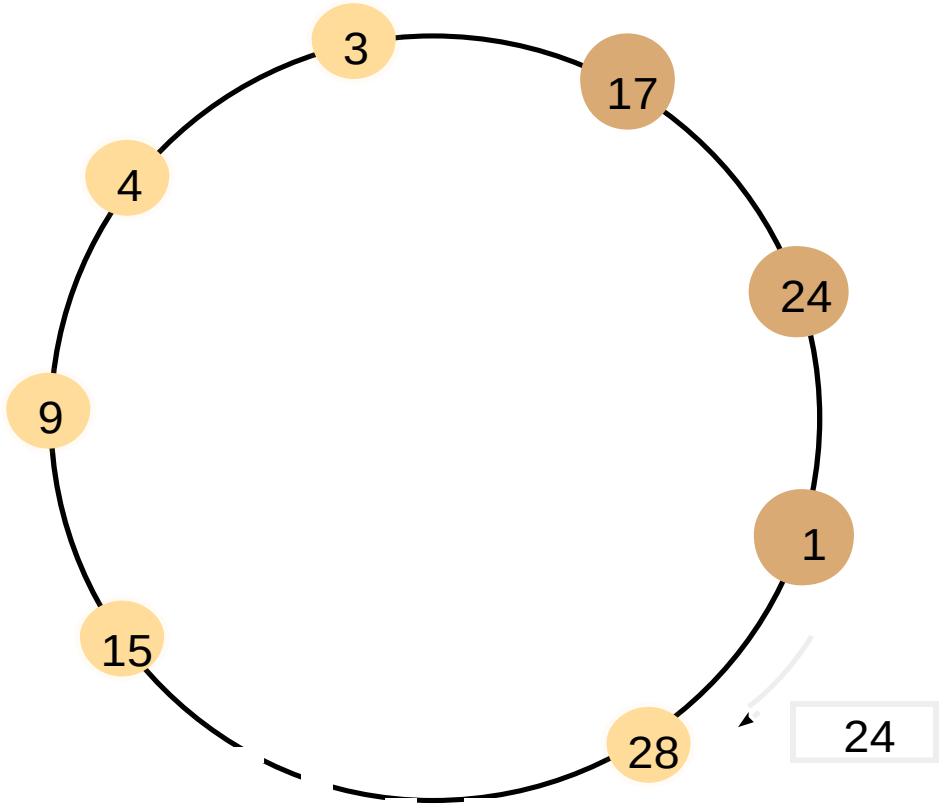
- Choosing a unique process to play a particular role
  - ✓ Central server for mutual exclusion
  - ✓ Ring master in token ring networks
- Election algorithm
  - ✓ Participant: meaning that it is engaged in some run of the election algorithm
  - ✓ Non-participant: meaning that it is not currently engaged in any election
  - ✓ An important requirement is for the choice of elected process to be unique, even if several processes call elections concurrently.
  - ✓ The elected process is chosen as the one with the largest identifier.
  - ✓ Properties
    - ❖ E1:(safety) - A participant process p_i has $elected_i = \perp$ or $elected_i = P$, where P is chosen as the non-crashed process at the end of the run with the largest identifier.
    - ❖ E2:(liveness) - All processes p_i participate and eventually set $elected_i \neq \perp$ – or crash.

Elections

– A ring-based election algorithm

- The algorithm of Chang and Roberts [1979], which is suitable for a collection of processes arranged in a logical ring
  - ✓ Each process p_i has a communication channel to the next process in the ring, $p_{(i+1) \bmod N}$, and all messages are sent clockwise around the ring.
  - ✓ Assumption
    - ❖ No failures occur and the system is asynchronous.
  - ✓ The goal of this algorithm is to elect a single process called the coordinator, which is the process with the largest identifier.
- Election algorithm (ring)
  - ✓ Initial phase
    - ❖ Every process is marked as a non-participant in an election.
    - ❖ It proceeds by marking itself as a participant, placing its identifier in an election message and sending it to its clockwise neighbour.

