CSD511 – Distributed Systems
分散式系統

Chapter 12
Coordination and Agreement

吳俊興
國立高雄大學 資訊工程學系

Chapter 12 Coordination and Agreement

12.1 Introduction

- Fundamental issue: for a set of processes, how to coordinate their actions or to agree on one or more values?
  - even no fixed master-slave relationship between the components
- Further issue: how to consider and deal with failures when designing algorithms
- Topics covered
  - mutual exclusion
  - how to elect one of a collection of processes to perform a special role
  - multicast communication
  - agreement problem: consensus and byzantine agreement

Failure Assumptions and Failure Detectors

- Failure assumptions of this chapter
  - Reliable communication channels
  - Processes only fail by crashing unless state otherwise
- Failure detector: object/code in a process that detects failures of other processes
- unreliable failure detector
  - One of two values: unsuspected or suspected
    - Evidence of possible failures
  - Example: most practical systems
    - Each process sends “alive/I’m here” message to everyone else
    - If not receiving “alive” message after timeout, it’s suspected
      - maybe function correctly, but network partitioned
- reliable failure detector
  - One of two accurate values: unsuspected or failure
  - few practical systems

12.2 Distributed Mutual Exclusion

- Process coordination in a multitasking OS
  - **Race condition**: several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access take place
  - **critical section**: when one process is executing in a critical section, no other process is to be allowed to execute in its critical section
  - **Mutual exclusion**: If a process is executing in its critical section, then no other processes can be executing in their critical sections
- Distributed mutual exclusion
  - Provide critical region in a distributed environment
  - message passing

  for example, locking files, lockd daemon in UNIX

  (NFS is stateless, no file-locking at the NFS level)

Algorithms for mutual exclusion

- Problem: an asynchronous system of *N* processes
  - processes don't fail
  - message delivery is reliable; not share variables
  - only one critical region
  - application-level protocol: enter(), resourceAccesses(), exit()
- Requirements for mutual exclusion
  - Essential
    - [ME1] safety: only one process at a time
    - [ME2] liveness: eventually enter or exit
  - Additional
    - [ME3] happened-before ordering: ordering of enter() is the same as HB ordering
- Performance evaluation
  - overhead and bandwidth consumption: # of messages sent
  - client delay incurred by a process at entry and exit
  - throughput measured by synchronization delay: delay between one's exit and next's entry

A central server algorithm

- server keeps track of a token---permission to enter critical region
  - a process requests the server for the token
  - the server grants the token if it has the token
  - a process can enter if it gets the token, otherwise waits
  - when done, a process sends release and exits

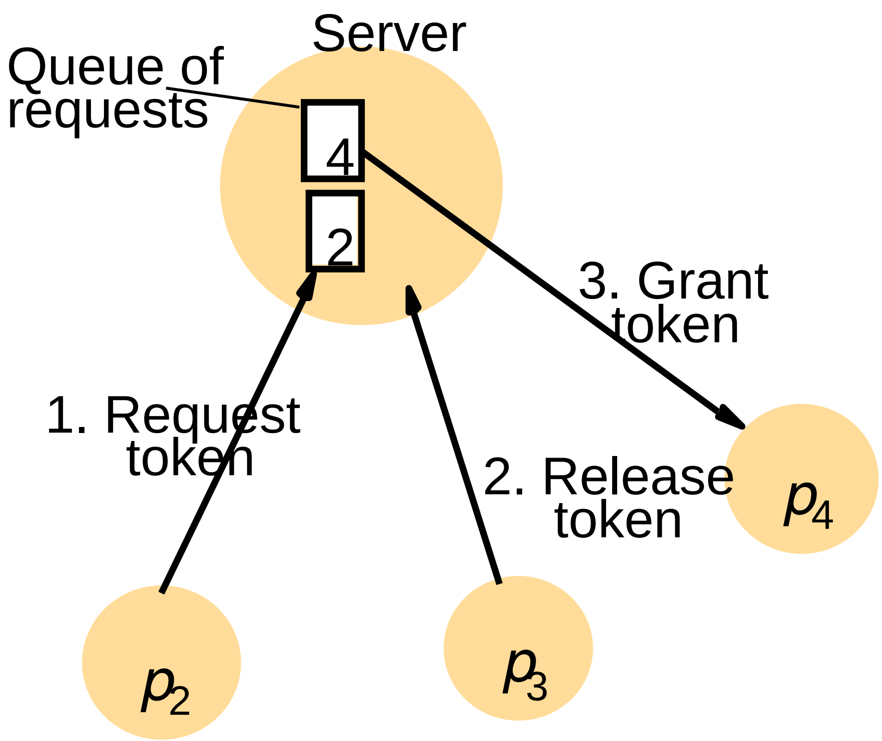


Figure 12.2 Server managing a mutual exclusion token for a set of processes

A central server algorithm: discussion

- Properties
  - safety, why?
  - liveness, why?
  - HB ordering not guaranteed, why?
- Performance
  - enter overhead: two messages (request and grant)
  - enter delay: time between request and grant
  - exit overhead: one message (release)
  - exit delay: none
  - synchronization delay: between release and grant
  - centralized server is the bottleneck

A ring-based algorithm

- Arrange processes in a logical ring to rotate a token
  - Wait for the token if it requires to enter the critical section
  - The ring could be unrelated to the physical configuration
- p_i sends messages to $p_{(i+1) \bmod N}$
  - when a process requires to enter the critical section, waits for the token
  - when a process holds the token
    - If it requires to enter the critical section, it can enter
      - when a process releases a token (exit), it sends to its neighbor
    - If it doesn't, just immediately forwards the token to its neighbor

