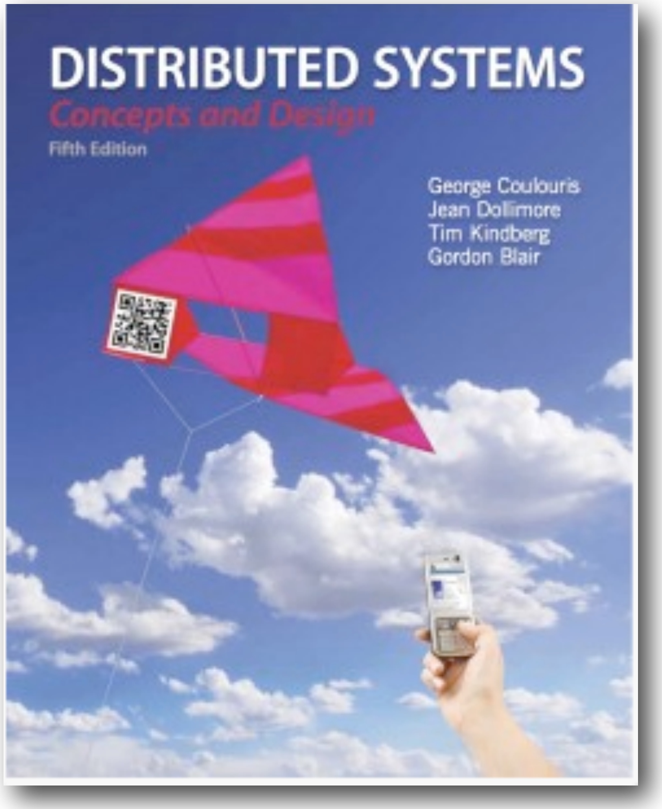
Coordination and Agreement

1 Introduction
2 Distributed Mutual Exclusion
3 Multicast Communication
4 Elections
5 Consensus and Related Problems



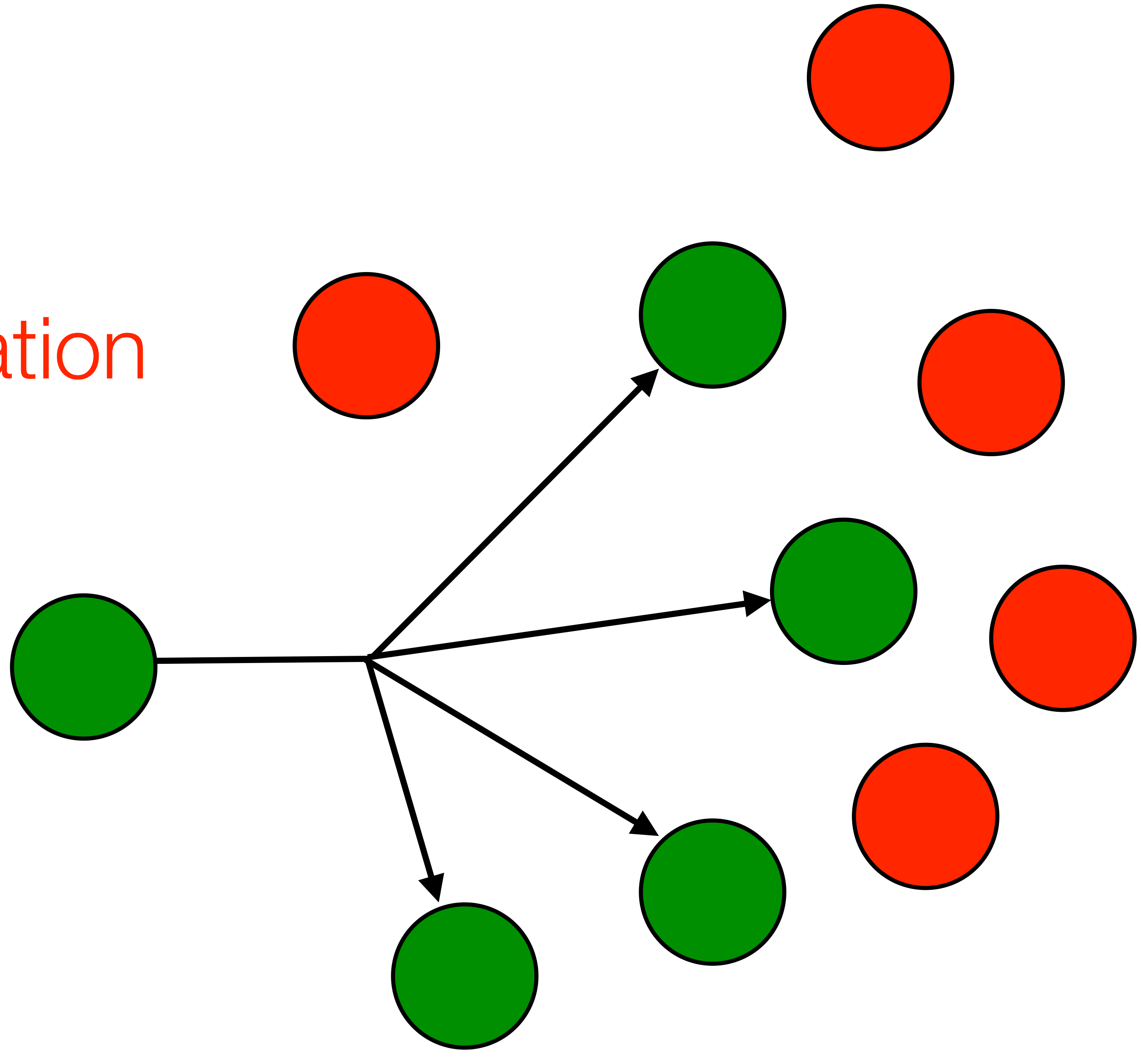
AIM: Coordination and/or Agreement

- Collection of algorithms whose goals vary

  but which share an aim that is fundamental in distributed systems

  for a set of distributed processes to coordinate their actions or to agree on one or more values

Multicast Communication



Group (or Multicast) Communication

- Some lectures ago... Java API to IP multicast: example of implementation of group communication

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[]){
         // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s =null;
         try {
                InetAddress group = InetAddress.getByName(args[1]);
                s = new MulticastSocket(6789);
                s.joinGroup(group);
                byte [] m = args[0].getBytes();
                DatagramPacket messageOut =
                        new DatagramPacket(m, m.length, group, 6789);
                s.send(messageOut);

