Distributed Systems

Aleardo Manacero Jr.

UNIVERSITY OF OREGON

Coordination and Agreement

Introduction

- The previous discussion over clocks is aimed to solve problems related to processes coordination
- Processes coordination may be understood as the distributed equivalent to processes synchronization in centralized systems
- Therefore, before discussing coordination it is necessary to discuss mutual exclusion and elections

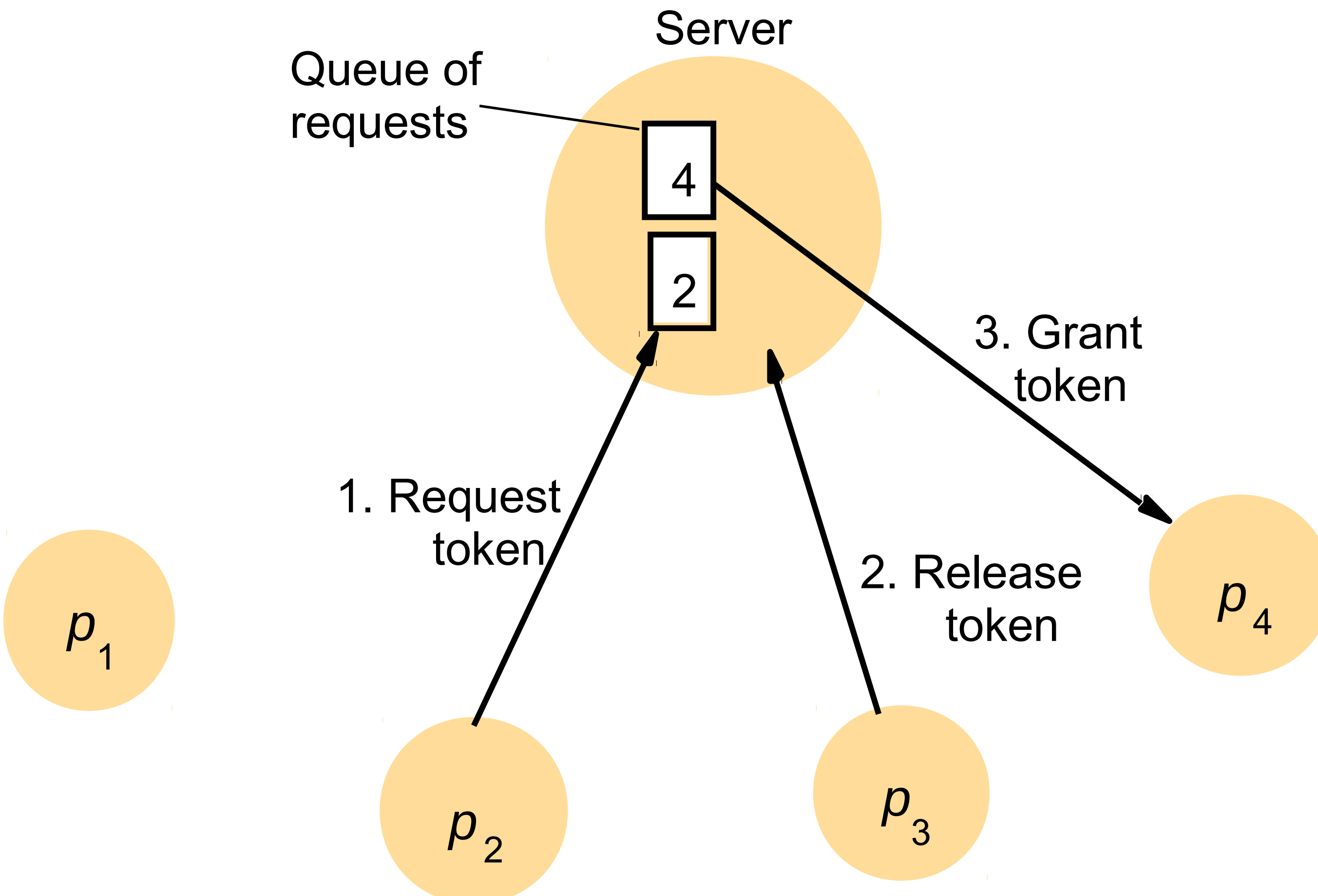
Distributed Mutual Exclusion

- Mutual exclusion is harder to accomplish in distributed systems by the existence of multiple processors and the lack of shared memory
- Solutions for distributed mutual exclusion may use centralized or distributed approaches
- Centralized approaches suffer, obviously, from the classical problems of concentrating control in a single place