Elections



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

Figure 15.7 A ring-based election in progress

Elections

– A ring-based election algorithm

- ✓ When a process receives an election message,
  - ❖ It compares the identifier in the message with its own.
  - ❖ If the arrived identifier is the greater, then it forwards the message to its neighbour and marks itself as a participant.
  - ❖ If the arrived identifier is smaller and the receiver is not a participant then it substitutes its own identifier in the message and forwards it.
  - ❖ It does not forward the message if it is already a participant.
- ✓ If the received identifier is that of the receiver itself,
  - ❖ This process's identifier must be the greatest and it becomes the coordinator.
  - ❖ The coordinator marks itself as a non-participant once more and sends an elected message to its neighbour, announcing its election and enclosing its identity.
- ✓ When a process p_i receives an elected message,
  - ❖ It marks itself as a non-participant, sets its variable $elected_i$ to the identifier in the message and, unless it is the new coordinator, forwards the message to its neighbour.

Elections

– A ring-based election algorithm

- Properties
  - ✓ The condition E1 is satisfied.
    - ❖ All identifiers are compared, since a process must receive its own identifier back before sending an elected message.
  - ✓ The condition E2 follows from the guaranteed traversals of the ring.
- Performance
  - ✓ The worst-performing case is when its anti-clockwise neighbour has the highest identifier.
    - ❖ A total of N-1 messages is then required to reach this neighbour, which will not announce its election until its identifier has completed another circuit, taking a further N messages.
  - ✓ The elected message is sent N times, making 3N-1 messages in all.
  - ✓ The turnaround time is also 3N-1, since these messages are sent sequentially.

Elections

- The bully algorithm
  - The algorithm of Garcia-Molina [1982], which allows processes to crash during an election
    - ✓ Assumption
      - ❖ The system is synchronous.
      - ❖ It uses timeouts to detect a process failure.
      - ❖ Each process knows which processes have higher identifiers, and it can communicate with all such processes.
    - ✓ Three types of messages
      - ❖ Election - announcing an election
      - ❖ Answer - in response to an election message
      - ❖ Coordinator - announcing the identity of the elected process
  - Election algorithm (bully)
    - ✓ Synchronous system - reliable failure detector
      - ❖ An upper bound on the total elapsed time from sending a message to another process to receiving a response : $2T_{trans} + T_{process}$

        T_{trans} : maximum message transmission delay **/** $T_{process}$: maximum delay

Elections

– The bully algorithm

- Election algorithm (bully)
  - ✓ Election phase
    - ❖ A process with a lower ID begins an election by sending an election message to those processes that have a higher ID and awaits an answer message in response.
    - ❖ If none arrives within time T, the process considers itself the coordinator and sends a coordinator message to all processes with lower IDs announcing this.
    - ❖ Otherwise, the process waits a further period T′ for a coordinator message to arrive from the new coordinator.
      - ➢ If none arrives, it begins another election.
    - ❖ If a process receives an election message, it sends back an answer message and begins another election - unless it has begun one already.

Elections

- The bully algorithm
  - Election algorithm (bully)
    - ✓ Coordinator phase
      - ❖ The process that knows it has the highest ID can elect itself as the coordinator by sending a coordinator message to all processes with lower IDs.
      - ❖ If a process p_i receives a coordinator message, it sets its variable $elected_i$ to the ID of the coordinator contained within it and treats that process as the coordinator.
    - ✓ When a process is started to replace a crashed process,
      - ❖ It begins an election.
      - ❖ If it has the highest process ID, then it will decide that it is the coordinator and announce this to the other processes.
      - ❖ Thus it will become the coordinator, even though the current coordinator is functioning. ⇒ "**Bully**" algorithm

Elections

- ✓ p_1 detects the failure of the coordinator p_4, and announces an election.
- ✓ p_2 and p_3 send answer messages to p_1 and begin their own election; p_3 sends an answer message to p_2, but p_3 receives no answer message from p_4.
- ✓ Process p_3 fails.
- ✓ When p_1's timeout period T′ expires, it begins another election.
- ✓ p_2 is elected coordinator.