         // this figure continued on the next slide
```

args[0] = msg contents
args[1] = multicast address

MulticastSocket creation on port 6789

joining a multicast group

sending a DatagramPacket message

Delivery Guarantees

- Group communication requires coordination and agreement

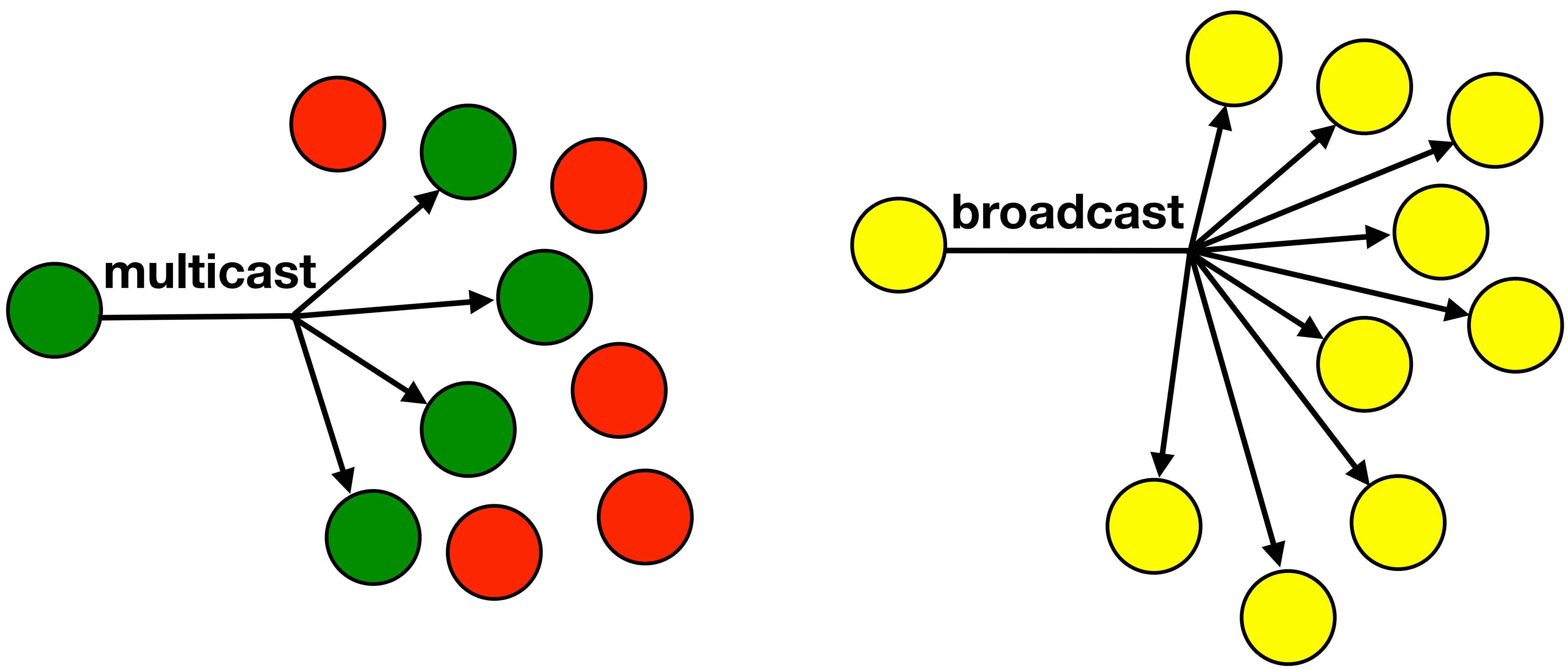
GOAL For each of a group of processes to receive copies of the messages sent to the group, satisfying some **delivery guarantees**

Delivery Guarantees

- **Agreement on the set of messages that every process in the group should receive**
- **Agreement on the delivery ordering across the group members**

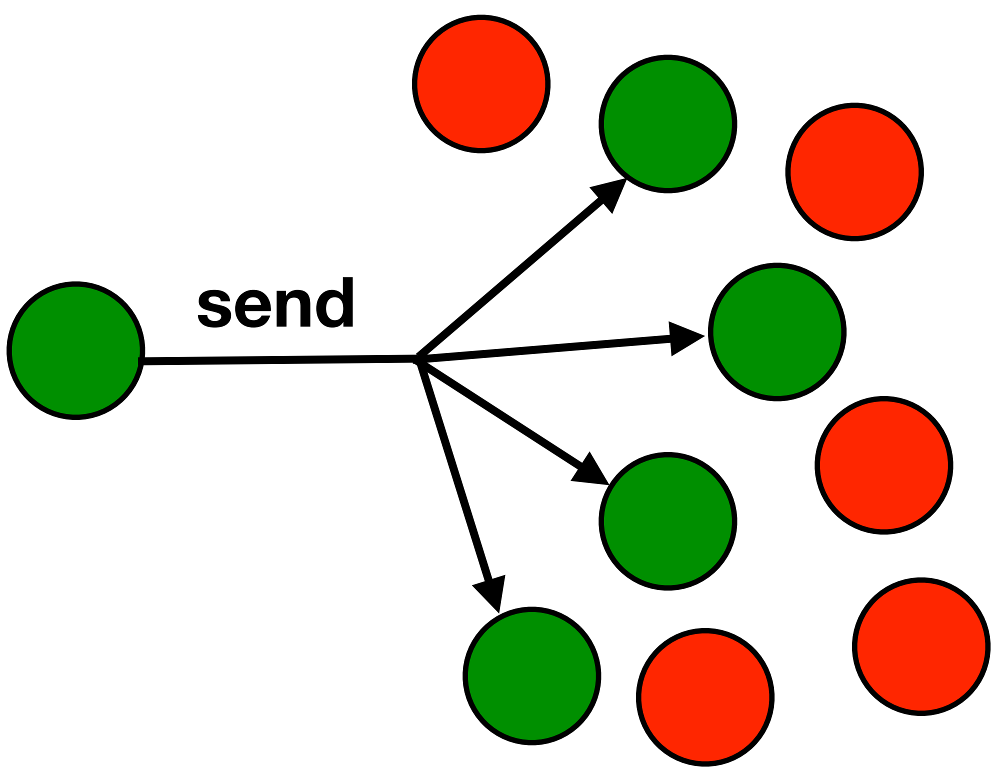
Multicast VS Broadcast

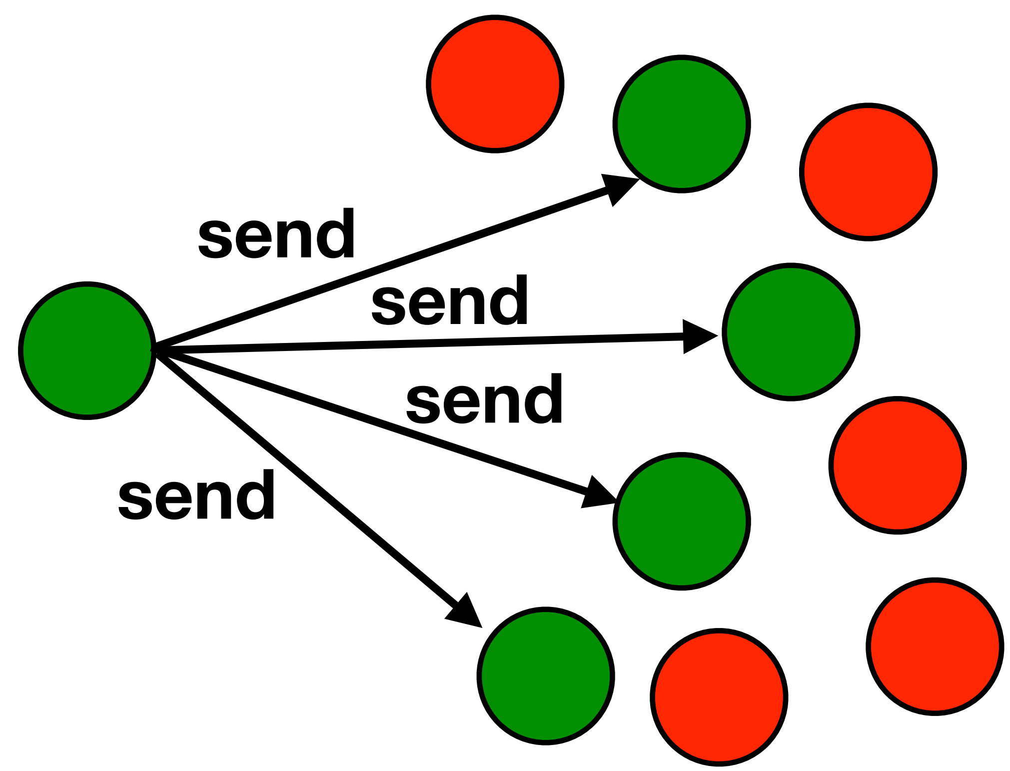
- Communication to all processes in the system, as opposed to a sub-group of them, is known as broadcast



Essential Feature

- **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





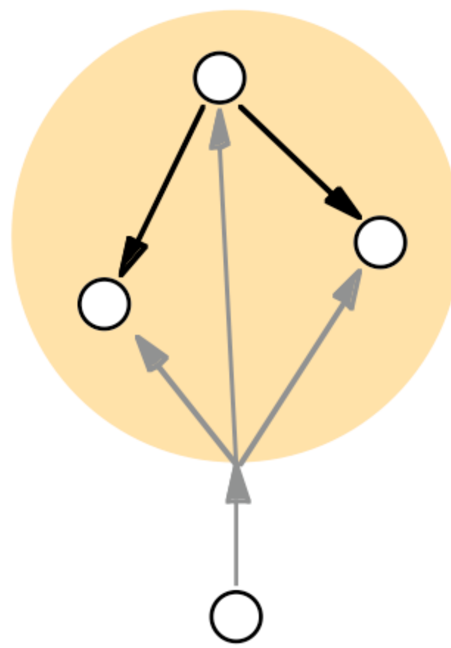
Example (from Java APIs)

- In Java, a multicast *send* primitive is provided by the MulticastSocket class: *aSocket.send(aMessage)*, where *aSocket* is an instantiated object of the class MulticastSocket (datagram interface to IP multicast)

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[]){
         // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s =null;
         try {
                InetAddress group = InetAddress.getByName(args[1]);
                s = new MulticastSocket(6789);
                s.joinGroup(group);
                byte [] m = args[0].getBytes();
                DatagramPacket messageOut =
                        new DatagramPacket(m, m.length, group, 6789);
                s.send(messageOut);
```