Centralized algorithm

- Uses a central server to control critical sections over shared resources
- Processes send requests to the central server, which attend the requests based on a FIFO queue
- After using a resource the process sends a release message to the server, which will then assign the resource to the next process in queue

Centralized algorithm



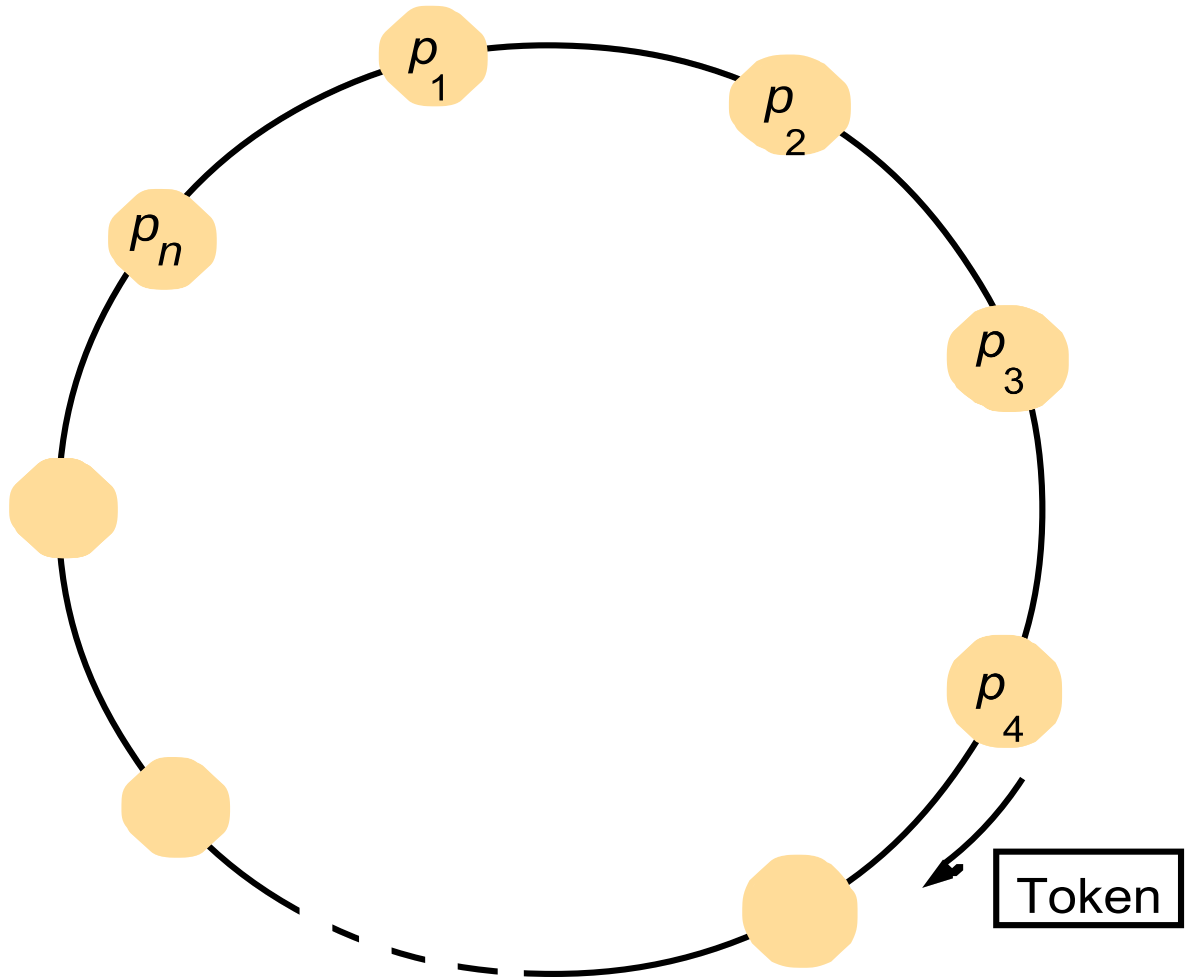
Distributed algorithms

- Several algorithms have been proposed to solve distributed mutual exclusion in a distributed approach
- We'll see two approaches here:
  - Token Ring algorithm
  - Ricart-Agrawala's algorithm
  - Maekawa's algorithm

Token ring

- The idea is identical to token bus algorithm for medium access control
- The advantage is the absence of single point of failure
- The disadvantage is that it continuously consumes bandwidth and processing time, except when a process is in its critical section

Token ring



Ricart-Agrawala

- This algorithm preserves bandwidth and processing while no process want to enter the critical section (CS)
- It uses logical clocks to establish the order of requisitions to access the CS
- When a process wants to enter the CS it sends messages to all other processes and waits until receiving replies from all of them

Ricart-Agrawala

- Processes give the reply if they do not hold the CS and, in case they want to enter it, their clock are lower than the clock of the requesting process
- The full algorithm follows...

