Chapter 12

Coordination and Agreement

Distributed algorithms for coordination and agreement

- Distributed mutual exclusion
  - Processes have to agree on the order of access on shared resources to avoid race conditions.
- Multicast reliability and ordering semantics
  - Recipients have to agree on which messages to receive and in which order.
- Consensus and byzantine agreement
  - Processes have to agree on a value to keep the system working correctly even if failures occur.

Distributed Mutual Exclusion

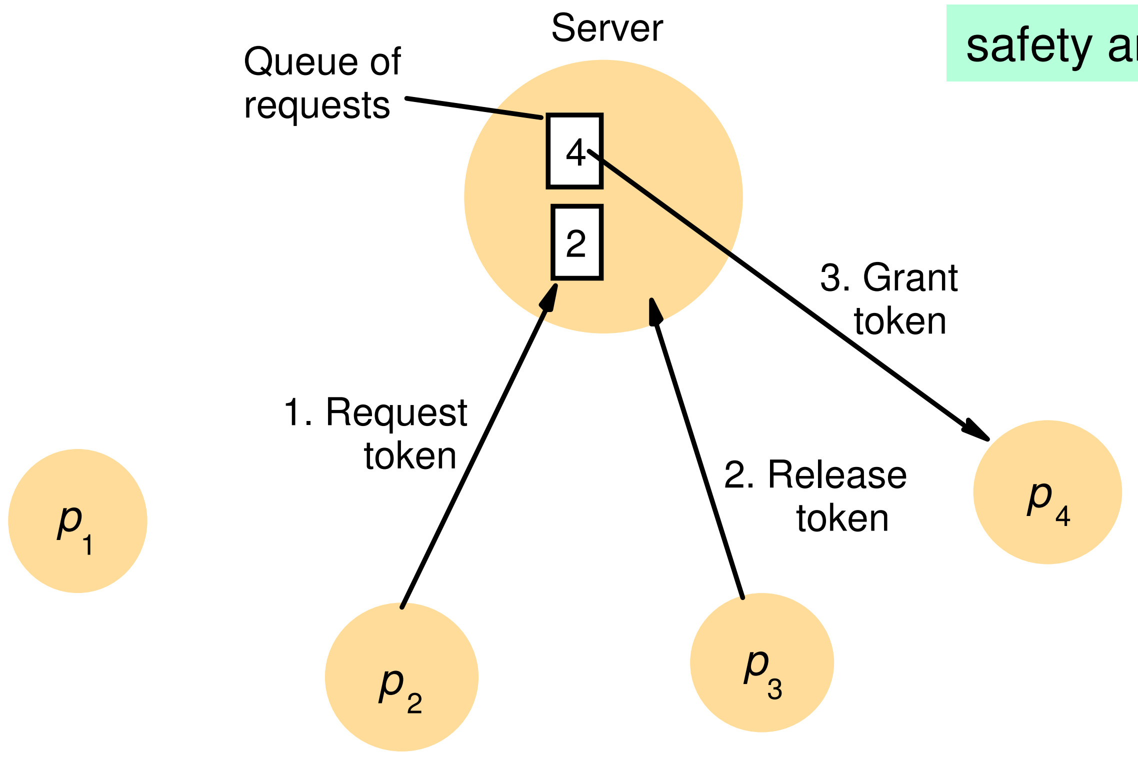
- To avoid inconsistency that may be caused by race conditions among distributed processes
  - Facilities in a single local kernel are not generally enough.
  - Algorithms are based on message passing.
  - Assume processes do not fail and message delivery is reliable (intact, and exactly once)
- Application-level protocol:

enter() – block if necessary
resourceAccesses() – work in CS
exit() – other process may enter

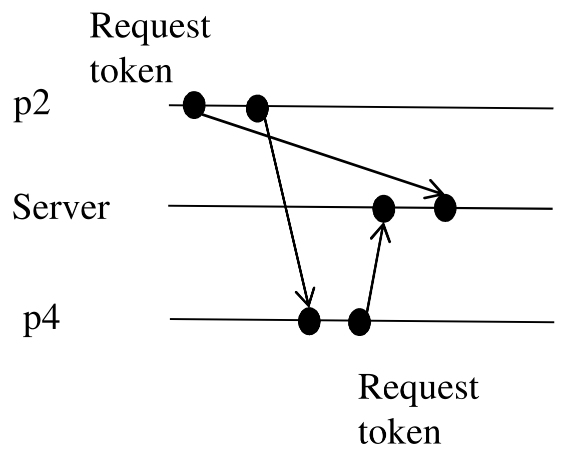
- Requirements:

Safety:	At most one process may execute in the CS at a time.
Liveness:	Requests to enter and exit the CS eventually succeed.
$\rightarrow$ ordering:	If one request to enter the CS happened-before another, then entry to the CS is granted in that order.

Central Server Algorithm



safety and liveness but not → ordering



Two messages (request and grant) on enter(), one message (release) on exit()

Client delay on enter() by a request-grant roundtrip (even though no process is in CS)

Synchronization delay by a release-grant roundtrip

Multicast and Logical Clocks-Based Algorithm (1)

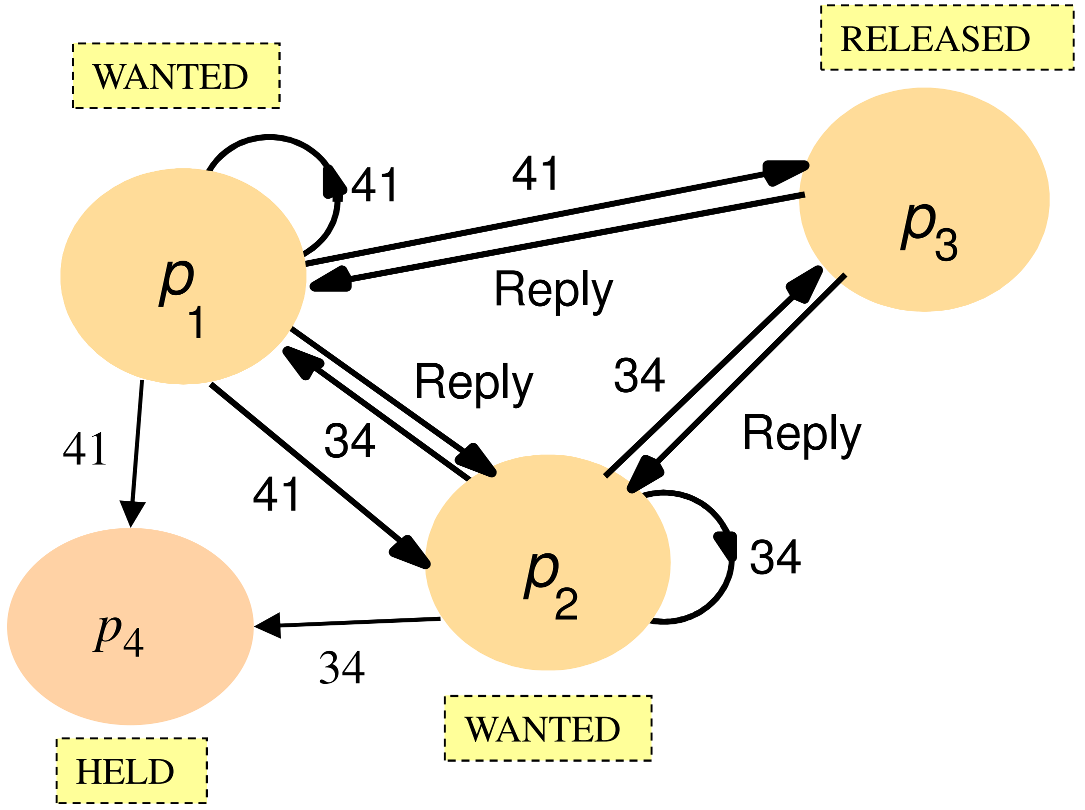
```
On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast request to all processes;
    T := request's timestamp;
    Wait until (number of replies received = (N – 1));
    state := HELD;

On receipt of a request <T_i, p_i> at p_j (i ≠ 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;
```

processing of other processes' requests is deferred here until all these are done.

Total order

Happened-before order is preserved.

Multicast and Logical Clocks-Based Algorithm (2)



safety, liveness, and → ordering

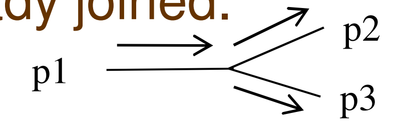
2(N-1) messages on enter() (or N messages (1+N-1) if hardware multicast is available)

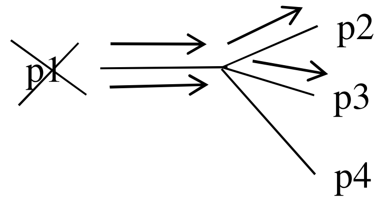
Client delay on enter() by a request-grants roundtrip

Synchronization delay by only one grant transmission time (of the process just exit())

Multicast Reliability and Ordering Semantics

- Multicast delivery should guarantee
  - Agreement on the messages each process in the group should receive
  - Ordering of the messages
- A process can multicast by the use of a single operation instead of a send to each member
  - For example in IP multicast, *aMulticastSocket.send*(*aMessage*) to the multicast address that *aMulticastSocket* has already joined.
  - The single operation allows for:
    - *Efficiency,* i.e. utilize bandwidth once on each link, using hardware multicast when available
    - *Delivery guarantees,* e.g. Agreement cannot be guaranteed if multicast is implemented as multiple sends and the sender fails half-way. Also, relative ordering of two messages to any two processes is undefined.





System Model for Multicast

- The system consists of a collection of processes which can communicate reliably over 1-1 channels.
- Processes fail only by crashing (no arbitrary failures).

Fail-stop

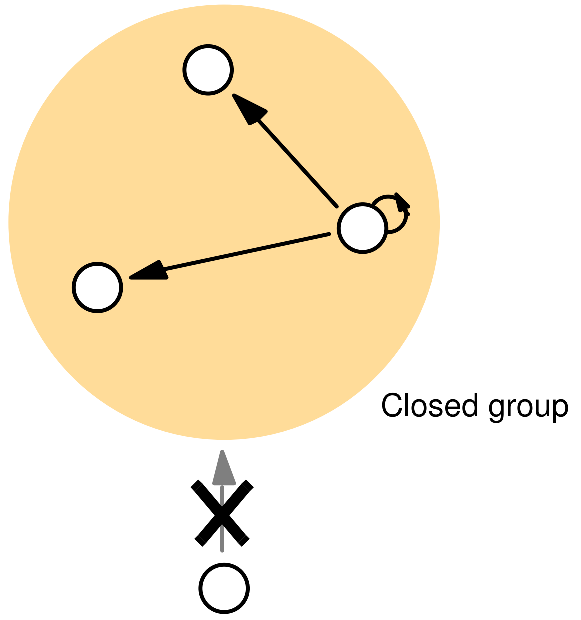
- In general process *p* can belong to more than one group.
- Operations:
  - *multicast*(*g*, *m*) sends message *m* to all members of process group *g*
  - *deliver* (*m*) delivers a message sent by multicast to the calling process. It is different from *receive* as it may be delayed to allow for ordering or reliability.