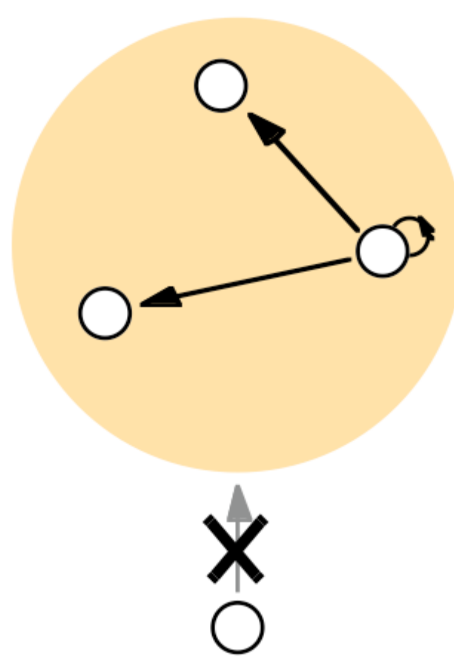
See lecture on Interprocess Communication
(--> JAVA API for IP Multicast)

Open VS Closed Group



Open Group:

Processes outside the group may send to it.



Closed Group:

Only members of the group can multicast to it.

A process delivers to itself any message that it multicasts to the group.

System Model

- Collection of processes, which communicate RELIABLY over 1-to-1 channels

Reliable (1-to-1) Communication (reliable 1-to-1 SEND primitive)

- **Validity**: if a correct process *p* sends a message *m* to a correct process *q*, then *q* eventually delivers *m*
- **No duplication**: no message is delivered by a process more than once
- **No creation**: if some process *q* delivers a message *m* with sender *p*, then *m* was previously sent to *q* by process *p*

- No Duplication + No Creation = Integrity property

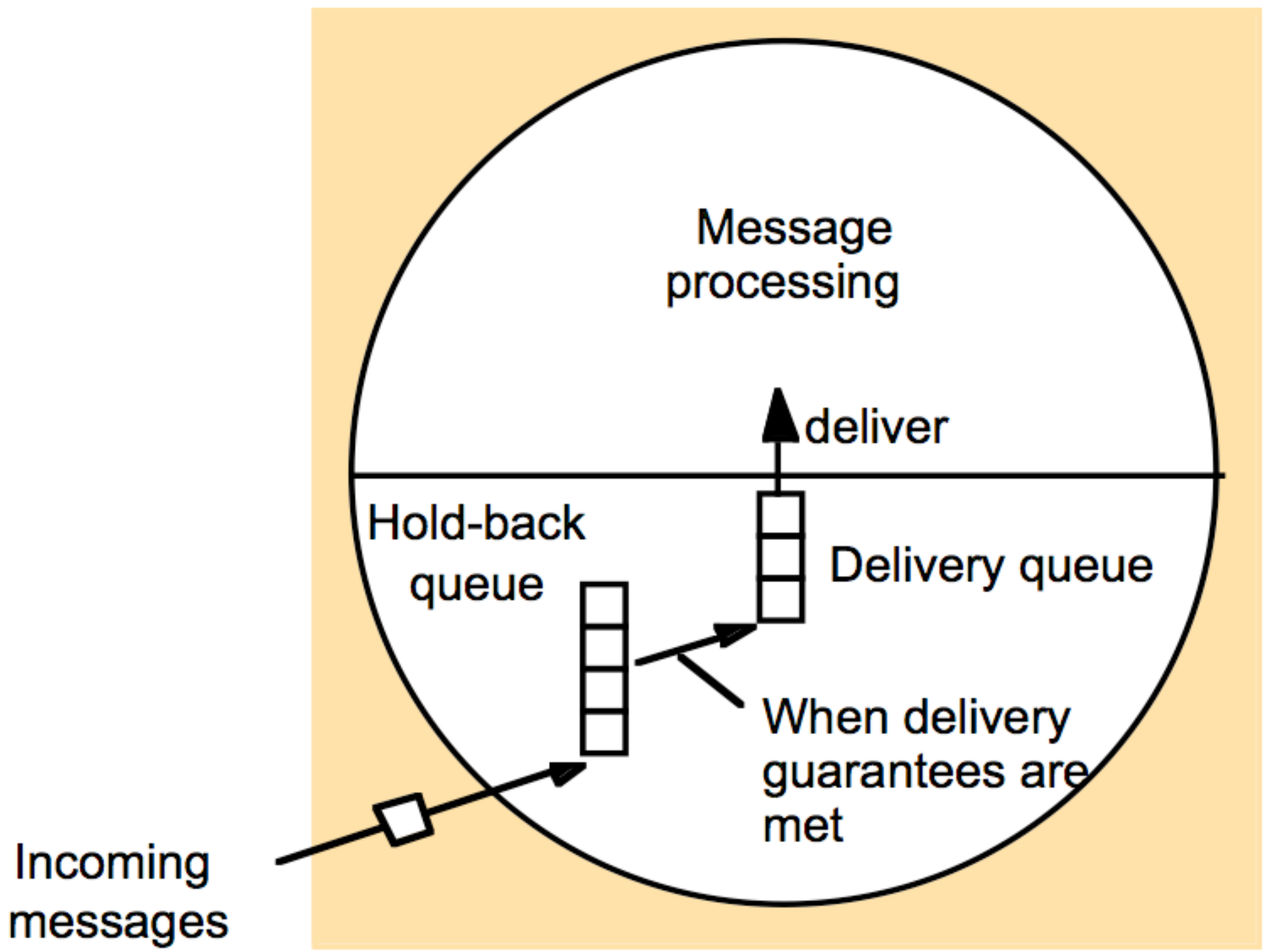
System Model (cont.)

- Processes may fail only by crashing
- Processes are members of groups, which are the destinations of messages sent with the *multicast* operation
- Communication primitives:
  - *multicast(g, m)*: sends a message *m* to all members of the group *g*
  - *deliver(m)*: delivers a message sent by multicast to the calling process

Why *deliver*
(and not *receive*)?