Ricart-Agrawala

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;

Ricart-Agrawala

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;

Maekawa's algorithm

- Maekawa reduces the amount of messagens exchanged between processes in R-A algorithm by creating overlapping subsets of processes
- The idea is to define a voting set Vi, associated with process Pi, where Vi is composed by a subset of processes containing Pi and some other processes

Maekawa's algorithm

- The sets are chosen so that:
  - $P_i \in V_i$
  - $V_i \cap V_j \neq \emptyset$
  - $|V_i| = K$
  - Each process is in M of voting sets
- For the optimal solution for N processes we have N approximately equal to K^2 and M=K
- The complete algorithm is shown next

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 (# 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

Maekawa's algorithm

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

Election algorithms

- A major problem in DS is the possibility of server crashes
- When a process in charge of any activity fails (crashes) the service becomes unavailable
- This unavailability may be overcome through the replacement of the faulty process
- This replacement can be done automatically through an election mechanism

Election algorithms

- An election algorithm must be activated by a process that identified the faulty server
- Therefore, it may happen to have several elections being conducted at the same time
- A correct election algorithm must assure that if more than one election is under way, all must result in the same elected process
- To simplify the election it is defined that the elected process is the one with highest identity value

Election requirements

- An election algorithm must guarantee that:
  - A participant Pi has $E_i = \perp$ or $E_i = P$ (safety)
  - All processes Pi participate and eventually either set $E_i \neq \perp$ or crash (liveness)