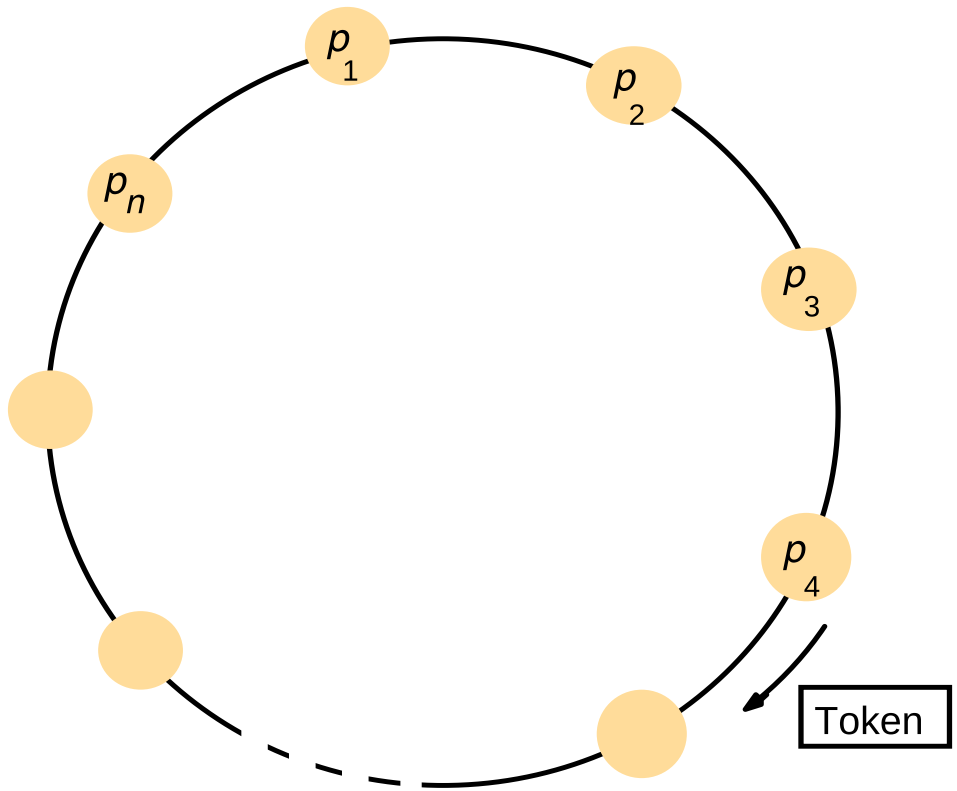


Figure 12.3 A ring of processes transferring a mutual exclusion token

A ring-based algorithm: discussion

- Properties
  - safety, why?
  - liveness, why?
  - HB ordering not guaranteed, why?
- Performance
  - bandwidth consumption: token keeps circulating
  - enter overhead: 0 to *N* messages
  - enter delay: delay for 0 to *N* messages
  - exit overhead: one message
  - exit delay: none
  - synchronization delay: delay for 1 to *N* messages

An algorithm using multicast and logical clocks

- Multicast a request message for the token (Ricart and Agrawala [1981])
  - enter only if all the other processes reply
  - totally-ordered timestamps: $<T, p_i>$
- Each process keeps a *state: RELEASED, HELD, WANTED*
  - if all have *state* = *RELEASED*, all reply, a process can hold the token and enter
  - if a process has *state* = *HELD*, doesn't reply until it exits
  - if more than one process has *state* = *WANTED*, process with the lowest timestamp will get all *N*-1 replies first

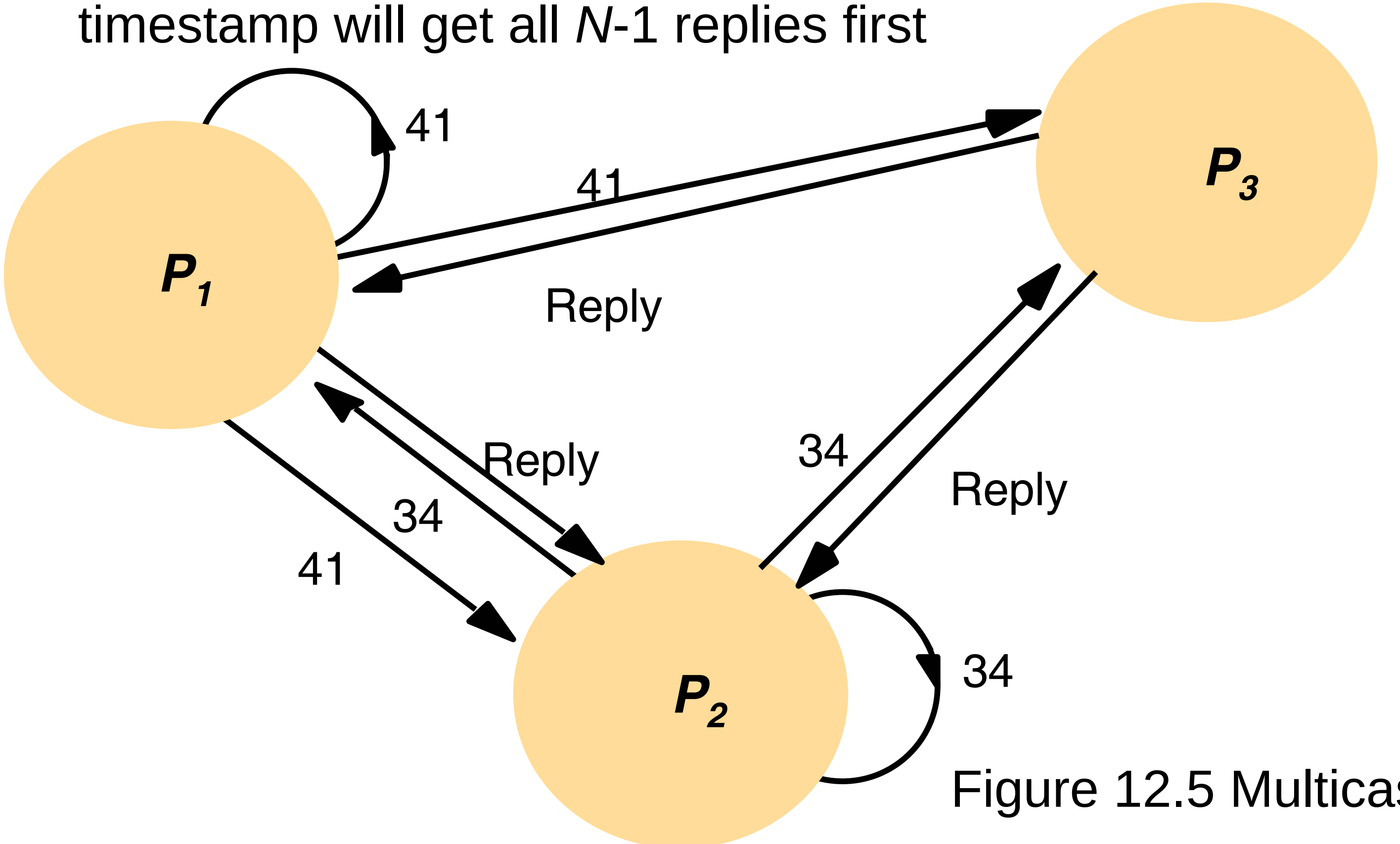


Figure 12.5 Multicast synchronization

Figure 12.4 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;