Message Delivery VS Message Receipt

A multicast message is not always handed to the application layer inside the process as soon as it is received at the process's node (it depends on the *multicast delivery semantics*...)



System Model (cont.)

- Every message *m* carries
  - the unique identifier of the process *sender(m)* that sent it
  - the unique destination group identifier *group(m)*
- We assume that processes do not lie about the origin or destinations of msgs

Basic Multicast

Basic Multicast - Specification

A **basic multicast** is one that satisfies the following properties:

- **Validity**: if a correct process multicasts message *m*, then every correct process eventually delivers *m*
- **No Duplication**: a correct process *p* delivers a message *m* at most once
- **No Creation**: if a correct process *p* delivers a message *m* with sender *s*, then *m* was previously multicast by process *s*

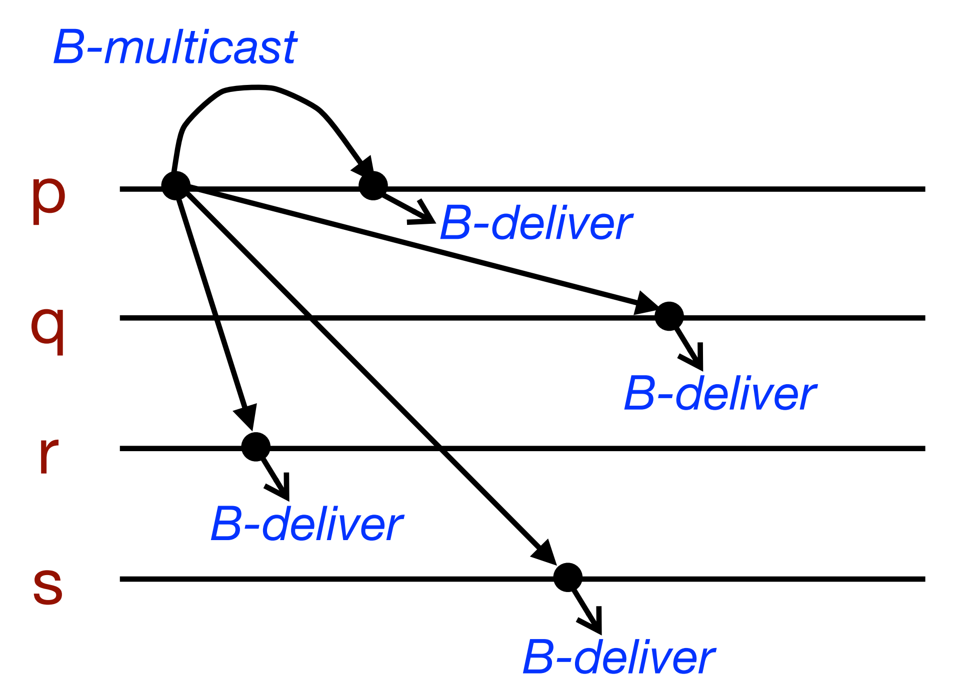
- Validity is a LIVENESS property (*something good eventually happens*)
- No Duplication and No Creation are SAFETY properties (*nothing bad happens*)
- No Duplication + No Creation = Integrity property

Basic Multicast - Algorithm

- Communication primitives:
  - *B-multicast*: basic multicast primitive
  - *B-deliver*: basic delivery primitive
- Implementation based on reliable 1-to-1 *send* operation:

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

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



Correctness of Basic Multicast Algorithm

A **basic multicast** is one that satisfies the following properties:

- **Validity**: if a correct process multicasts message *m*, then every correct process eventually delivers *m*
- **No Duplication**: a correct process *p* delivers a message *m* at most once
- **No Creation**: if a correct process *p* delivers a message *m* with sender *s*, then *m* was previously multicast by process *s*

- **Correctness** means that **a basic multicast algorithm must satisfy the validity, no duplication and no creation properties**
  - *Derived from the properties of the underlying RELIABLE channels*

Correctness of Basic Multicast: No Creation

- Properties derived from the properties of the underlying RELIABLE channels

  - B-multicast is based on *1-to-1 reliable send* primitive

  - No Creation [reliable channel]: if some process *q* delivers a message *m* with sender *p*, then *m* was previously sent to *q* by process *p*

No Creation [B-multicast]: if a correct process *p* delivers a message *m* with sender *s*, then *m* was previously multicast by process *s*

Correctness of Basic Multicast: No Duplication

- Properties derived from the properties of the underlying RELIABLE channels
  - No Duplication [reliable channel]: no message is delivered by a process more than once

No Duplication [B-multicast]: a correct process *p* delivers a message *m* at most once

Correctness of Basic Multicast: Validity

- Properties derived from the properties of the underlying RELIABLE channels
  - the sender sends the msg to every other process in the group (by means of a reliable 1-to-1 *send* primitive)
  - the *validity property* of the communication channels: if a correct process *p* sends a message *m* to a correct process *q*, then *q* eventually delivers *m*

Validity [B-multicast]: if a correct process multicasts message *m*, then every correct process eventually delivers *m*

Basic Multicast: Ack-Implosion Problem

- The implementation may use threads to perform the *send* operations concurrently, in an attempt to reduce the total time taken to deliver the msg

- Liable to suffer from ACK-IMPLOSION if the number of processes is large

  - The acknowledgements sent as part of the reliable *send* operation are liable to arrive from many processes at about the same time

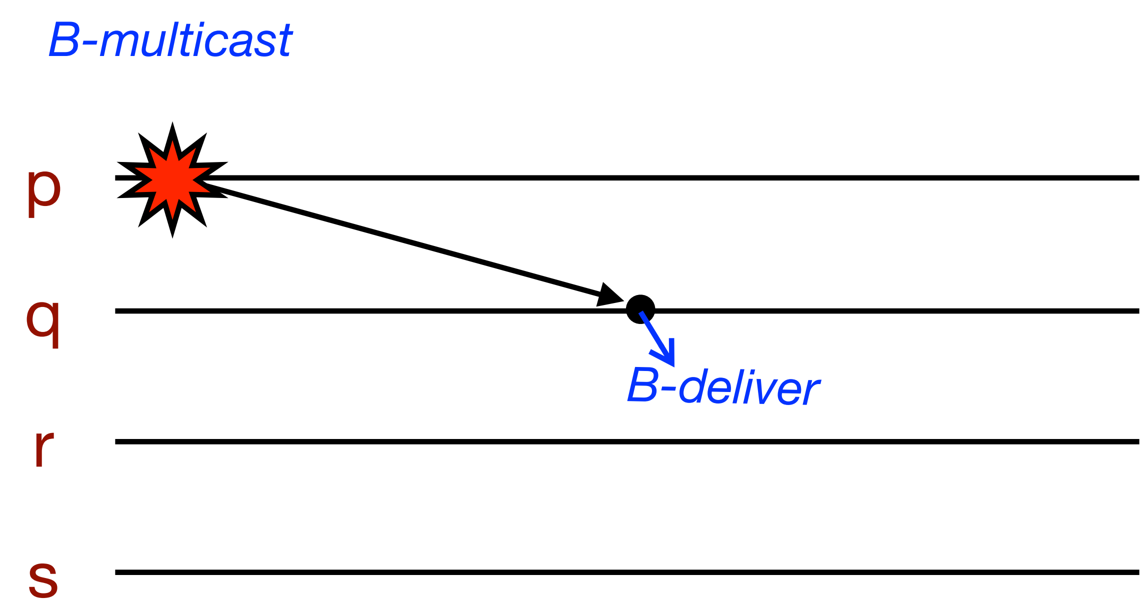
  - The multicasting process's buffer will rapidly fill and it is liable to drop acknowledgments

    ➡ It will therefore retransmit the msg, leading to yet more acks and further waste of network bandwidth

Scenario: Faulty Sender

- If the sender fails, some processes might deliver the message and other might not deliver it

THE PROCESSES DO NOT *AGREE* ON THE DELIVERY OF THE MESSAGE!