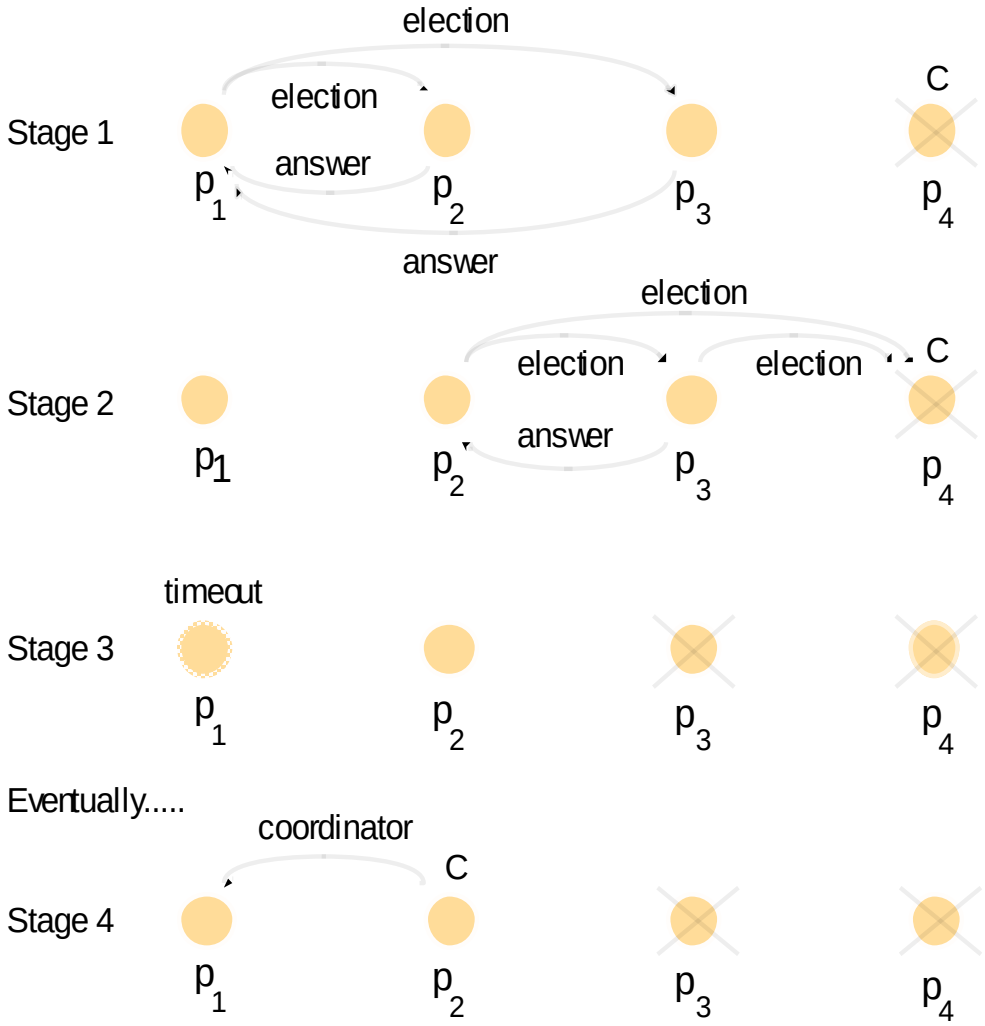


Figure 15.8 The bully algorithm

Elections

– The bully algorithm

- Properties
  - ✓ E1 is satisfied if no process is replaced.
  - ✓ E2 is satisfied by the assumption of reliable message delivery.
  - ✓ E1 is not guaranteed If processes that have crashed are replaced by processes with the same ID.
    - ❖ A process that replaces a crashed process *p* may decide that it has the highest ID just as another process (which has detected *p*'s crash) has decided that it has the highest ID. Two processes will announce themselves as the coordinator concurrently.
- Performance
  - ✓ Best case - The process with the second highest ID notices the coordinator's failure.
    - ❖ elect itself coordinator and send N-2 coordinator messages
    - ❖ Turnaround time is one message
  - ✓ Worst case - $O(N^2)$ messages
    - ❖ When the process with the least ID first detects the coordinator's failure