An algorithm using multicast: discussion

- Properties
  - safety, why?
  - liveness, why?
  - HB ordering, why?
- Performance
  - bandwidth consumption: no token keeps circulating
  - entry overhead: $2(N-1)$, why? [with multicast support: $1 + (N - 1) = N$]
  - entry delay: delay between request and getting all replies
  - exit overhead: 0 to N-1 messages
  - exit delay: none
  - synchronization delay: delay for 1 message (one last reply from the previous holder)

Maekawa's voting algorithm

- Observation: not all peers to grant it access
  - Only obtain permission from subsets, overlapped by any two processes
- Maekawa's approach
  - subsets Vi,Vj for process Pi, Pj
    - $Pi \in Vi$, $Pj \in Vj$
    - $Vi \cap Vj \neq \emptyset$, there is at least one common member
    - subset |Vi|=K, to be fair, each process should have the same size
  - Pi cannot enter the critical section until it has received all K reply messages
  - Choose a subset
    - Simple way (2√N): place processes in a √N by √N matrix and let Vi be the union of the row and column containing Pi
    - Optimal (√N): non-trivial to calculate (skim here)
  - Deadlock-prone
    - V1={P1, P2}, V2={P2, P3}, V3={P3, P1}
    - If P1, P2 and P3 concurrently request entry to the critical section, then its possible that each process has received one (itself) out of two replies, and none can proceed
    - adapted and solved by [Saunders 1987]

Figure 12.6 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;
  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

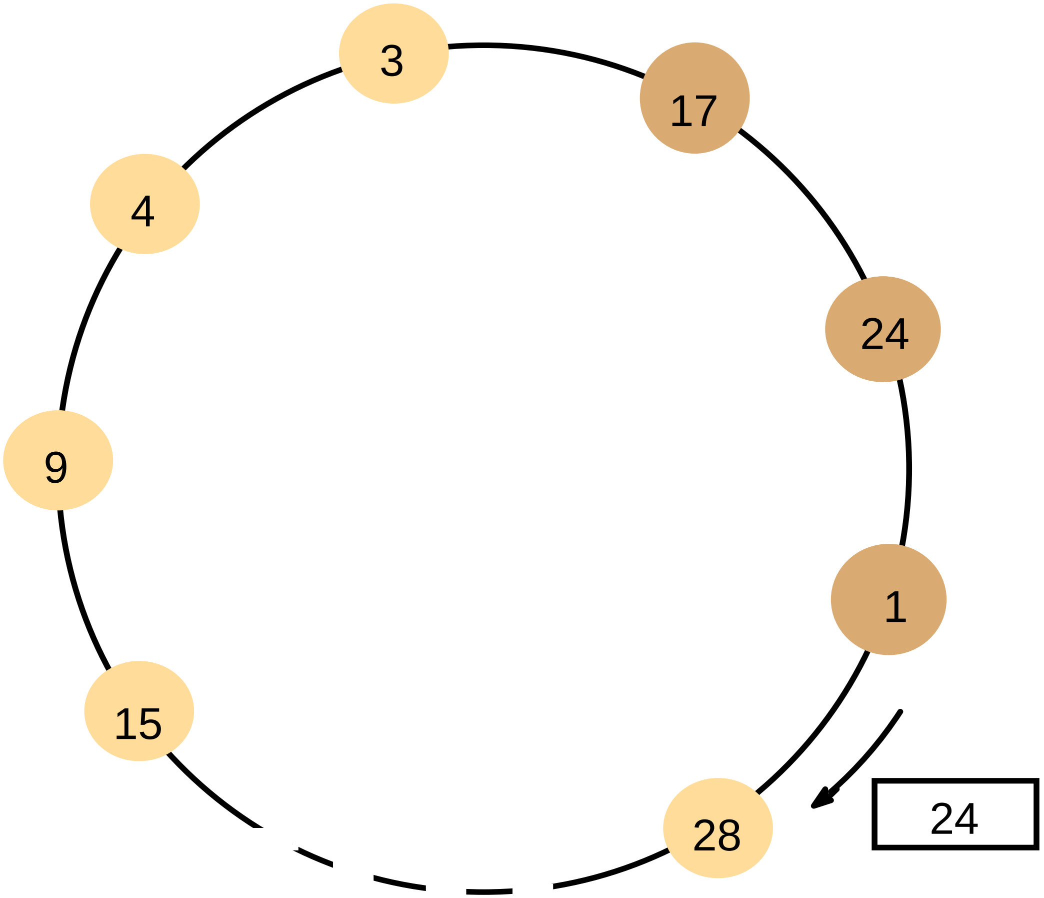
12.3 Elections

- Election: choosing a unique process for a particular role
  - All the processes agree on the ***unique*** choice
  - For example, server in dist. mutex
- Assumptions
  - Each process can call only one election at a time
  - multiple concurrent elections can be called by different processes
  - Participant: engages in an election
    - each process p_i has variable $elected_i$ = ? (don't know) initially
    - process with the *largest* identifier wins
      - The (unique) identifier could be any useful value
- Properties
  - [E1] $elected_i$ of a "participant" process must be P (elected process=largest id) or ⊥ (undefined)
  - [E2] liveness: all processes participate and eventually set $elected_i$!= ⊥ (or crash)
- Performance
  - overhead (bandwidth consumption): # of messages
  - turnaround time: # of messages to complete an election

A ring-based election algorithm

- Arrange processes in a logical ring
  - p_i sends messages to $p_{(i+1) \bmod N}$
  - It could be unrelated to the physical configuration
  - Elect the coordinator with the largest id
  - Assume no failures
- Initially, every process is a non-participant. Any process can call an election
  - Marks itself as participant
  - Places its id in an *election* message
  - Sends the message to its neighbor
  - Receiving an election message
    - if *id* > *myid*, forward the msg, mark participant
    - if *id* < *myid*
      - non-participant: replace *id* with *myid*: forward the msg, mark participant
      - participant: stop forwarding (why? Later, multiple elections)
    - if *id* = *myid*, coordinator found, mark non-participant, $elected_i$:= *id*, send *elected* message with *myid*
  - Receiving an elected message
    - *id* != *myid,* mark non-participant, $elected_i$:= *id* forward the msg
    - if *id* = *myid*, stop forwarding

Figure 12.7 A ring-based election in progress



- Receiving an election message:
  - if *id* > *myid*, forward the msg, mark participant
  - if *id* < *myid*
    - non-participant: replace *id* with *myid*: forward the msg, mark participant
    - participant: stop forwarding (why? Later, multiple elections)
  - if *id* = *myid*, coordinator found, mark non-participant, $elected_i$:= *id*, send *elected* message with *myid*
- Receiving an elected message:
  - *id* != *myid*, mark non-participant, $elected_i$:= *id* forward the msg
  - if *id* = *myid*, stop forwarding