- (Actually, even if the process sends the msg to all processes BEFORE crashing, the delivery is NOT ensured because reliable channels do not enforce the delivery when the sender fails!!)

We want to ensure AGREEMENT even when the sender fails

Reliable Multicast

Reliable Multicast - Specification

- Based on 2 primitives: *R-multicast* and *R-deliver*

A **reliable multicast** is one that satisfies the following properties:

- **No Duplication**: a correct process *p* delivers a message *m* at most once
- **No Creation**: if a correct process *p* delivers a message *m* with sender *s*, then *m* was previously multicast by process *s*
- **Validity**: if a correct process multicasts message *m* then it will eventually deliver *m*
- **Agreement**: if a correct process delivers message *m*, then all the other correct processes in *group(m)* will eventually deliver *m*

- **Validity** --> **Liveness** for the sender
- **Validity** + **Agreement** --> **Liveness** for the system

Reliable Multicast - Algorithm

- Implemented over B-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*)
    then
        Received := *Received* $\cup$ { *m* };
        if ($q \neq p$) *then B-multicast(g, m); end if*
        R-deliver m;
    end if

To *R-multicast* a message, a process *B-multicasts* the message to the processes in the destination group (including itself)

Reliable Multicast - Algorithm

- Implemented over B-multicast

When the message is *B-delivered*:
- the recipient in turn *B-multicasts* the message to the group (if it is not the original sender)
- then it *R-delivers* the message

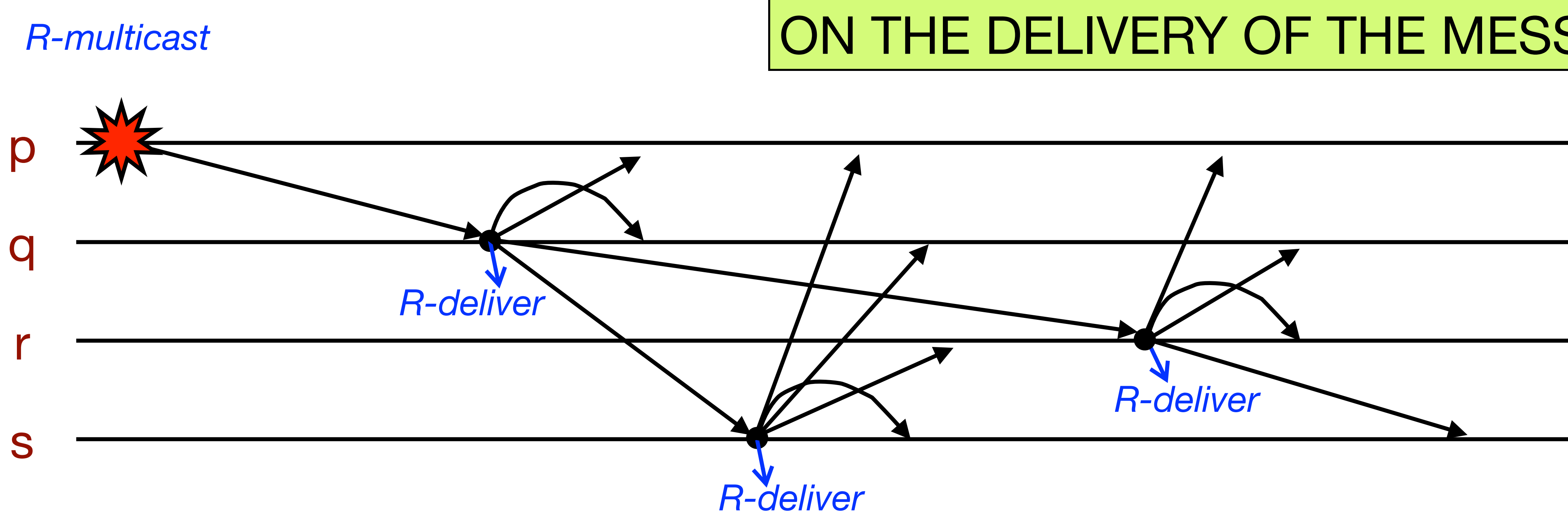
```
On initialization
    Received := {};

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

On B-deliver(m) at process q with g = group(m)
    if (m ∉ Received)
    then
                Received := Received ∪ {m};
                if (q ≠ p) then B-multicast(g, m); end if
                R-deliver m;
    end if
```

since a message may arrive more than once, duplicates of the message are detected and not delivered

Scenario: Faulty Sender

- process *p* crashes and its message is not *B-delivered* by processes *r* and *s*
- however, process *q* retransmits the message (i.e., *B-multicast* it)
- consequently, the remaining *correct* processes also *B-deliver* it and subsequently *R-deliver* it

THE CORRECT PROCESSES *AGREE* ON THE DELIVERY OF THE MESSAGE!



