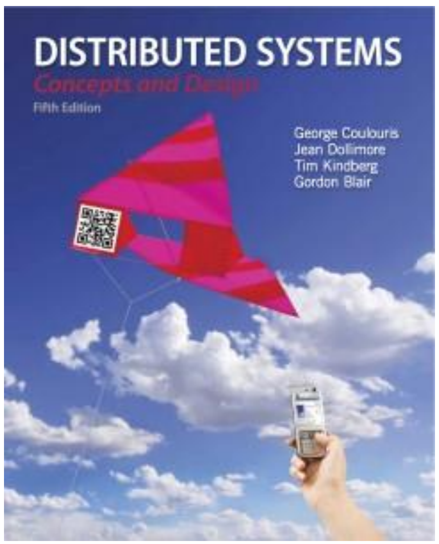
Coordination and Agreement (II)



From **Coulouris, Dollimore, Kindberg and Blair**

Distributed Systems: Concepts and Design, 5e

Election algorithms

- Algorithms for **choosing a unique process** to play **a particular role**
  - E.g. choose mutual exclusion central server from among the processes that need to use the critical section
  - Essential that all processes agree on the choice
  - If elected process retires, another election is required

- Two or more processes can call **concurrent elections**
  - **Choice** of elected process must be **unique**, even if several processes call elections **concurrently**

- At any point in time, a process p_i is either
  - A participant – is **engaged** in some run of the election algorithm
  - A non-participant – **not engaged** in any election
  - has a variable $elected_i$, which will contain the identifier of the **elected process**

- (Without loss of generality) The **elected** process chosen be the one with **the largest 'identifier'**
  - ID may be **any useful value**, as long as IDs are **unique** and **totally ordered**
  - E.g. could elect the process with the **lowest computational load** by having each process use $<1/load,i>$ as its identifier, where *load* > 0 and the index *i* is used to **order identifiers with the same load**

Requirements for leader election algorithms (cont.)

- During any particular run of the algorithm:
  - E1 (safety)
    A process p_i has $elected_i$ = φ 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 $elected_i \neq$ φ – or crash

- Note that there may be processes p_j that are **not yet participants**, which record in $elected_j$ the identifier of **the previously elected process**

- The **performance** of an election algorithm:
  - Total **network bandwidth** utilisation (total messages sent)
  - **Turnaround time** – number of serialised message transmission times between **initiation** and **termination** of a **single run**

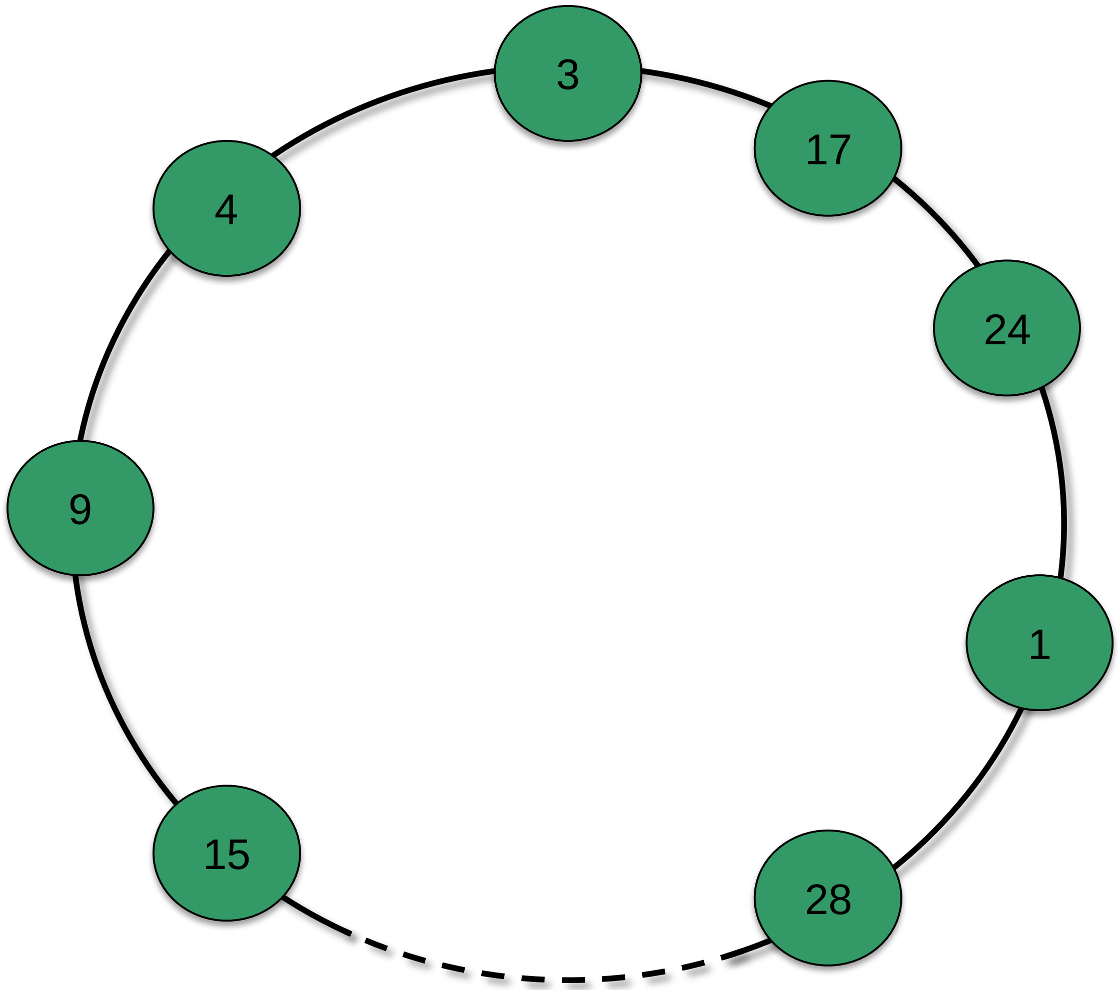
Chang and Roberts algorithm (ring-based election)

- Processes are arranged in a **logical ring**
- Each process p_i has a communication channel to the next process in the ring, $p_{(i+1) mod N}$
- All messages sent **clockwise** around the ring
- Assume **no failures** occur and system is **asynchronous**
- Goal of the algorithm: to elect the **coordinator**, the process with the **largest identifier**
- **Initially**, each process is a **non-participant** in the election
- Any process can **initiate** an election
  - **Marks** itself as a **participant**
  - Places its **identifier** in an *election* message
  - **Sends** the message to its **clockwise neighbour**