Consensus and related problems

- Consensus
  - Problems of agreement
    - ✓ To agree on a value after one or more of the processes has proposed what that value should be
      - ❖ A situation in which two armies should decide consistently to attack or retreat
      - ❖ All the correct computers controlling a spaceship's engines should decide 'proceed', or all of them decide 'abort'.
      - ❖ In a transaction to transfer funds from one account to another, the computers involved must consistently agree to perform the respective debit and credit.
      - ❖ In mutual exclusion, the processes agree on which process can enter the critical section.
      - ❖ In an election, the processes agree on which is the elected process.
      - ❖ In a totally ordered multicast, the processes agree on the order of message delivery.

Consensus and related problems

- System model and problem definitions
  - ✓ A collection of processes p_i (i=1, …, N) communicating by message passing
  - ✓ Is for consensus to be reached even in the presence of faults
  - ✓ Communication is reliable but that processes may fail.
- Definition of the consensus problem
  - ✓ To reach consensus, every process p_i begins in the undecided state and proposes a single value v_i
  - ✓ The processes communicate with one another, exchanging values.
  - ✓ Each process sets the value of a decision variable d_i. (enters the decided state)
  - ✓ The requirements of a consensus algorithm
    - ❖ Termination: Eventually each correct process sets its decision variable.
    - ❖ Agreement: The decision value of all correct processes is the same: if p_i and p_j are correct and have entered the decided state, then $d_i=d_j$.
    - ❖ Integrity: If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value.

Consensus and related problems

- Definition of the consensus problem
  - ✓ In figure 15.16,
    - ❖ Three processes engaged in a consensus algorithm
    - ❖ Two processes propose '**proceed**' and a third proposes '**abort**' but then crashes.
    - ❖ The two processes that remain correct each decide 'proceed'.