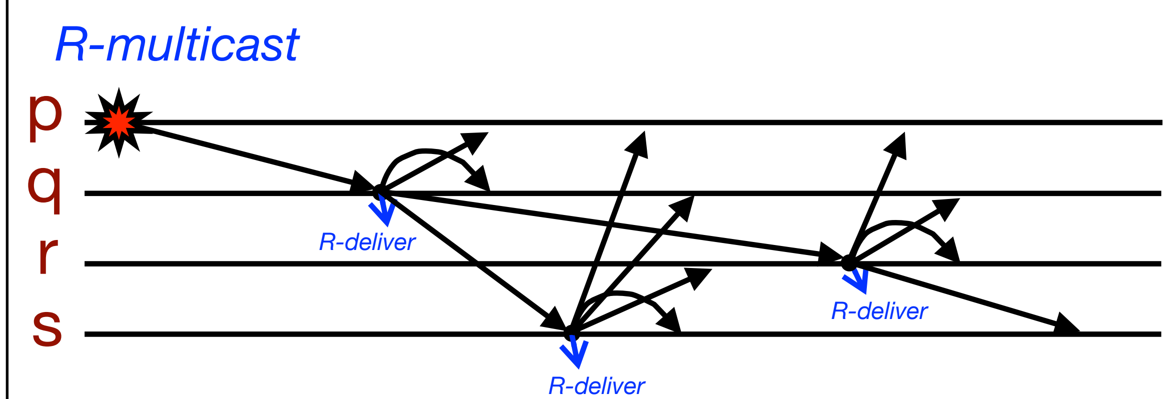
On the Agreement Property: Atomicity

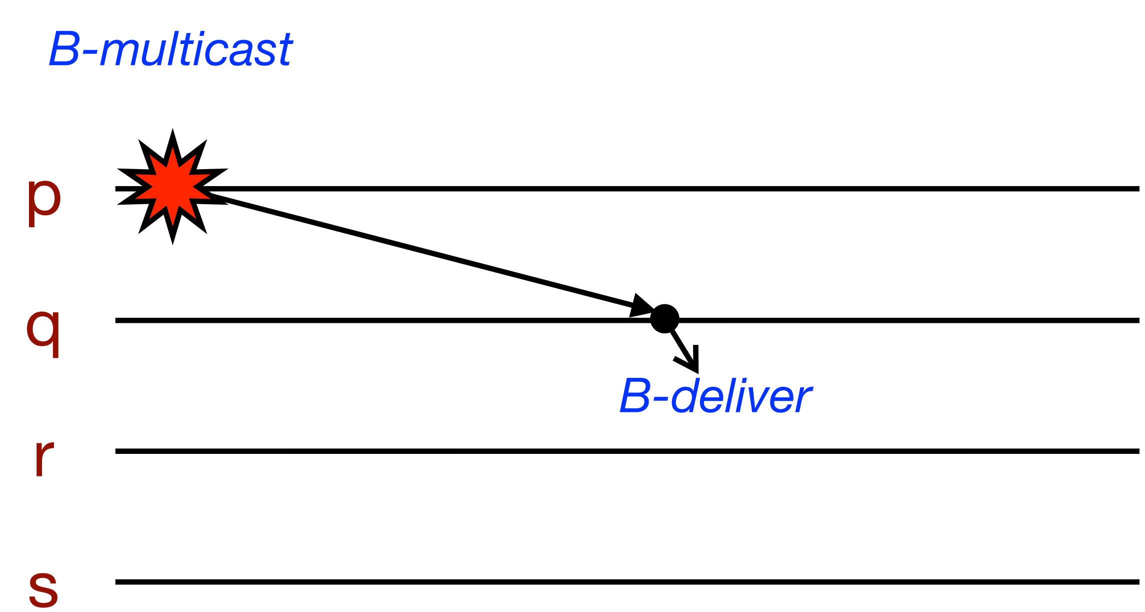
Property of "atomicity": all or nothing

- *If a process that multicasts a message crashes before is has delivered it, than it it is possible that the message will not be delivered to any process in the group*
- *But if it is delivered to some correct process, then all the other correct processes will deliver it*

Note: **NOT** a property of the B-multicast algorithm!

The sender may fail at any point while *B-multicast* proceeds, so some processes may deliver a msg while others do not





Algorithm Analysis + HOMEWORK

- The algorithm satisfies validity, since a correct process will eventually *B-deliver* the message to itself

- The algorithm satisfies integrity, because of

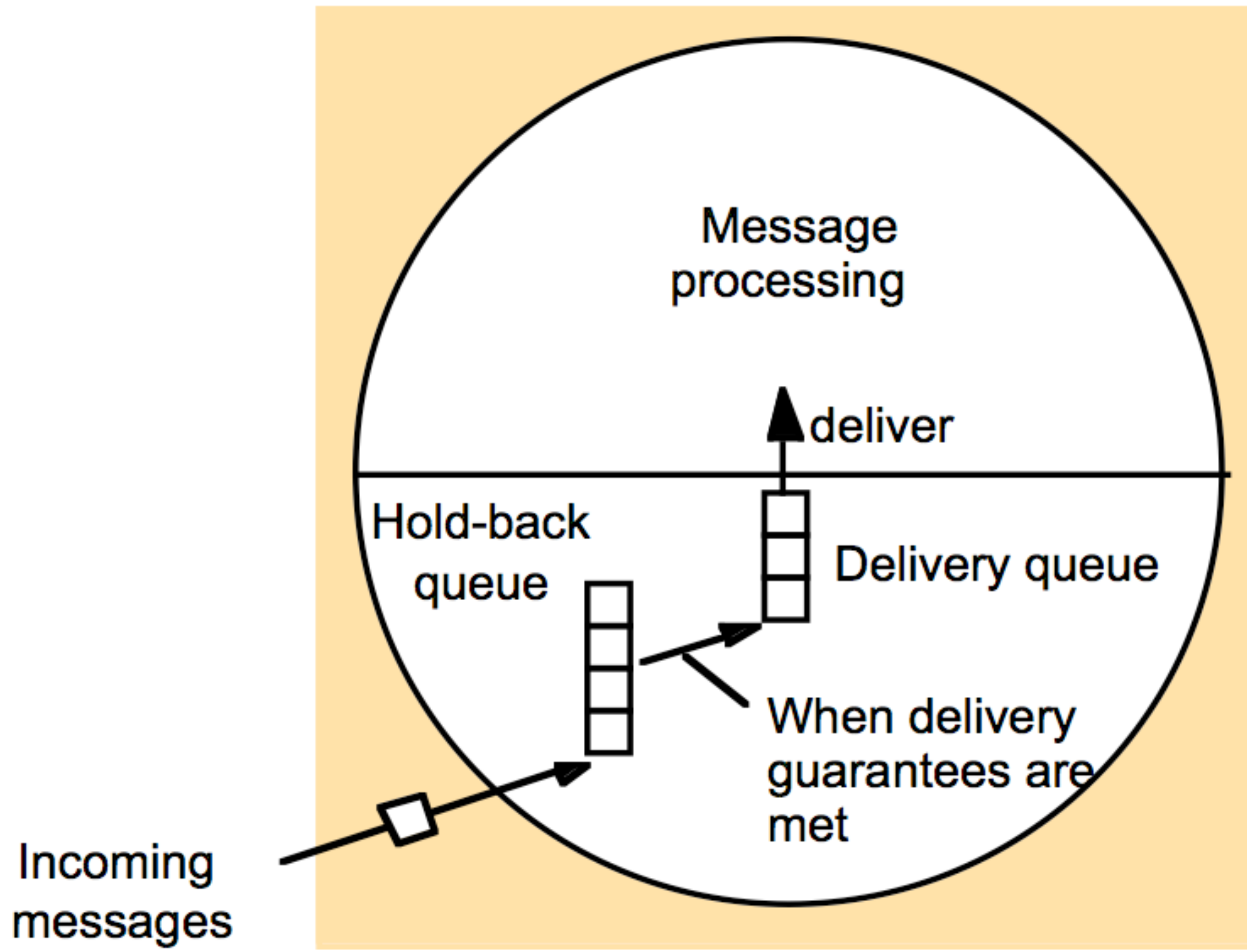
  (1) the integrity property of the underlying communication channels

  (2) the fact that duplicates are not delivered

  What about agreement? It follows because... HOMEWORK! :-)

- The algorithm is **correct** in an asynchronous system BUT **inefficient** for practical purpose: each message sent |g| times to each process ($O(|g|^2)$ messages)