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

A ring-based election algorithm: discussion

- Properties
  - safety: only the process with the largest id can send an *elected* message
  - liveness: every process in the ring eventually participates in the election; extra elections are stopped
- Performance
  - one election, best case, when?
    - *N election* messages
    - *N elected* messages
    - turnaround: 2*N* messages
  - one election, worst case, when?
    - 2*N* - 1 election messages
    - *N elected* messages
    - turnaround: 3*N* - 1 messages
  - can't tolerate failures, not very practical

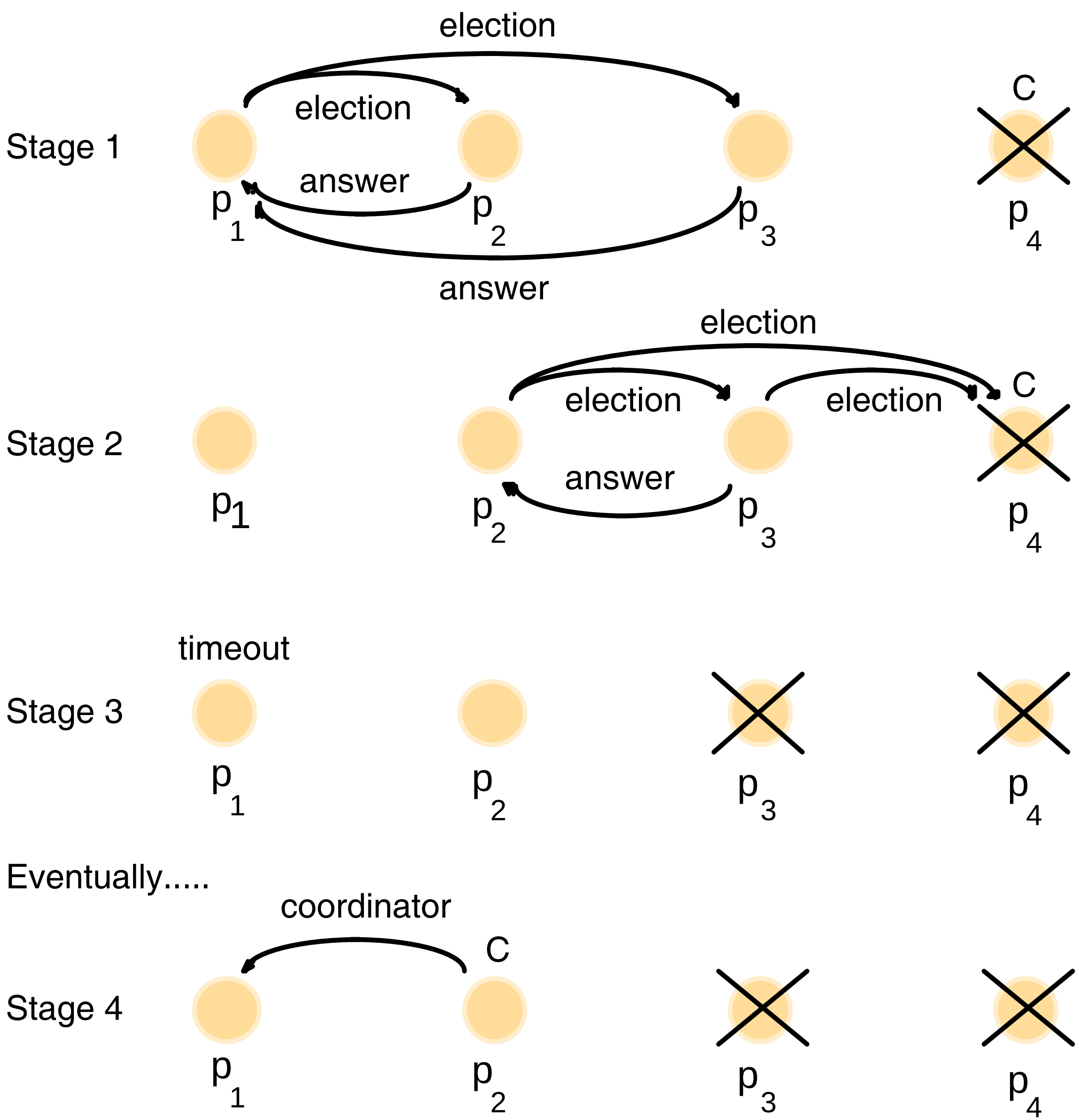
The bully election algorithm

- Assumption
  - Each process knows which processes have higher identifiers, and that it can communicate with all such processes
- Compare with ring-based election
  - Processes can crash and be detected by timeouts
    - synchronous
    - timeout $T = 2T_{transmitting}$ (max transmission delay) + $T_{processing}$ (max processing delay)
- Three types of messages
  - Election: announce an election
  - Answer: in response to Election
  - Coordinator: announce the identity of the elected process

The bully election algorithm: howto

- Start an election when detect the coordinator has failed or begin to replace the coordinator, which has lower identifier
  - Send an election message to all processes with higher id's and waits for answers (except the failed coordinator/process)
    - If no answers in time *T*
      - Considers it is the coordinator
      - sends coordinator message (with its id) to all processes with lower id's
    - else
      - waits for a coordinator message and starts an election if T' timeout
  - To be a coordinator, it has to start an election
    - A higher id process can replace the current coordinator (hence "bully")
      - The highest one directly sends a coordinator message to all process with lower identifiers
- Receiving an election message
  - sends an answer message back
  - starts an election if it hasn't started one—send election messages to all higher-id processes (including the "failed" coordinator—the coordinator might be up by now)
- Receiving a coordinator message
  - set $elected_i$ to the new coordinator

Figure 12.8 The bully algorithm



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

The bully election algorithm: discussion

- Properties
  - safety:
    - a lower-id process always yields to a higher-id process
    - However, it's guaranteed
      - if processes that have crashed are replaced by processes with the same identifier since message delivery order might not be guaranteed and
      - failure detection might be unreliable
  - liveness: all processes participate and know the coordinator at the end
- Performance
  - best case: when?
    - overhead: N-2 *coordinator* messages
    - turnaround delay: no *election/answer* messages
  - worst case: when?
    - overhead:
    - 1+ 2 + ...+ (N-2) + (N-2)= (N-1)(N-2)/2 + (N-2) *election* messages,
    - 1+...+ (N-2) *answer* messages,
    - N-2 *coordinator* messages,
    - total: (N-1)(N-2) + 2(N-2) = (N+1)(N-2) = O(N^2)
  - turnaround delay: delay of election and answer messages