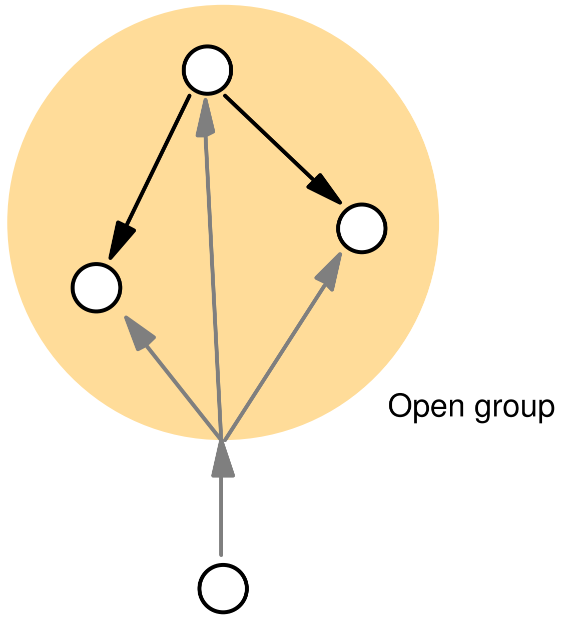
Application layer

- Multicast message *m* carries the id of the sending process *sender*(*m*) and the id of the destination group *group*(*m*).
- We assume there is no falsification of the origin and destination of messages.

Closed vs. Open Multicast Group

- Closed group
  - Only members can send to group; a member delivers to itself.
  - Useful for coordination of a group of cooperating servers
- Open group
  - Useful for notification of events to a group of interested processes





Basic Multicast (1)

- A basis for building reliable multicast and ordered multicast.
- Assume a correct process (i.e. never fails) will eventually deliver the message, provided that the multicaster does not crash.
- Use a reliable 1-1 send which has the following properties:
  - *Validity* : Any message in the outgoing message buffer is eventually delivered to the incoming message buffer.
    - Achieved by use of acknowledgements and retries
  - *Integrity* : The message received is identical to the one sent, and no messages are delivered twice.
    - Achieved by use of checksums, reject duplicates (e.g. due to retries).
    - If allowing for malicious users, use security techniques.

Basic Multicast (2)

- The primitives are called *B-multicast* and *B-deliver.*

  To *B-multicast*(*g*,*m*) *:* for each process $p \in g$, *send*(*p*, *m*)

  On *receive* (*m*) at *p* : *B-deliver* (*m*) at *p*

Use threads to send

- If the number of processes is large, the protocol will suffer from *ack-implosion.*
  - Multicasting process's buffer will fill rapidly and drop acknowledgements, causing the message to be retransmitted.
- Basic multicast can be built using IP multicast.

UDP; Need to implement ack and retry at application level.

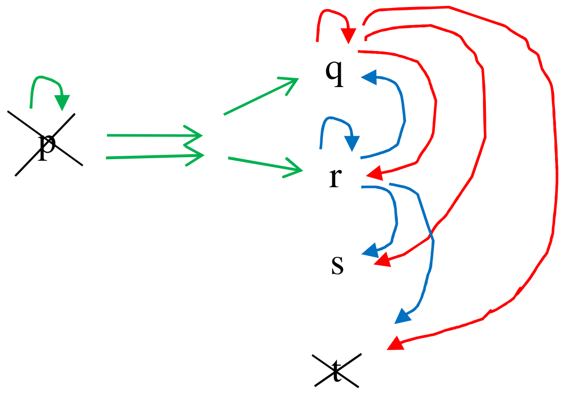
Reliable 1-1 but multicaster may crash half-way.

Reliable Multicast

- The protocol is correct even if the multicaster crashes.
- It provides operations *R-multicast* and *R-deliver.*
- It satisfies the following properties:
  - *Integrity* : A correct process *p* delivers *m* at most once. Also $p \in$ *group*(*m*) and *m* was supplied to a multicast operation by *sender*(*m*).
  - *Validity* : If a correct process multicasts *m*, it will eventually deliver *m.*
  - *Agreement* : If a correct process delivers *m,* then all other correct processes in *group*(*m*) will eventually deliver *m.* (This is *atomicity* or *all-or-nothing* over correct processes).

Reliable Multicast over Basic Multicast

```
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 ($m \notin Received$)    reject duplicate
    then
            $Received := Received \cup \{m\}$;
            if ($q \neq p$) then B-multicast(g, m); end if    q multicasts to others too since p may crash half-way.
            R-deliver m;    q uses m.
    end if
```



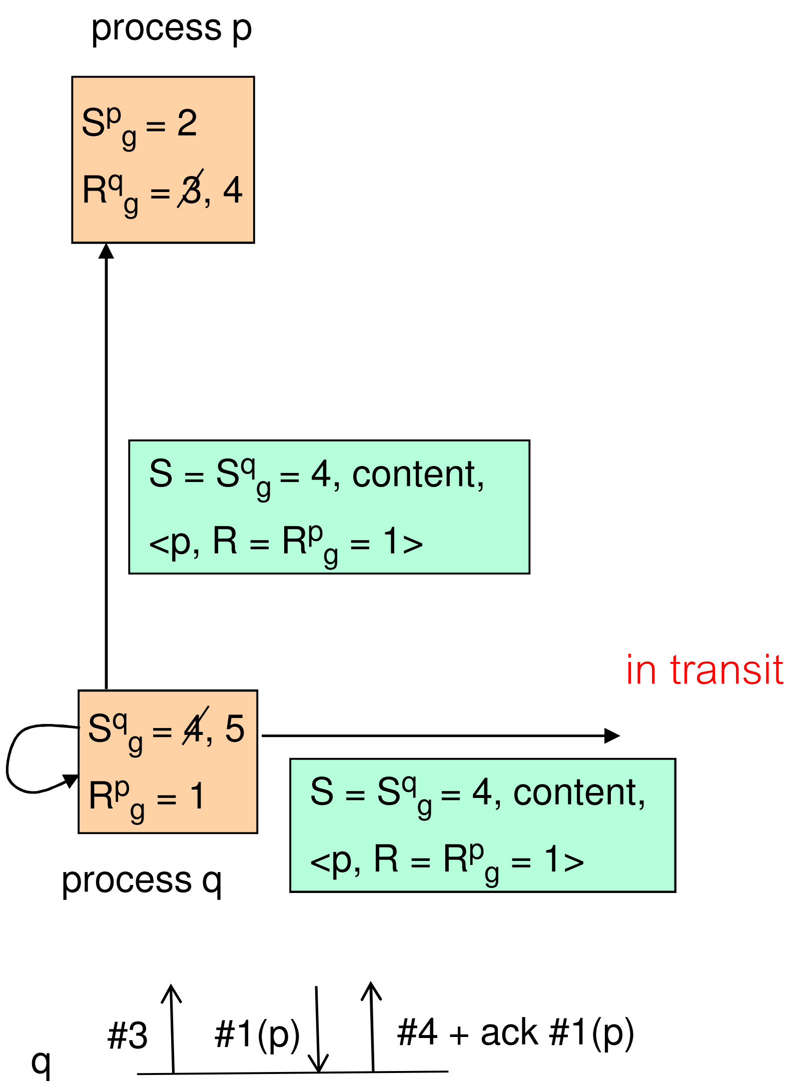
- Processes can belong to several closed groups.
- Validity : a correct process will *B-deliver* to itself.
- Integrity : reliable 1-1 channels used for *B-multicast* guarantee integrity.
- Agreement : all correct processes will eventually deliver *m* even though the multicaster crashes half-way.
- But this implementation is inefficient for practical purposes.

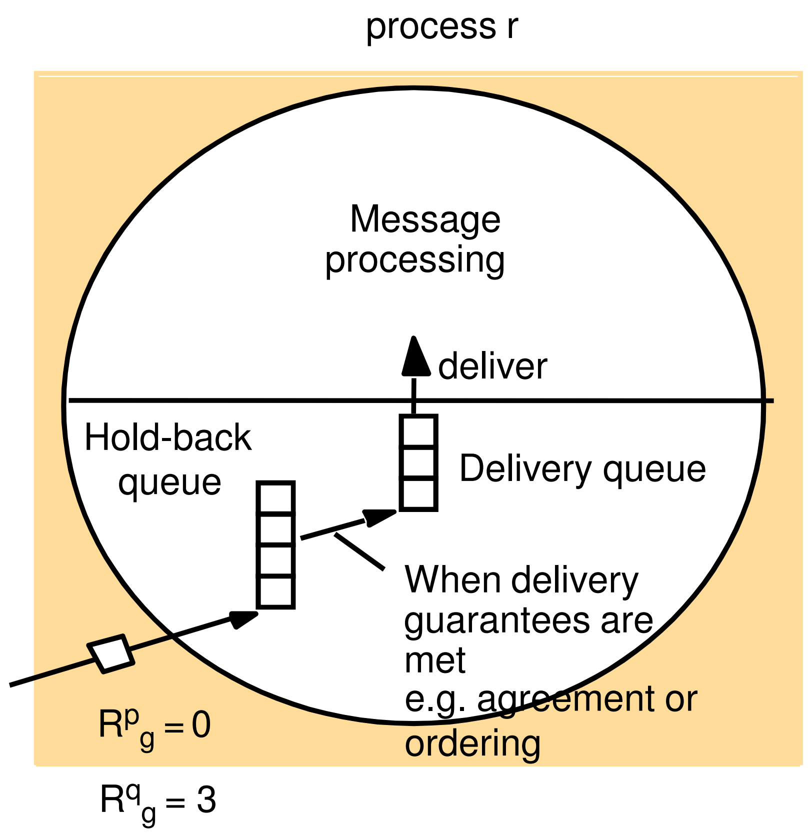
Reliable Multicast over IP multicast

- This protocol assumes groups are closed. It uses:
  - Piggybacked acknowledgement messages
  - Negative acknowledgements when messages are missed
- Process *p* maintains:
  - S^p_g : a message sequence number (for send) for each group that *p* belongs to (initially zero)
  - R^q_g : a sequence number of the latest message from process *q* to *g* that *p* has delivered since its last multicast
- For process *p* to *R-multicast* message *m* to group *g*
  - Piggyback $S = S^p_g$ and acks for messages received in the form $<q, R = R^q_g>$.
  - IP multicasts the message to *g*, then increments S^p_g by 1.
- A process *r* on receipt of a message to *g* with *S* from *p*
  - Iff $S = R^p_g + 1$, *R-deliver* the message and increment R^p_g by 1.