where $\perp$ means that no process was chosen yet

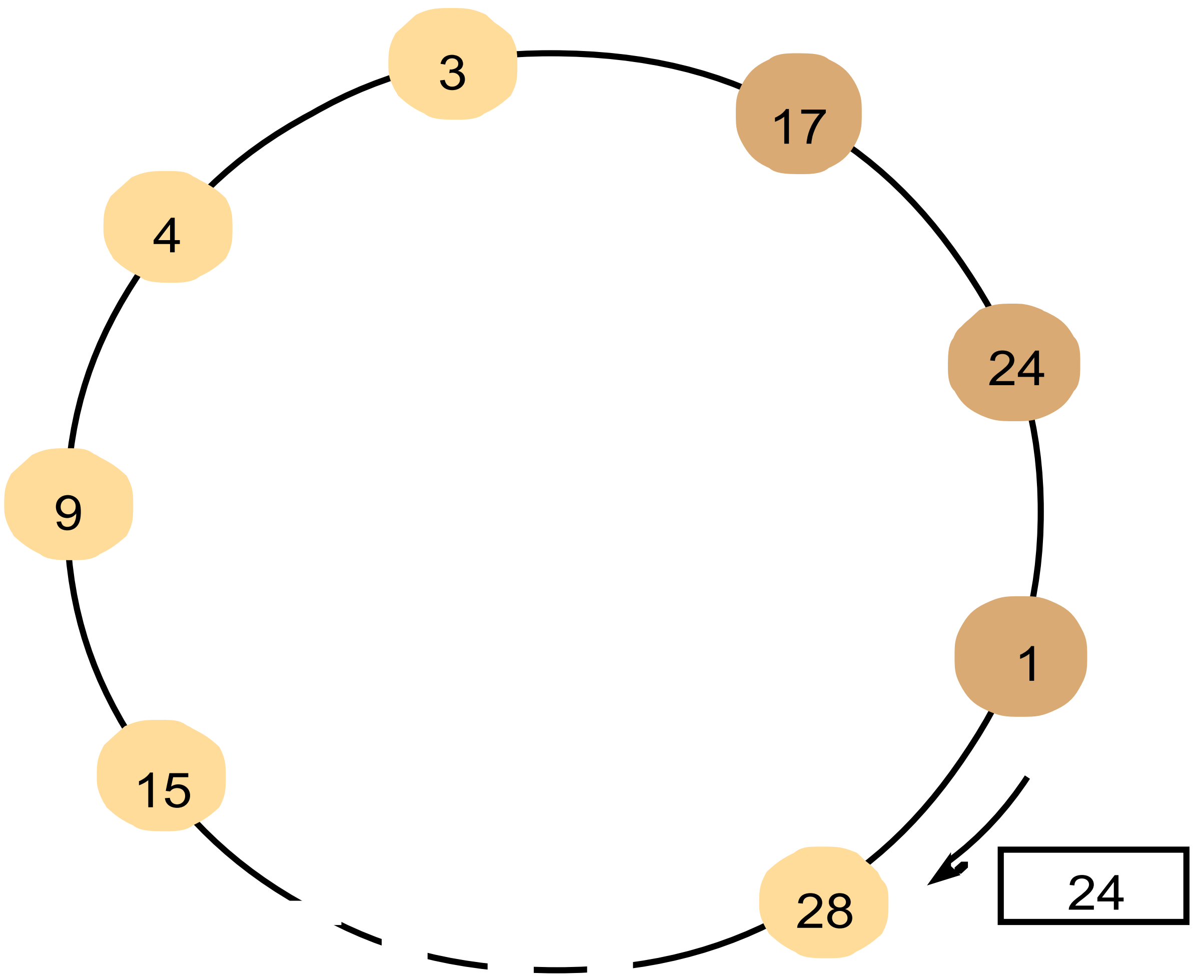
Ring based election

- Processes are organized in a logical ring
- If one process recognizes a server's failure, it will start the election, sending an election request token to its neighbour
- The token contains the identification of the process currently with the highest identification
- When receiving a token a process either forwards it, discards it if or changes it to an elected message, depending on token value and process participation

Ring based election

- Upon receiving an election message each process mark itself as "participant" and forwards the message with the higher value between the received one and its own
- If it was already "participant", it forwards the message if the value is higher than its own, or discards it if smaller
- If it was already "participant" and the value is equal to its own, then it changes its status to "non-participant" and sends an elected message
- Upon receiving an elected message each process changes it status to "non-participant"