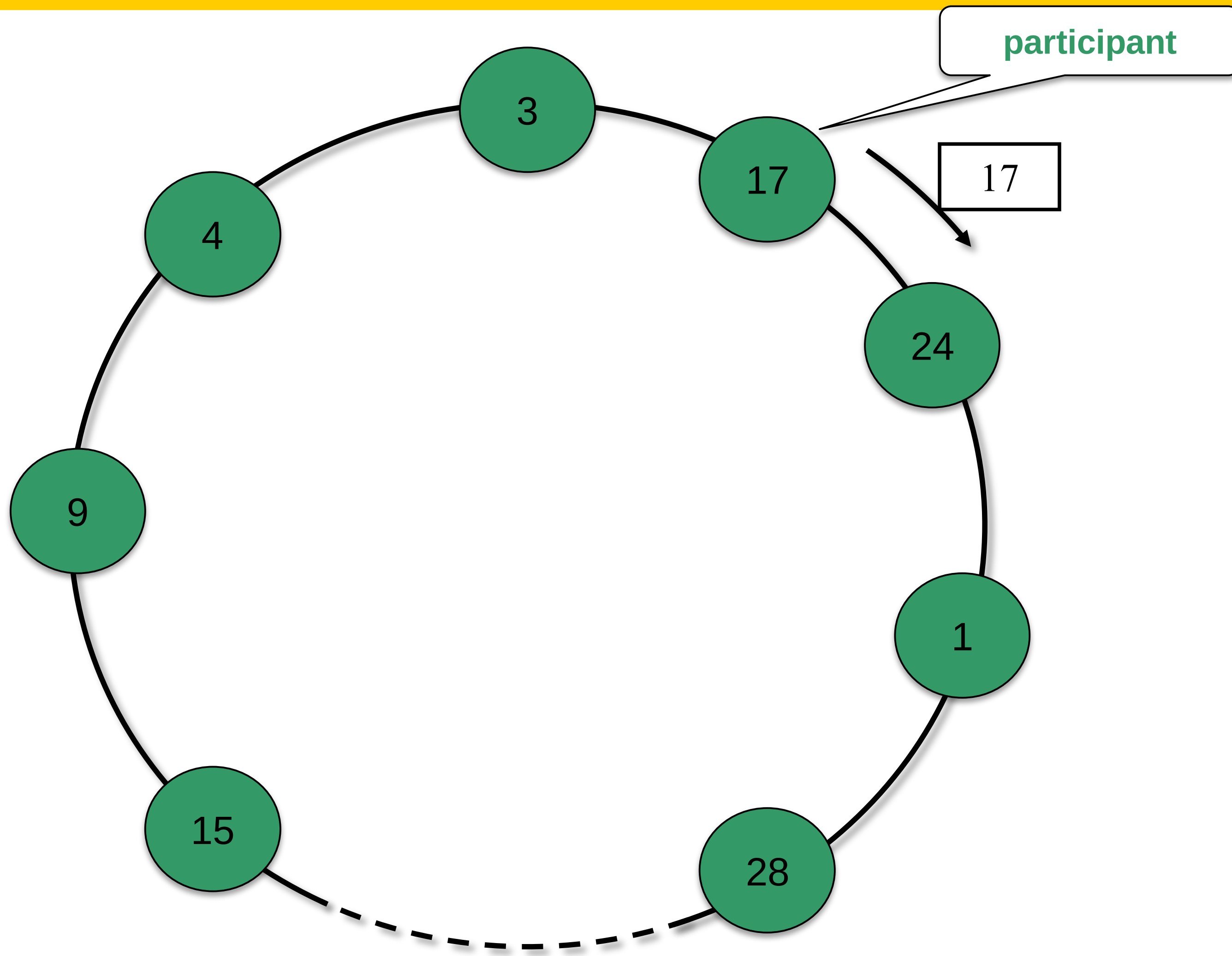
Chang and Roberts algorithm (cont.)

- On reception of an **election** message with enclosed ID p_i by a process p_k
  - If p_k non-participant, forward $max(p_i, p_k)$; mark itself as participant
  - If p_k participant, forward p_i only if $p_i > p_k$ otherwise do nothing
- If ($p_i == p_k$)
  - p_k becomes **coordinator**
  - Mark itself **non-participant**; send **elected** message with **its ID** to neighbour
- On reception of an elected message with enclosed ID p_i by a process p_k
  - Mark itself as **non-participant**
  - Set $elected_k = p_i$
  - If (p_k != coordinator)
    - Forward message to neighbour