Ordered Multicast

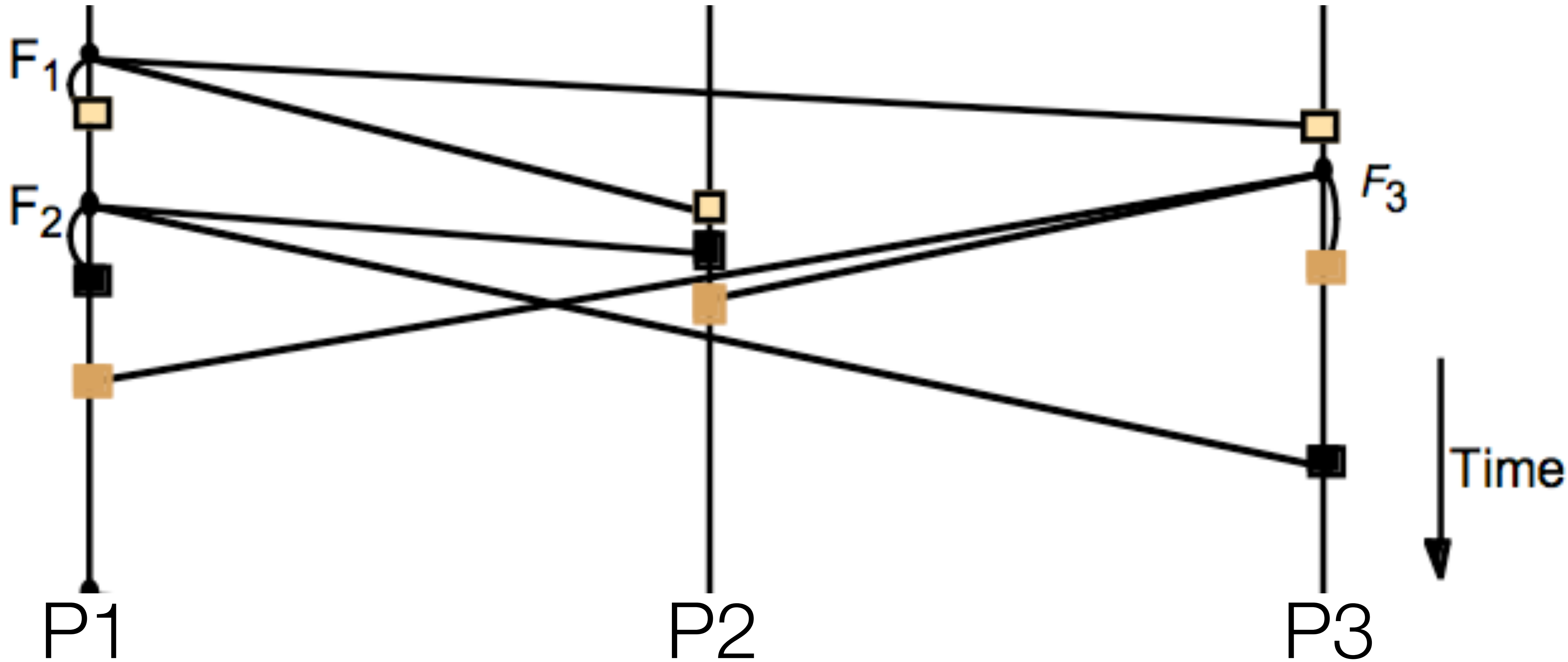


Ordered Multicast

- The B- and R- multicast algorithms deliver messages to processes in an arbitrary order, due to arbitrary delays in the 1-to-1 *send* operations
- Common **ordering requirements**:
  - **FIFO ordering**: if a correct process issues *multicast(g, m)* and then *multicast(g, m')* (*multicast(g, m)* $\rightarrow_i$ *multicast(g, m')*), then every correct process that delivers *m'* will deliver *m* before *m'*; partial relation
  - **Causal ordering**: *multicast(g, m)* $\rightarrow$ *multicast(g, m')*, then any correct process that delivers *m'* will deliver *m* before *m'*; partial relation
  - **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'*.
- N.B.: *causal ordering implies FIFO ordering*

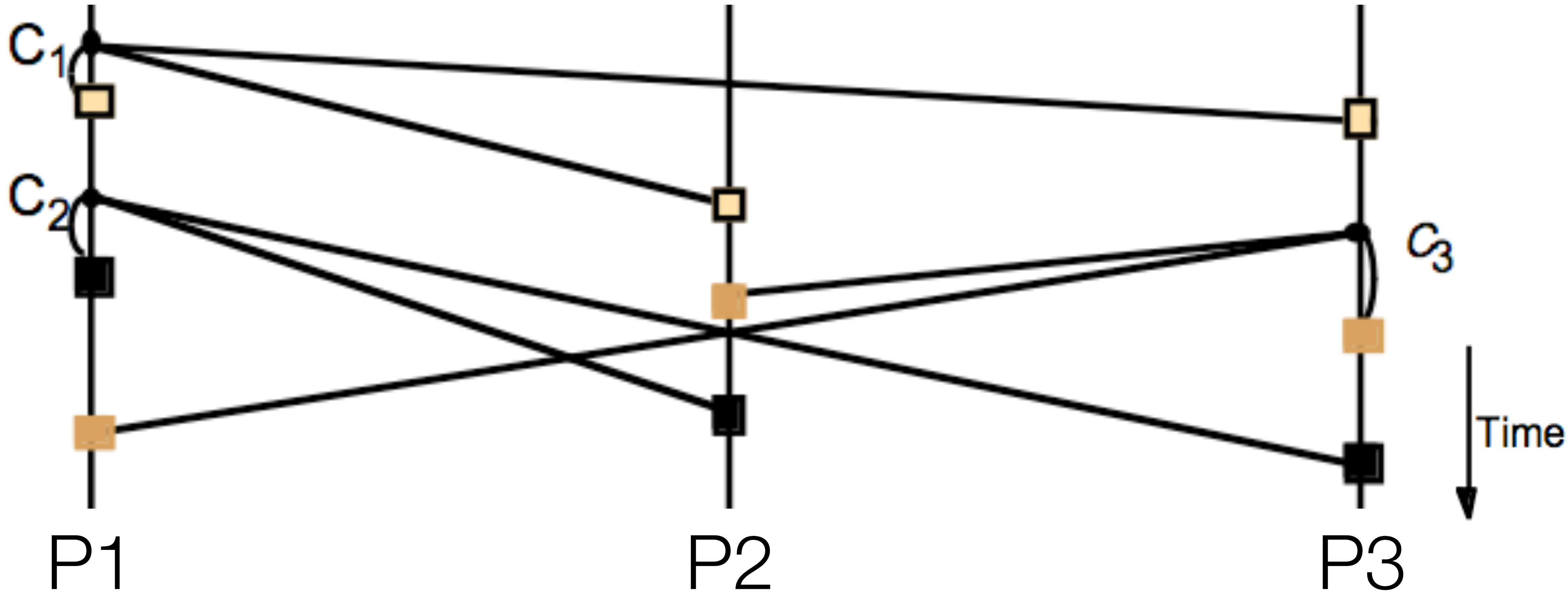
Example: FIFO Ordering

- FIFO ordering: if a correct process p_i issues *multicast(g, m)* and then *multicast(g, m')* (*multicast(g, m)* $\rightarrow_i$ *multicast(g, m')*), then every correct process that delivers *m'* will deliver *m* before *m'*



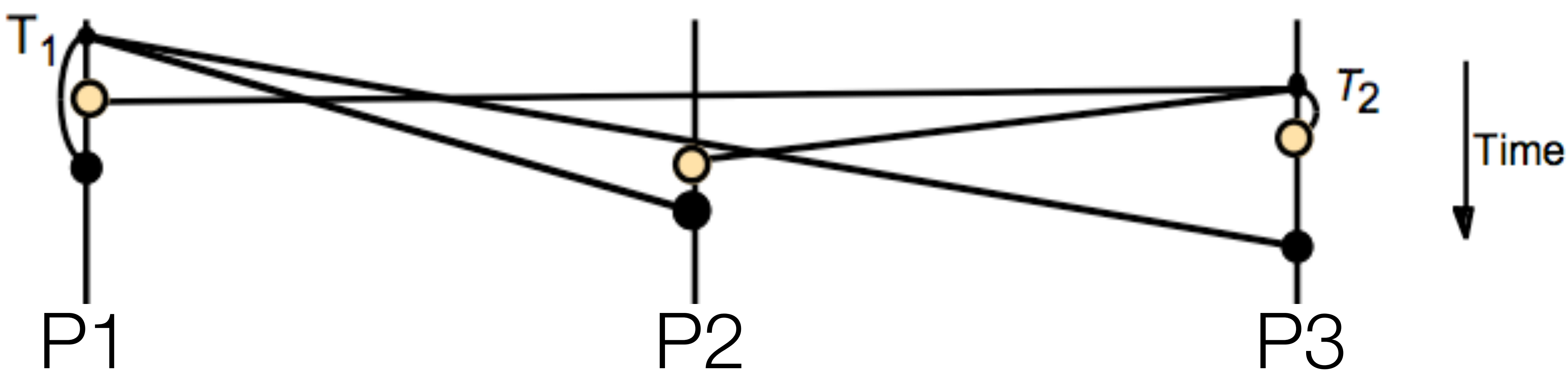
Example: Causal Ordering

- Causal ordering: *multicast(g, m)* → *multicast(g, m')*, then any correct process that delivers *m'* will deliver *m* before *m'*