Ring based election



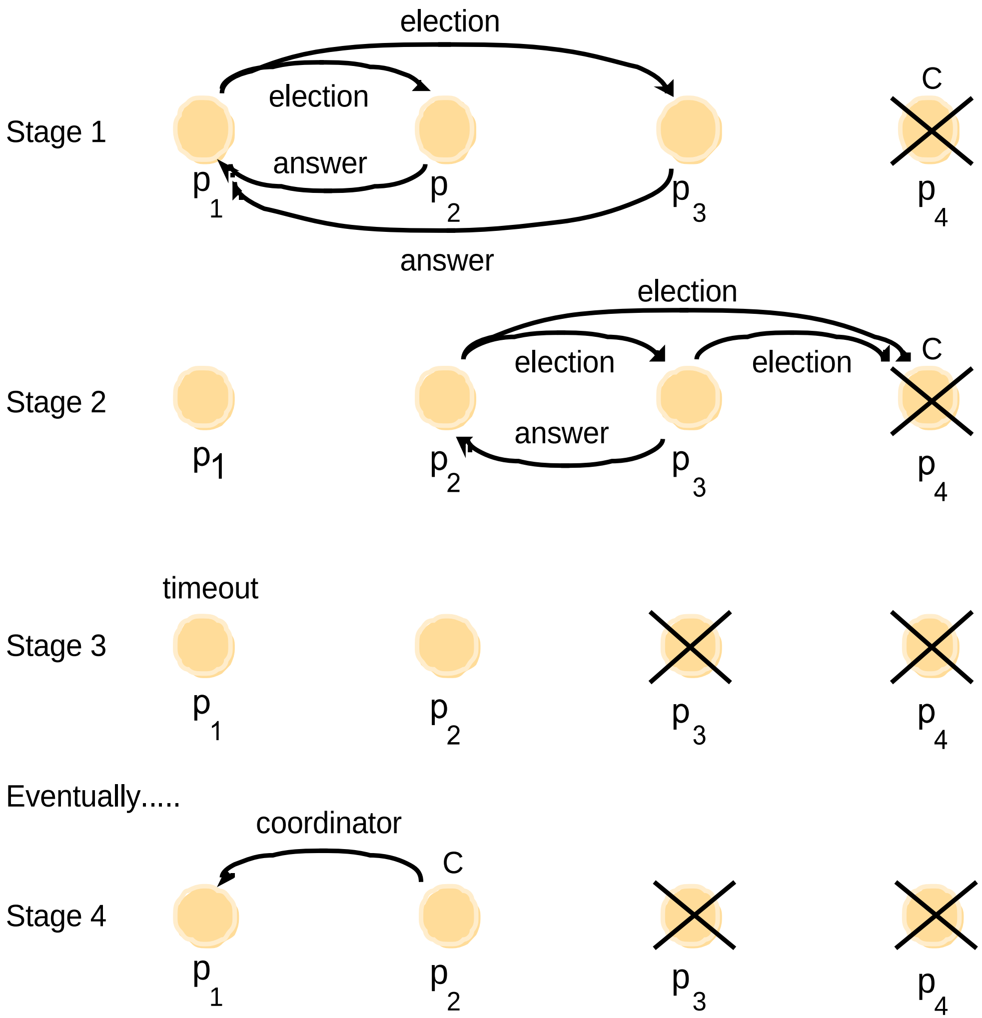
Bully algorithm

- In the Bully algorithm the idea is to allow for processes to crash during the election procedure
- It is supposed that each process knows its rank and the rank of other processes too
- It uses time-outs to detect crashed processes

Bully algorithm

- The messages in the algorithm are:
  - Election, where a process announces the election for processes with higher ranks
  - Answer, where a process replies to the election messages
  - Coordinator, where a process assumes the coordination position
- If a process do not get the coordinator message in a given period it assumes that all higher-ranked processes crashed and becomes the new coordinator

Bully election process



Coordination and agreement

- Coordination and agreement among processes in group communication is needed in order to assure correct event ordering and system reliability
- Reliability is viewed in terms of validity, integrity and agreement