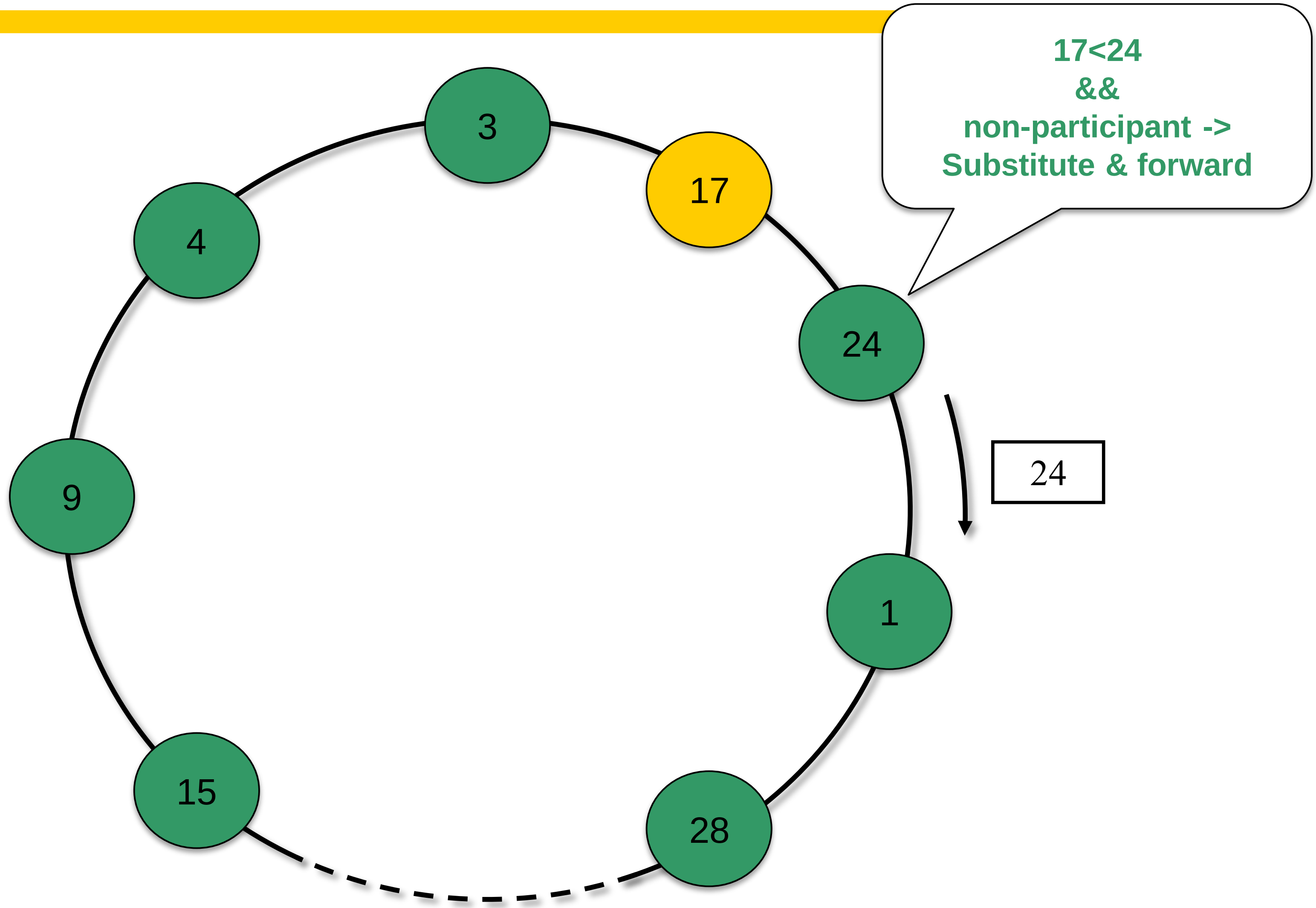
Ring-based Election in Action



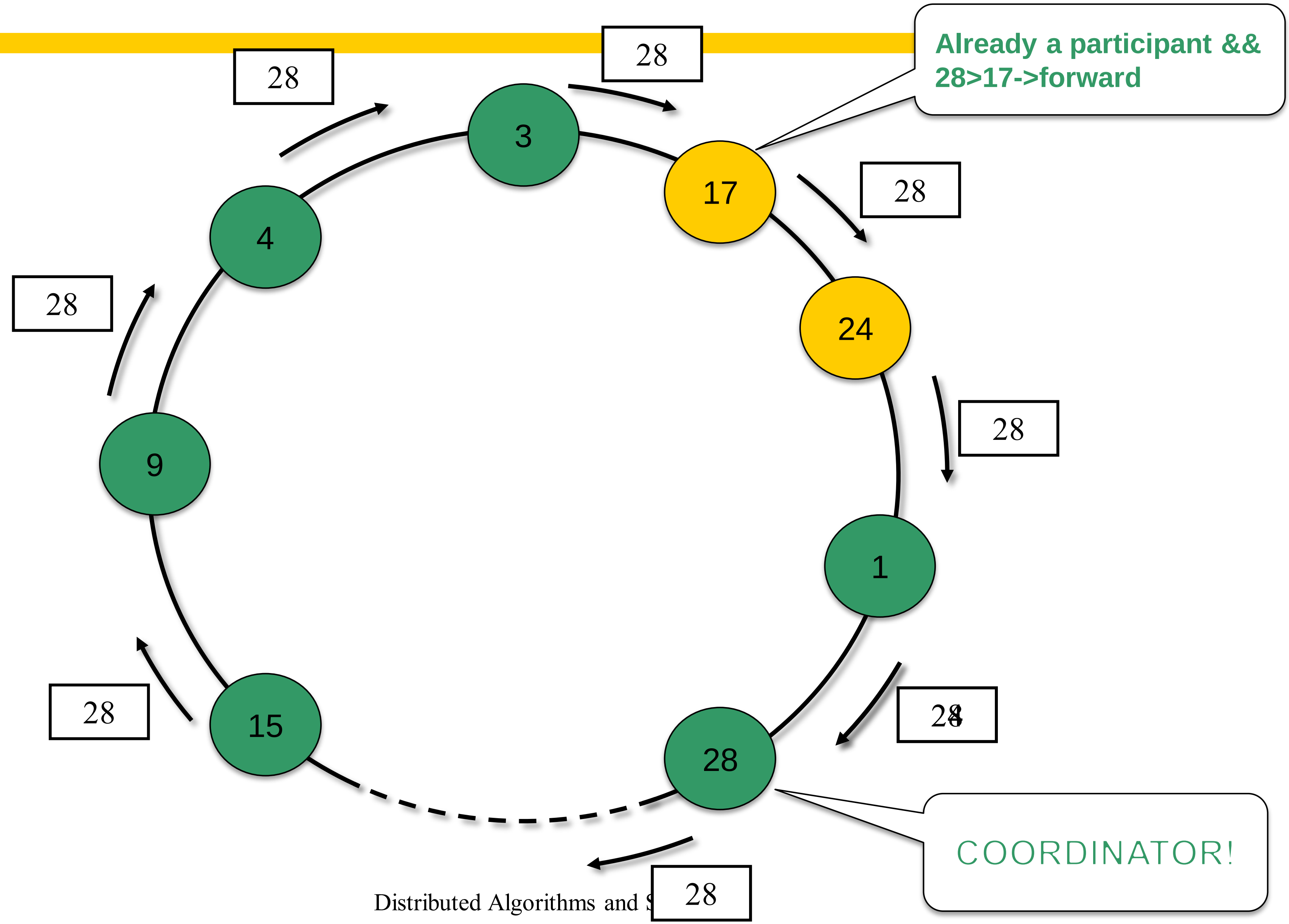
Ring-based Election in Action



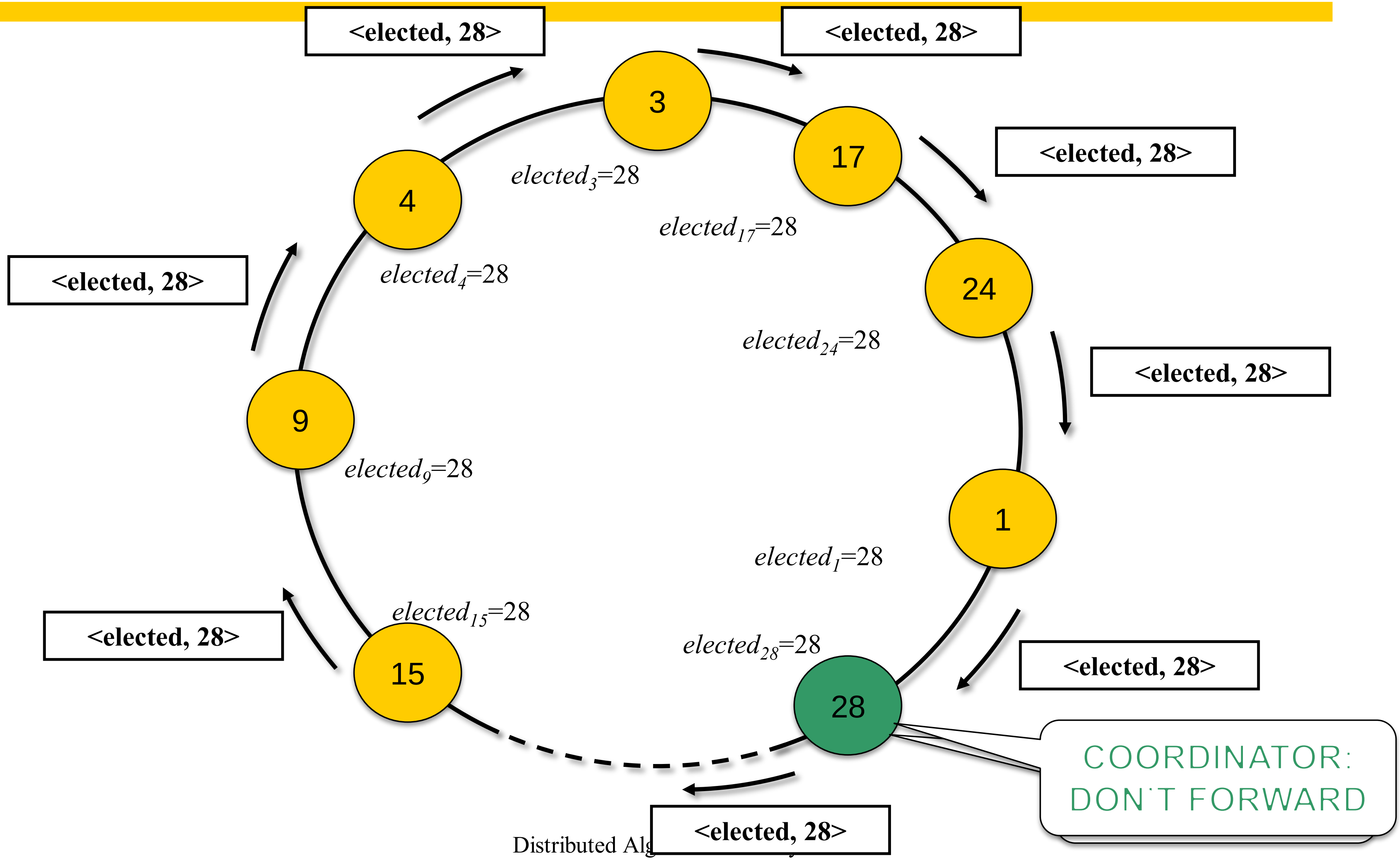
Ring-based Election in Action



Ring-based Election in Action



Ring-based Election in Action



Performance of ring-based election algorithm

- E1 is met
  - All identifiers are compared; a process must receive its **own identifier back** before sending an *elected* message
  - For any two processes, the one with the **larger** identifier **will not pass on** the other's identifier $\Rightarrow$ impossible that both should receive their own identifiers back

- E2 follows immediately from the **guaranteed traversals** of the ring (**no failures**)
  - **non-participant** and **participant** states used so that messages arising when another process starts a **concurrent election** are **extinguished** as soon as possible, before the winning election result has been announced

Performance of ring-based election algorithm (cont.)