    All previous messages from p have been delivered.
  - If $S > R^p_g + 1$ or if $R > R^q_g$,
    - *r* has missed messages (from *p* or *q* respectively) and requests them from either *p* or the original multicaster, with negative acknowledgements.
    - *r* puts the too-early message in its *hold-back queue* for later delivery.
  - If $S \leq R^p_g$, discard the message (it has arrived before from other channel and *r* has delivered it).

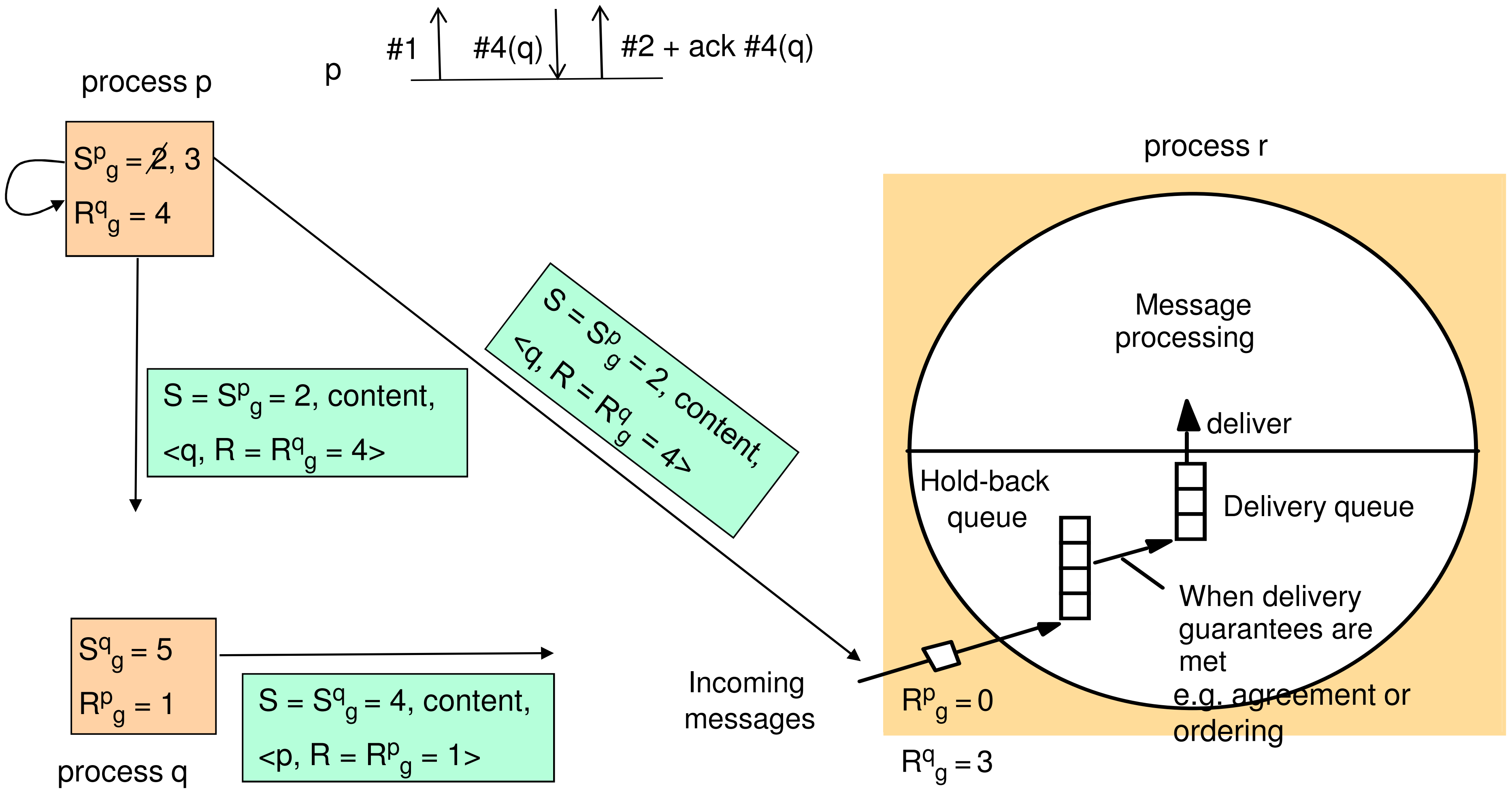
Hold-back queue for arriving multicast message (1)





msg $S^p_g = 1$ has been delivered by q but not arrived at r.

Hold-back queue for arriving multicast message (2)

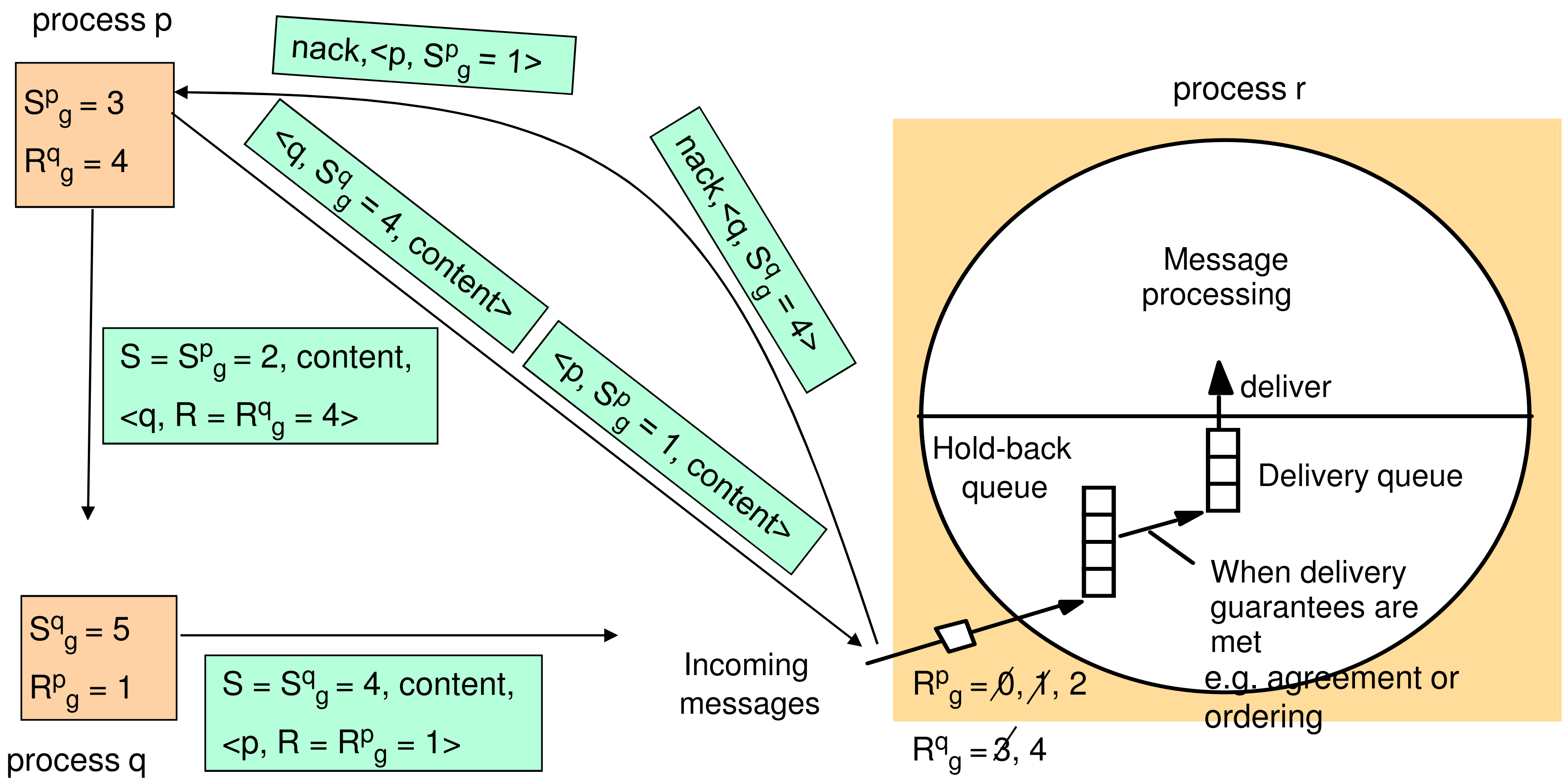


msg $S^p_g = 1$ is lost, msg $S^q_g = 4$ is late

msg $S^p_g = 2$ is put in hold-back queue, not deliver queue yet.

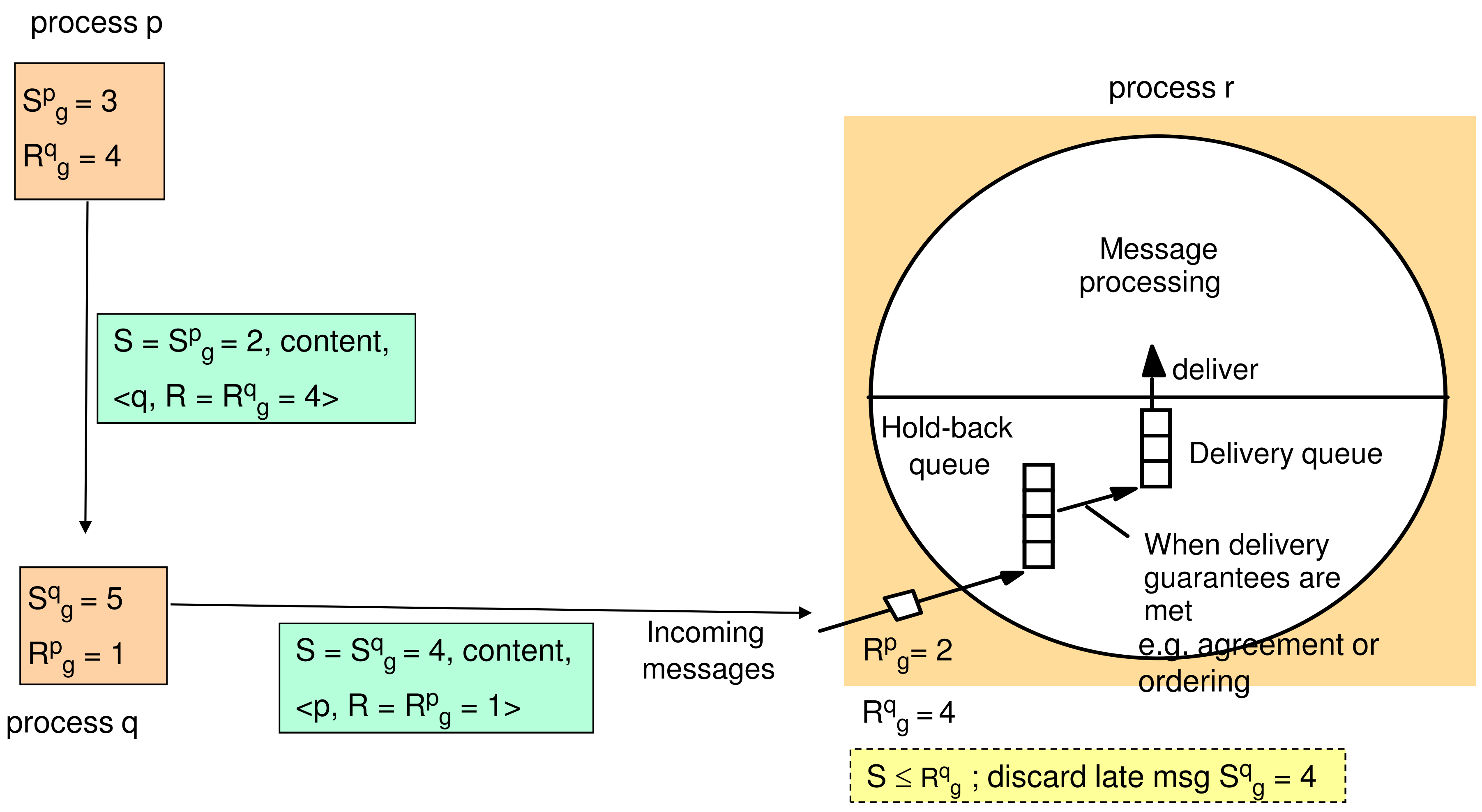
Hold-back queue for arriving multicast message (3)