12.4 Multicast Communication

- Group (multicast) communication: for each of a group of processes to receive copies of the messages sent to the group, often with delivery guarantees
  - The set of messages that every process of the group should receive
  - On the delivery ordering across the group members
- Challenges
  - Efficiency concerns include minimizing overhead activities and increasing throughput and bandwidth utilization
  - Delivery guarantees ensure that operations are completed
- Types of group
  - Static or dynamic: whether joining or leaving is considered
  - Closed or open
    - A group is said to be closed if only members of the group can multicast to it. A process in a closed group sends to itself any messages to the group
    - A group is open if processes outside the group can send to it

Reliable Multicast

- Simple basic multicasting (B-multicast) is sending a message to every process that is a member of a defined group
  - B-multicast(g, m) for each process p $\in$ group g, send(p, message m)
  - On receive(m) at p: B-deliver(m) at p
- Reliable multicasting (R-multicast) requires these properties
  - Integrity: a correct process sends a message to only a member of the group and does it only once
  - Validity: if a correct process sends a message, it will eventually be delivered
  - Agreement: if a message is delivered to a correct process, all other correct processes in the group will deliver it

Figure 12.10 Reliable multicast algorithm

On initialization
  $Received := \{\};$

For process p to R-multicast message m to group g
  $B\text{-}multicast(g, m);$ // $p \in g$ is included as a destination

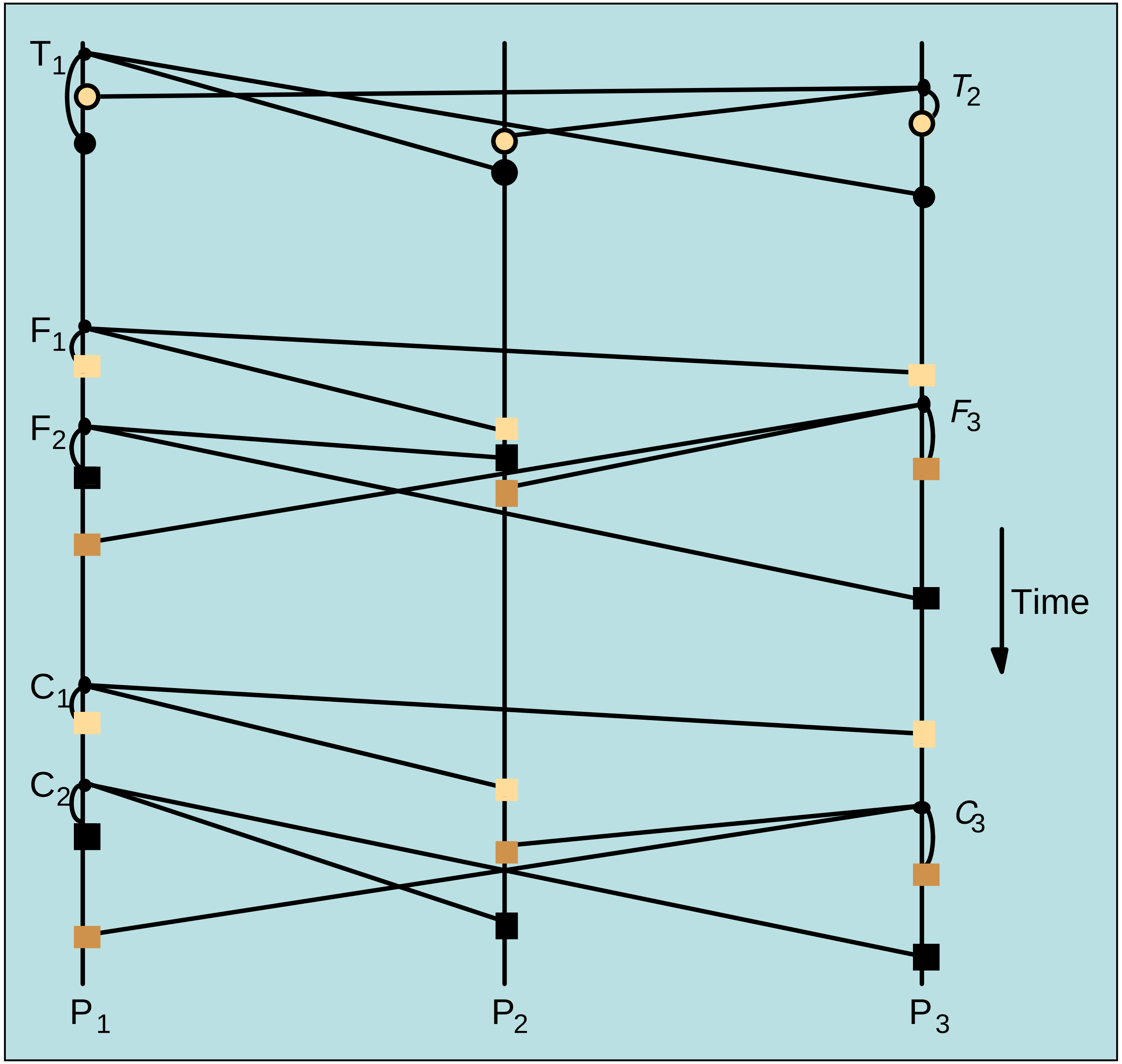
On B-deliver(m) at process q with g = group(m)
  $if\ (m \notin Received)$
  then
    $Received := Received \cup \{m\};$
    $if\ (q \neq p)\ then\ B\text{-}multicast(g, m);\ end\ if$
    $R\text{-}deliver\ m;$
  end if

Implementing reliable R-multicast over B-multicast
- When a message is delivered, the receiving process multicasts it
- Duplicate messages are identified (possible by a sequence number) and not delivered

Types of message ordering

- Three types of message ordering
  - ***FIFO (First-in, first-out) ordering***: if a correct process delivers a message before another, every correct process will deliver the first message before the other
  - ***Casual ordering***: any correct process that delivers the second message will deliver the previous message first
  - ***Total ordering***: if a correct process delivers a message before another, any other correct process that delivers the second message will deliver the first message first
- Note that
  - FIFO ordering and casual ordering are only partial orders
  - Not all messages are sent by the same sending process
  - Some multicasts are concurrent, not able to be ordered by happened-before
  - Total order demands consistency, but not a particular order