- If only a single process starts an election, worst case is when **anti-clockwise neighbour** has the **highest identifier**
  - *N-1* messages needed to reach this neighbour
  - *N* messages required for this neighbour to receive its own identifier
  - *elected* message is sent N times
- Worst case bandwidth utilisation – *3N-1* messages
- Worst case turnaround time – O(3N-1)

The bully algorithm

- Can we let processes **crash** during the election?
- We can if we assume that the system is **synchronous**, so we can use **timeouts** to **detect** process failure
- Bully algorithm assumes that each process **knows** which processes have **higher identifiers**, and **can communicate** with **each** such process
- **Three** types of messages in this algorithm
  - *election* – **announces** an election
  - *answer* – sent in **response** to an *election* message
  - *coordinator* – **announces** identity of the **elected** process
- A process initiates an election when it **notices**, through **timeouts**, that the current **coordinator has failed** – several processes may discover this **concurrently**

The bully algorithm (cont.)

- **Synchronous** system assumptions enables construction of a **reliable failure detector**
  - T_{trans} is maximum transmission delay
  - $T_{process}$ is maximum processing delay in a process
  - $T = 2\, T_{trans} + T_{process}$ is an **upper bound** on the **total elapsed time** from sending a message to another process to receiving a response
  - If **no response** arrives within time **T**, then local failure detector can report that the intended recipient of the request has **failed**

The bully algorithm (cont.)

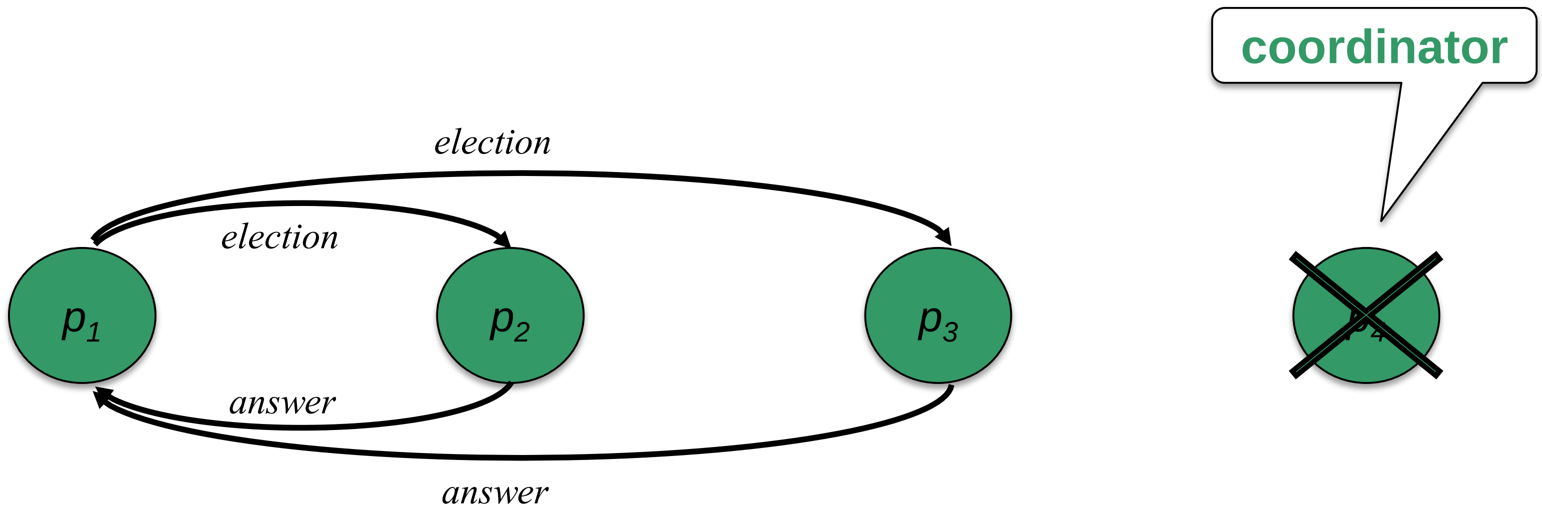
- Process that **knows** it has the **highest identifier** can **elect itself** as the coordinator by sending a *coordinator* message to all processes with smaller identifiers

- Process with **lower** identifier
  - Sends an *election* message to processes **with higher identifiers** and **awaits** an *answer* message in response
  - if none arrives within time *T*, process considers **itself the coordinator** and sends a *coordinator* message to all **processes** with **lower identifiers**
  - else waits *T'* for a *coordinator* message to **arrive** from the new coordinator – if none arrives, it calls (begins) another election

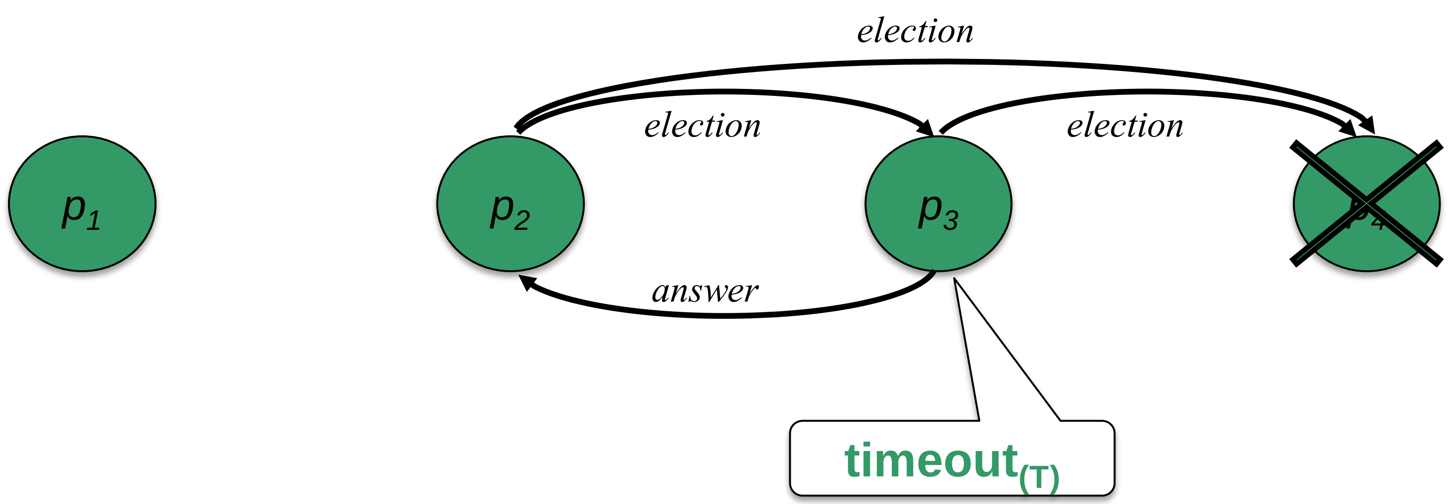
The bully algorithm (cont.)