nack = negative acknowledgement


process p
$S^p_g = 3$
$R^q_g = 4$
nack,<p, $S^p_g = 1$>
<q, $S^q_g = 4$, content>
nack,<q, $S^q_g = 4$>
<p, $S^p_g = 1$, content>
$S = S^p_g = 2$, content, <q, $R = R^q_g = 4$>
$S^q_g = 5$
$R^p_g = 1$
process q
$S = S^q_g = 4$, content, <p, $R = R^p_g = 1$>
Incoming messages
process r
Message processing
deliver
Hold-back queue
Delivery queue
When delivery guarantees are met
e.g. agreement or ordering
$R^p_g = \cancel{0}, \cancel{1}, 2$
$R^q_g = \cancel{3}, 4$

When missing messages arrive, put in hold-back queue to sort and move to deliver queue by sequence number.

Hold-back queue for arriving multicast message (4)



duplicate

Reliability Properties of Reliable Multicast over IP Multicast

- Reliability is achieved by assuming each process multicasts messages indefinitely.
  - Missing messages are detected as further messages are received.
- *Integrity* : duplicate messages are detected and rejected, and IP multicast uses checksums to reject corrupt messages.
- *Validity* : due to IP multicast in which the multicaster delivers to itself.
- *Agreement* : processes must keep copies of messages they have delivered so that they can retransmit them to others.
  - To discard copies of messages that are no longer needed :
    - Each process has to collect piggybacked acknowledgements of a message from all other processes in *g.*
    - Problem of a process that stops sending - use ‘heartbeat’ messages.

Ordered Multicast

- FIFO ordering
  - If a correct process issues *multicast*(*g*, *m*) and then *multicast*(*g*, *m'*), then every correct process that delivers *m'* will deliver *m* before *m'* .
- Causal ordering
  - If *multicast*(*g*, *m*) → *multicast*(*g*, *m'*), where → is the happened-before relation between messages in group *g*, then any correct process that delivers *m'* will deliver *m* before *m'* .
    - Causal ordering implies FIFO ordering.

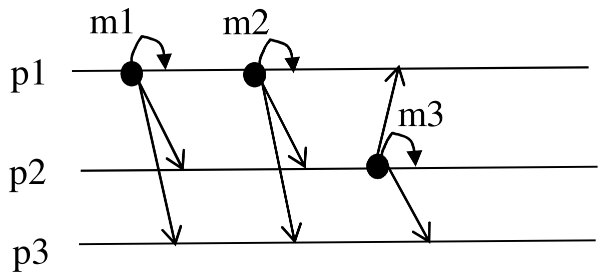
FIFO has intraprocess HB.

- Total ordering
  - If a correct process delivers message *m* before it delivers *m'*, then any other correct process that delivers *m'* will deliver *m* before *m'*.
    - Order is the same at every process.
    - We can have arbitrary-total, FIFO-total or causal-total ordering.
- Ordered multicast is expensive in delivery latency and bandwidth consumption but is a requirement in many applications.
  - Less expensive orderings (e.g. FIFO or causal) are chosen for applications for which they are suitable.

arbitrary-total: m3, m2, m1

FIFO-total: m1, m3, m2

causal-total: m1, m2, m3



first-in first-out

Total, FIFO, and Causal Ordering of Multicast Messages

cause & effect

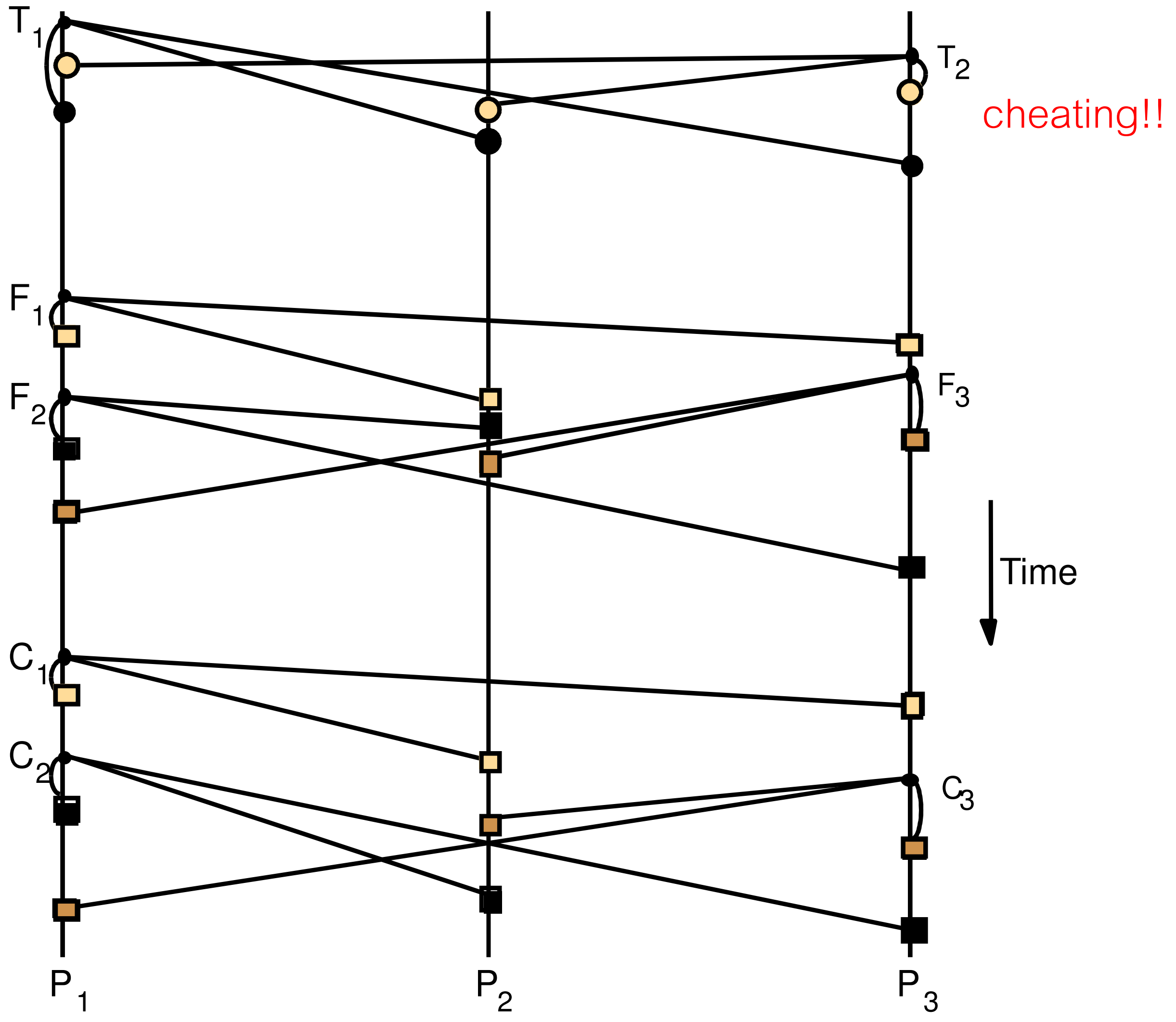
Consistent ordering of totally ordered messages T_1 and T_2.
They are opposite to real time.
The order can be arbitrary; it need not be FIFO or causal.

FIFO-related messages F_1 and F_2

C1 -> C3, C1 -> C2, C2 || C3

Causally-related messages C_1 and C_3

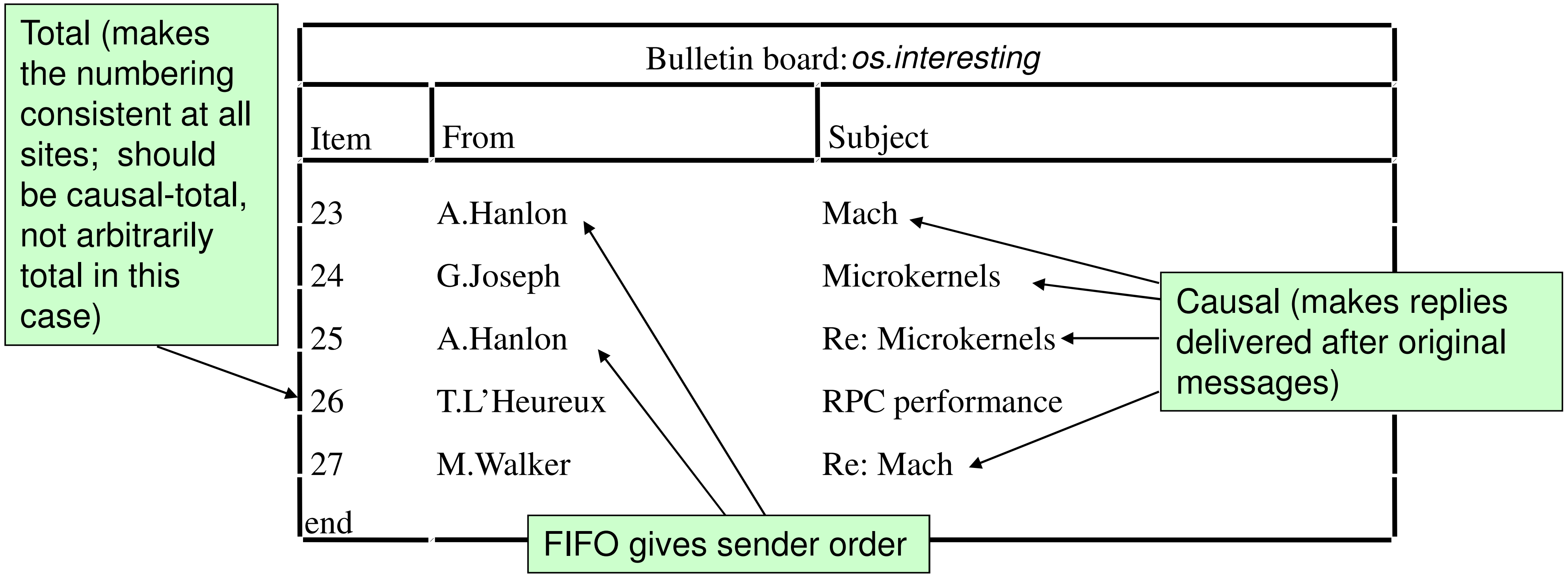
Ordered multicast does not imply reliability (e.g. in total ordering, p_1 may deliver *m*, *m'*, *m''* while p_2 delivers *m*, *m'*).
We can form ordered and reliable multicast (e.g. totally-ordered reliable multicast is referred to as atomic multicast).