Figure 12.12 Total, FIFO and causal ordering of multicast messages



Notice

- the consistent ordering of totally ordered messages T_1 and T_2,
- the FIFO-related messages F_1 and F_2 and
- the causally related messages C_1 and C_3
  – and
- the otherwise arbitrary delivery ordering of messages

Note that T_1 and T_2 are delivered in opposite order to the physical time of message creation

Bulletin board example (FIFO ordering)

- A bulletin board such as Web Board at NJIT illustrates the desirability of consistency and FIFO ordering. A user can best refer to preceding messages if they are delivered in order. Message 25 in Figure 12.13 refers to message 24, and message 27 refers to message 23.
- Note the further advantage that Web Board allows by permitting messages to begin threads by replying to a particular message. Thus messages do not have to be displayed in the same order they are delivered

Bulletin board: *os.interesting*

Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

Figure 12.13 Display from bulletin board program

Implementing total ordering

- The normal approach to total ordering is to assign totally ordered identifiers to multicast messages, using the identifiers to make ordering decisions.
- One possible implementation is to use a sequencer process to assign identifiers. See Figure 12.14. A drawback of this is that the sequencer can become a bottleneck.
- An alternative is to have the processes collectively agree on identifiers. A simple algorithm is shown in Figure 12.15.

Figure 12.14 Total ordering using a sequencer

1. Algorithm for group member p

On initialization: $r_g := 0$;

To TO-multicast message m to group g
B-multicast$(g \cup \{sequencer(g)\}, <m, i>)$;

On B-deliver$(<m, i>)$ *with* $g = group(m)$
Place $<m, i>$ in hold-back queue;

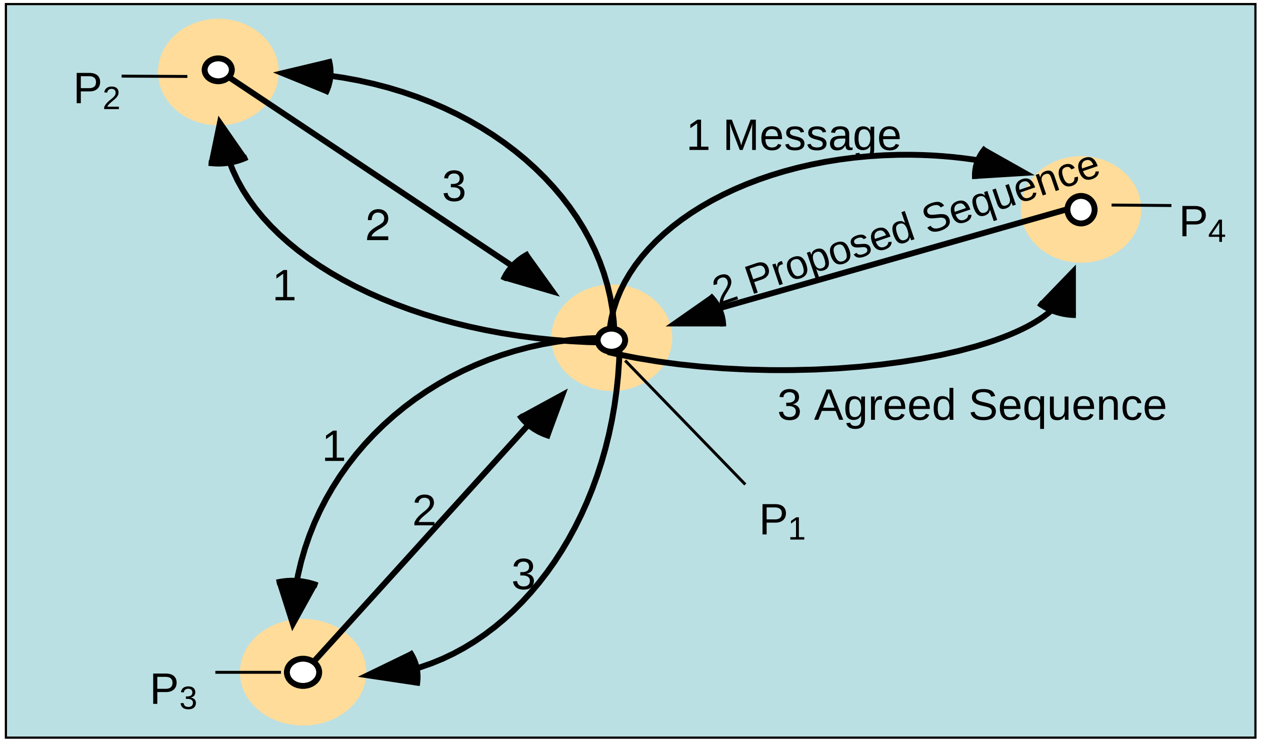
On B-deliver$(m_{order} = <\text{"order"}, i, S>)$ *with* $g = group(m_{order})$
wait until $<m, i>$ in hold-back queue and $S = r_g$;
TO-deliver m; // (after deleting it from the hold-back queue)
$r_g = S + 1$;

2. Algorithm for sequencer of g

On initialization: $s_g := 0$;

On B-deliver$(<m, i>)$ *with* $g = group(m)$
B-multicast$(g, <\text{"order"}, i, s_g>)$;
$s_g := s_g + 1$;

Figure 12.15 The ISIS algorithm for total ordering



Each process q in group g keeps

- A^q_g: the largest agreed sequence number it has observed so far for the group g
- P^q_g: its own largest proposed sequence number

Algorithm for process p to multicast a message m to group g

1. p B-multicasts <m, i> to g, where i is a unique identifier for m
2. Each process q replies to the sender p with a proposal for the message's agreed sequence number of P^q_g :=Max(A^q_g, P^q_g)+1
3. p collects all the proposed sequence numbers and selects the largest one a as the next agreed sequence number. It then B-multicasts <i, a> to g.
4. Each process q in g sets A^q_g := Max(A^q_g, a) and attaches a to the message identified by i

Implementing casual ordering

- Causal ordering using vector timestamps (Figure 12.16)
  - Only orders multicasts, and ignores one-to-one messages between processes
  - Each process updates its vector timestamp before delivering a message to maintain the count of precedent messages

Algorithm for group member p_i $(i = 1, 2 \ldots, N)$

On initialization
$V_i^g[j] := 0 \ (j = 1, 2 \ldots, N);$

To CO-multicast message m to group g
$V_i^g[i] := V_i^g[i] + 1;$
B-multicast$(g, < V_i^g, m>);$

On B-deliver$(< V_j^g, m>)$ *from* p_j, *with* $g = group(m)$
place $< V_j^g, m>$ in hold-back queue;
wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \le V_i^g[k] \ (k \ne j);$
CO-deliver m; // after removing it from the hold-back queue
$V_i^g[j] := V_i^g[j] + 1;$