- If p_i receives a *coordinator* message, sets $elected_i$ to the identifier of the coordinator contained in the message
- If a process receives an *election* message, it sends back an *answer* message and **begins another election**, unless it has begun one already
- When a process is **started to replace a crashed process**, it **begins** an election
  - In particular, if it has the **highest identifier**, then it will **announce itself** as the new coordinator, even though the current coordinator is functioning
  - i.e. it "bullies" its way into being the coordinator

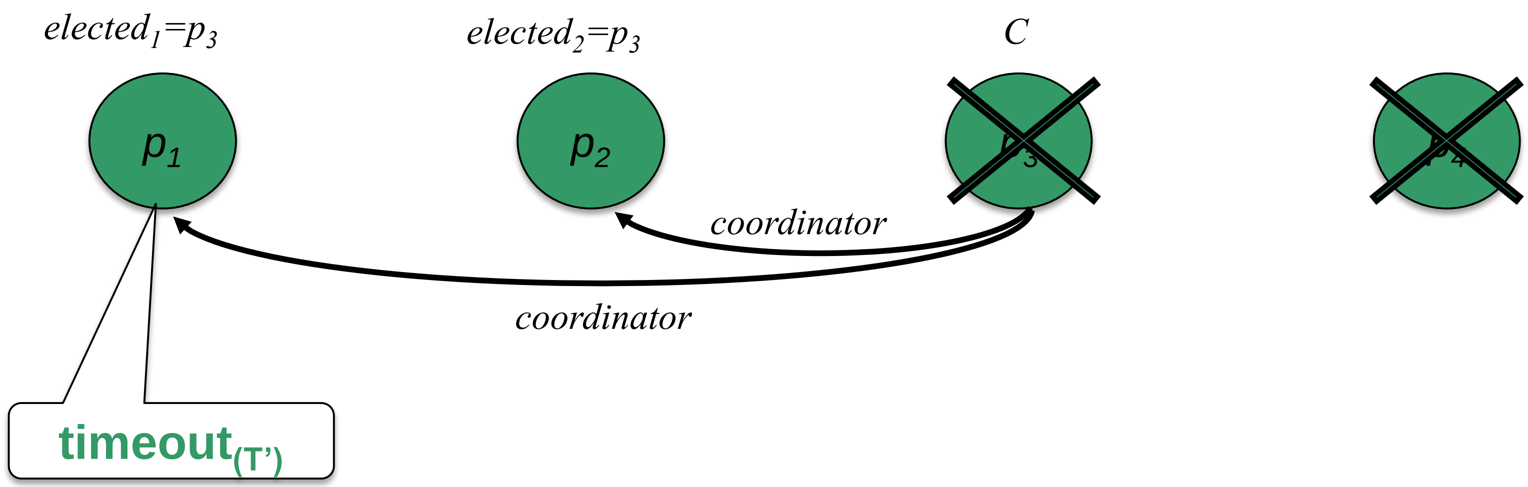
"Bully" election in action



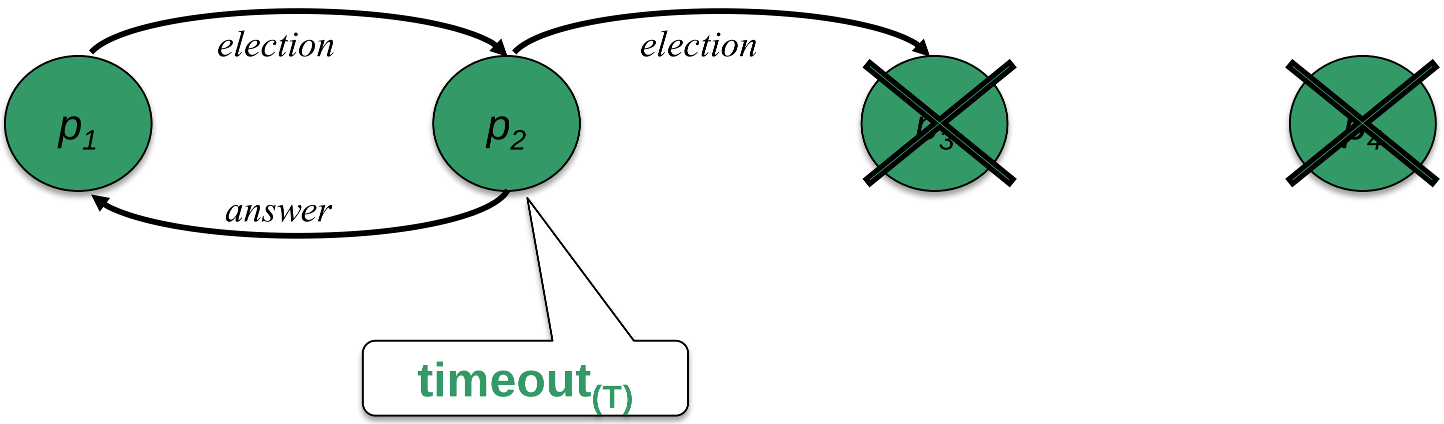
“Bully” election in action



“Bully” election in action - Process failure during election



“Bully” election in action



"Bully" election in action

Performance of bully algorithm

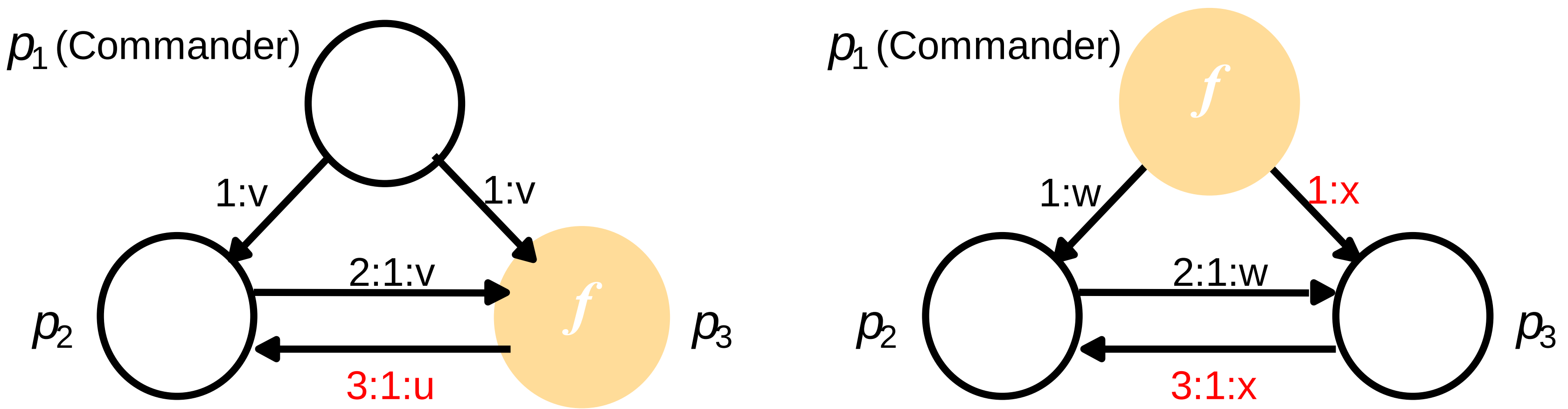
- E2 (liveness) is met by the assumption of **reliable message delivery**
- E1 (safety) is met under the assumption that **no process is replaced**
- E1 (safety) is **not met** if processes that **crash** are **replaced** by processes with the **same identifier**
  - Multiple processes may announce themselves coordinators concurrently
- E1 may also be broken if the assumed **timeout** values are **inaccurate** – i.e. the failure detector is unreliable
- Performance
  - Best case – **second highest identifier** notices coordinator's failure, declares itself coordinator, sends **N-2 messages** to others; turnaround time is 1 message transmission time
  - Worst case – **lowest identifier** notices coordinator's failure, since **N-1** processes **begin** elections, $O(N^2)$ messages; approx. N+1 transmission times

Consensus

- Processes need to **agree on a value** after one or more of the processes have proposed what that value should be
- Important to be able to do so even in the **presence of faults** (crashed processes)