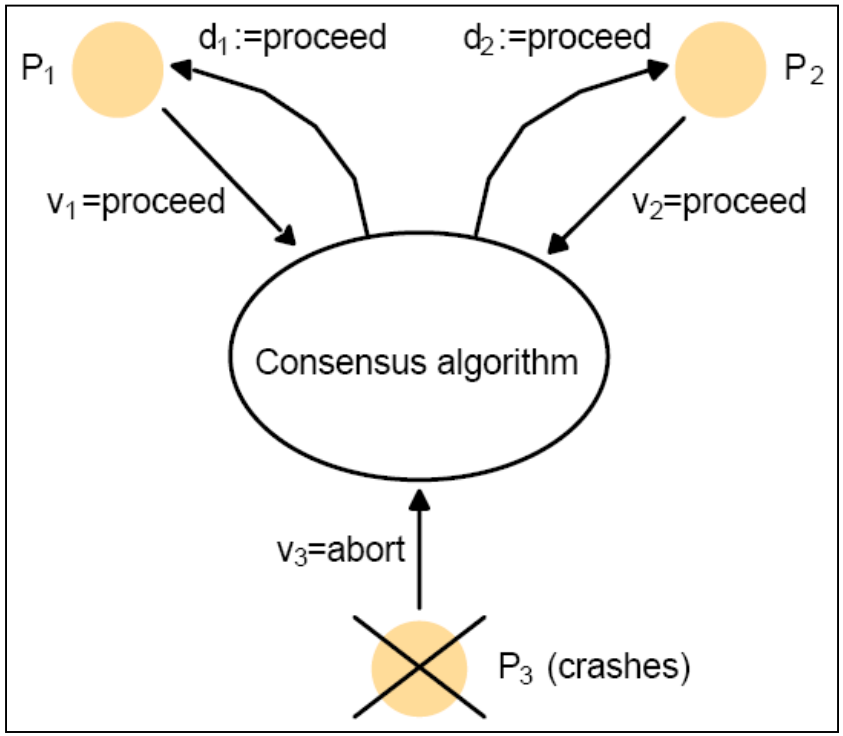


Figure 15.16
Consensus for three processes

Consensus and related problems

- Definition of the consensus problem
  - ✓ To solve consensus
    - ❖ Collect the processes into a group and have each process reliably multicast its proposed value to the members of the group
    - ❖ Each process waits until it has collected all N values.
    - ❖ It evaluates the function majority(v_1, … , v_N), which returns the value that occurs most often among its arguments, or special value $\perp \notin D$ if no majority exists.
  - ✓ Properties
    - ❖ **Termination**: guaranteed by the reliability of the multicast operation
    - ❖ **Agreement** and **integrity**: guaranteed by the definition of majority, and integrity property of a reliable multicast
  - ✓ If processes can crash,
    - ❖ Introduces the complication of detecting failures.
    - ❖ A run of the consensus algorithm can terminate?
  - ✓ If processes can fail in arbitrary (byzantine) ways,
    - ❖ Faulty processes can communicate random values to the others.
    - ❖ Not be accidental but the result of mischievous or malevolent operation

Consensus and related problems

- The byzantine generals problem
  - ✓ Three or more generals are to agree to attack or to retreat
  - ✓ One, the commander, issues the order. The others, lieutenants to the commander, are to decide to attack or retreat.
  - ✓ One or more of the generals may be 'treacherous' (faulty).
    - ❖ If the commander is treacherous, he proposes attacking to one general and retreating to another.
    - ❖ If a lieutenant is treacherous, he tells one of his peers that the commander told him to attack and another that they are to retreat.
  - ✓ The byzantine generals problem differs from consensus.
    - ❖ A distinguished process supplies a value that the others are to agree upon, instead of each of them proposing a value.
  - ✓ Requirements
    - ❖ Termination: Eventually each correct process sets its decision variable.
    - ❖ Agreement: The decision value of all correct processes is the same: if p_i and p_j are correct and have entered the decided state, then $d_i=d_j$.
    - ❖ Integrity: If the commander is correct, then all correct processes decide on the value that the commander proposed.

Consensus and related problems

- The byzantine generals problem in a synchronous system
  - ✓ Assume that processes can exhibit arbitrary failures
  - ✓ A faulty process may send any message with any value at any time; and it may omit to send any message.
  - ✓ Up to f of the N processes may be faulty.
  - ✓ Correct processes can detect the absence of a message through a timeout, but they cannot conclude that the sender has crashed, since it may be silent for some time and then send messages again.
  - ✓ Assume that the communication channels between pairs of processes are private.
    - ❖ No faulty process can inject messages into the communication channel between correct processes.
  - ✓ The case of three processes that send unsigned messages to one another
    - ❖ No solution that guarantees to meet the conditions of the byzantine generals problem if one process is allowed to fail
    - ❖ No solution exists if $N \leq 3f$
    - ❖ Give an algorithm for $N \geq 3f+1$

Consensus and related problems

- Impossibility with three processes
  - ✓ In figure 15.18
    - ❖ One of three processes is faulty.
    - ❖ Two rounds of messages - the values the commander sends, and the values that the lieutenants subsequently send to each other.
    - ❖ The numeric prefixes - the sources of messages and different rounds
    - ❖ Symbol ':' (says) - '3:1:u' (3 says 1 says u)
    - ❖ In the left-hand scenario
      - ➢ One of the lieutenants, p_3 is faulty.
      - ➢ The commander sends the value v to each of the other two processes.
      - ➢ p_2 sends the value v to p_3, but p_3 sends a value $u \neq v$ to p_2.
      - ➢ All p_2 knows at this stage is that it has received differing values; it cannot tell which were sent out by the commander.
    - ❖ In the right-hand scenario
      - ➢ The commander, p_1 is faulty, and sends differing values to lieutenants.
      - ➢ After p_3 has correctly echoed the value x that it received, p_2 is in the same situation as it was in when p_3 was faulty: it has received two differing values.