- Ordering is viewed from FIFO, causal and total ordering perspectives

Basic multicast

- B-multicast guarantees that a message gets delivered if the sender does not crash
- It executes as:
  - To B-multicast(g,m): for each process $p \in g$, send(p,m)
  - On receive(m) at p: B-deliver(m) at p

The problem with this approach is the possibility of losing ack messages

Reliable multicast

- A reliable multicast is defined as one that attends the following properties:
  - Integrity, where a process delivers a message at most once
  - Validity, where if a correct process multicasts a message, then it will eventually deliver m
  - Agreement, where if a correct process delivers m, then all other correct process will also deliver m

Reliable multicast

- A reliable solution may be built over B-multicast by:

On initialization
    Received := { };

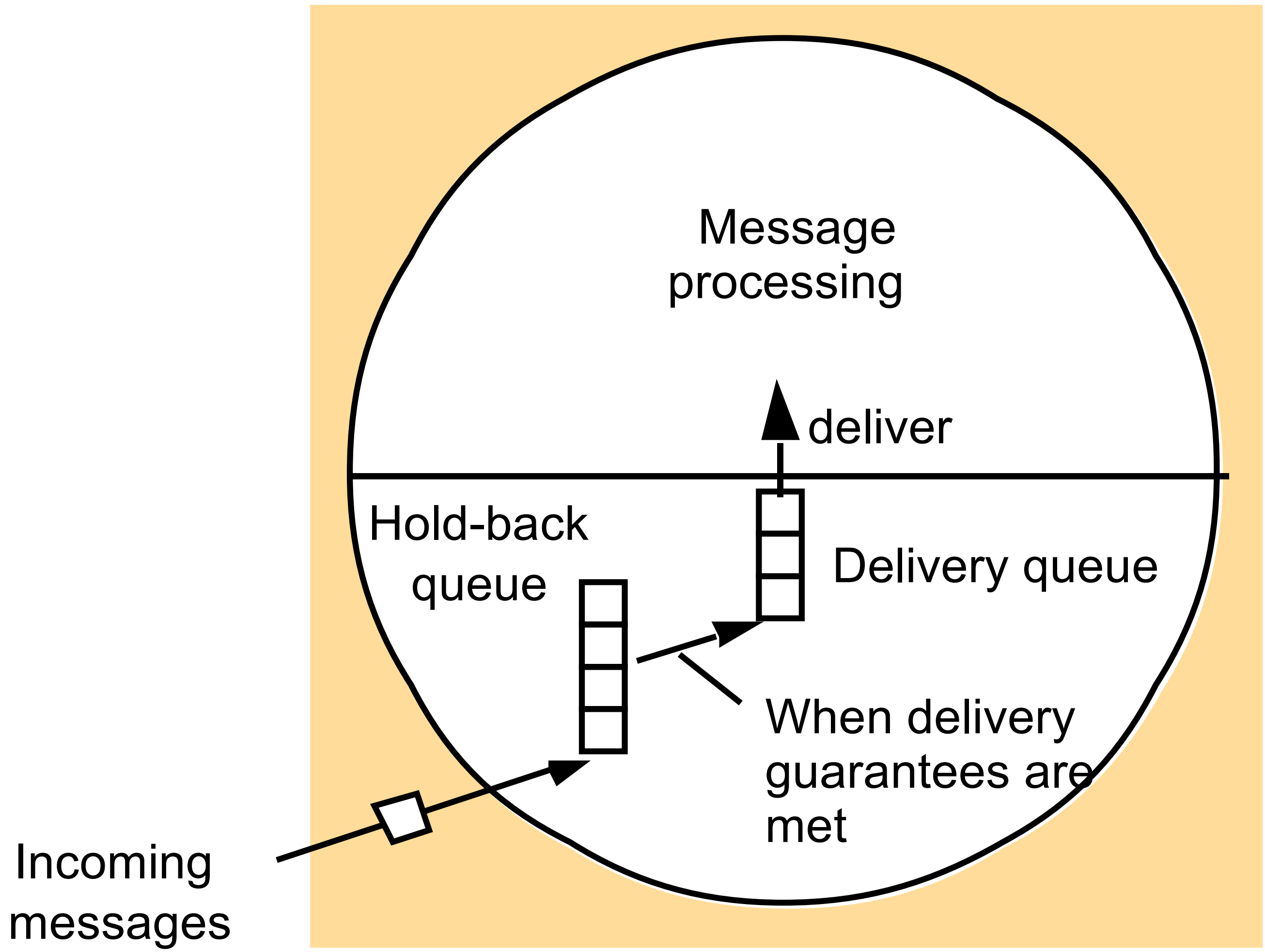
For process p to R-multicast message m to group g
    B-multicast(g, m);   // $p \in g$ is included as a destination