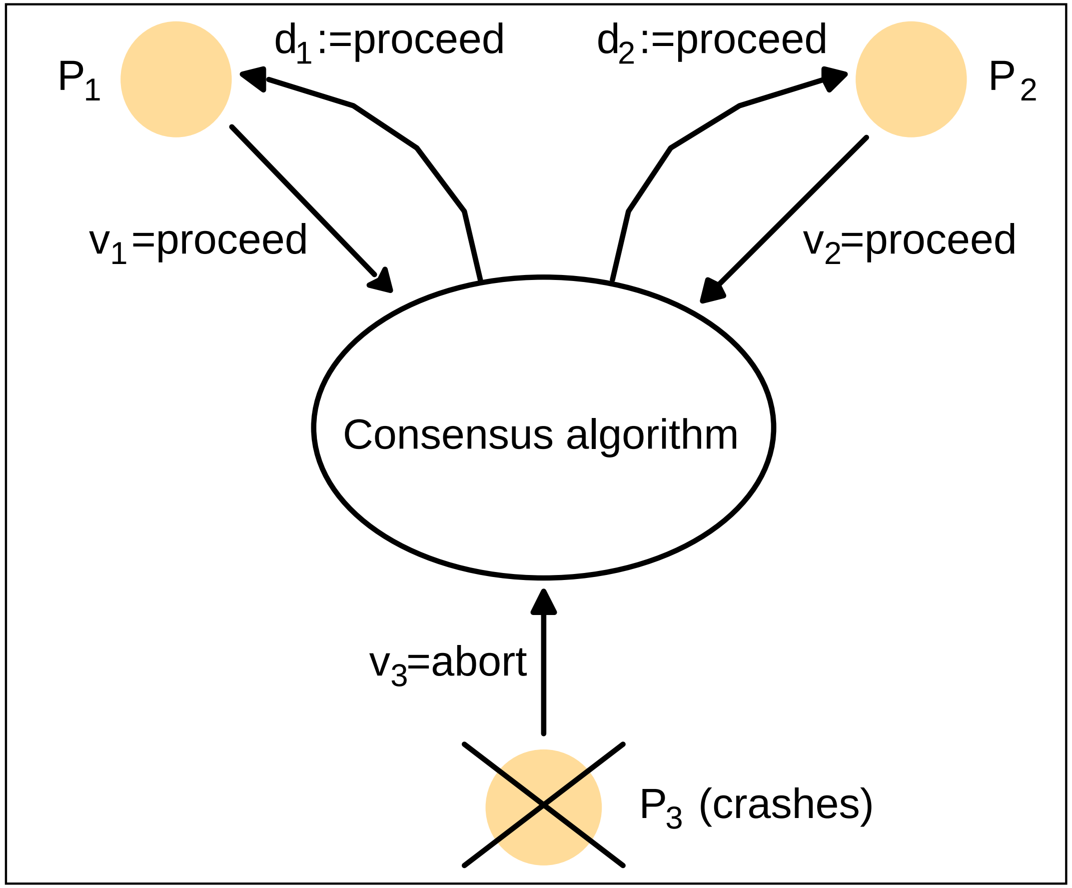
Figure 12.16 Causal ordering using vector timestamps

12.5 Consensus and related problems

- Problems of agreement
  - For processes to agree on a value (consensus) after one or more of the processes has proposed what that value should be
  - Covered topics: byzantine generals, interactive consistency, totally ordered multicast
    - The byzantine generals problem: a decision whether multiple armies should attack or retreat, assuming that united action will be more successful than some attacking and some retreating
    - Another example might be space ship controllers deciding whether to proceed or abort. Failure handling during consensus is a key concern
- Assumptions
  - communication (by message passing) is reliable
  - processes may fail
    - Sometimes up to f of the N processes are faulty

Consensus Process

1. Each process p_i begins in an undecided state and proposes a single value v_i, drawn from a set D (i=1…N)
2. Processes communicate with each other, exchanging values
3. Each process then sets the value of a decision variable d_i and enters the decided state



Two processes propose "proceed." One proposes "abort," but then crashes. The two remaining processes decide proceed.

Figure 12.17 Consensus for three processes

Requirements for Consensus

- Three requirements of a consensus algorithm
  - ***Termination***: Eventually every correct process sets its decision variable
  - ***Agreement***: The decision value of all correct processes is the same: if pi and pj are correct and have entered the *decided* state, then di=dj (i,j=1,2, …, N)
  - ***Integrity***: If the correct processes all proposed the same value, then any correct process in the *decided* state has chosen that value

The byzantine generals problem

- Problem description
  - Three or more generals must agree to *attack* or to *retreat*
  - One general, the *commander*, issues the order
  - Other generals, the *lieutenants*, must decide to attack or retreat
  - One or more generals may be treacherous
    - A *treacherous general* tells one general to attack and another to retreat
- Difference from consensus is that a single process supplies the value to agree on
- Requirements
  - *Termination*: eventually each correct process sets its decision variable
  - *Agreement*: the decision variable of all correct processes is the same
  - *Integrity*: if the commander is correct, then all correct processes agree on the value that the commander has proposed (but the commander need not be correct)

The interactive consistency problem

- Interactive consistency: all correct processes agree on a vector of values, one for each process. This is called the decision vector
  - Another variant of consensus
- Requirements
  - *Termination*: eventually each correct process sets its decision variable
  - *Agreement*: the decision vector of all correct processes is the same
  - *Integrity*: if any process is correct, then all correct processes decide the correct value for that process

Relating consensus to other problems

- Consensus (C), Byzantine Generals (BG), and Interactive Consensus (IC) are all problems concerned with making decisions in the context of arbitrary or crash failures
- We can sometimes generate solutions for one problem in terms of another. For example
  - We can derive IC from BG by running BG N times, once for each process with that process acting as commander
  - We can derive C from IC by running IC to produce a vector of values at each process, then applying a function to the vector's values to derive a single value.
  - We can derive BG from C by
    - Commander sends proposed value to itself and each remaining process
    - All processes run C with received values
    - They derive BG from the vector of C values

Consensus in a Synchronous System

- Up to f processes may have crash failures, all failures occurring during f+1 rounds. During each round, each of the correct processes multicasts the values among themselves
- The algorithm guarantees all surviving correct processes are in a position to agree
- Note: any process with f failures will require at least f+1 rounds to agree

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization

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

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

$B\text{-}multicast(g,\ Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

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

while (in round r)