Bulletin Board Program

This is not a web board!

- Users run bulletin board applications which multicast messages.
- One multicast group per topic (e.g. *os.interesting*)
- Reliable multicast : so that all members receive messages.
- Ordering:


Total (makes the numbering consistent at all sites; should be causal-total, not arbitrarily total in this case)
Causal (makes replies delivered after original messages)
FIFO gives sender order

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		

Implementation cost outweighs advantages of ordering for USENET.

Implementing FIFO ordering over basic multicast

- Assume each process belongs to a single group.
- Operations : *FO-multicast* and *FO-deliver.*
- Each process *p* holds:
  - S^p_g : a count of messages sent by *p* to *g*
  - R^q_g : the sequence number of the latest message to *g* from *q* that *p* has delivered
- For *p* to *FO-multicast* a message to *g*
  - Piggyback S^p_g on the message
  - *B-multicast* it and increment S^p_g by 1
- On *p*'s receipt of a message from *q* with sequence number *S*
  - Check whether $S = R^q_g + 1$. If so, *FO-deliver* it.
  - If $S > R^q_g + 1$ then *p* places message in its hold-back queue until intervening messages have been delivered. (Note that *B-multicast* eventually delivers messages unless the sender crashes.)
- This is similar to the previous reliable multicast over IP multicast except that there is no ack/nack part. Only order messages from each sender .
  - The previous reliable multicast over IP multicast already guarantees FIFO ordering.

Implementing total ordering over basic multicast

- Assume each process belongs to a single group.
- Attach group-specific (totally ordered) sequence numbers to multicast messages.

Use sequence number run by group

  - Similar to FIFO algorithm, but processes keep group-specific sequence numbers.
    - Each receiving process makes the same ordering decisions based on these sequence numbers.
  - Like *B-multicast*, if the multicaster does not crash, all members receive the message.
- Operations : *TO-multicast* and *TO-deliver*
- Two approaches
  - Use a sequencer (although it can be a bottleneck).
  - Processes in a group collectively agree on a sequence number for each message.