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-multicast*(g, m); *end if*
        R-deliver m;
    end if

Reliable multicast over IP multicast

- A sounder implementation may be achieved using piggybacking and negative acknowledgments
- Positive acks are piggybacked into other messages, in order to enable a process to identify a lost message, requesting it by a negative acknowledge
- It may use a hold-back queue to facilitate the protocol

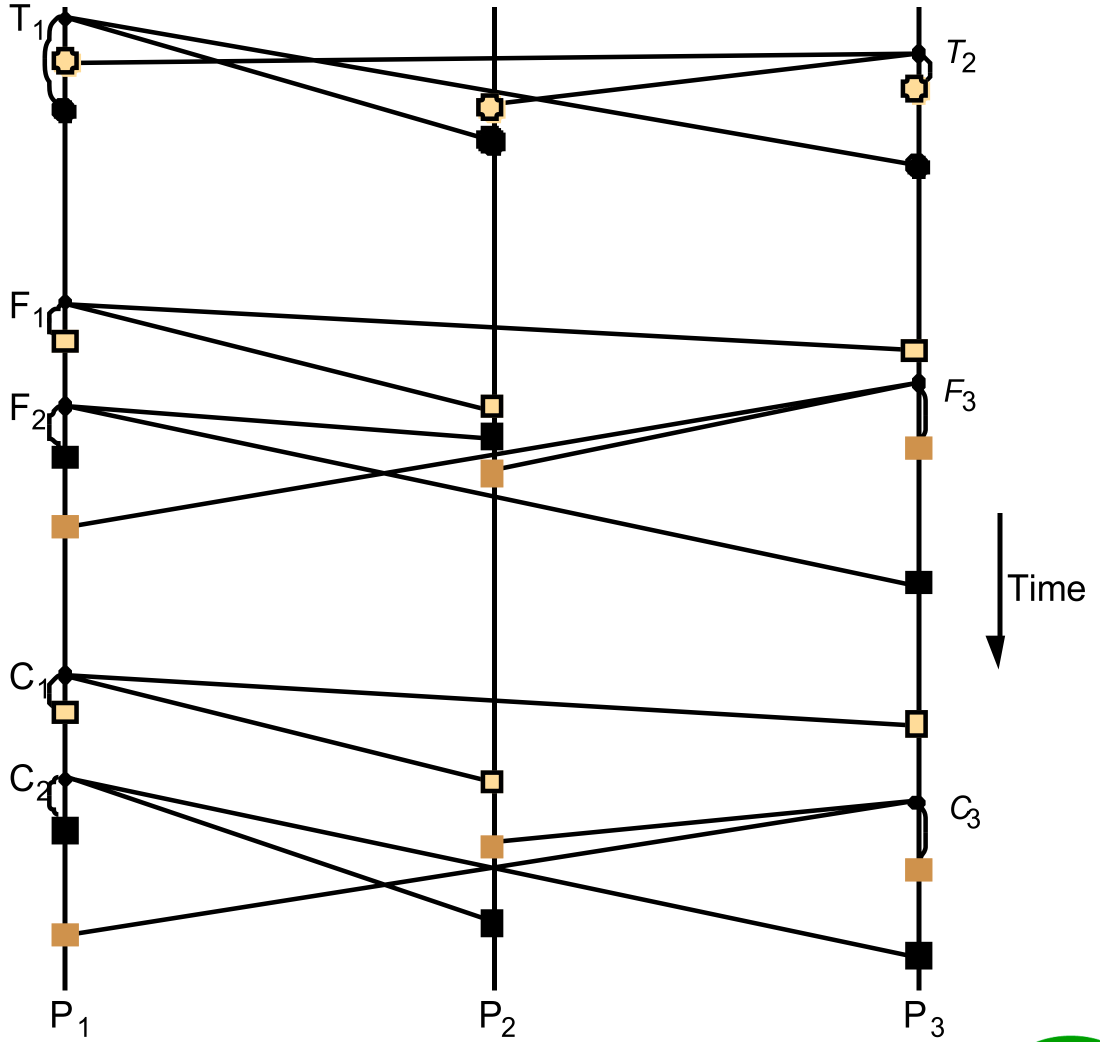
Hold-back queue



Reliable multicast over IP multicast

- Ordering events in multicast may follow one of these semantics:
  - FIFO, where if a process issues multicast(g,m) before multicast(g,m'), then every correct process that delivers m' will deliver m before m'
  - Causal, where if multicast(g,m) → multicast(g,m'), then any correct process delivering m' will deliver m first
  - Total, where if a correct process delivers m before m', then every process that delivers m' will deliver m before m'

Ordering of multicast messages



Implementing ordering

- The ordering semantics can be implemented by the enforcement of sequence numbers
- To implement FIFO ordering is enough to use an array of sequence numbers and enforcing the delivering of received messages be made in strictly sequential order, storing out-of-order messages in a hold-back queue

Consensus

- Consensus is one form of agreement between processes, where all processes involved in a group must agree on a value proposed by one of them
- The Byzantine generals problem falls in this category and will be discussed further here

Requirements for consensus

- Consensus is reached if attends these conditions:
  - Termination – when every correct process sets its decision variable (the decided state)
  - Agreement – when the decision variable of every correct process is the same at the decided state
  - Integrity – if all correct processes proposed the same value, then any correct process in the decided state has that value

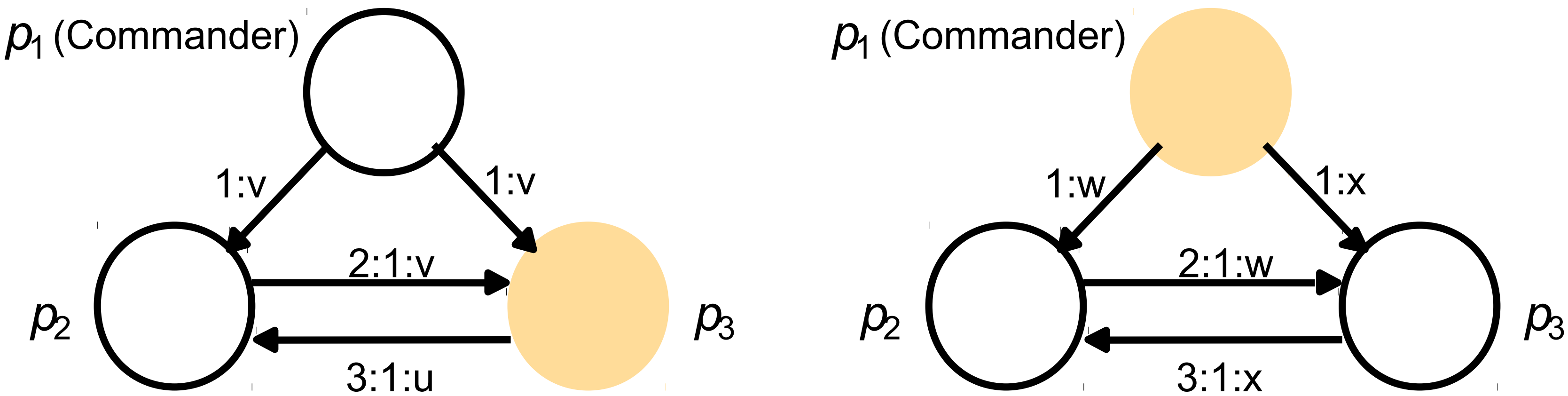
Requirements for consensus

- Consensus is reached if attends these conditions:
  - Termination – when every correct process sets its decision variable (the decided state)
  - Agreement – when the decision variable of every correct process is the same at the decided state
  - Integrity – if all correct processes proposed the same value, then any correct process in the decided state has that value