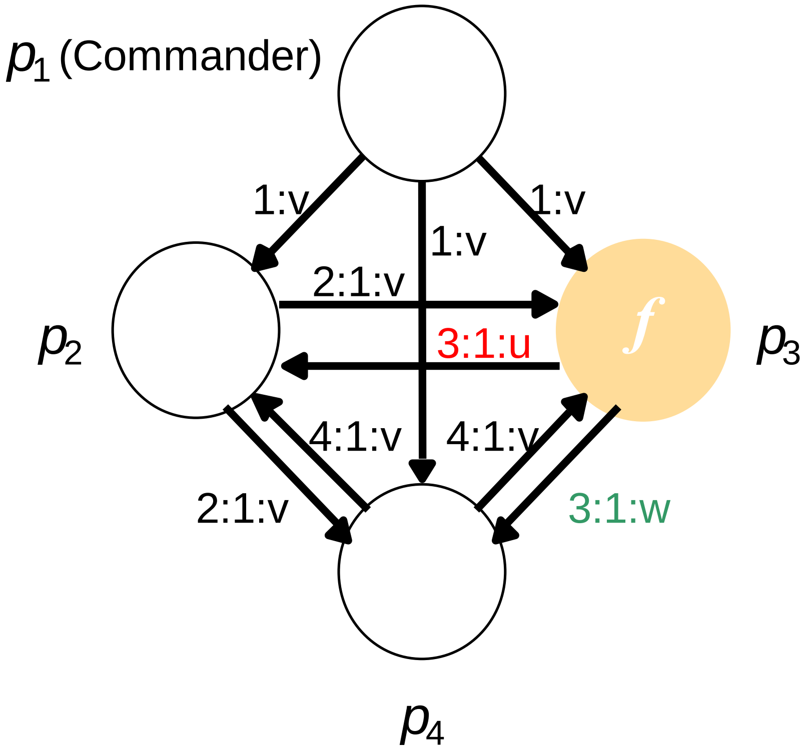
- Definition of the consensus problem:
  - every process p_i begins in the ***undecided*** **state** and proposes a single value, v_i, drawn from a set of values
  - the processes communicate with one another, exchanging values
  - each process then sets the value of a *decision variable*, d_i, and enters the ***decided*** **state**, after which it can **no longer change** d_i
- Requirements of every execution of a consensus algorithm:
  - **Termination** – eventually each correct process sets its decision variable
  - **Agreement** – the decision value of all correct processes is the same
  - **Integrity** – if the correct processes all proposed the same value, then any correct process in the *decided* state has chosen that value
- Choosing the value for the decision variable
  - majority (of values offered) / minimum or maximum (of values offered)

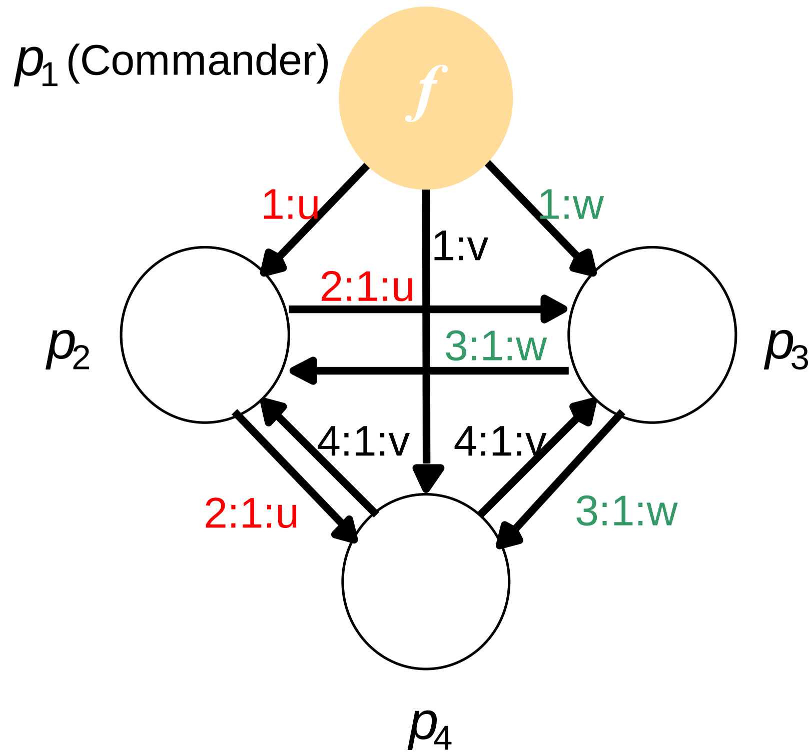
3 Byzantine Generals (N ≤ 3*f*) - Impossibility



- Left: p_2 receives differing values (cannot tell correct)
- Right: commander sends differing values
  - p_2 in the same situation (received differing values)
- Assuming there exist solution, p_2 needs to decide
  - Say, choose value from commander
  - p_3 will do the same and decide, BUT contradicts agreement condition!

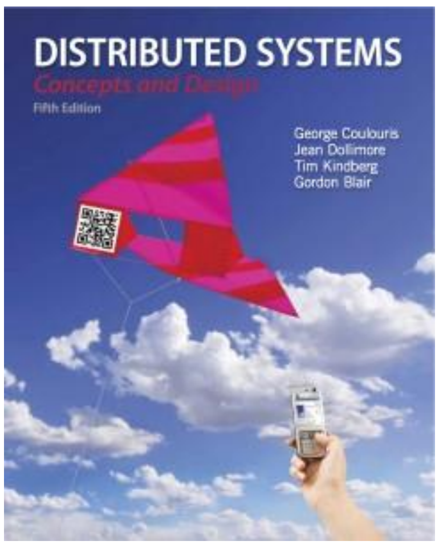
4 Byzantine Generals (N ≥ 3f + 1)





- If one faulty process, all receive N-2 correct values
  - If N ≥ 4, N – 2 ≥ 2 ⇒ can apply majority function
  - Can use special value (e.g., φ) to denote absence of majority

Coordination and Agreement (III)



From **Coulouris, Dollimore, Kindberg and Blair**
Distributed Systems: Concepts and Design, 5e

Multicast communication basics

- A process issues only **one** *multicast* operation to send a message to **each** of a **group** of processes
  - Instead of issuing multiple send operations to individual processes
  - Communication to **all** processes in the system is known as a *broadcast*
- Use of a single *multicast* operation instead of multiple send operations enables efficiency and delivery guarantees
- Efficiency
  - Efficient **bandwidth utilisation**; can send a message **no more than once** over a communication link by using distribution trees
  - Can use **network hardware** to support multicast where available
  - Can **minimize total time taken** to deliver a message to all of its destinations, **instead** of transmitting it **separately** and **serially**
- Delivery guarantees
  - Can **guarantee delivery order** to all members of the group
  - **Impossible** to do so if process issues multiple **independent** *send* operations, e.g. if sender fails halfway through sending

Multicast system model

- System consists of a collection of processes
  - They can communicate reliably over 1-1 channels
  - They can fail only by crashing
- Processes are members of groups
  - **Groups** are the **destinations** of messages sent using the *multicast* operation
  - A **process** can be a **member of multiple groups** simultaneously; we will restrict membership to a single group
- *multicast(g, m)* sends message *m* to all members of the group *g*
- *deliver(m)* delivers a message sent by multicast to the calling process
  - A multicast message not always handed to the application layer inside the process as soon as it's received by process's node (use deliver rather than receive)
- **Every message** m carries the **unique identifier** of the process *sender(m)* that sent it, and the **unique destination group identifier** *group(m)*

Basic multicast

- IP multicast provides no guarantees
- We define a Basic multicast (*B-multicast*) primitive
  - **Guarantees** that a **correct process** will **eventually deliver** a message as long as the multicasting process does not crash
  - *B-deliver* the corresponding basic delivery primitive
- Straightforward implementation by using a **reliable one-to-one** *send* operation
  - To *B-multicast(g, m)*: for each process $p \in g$, *send(p, m)*
  - On *receive(m)* at *p*: *B-deliver(m)* at *p*
- This implementation may use **threads** to perform *send* operations concurrently
  - **Reduce the total time** taken to deliver a message
- Liable to *ack-implosion* if the number of processes are large
  - Acks arriving at the same time may **fill the buffers** of sender, resulting in **dropping** acks
  - **Retransmit** messages, get **even more** acks, waste network bandwidth further