Consensus and related problems

- Impossibility with N ≤ 3f
  - 3 processes that send messages to one another
  - 1 process allowed to fail

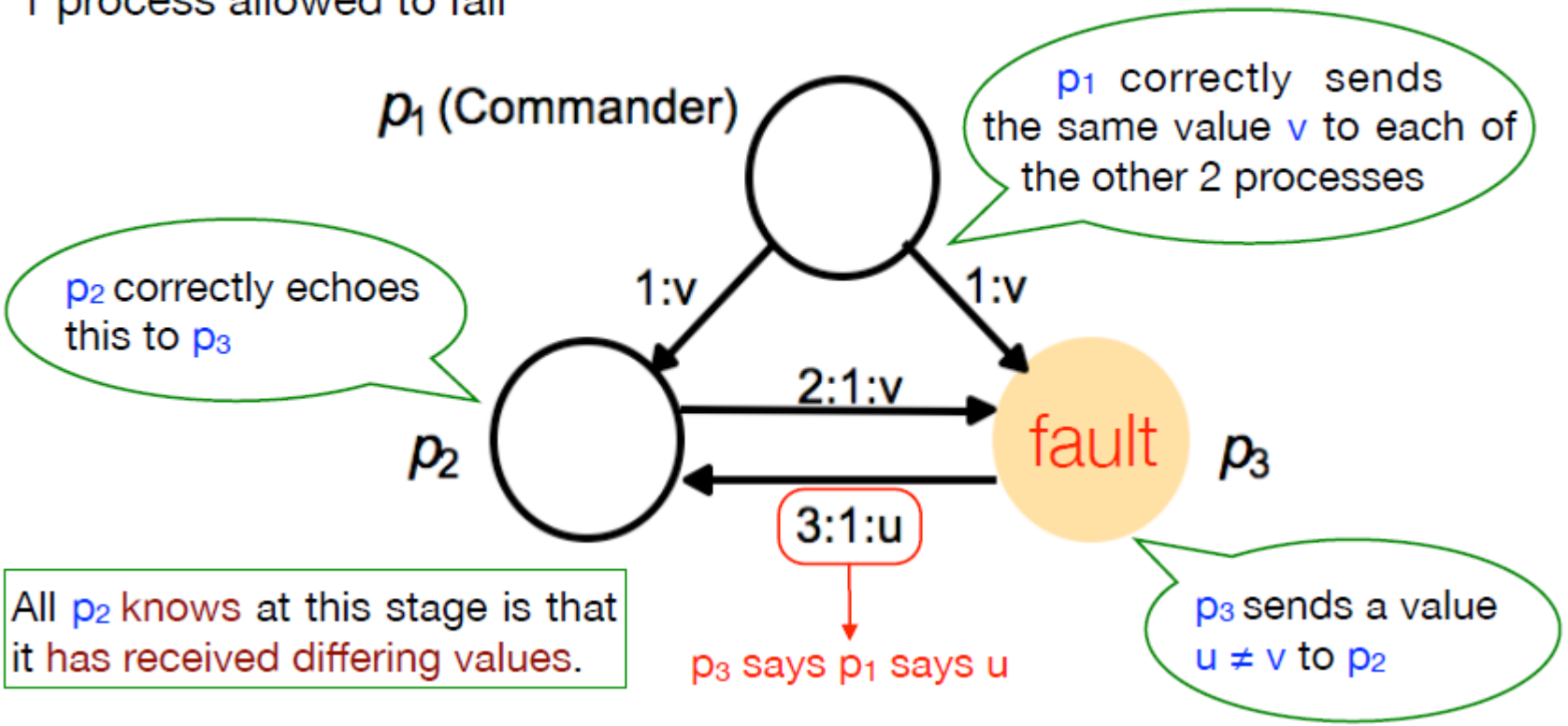


Figure 15.18 Three byzantine generals

Consensus and related problems

- Impossibility with N ≤ 3f
  - 3 processes that send messages to one another
  - 1 process allowed to fail