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($m_{order}$ = <"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, <"order", i, $s_g$>);
    $s_g := s_g + 1$;
```

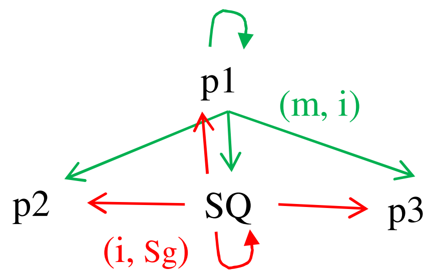
unique identifier of message

Message is always put in the hold-back queue, waiting to hear from the sequencer about the associated sequence number.

The message is delivered in total ordering.

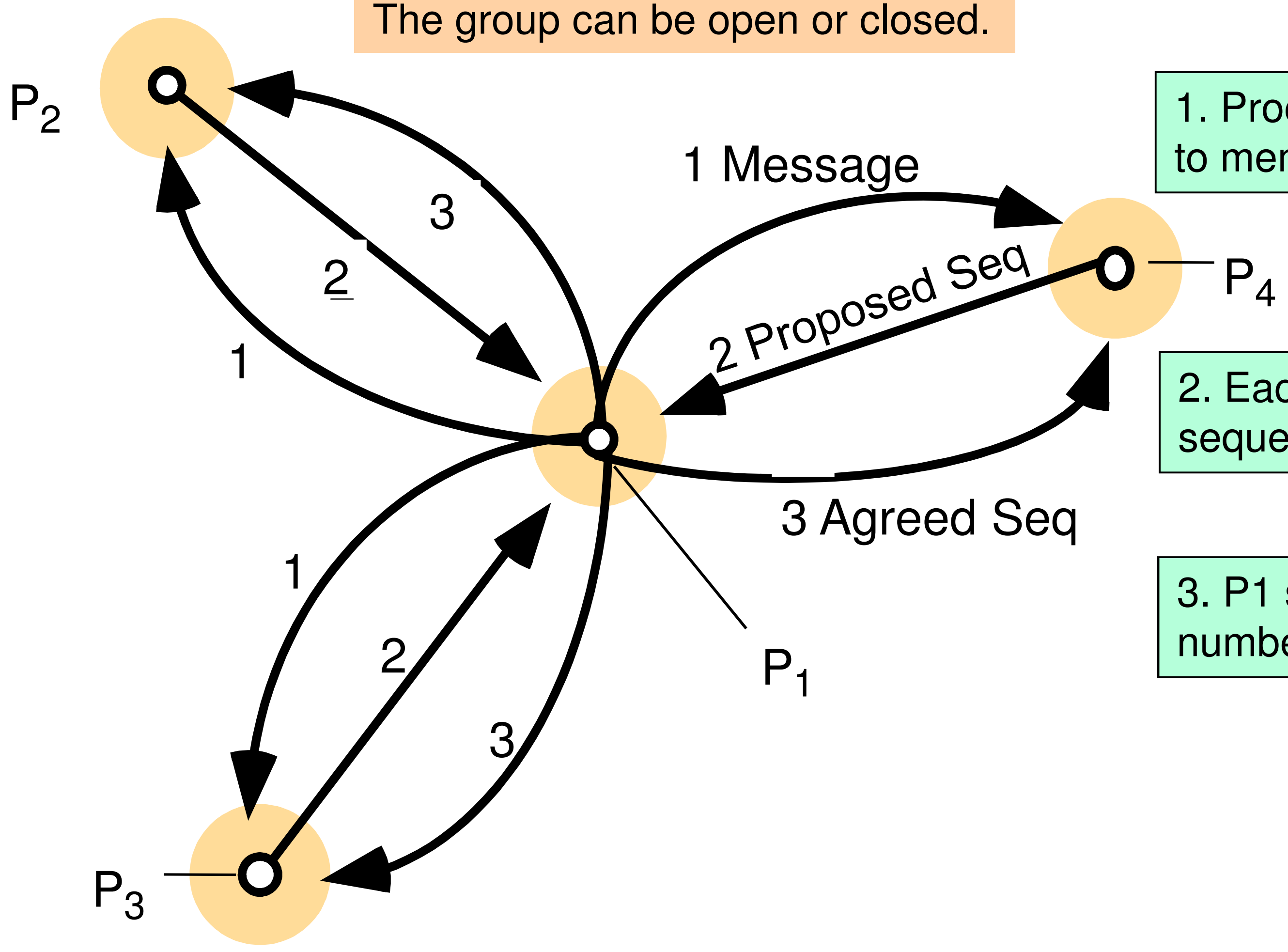
group sequence number

assign sequence number to i



Total ordering using ISIS algorithm (1)

The group can be open or closed.



1. Process P_1 *B-multicasts* a message to members of the group.

2. Each receiving process proposes a sequence number and returns it to P_1.

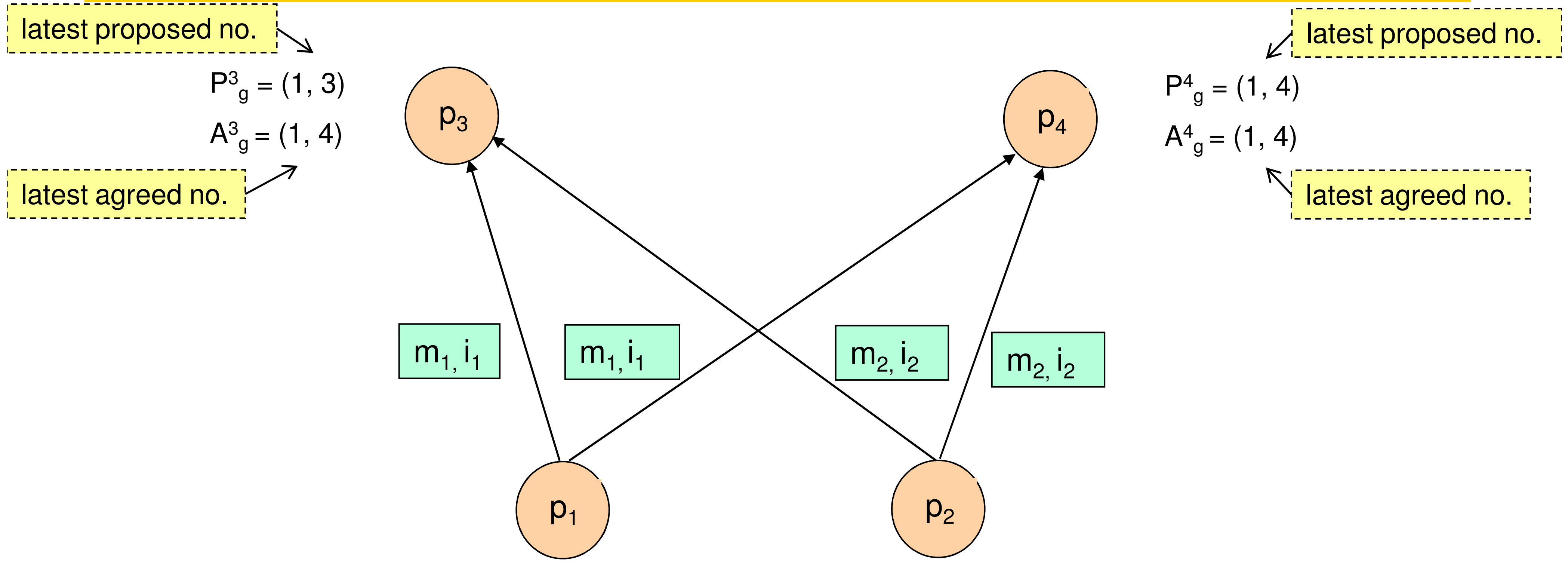
3. P1 selects an agreed sequence number, from the proposed numbers.

It has higher latency than the sequencer method (3 messages).

Total ordering using ISIS algorithm (2)

- Each process *q* keeps:
  - A^q_g : the largest agreed sequence number it has seen so far
  - P^q_g : its own largest proposed sequence number
- Process *p B-multicasts* <*m*, *i*> to *g*, where *i* is a unique identifier for *m*.
- Each process *q* proposes to the sender *p* a sequence number $P^q_g = \text{Max}(A^q_g, P^q_g)+1$ for *m.*
  - The proposed number includes *q*'s process identifier to ensure total ordering.
  - *q* assigns the proposed sequence number to *m* and places *m* in its hold-back queue in ascending order.
- *p* collects all the proposed sequence numbers and selects the largest as the next agreed sequence number, *a*.
  - *p B-multicasts* <*i*, *a*> to *g*.
  - Each recipient sets $A^q_g = \text{Max}(A^q_g, a)$, attaches *a* to *m,* and reorders hold-back queue.
  - The message with an agreed sequence number and also at the head of hold-back queue will be transferred to the tail of delivery queue.
- This is arbitrary total ordering.

Agreement on a sequence number in ISIS (1)

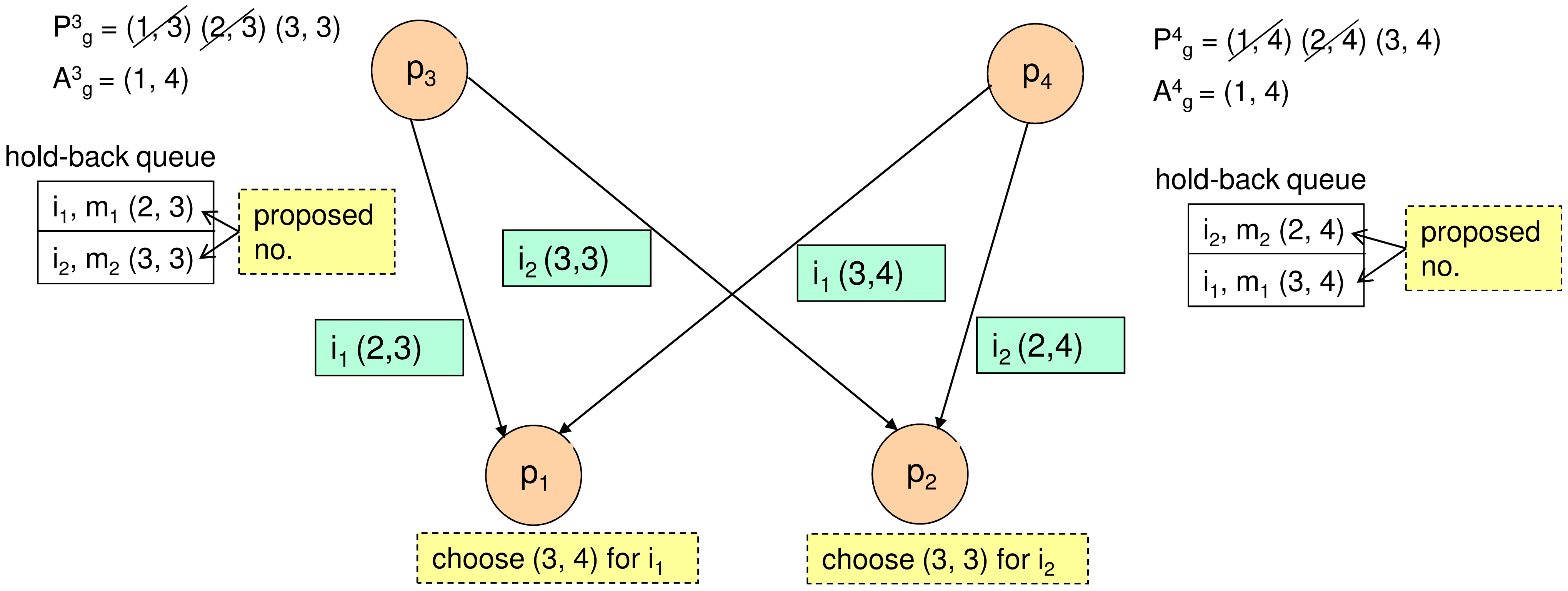


$p_3, p_4 \in g$

p_1, p_2 are outside multicaster

(x, y) is sequence number where x = running number, y = process identifier

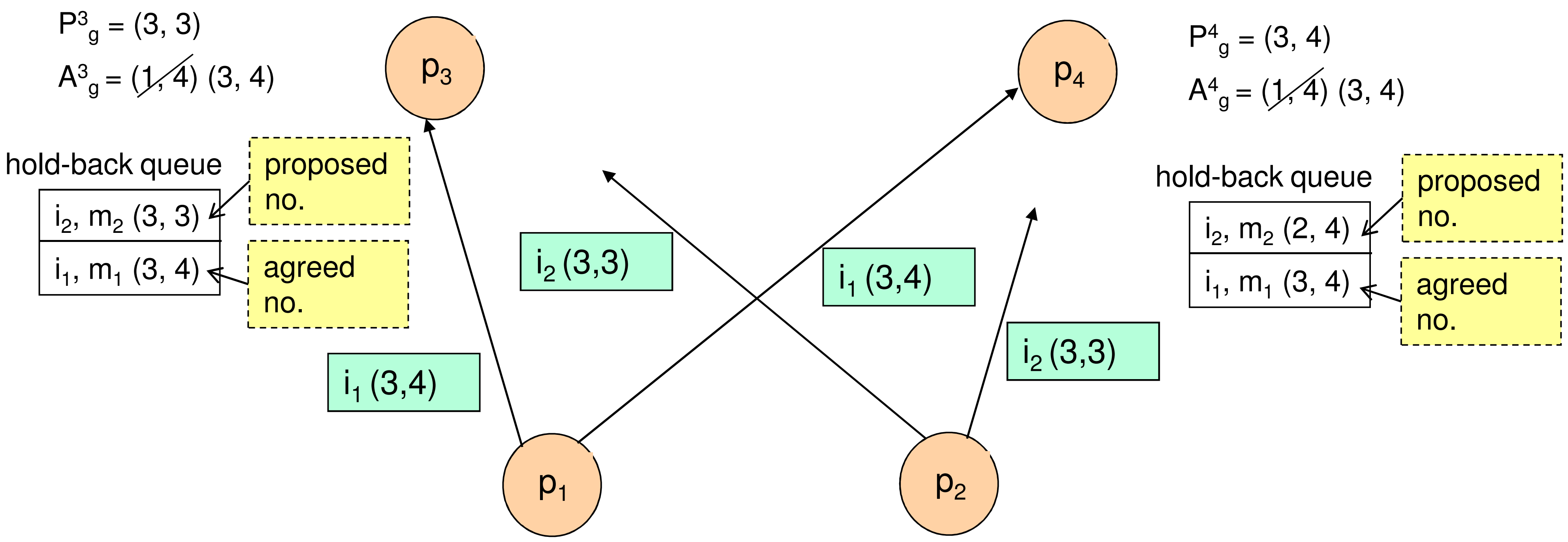
Agreement on a sequence number in ISIS (2)



Assume p_3 responds to m_1 first and p_4 responds to m_2 first.

By total ordering, sequence number $(x_1, y_1) > (x_2, y_2)$ iff $(x_1 > x_2)$ or $((x_1 = x_2)$ and $(y_1 > y_2))$

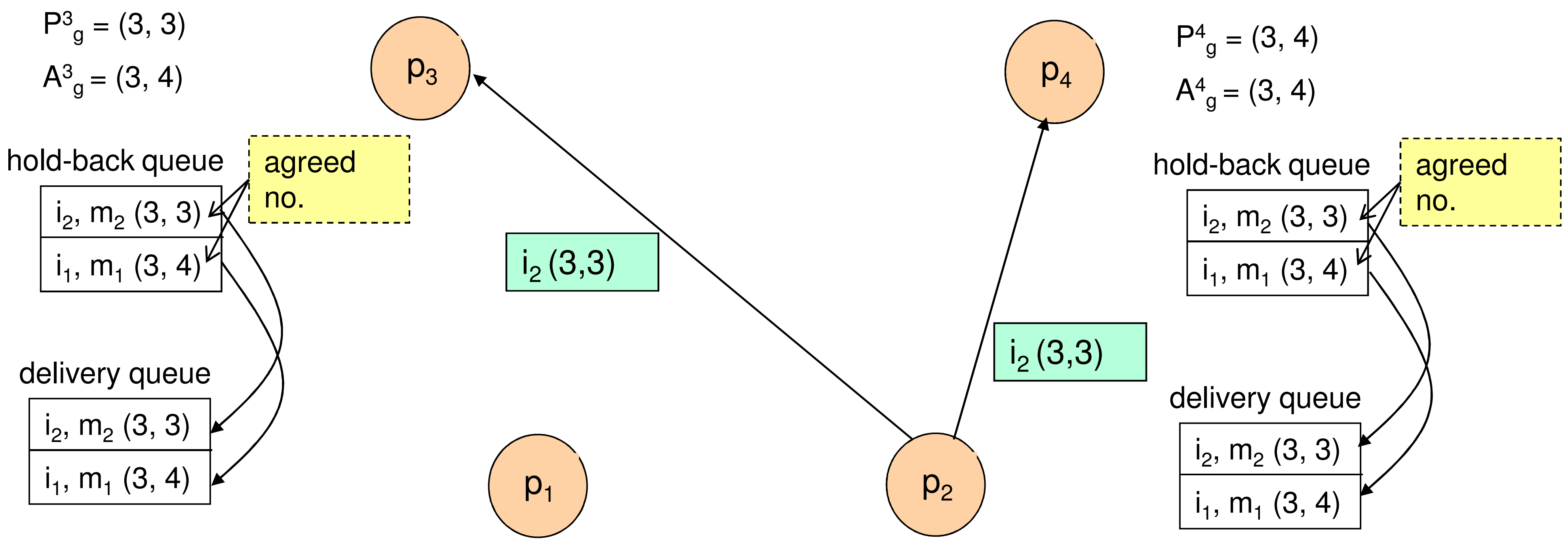
Agreement on a sequence number in ISIS (3)



Assume agreed sequence number from p_2 arrives later.

m_1 will be kept waiting in the hold-back queue.

Agreement on a sequence number in ISIS (4)



p_3 and p_4 will deliver m_2 and then m_1 in the same order.

Agreement Problems