{

On B-deliver(V_j) *from some* p_j

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

}

After $(f + 1)$ *rounds*

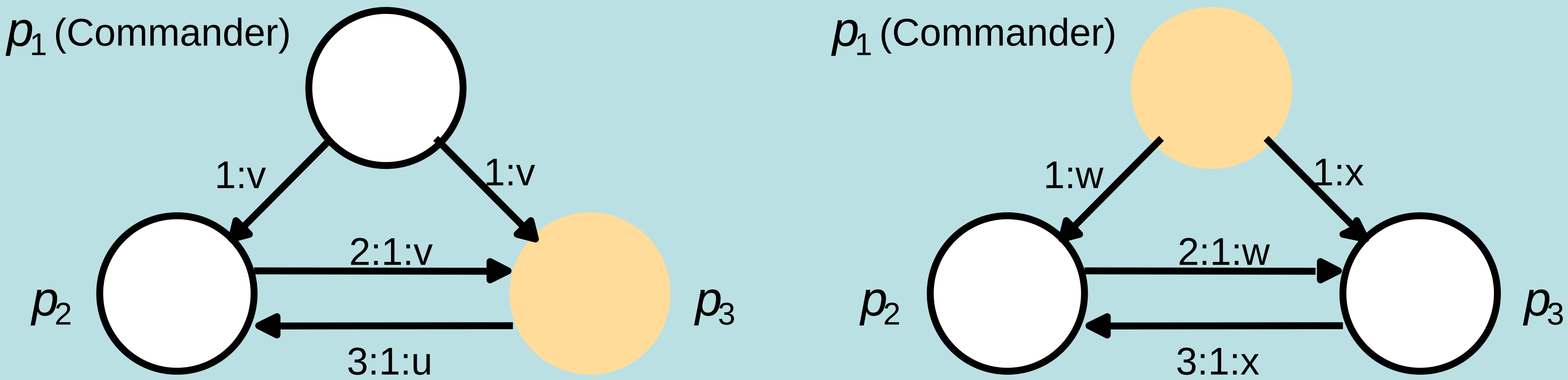
Assign $d_i = minimum(Values_i^{f+1});$

Figure 12.18 Consensus in a synchronous system

Limits for solutions to Byzantine Generals

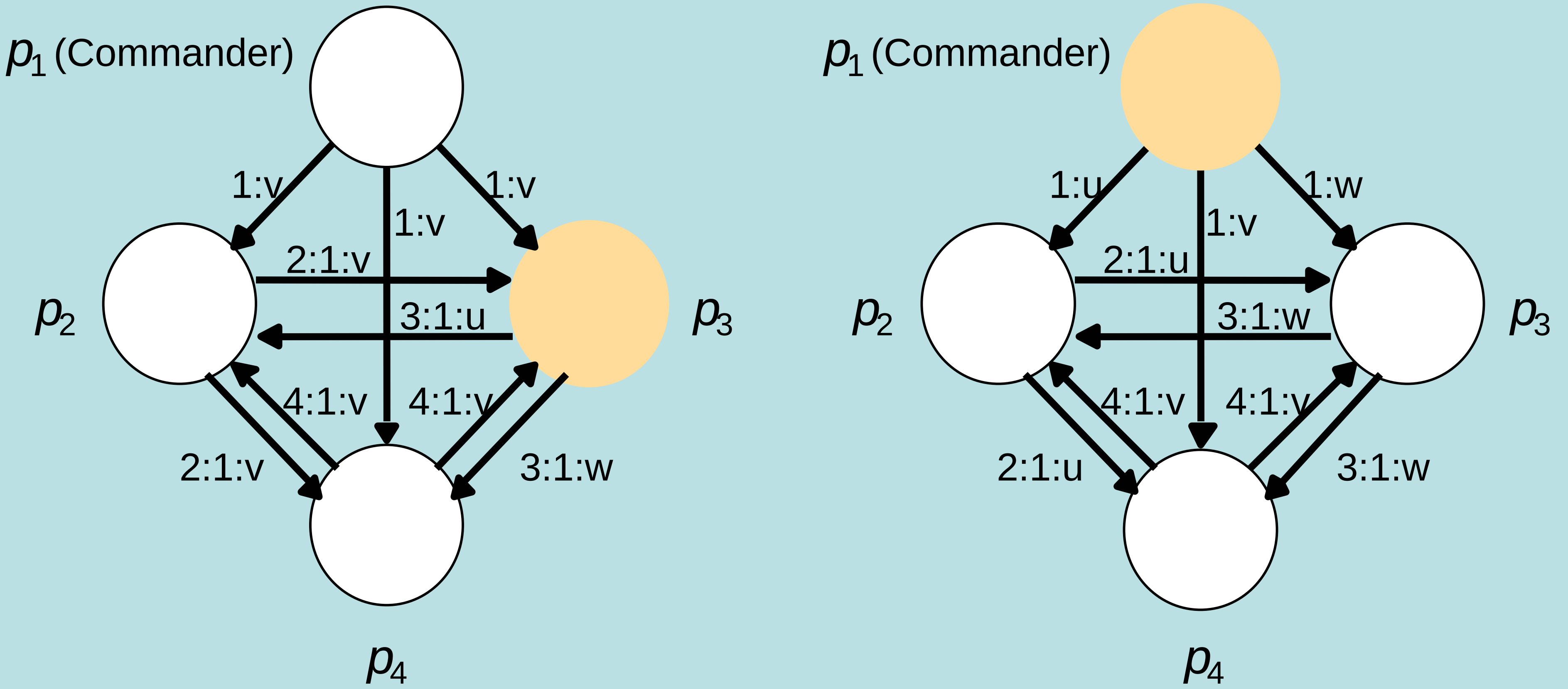
- Some cases of the Byzantine Generals problems have no solutions
  - Lamport *et al* found that if there are only 3 processes, there is no solution
  - Pease *et al* found that if the total number of processes is less than three times the number of failures plus one, there is no solution
- Thus there is a solution with 4 processes and 1 failure, if there are two rounds
  - In the first, the commander sends the values
  - while in the second, each lieutenant sends the values it received

Figure 12.19 Three Byzantine generals



Faulty processes are shown coloured

Figure 12.20 Four Byzantine generals



Faulty processes are shown coloured

Asynchronous Systems

- All solutions to consistency and Byzantine generals problems are limited to synchronous systems
- Fischer *et al* found that there are no solutions in an asynchronous system with even one failure
- This impossibility is circumvented by *masking faults* or using *failure detection*
- There is also a partial solution, assuming an *adversary* process, based on *introducing random values* in the process to prevent an effective thwarting strategy. This does not always reach consensus