Byzantine generals problem

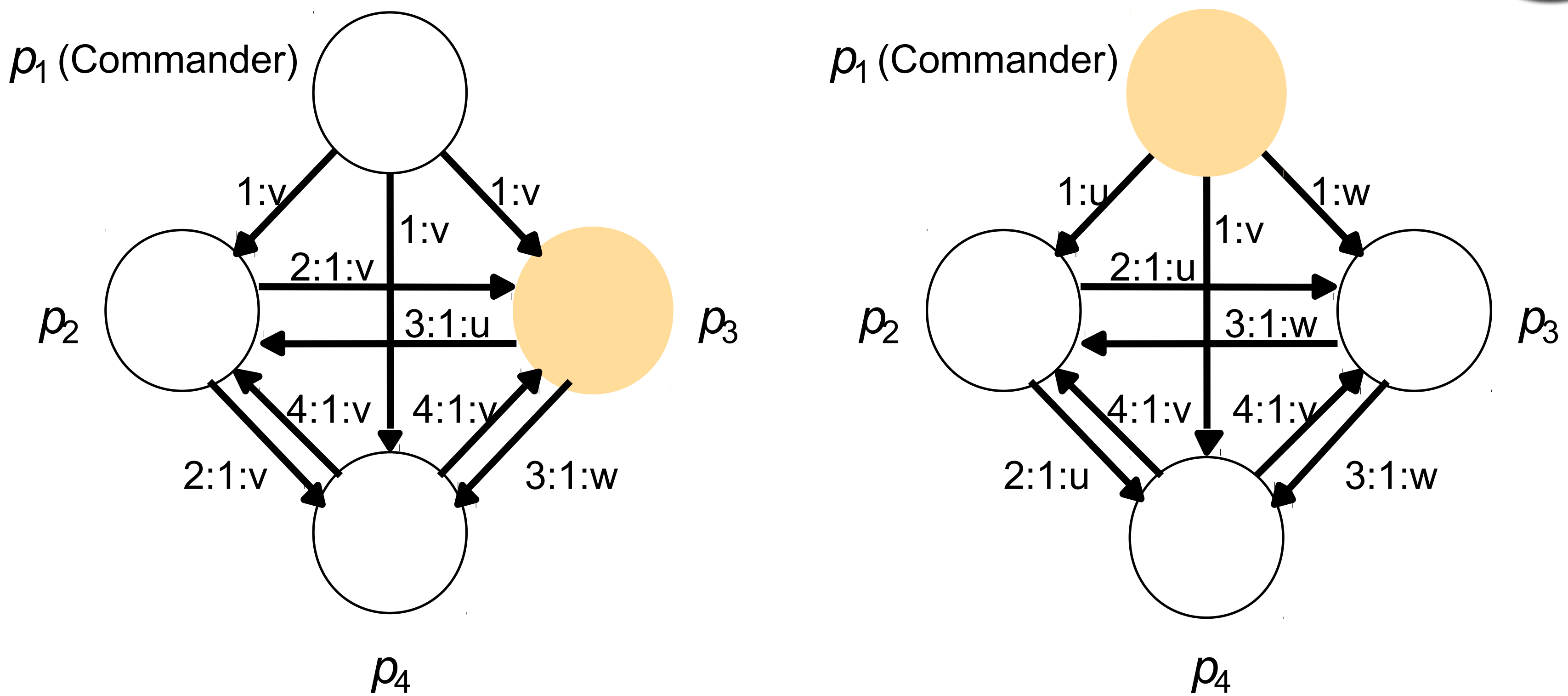
- Here one process proposes a value and the others must agree with it. The problem is to identify faulty processes that may provide false values
- If the faulty process is the commander, then some generals will receive a different value
- If the faulty process is one of the generals, then he will deliver different values to his peers

Byzantine agreement



Faulty processes are shown coloured

Lamport's clock progression



Faulty processes are shown coloured

Consensus in synchronous systems

- Consensus may be reached in synchronous systems with f faulty processes through a sequence of multicast messages
- After f+1 rounds it is expected that all correct processes have agreed on a decision value
- An algorithm for this is given in the next slide

Consensus algorithm

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

THAT'S IT FOR TODAY !!