- Processes usually have to agree on a value after one or more processes have proposed what that value should be.
  - Computers in a spaceship have to agree on whether to proceed or abort.
  - Computers in a money transfer transaction have to agree on respective debit and credit.
  - The previous ISIS algorithm
- The practical requirement is that agreement should be reached even though there are faults.
- We will look at consensus problem and byzantine generals problem.

System Model

- Assume communication is reliable but processes may fail.
  - Crash failure – process either functions correctly or crashes.
  - Byzantine (arbitrary) failure – process may send arbitrary messages or omit to send messages.
- Upto a certain number of processes may fail and still cannot harm other correct processes.
  - Correct process means the process works correctly and communication channel works correctly and safely.
- Environment with message signing is less prone to the harm from faulty processes. Digital signature
  - Processes can be sure that the messages are really from the claimed sources and the contents are not interfered during transmission.
  - Faulty processes cannot make false claim about the messages from other correct processes.

Consensus Problem

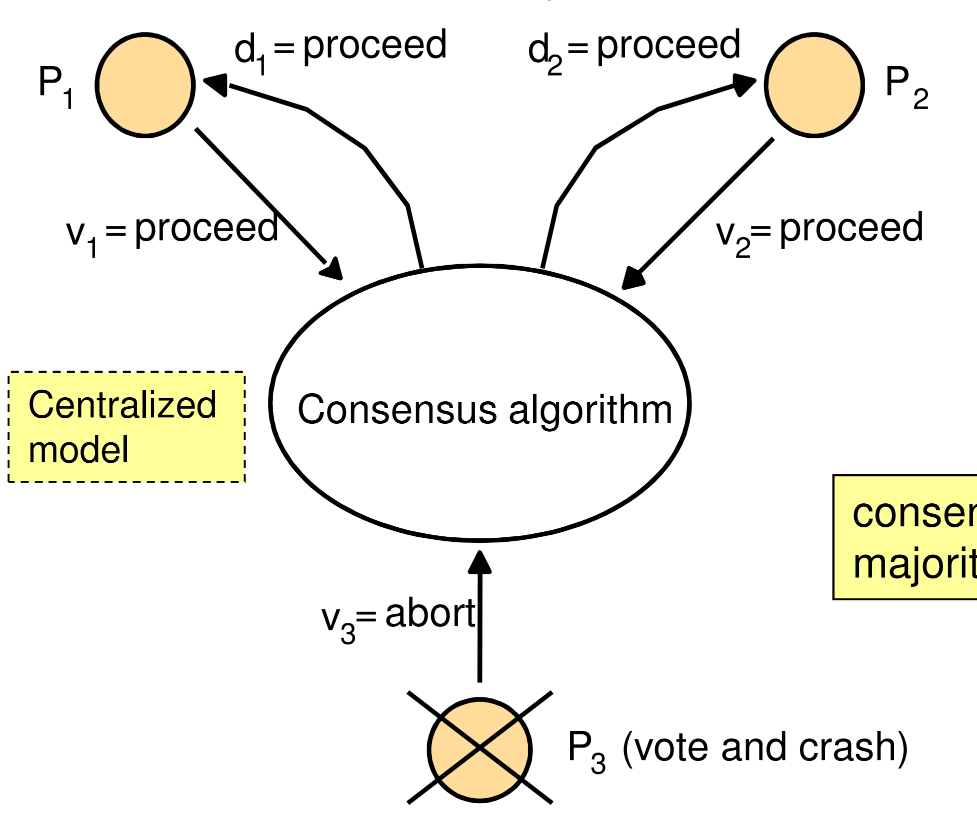
majority or unanimous

- Processes will agree on a value by finding a consensus.
- They start with the undecided state and then propose a value to others.
- Consensus is reached by finding a majority of values (or min, max etc. where appropriate).

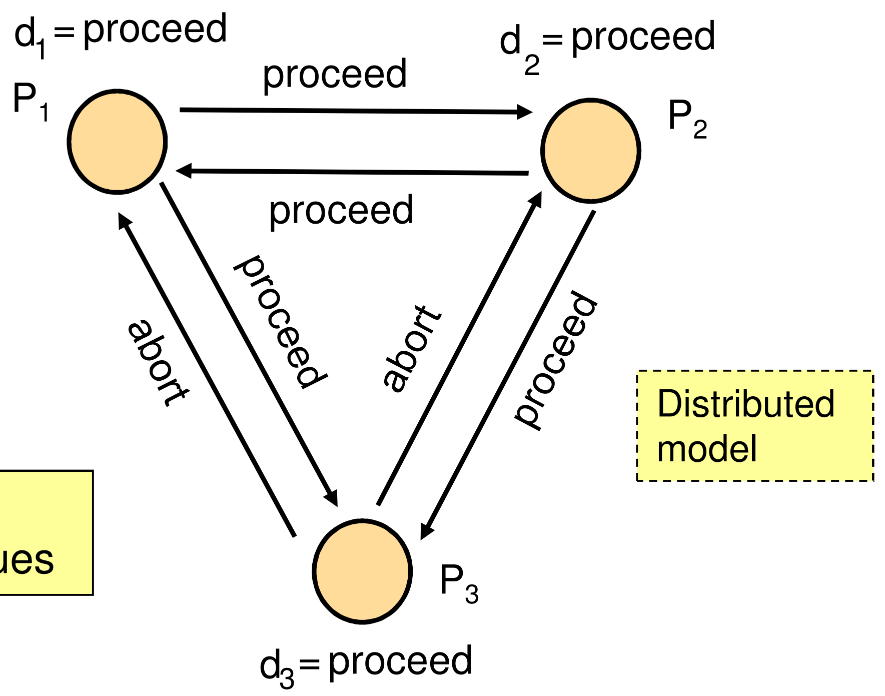
ISIS decides on max value.

- Processes will set its decision variable according to the consensus value.
- Requirement:
  - 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.

Consensus with crash failure or no process failure



consensus by majority of values

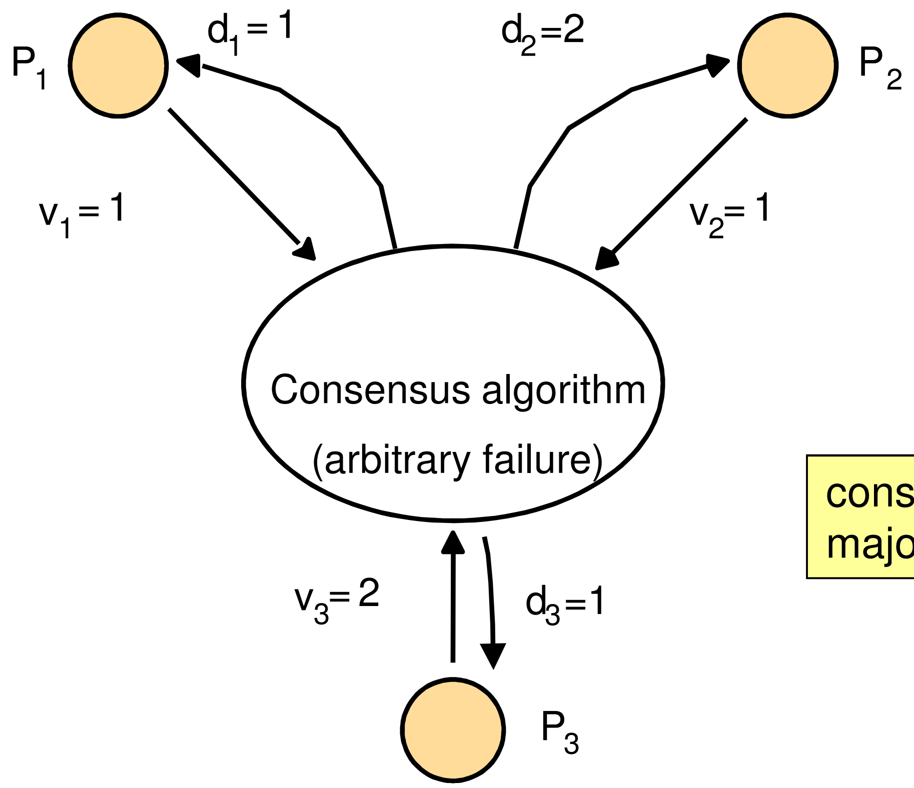


Termination is guaranteed by reliability of multicast. Agreement and integrity are guaranteed by majority.

All correct processes agree and set their decision value.

Consensus assumes that votes are from the claimed sources and not interfered during transmission.

Consensus with arbitrary failure

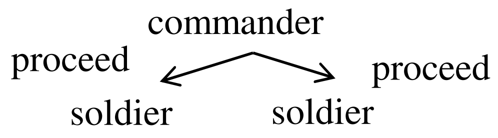


consensus by majority of values

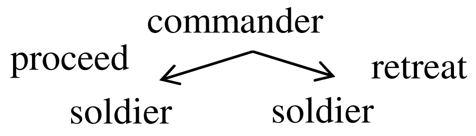
If processes receive random values from buggy processes or by malicious attack (i.e. arbitrary failure), correct processes may not agree on the same value; they must compare what they receive with what other processes claim to have received.

Byzantine Generals Problem

- This differs from consensus problem; a distinguished process (commander) supplies a value that others (soldiers) are to agree upon, instead of each of them proposing a value.