Reliable multicast

- Reliable multicast properties:
  - Integrity (analogous to that of reliable one-to-one communication)
    - A **correct process** *p* delivers a message *m* **at most once**
    - $p \in group(m)$ and *m* was supplied to a multicast operation by *sender(m)*
  - Validity (guarantees liveness for the sender) – if a **correct** process **multicasts** message *m* then it will eventually **deliver** *m* (self-delivery)
  - Agreement – if a correct process **delivers** message *m*, then **all other correct processes** in *group(m)* will eventually deliver *m*

- Validity property is with respect to the sender – note that validity + agreement together provide an overall liveness requirement

- The agreement property implies “all or nothing”

Reliable multicast algorithm

On initialization

$\quad Received := \{\};$

For process p to R-multicast message m to group g

$\quad B\text{-}multicast(g, m);$ // $p \in g$ is included as a destination

On B-deliver(m) at process q with g = group(m)

$\quad if\ (m \notin Received)$

$\quad then$

$\qquad Received := Received \cup \{m\};$

$\qquad if\ (q \neq p)\ then\ B\text{-}multicast(g, m);\ end\ if$

$\qquad R\text{-}deliver\ m;$

$\quad end\ if$

Correct but inefficient algorithm: each message is sent |g| times to each process

Reliable multicast over IP multicast

- Use IP multicast to **efficiently distribute** messages to group members
- Use **positive** and **negative** acknowledgements
- Piggy-back acknowledgements on other messages sent to the group
  - Processes do not send separate acknowledgments
  - Send separate response **only** when processes detect they have missed a message (negative ack)
- Each process *p* maintains a **sequence number** S_g^p **for each group** *g* to which it belongs, initially has a value of 0
- Also records R_g^q, the **sequence number** of the **latest message** it has **delivered** from process *q* that was sent to group *g*

Reliable multicast over IP multicast (cont.)

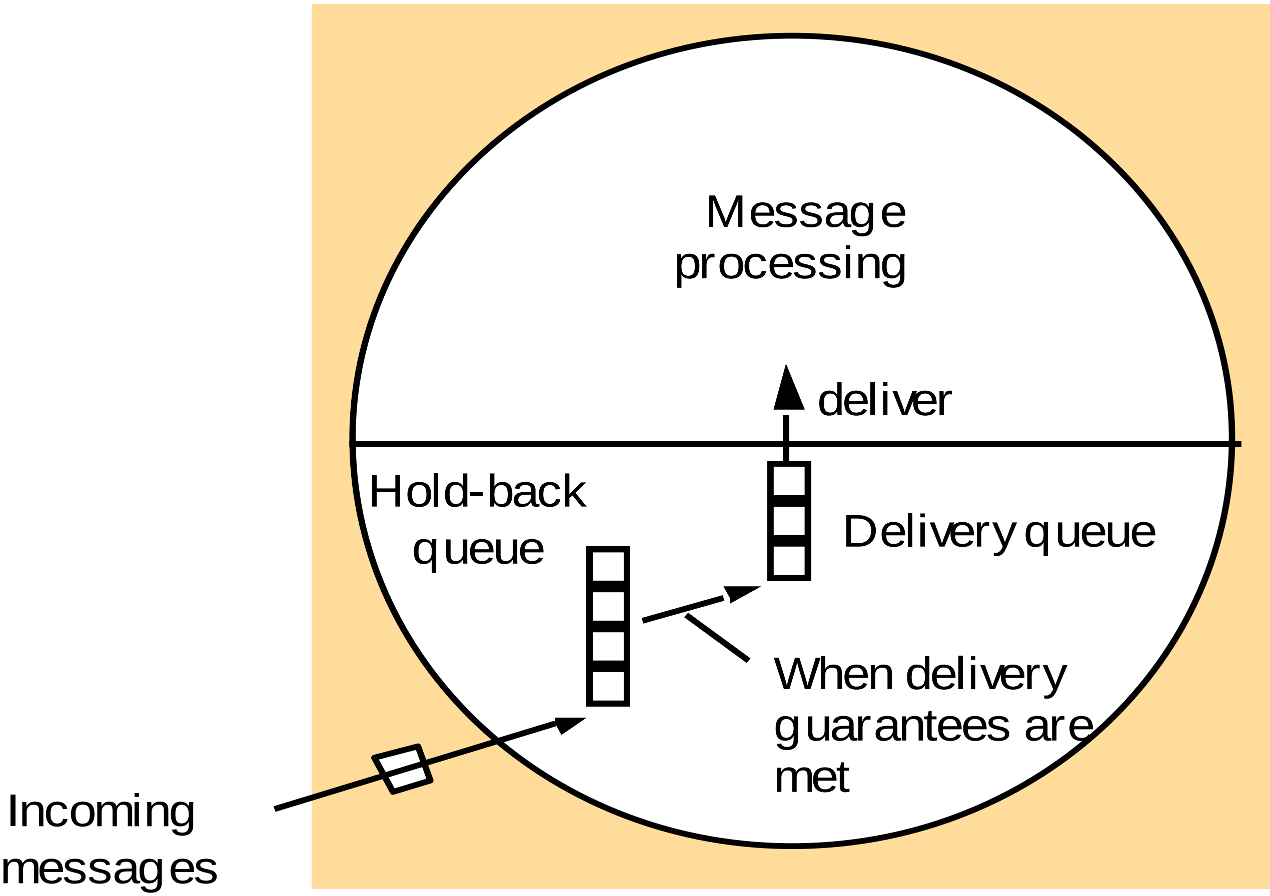
- For p to *R-multicast* a message to group g
  - **Piggybacks** the value of S_g^p and **acknowledgements** of the form
  - <q, R_g^q > onto the actual message
  - **IP-multicasts** the entire message to g and **increments** S_g^p by one
  - Acknowledgements enable processes to **learn about messages** that they have not received

- A process *R-delivers* a message destined for g bearing the sequence number S from p **iff** $S = R_g^p + 1$, and it **increments** R_g^p immediately after delivery

- If $S \leq R_g^p$, then the process **has delivered** the message before and **discards** it

- If $S > R_g^p+1$, then there are one or more messages that it has not yet received
  - Places the message in a **hold back queue**
  - Sends **negative acknowledgements** to obtain the missing messages

The hold-back queue for arriving multicast messages



- *Integrity* follows from **detection of duplicates** and underlying properties of IP multicast (checksums)
- Validity follows from IP multicast
- Agreement requires **always detecting** missing messages
  - Need infinite messages
  - Copies of all messages always available (at sender)

Ordered multicast

- Basic multicast algorithm delivers messages to processes in **arbitrary orders** due to **arbitrary delays** in the underlying one-to-one send operations