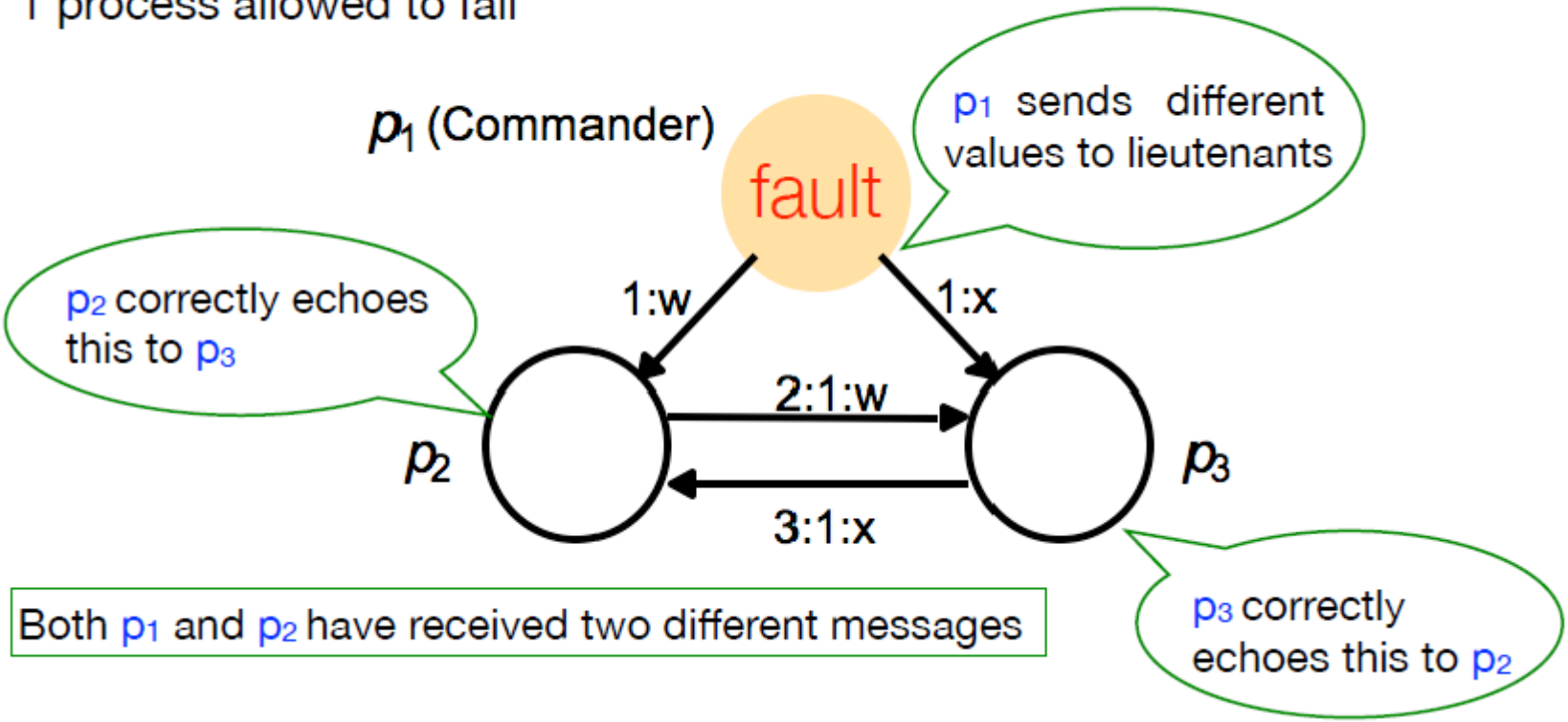


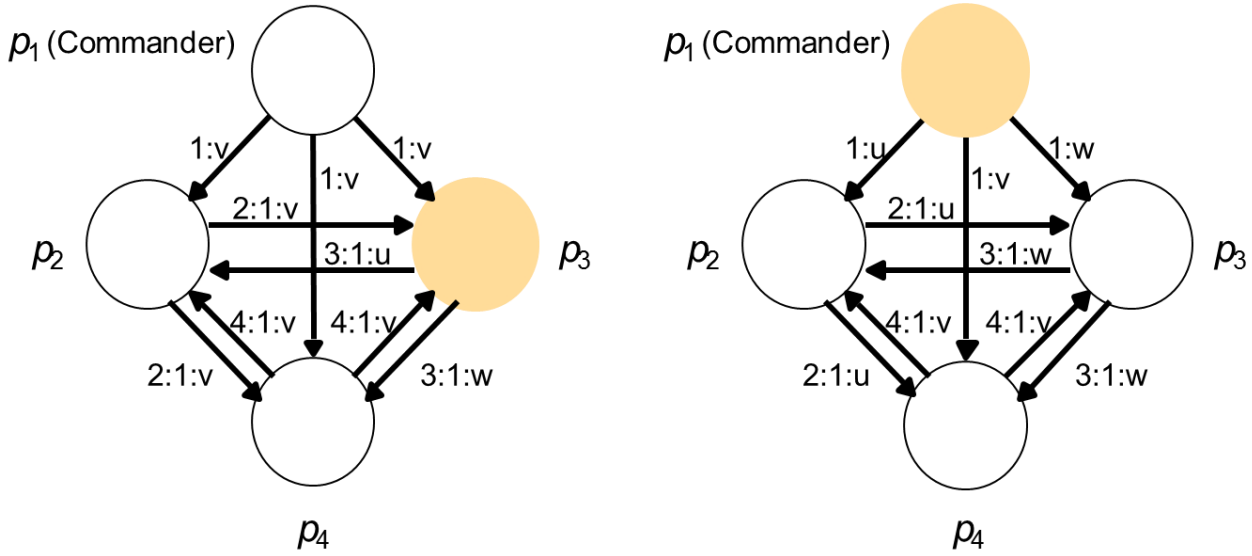
Figure 15.18 Three byzantine generals

Consensus and related problems

- Solution with one faulty process
  - ✓ Is complex to describe fully the algorithm of Pease that solves the byzantine generals problem in a synchronous system with $N \geq 3f+1$.
  - ✓ The operation of the algorithm for the case $N \geq 4$, $f = 1$ and illustrate it for $N = 4$, $f = 1$.
    - ❖ Correct generals reach agreement in two rounds of messages:
      - ➢ In the first round, the commander sends a value to each of the lieutenants.
      - ➢ In the second round, each of the lieutenants sends the value it received to its peers.
    - ❖ A lieutenant receives a value from the commander, plus N-2 values from its peers.
    - ❖ If the commander is faulty, then all the lieutenants are correct and each will have gathered exactly the set of values that the commander sent out.
    - ❖ If one lieutenant is faulty, each of peers receives N-2 copies of the value that the commander send, plus a value that the faulty lieutenant sent to it.

Consensus and related problems

- Solution with one faulty process
  - ❖ Correct lieutenants need only apply a simple majority function to the set of values they receive. Since N ≥ 4, (N-2) ≥ 2.
    - ➢ The majority function will ignore any value that a faulty lieutenant sent, and it will produce the value that the commander sent if the commander is correct.
  - ❖ In the left-hand case, two correct lieutenant processes agree.
    - ➢ p_2 and p_4 decide on majority(v, u, v) = v and majority(v, v, w) = v.
  - ❖ In the right-hand case the commander is faulty, but the three correct processes agree: p_2, p_3 and p_4 decide on majority(u, v, w) = ⊥.



Faulty processes are shown in color

Figure 15.19 Four byzantine generals