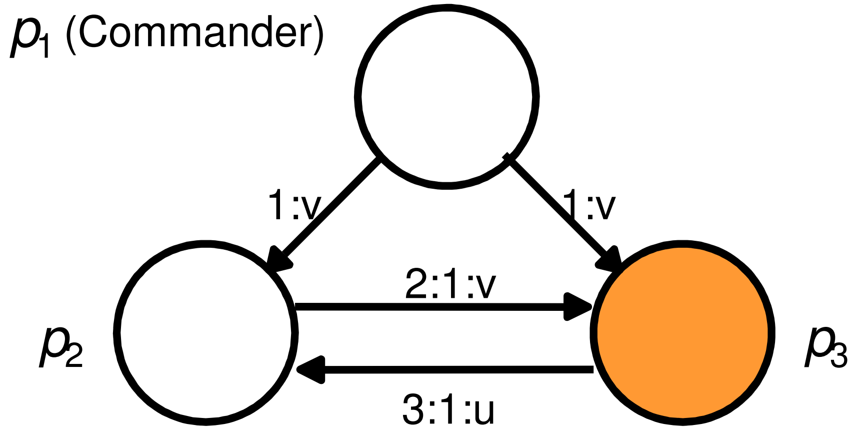
- Requirements:
  - Termination : Eventually each correct process sets its decision variable.
  - Agreement : The decision value of all correct processes is the same.
  - Integrity : If the commander is correct, then all correct processes decide on the value that the commander proposed.
    - This implies agreement when the commander is correct.
    - If the commander is faulty, soldiers have to ask each other what they heard from the commander.

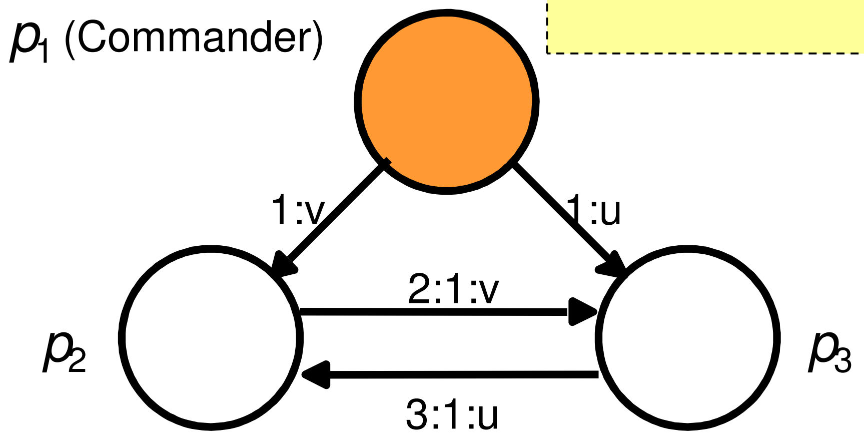


Byzantine generals with unsigned messages (1)

- Agreement cannot be reached with $N \leq 3f$ in an unsigned environment.
  - Unsigned environment contributes to arbitrary failure.

Even asking each other what one heard from the commander, correct soldier cannot be sure of the decision value in unsigned environment.



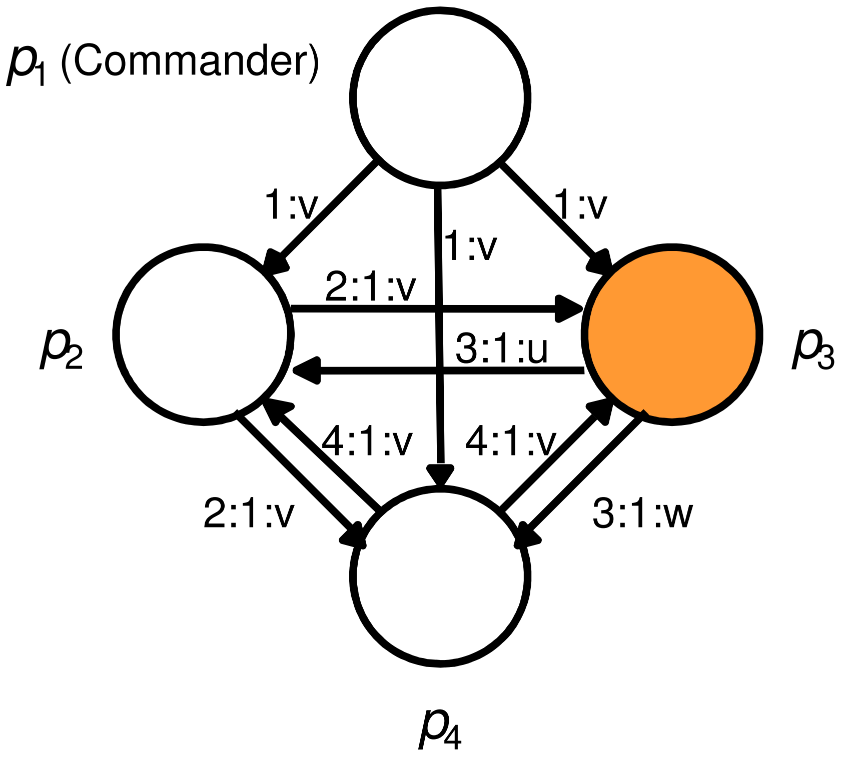


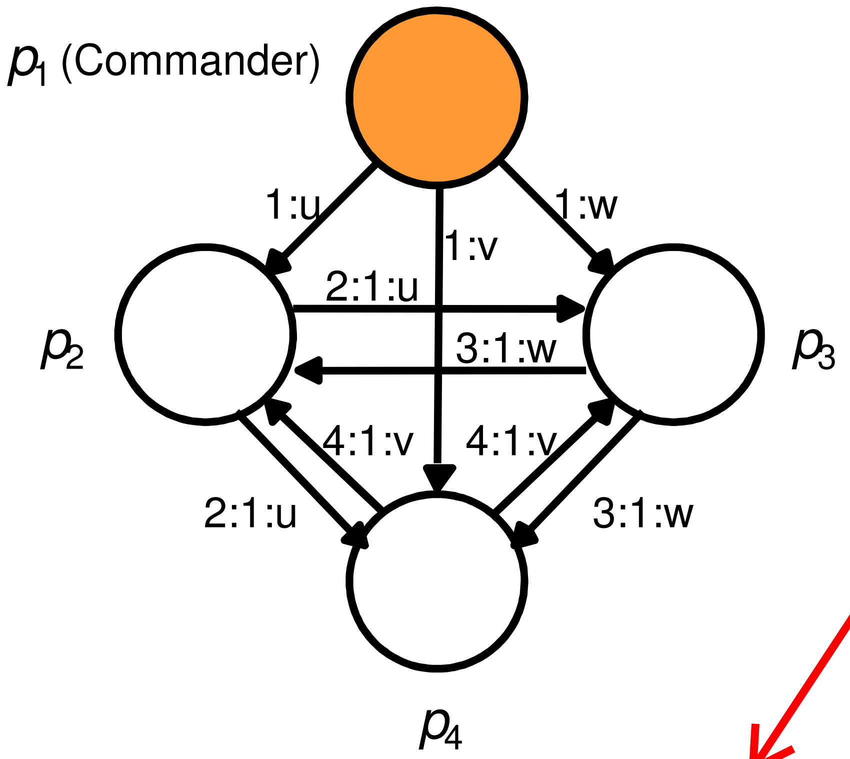
The faulty process may really be buggy and send a random message. Or it is correct but its unsigned message is altered in the channel.

Correct process P_2 cannot distinguish these two scenarios. If, by integrity, it believes the commander, then what if what is happening is the right-side? P_3 may also believe the commander, and so p_2 and p_3 do not agree on the same value.

Byzantine generals with unsigned messages (2)

- Agreement can be reached if N ≥ 3f + 1.





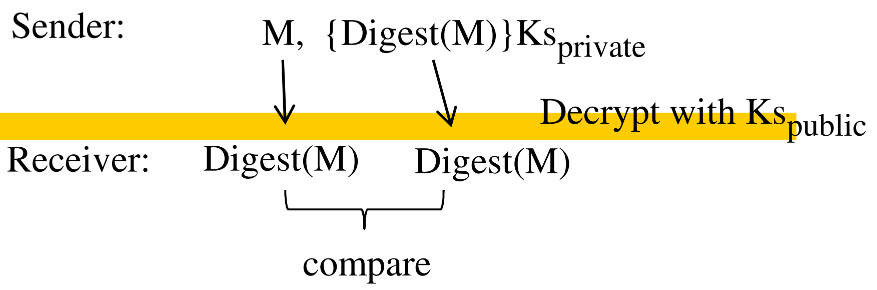
p_2 decides on majority(v, u, v) = v
p_4 decides on majority(v, v, w) = v

null

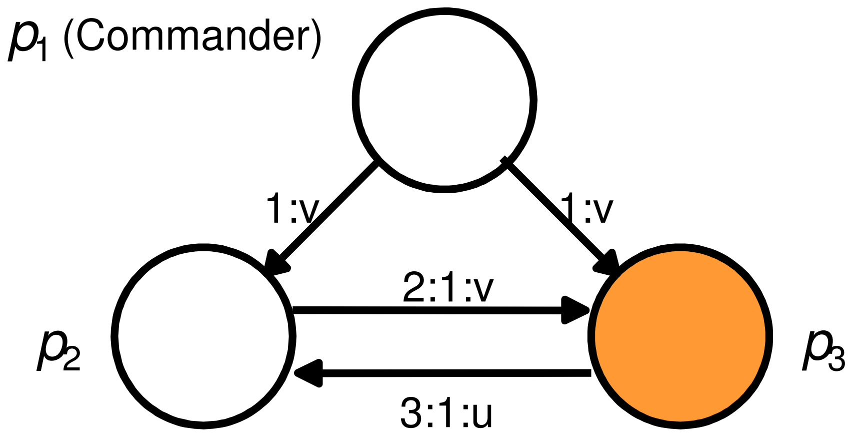
p_2 decides on majority(u, w, v) = ⊥
p_3 decides on majority(w, u, v) = ⊥
p_4 decides on majority(v, u, w) = ⊥

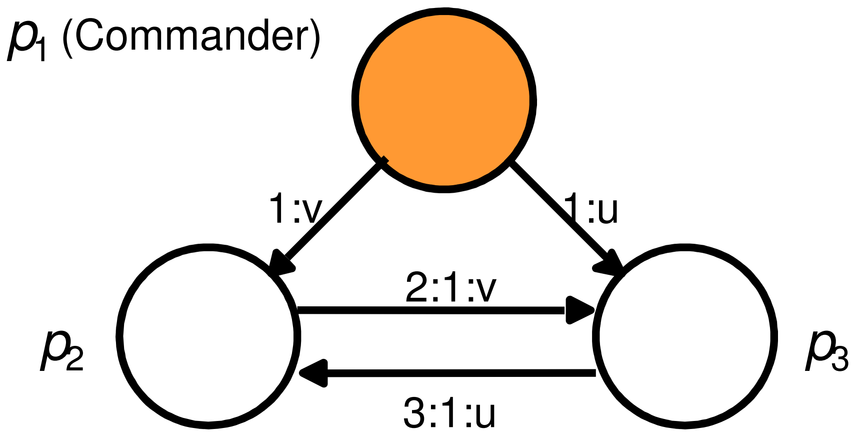
Correct processes agree on the same value.

Byzantine generals with signed messages



- Agreement can be reached.





If p_3 really says 3:1:u, or it says 3:1:v but the message is altered to 3:1:u, then p_2 could detect that either p_3 or the channel is faulty (p_2 can check if the message from p_3 can be decrypted with p_3's and p_1's keys.)

p_2 will believe the commander p_1.

p_2 can detect that either the commander p_1 or the channel is faulty. (It can check if the messages can be decrypted by p1's and p3's keys. And if so, the values are inconsistent).

Analogously for p_3.

p_2 and p_3 will agree on $\perp$.

Complexity on reaching agreement

- Solutions are complex and costly.

BUT unavoidable !!

- System design should be based on the fault model that is allowed or is likely to occur.
  - System design with crash failure only or with digital signatures would be simpler than that which allows arbitrary failure.