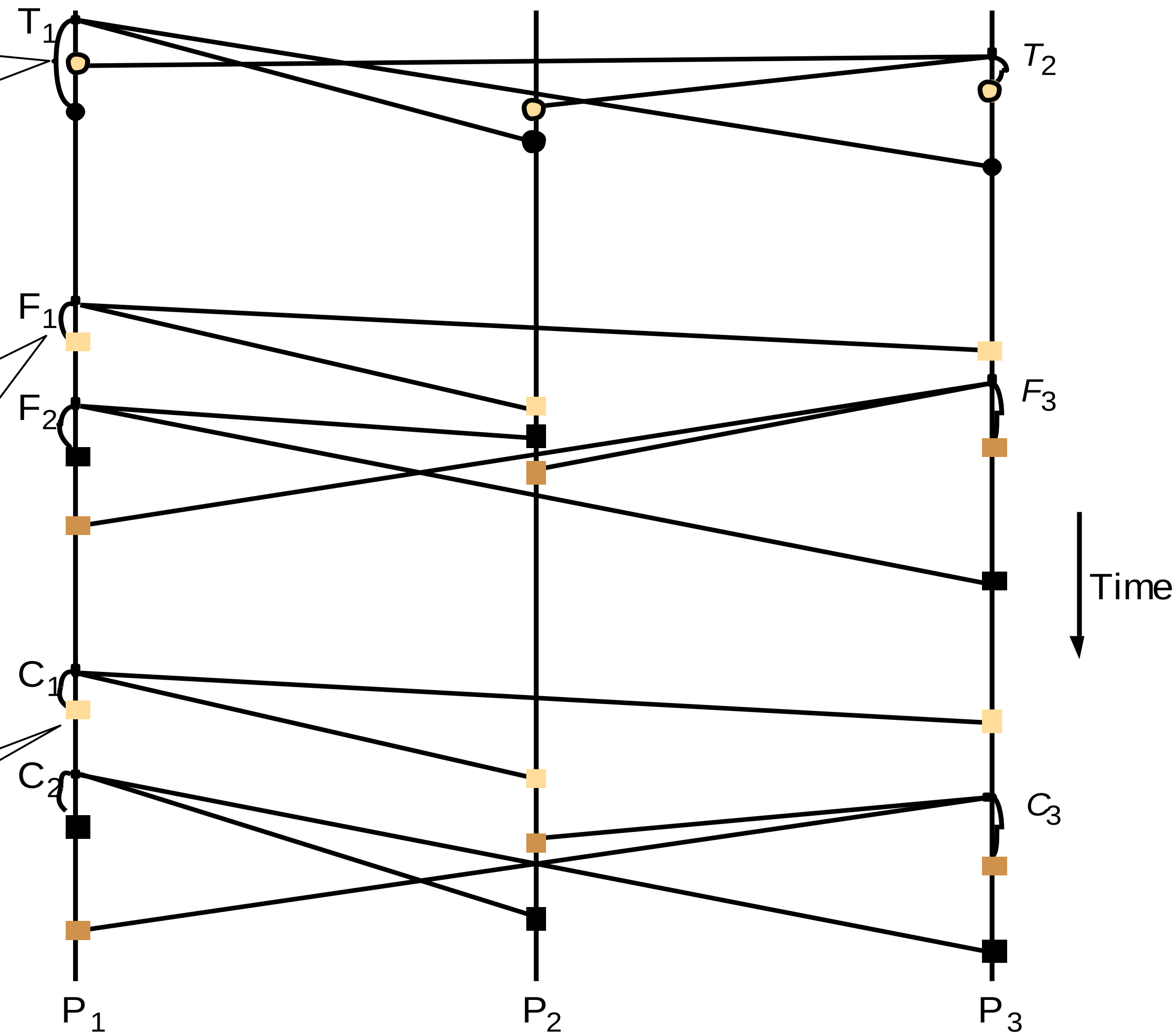
- Three typical types of ordering guarantees are often considered
  - FIFO – if a correct process **issues** *multicast(g, m)* **before** it issues *multicast(g, m')*, then every correct process that delivers *m'* will **deliver** *m* **before** *m'*
  - Causal – if ***multicast(g, m)*** → ***multicast(g, m')***, where → is the happened-before relation, then any correct process that delivers *m'* will **deliver** *m* **before** *m'*
  - Total – if a correct process **delivers** message *m* **before** it delivers *m'*, then any other correct process that delivers *m'* will deliver *m* before *m'*

Total, FIFO and causal ordering of multicast messages

Total ordering: Message T_2 delivered before T_1 by all processes

FIFO ordering: Message F_1 delivered before F_2 (issued by the same process) by all processes; Message F_3 (issued by different process) delivered with arbitrary ordering

Causal ordering: Message $C_1 \rightarrow C_3$; delivered first by all processes



Implementing FIFO ordering

- Achieved with **sequence numbers**, much as we would achieve it for one-to-one communication
  - Reliable multicast implements FIFO ordering
- In general, for multicast group *g*, process *p* maintains:
  - S_g^p, a count of **how many messages** *p* has sent to *g*
  - R_g^q, the sequence number of the **latest message** *p* has delivered from process *q* that was sent to group *g*
- For *p* to FO-multicast a message to group g, it **piggybacks** S_g^p onto the message, B-multicasts it to *g*, and increments S_g^p
- **Upon receipt** of a message from *q* bearing sequence number S, checks whether $S = R_g^q + 1$;
  - If true (the message is the next one expected), FO-delivers it, setting $R_g^q = S$
  - If false, places in hold-back queue until the intervening messages have been delivered

Implementing total ordering

- Assign **totally ordered identifiers** to multicast messages so that **each process** makes the **same ordering decision** based upon these identifiers

- Delivery algorithm very similar to FIFO ordering
  - Difference is that processes keep **group-specific** instead of process-specific **sequence numbers**

- Possible approach is to use a **sequencer process** to **assign** identifiers
  - Process wishing to TO-multicast a message *m* to group *g* attaches a **unique identifier** *id(m)* to it
  - Message is sent to **sequencer** as well as to the members of *g*
  - The sequencer maintains a **group-specific sequence number** which it uses to assign **increasing** and **consecutive sequence numbers** to the messages received
    - It then *B-multicasts* an **order message** to the members of the group

Total ordering using a sequencer

1. Algorithm for group member *p*

On initialization: $r_g := 0$;

Sequence number of the latest message *p* has delivered that was sent to *g*

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

Unique identifier

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$(g, <\text{“order”}, i, s_g>)$;
    $s_g := s_g + 1$;

Implementing causal ordering

- In essence, multicast **message delivery order** conforms to the **happened-before relation**
  - Does not take into account one-to-one messages between processes (only multicast)

- Each process p_i *(i=1, 2, …, N)* maintains its **own vector timestamp**
  - Entries in timestamp count the **number of multicast messages from each process** that **happened-before** the **next** message to be multicast

- To *CO-multicast* a message to group *g*
  - Process adds 1 to **its entry in the timestamp**; then *B-multicasts* the message along with the timestamp to *g*
- When a process p_i *B-delivers* a message from p_j
  - Must place it in the **hold-back queue** before it can *CO-deliver* it
  - i.e. until it is assured that it **has delivered** any messages that **causally preceded it**
  - To establish this, it **waits until** *(a)* has **delivered earlier messages** sent by p_j and *(b)* it has **delivered** any message that $\boldsymbol{p_j}$ **had delivered at the time** it multicast the message

Causal ordering using vector timestamps

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

$V_i^g[j] := V_i^g[j] + 1$;

p_i has delivered earlier messages sent by p_j

p_i has delivered any message that p_j had delivered at the time it multicast the message

p_i updates its own vector timestamp (j^{th} entry) upon delivering a message from p_j

Summary

- Distributed mutual exclusion
  - Locks not always implemented by servers that manage shared resources
  - Considered three algorithms: **central server**, **ring-based**, **multicast-based** using logical clocks
  - They can be modified to tolerate some faults
- Algorithms to uniquely **elect** a process from a given set
  - E.g. elect a new master time server
  - **Ring-based**, **bully** algorithms
- Consensus algorithms to survive *f* faults
  - Through a voting function, e.g., majority, min-max, etc.
- Multicast communication
  - **Reliable multicast**: processes agree on the set of messages to be delivered
  - **FIFO**, **causal**, **total** delivery ordering