Example: Total Ordering

- 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'*



Example: Bulletin Board

- Consider an application in which users post messages to bulletin boards
- Each user runs a bulleting-board application process
- Every topic of discussion has its own process group
- When a user posts a message to a bulletin board, the application multicasts the user's posting to the corresponding group
- Each user's process is a member of the group for the topic he/she is interested ==> the user will receive just the postings concerning that topic

[Bulletin Board] Ordering Requirements

- Reliable multicast required if every user is to receive every posting eventually

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		

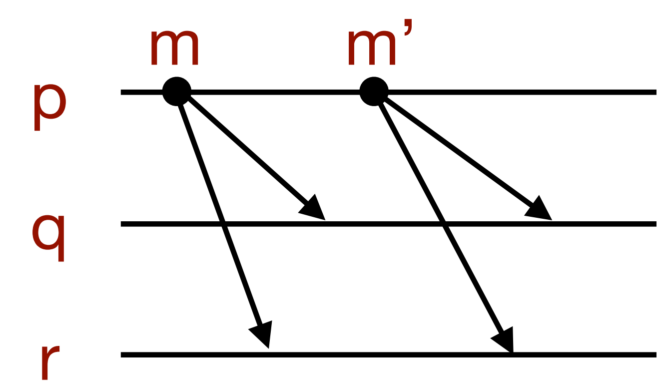
FIFO ordering desirable since then every posting from a given user will be received in the same order

Causal ordering needed to guarantee this relationship

If multicast delivery was totally ordered, then the items would be consistent between the users (users could refer unambiguously, for example, to "message 24")

Implementing FIFO Ordering

FIFO ordering: if a correct process p_i issues *multicast(g, m)* and then *multicast(g, m')* (*multicast(g, m)* $\rightarrow_i$ *multicast(g, m')*), then every correct process that delivers *m'* will deliver *m* before *m'*



- Two primitives: *FO-multicast* and *FO-deliver*
- Achieved with sequence numbers
- We assume non-overlapping groups
- A process p has variables (storing *sequence numbers*):
  - S^p_g : how many messages p has sent to g
  - D^q_g : sequence number of the latest message p has delivered from process q that was sent to g

Basic FIFO Multicast: FO-Multicast and FO-Deliver

- For p to *FO-multicast* a message to group g:

  it piggy backs the value S^p_g onto the message;

  it *B-multicasts* the message to g;

  $S^p_g = S^p_g + 1$

If we use *R-multicast* instead of *B-multicast*, then we obtain a reliable FIFO multicast

- Upon a receipt of a message from q bearing the seq. number S, p checks:

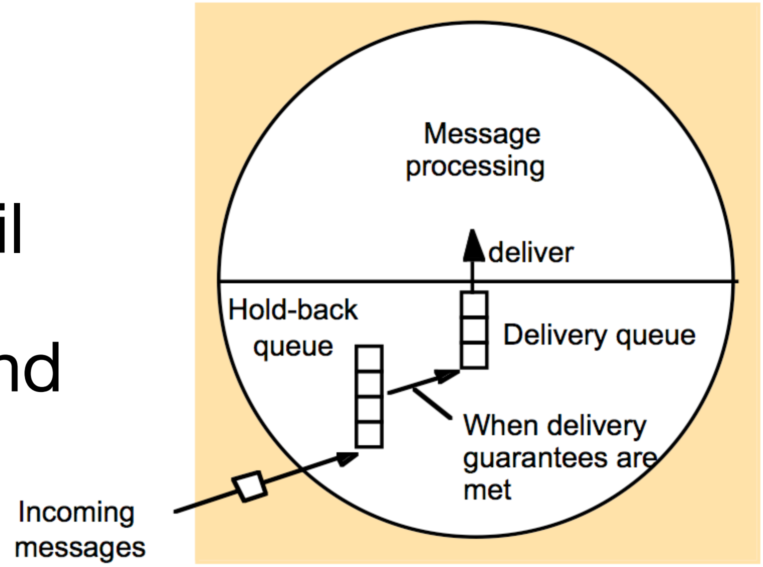
  IF ($S = D^q_g + 1$) **THEN** it *FO-delivers* the message, setting $D^q_g := S$

  ELSIF ($S > D^q_g + 1$) **THEN**

  it places the message in its *hold-back queue* until

  the intervening messages have been delivered and

  $S = D^q_g + 1$



Condition for FIFO Ordering Satisfied Because...

- Upon a receipt of a message from q bearing the seq. number S, p checks:

  IF ($S = D^q_g + 1$) **THEN** it *FO-delivers* the message, setting $D^q_g := S$

  ELSIF ($S > D^q_g + 1$) **THEN** it places the message in its *hold-back queue* until the intervening messages have been delivered and $S = D^q_g + 1$

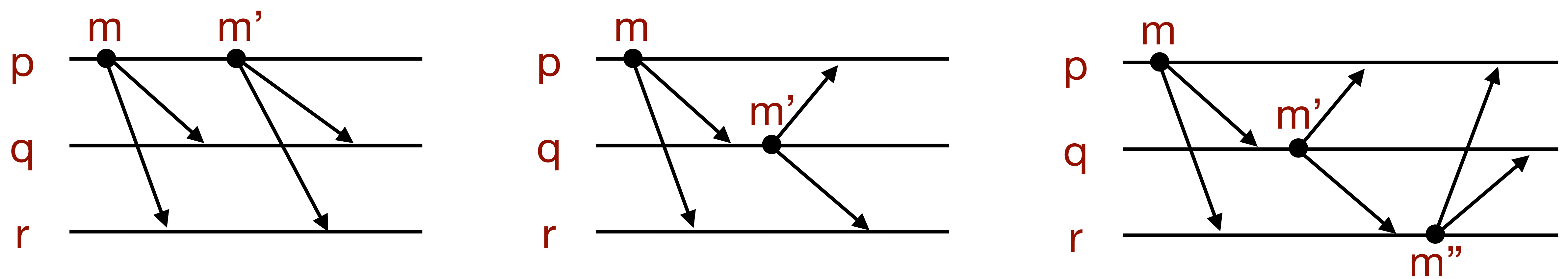
1. All messages from a given sender are delivered in the same sequence

2. Delivery of a message is delayed until its sequence number has been reached

- N.B.: this is so only under the assumption that *groups are NON-overlapping!*

Implementing Causal Ordering

Causal ordering: *multicast(g, m)* → *multicast(g, m')*, then any correct process that delivers *m'* will deliver *m* before *m'*



- Algorithm for *non-overlapping closed groups* (Birman et al., 1991)
- It takes into account of the happened-before relationship only as it is established by multicast messages
- Each process maintain its own vector timestamp: the entries count the number of multicast messages from each process that happened-before the next message to be multicast

Causal Ordering Using Vector Timestamps

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

On initialization

$V_i^g[j] := 0$ $(j = 1, 2..., N)$;

To CO-multicast message m to group g

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

B-multicast$(g, <V_i^g, m>)$;

the process add 1 to its entry in the timestamp and *B-multicasts* the msg along with its timestamp to g

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 $V_j^g[k] \leq V_i^g[k]$ $(k \neq j)$;

CO-deliver m; // after removing it from the hold-back queue

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

Causal Ordering Using Vector Timestamps

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

On initialization

$V_i^g[j] := 0$ $(j = 1, 2..., N)$;

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 $V_j^g[k] \le V_i^g[k]$ $(k \neq j)$;

CO-deliver m; // after removing it from the hold-back queue

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

p_i has delivered any message that p_j had delivered

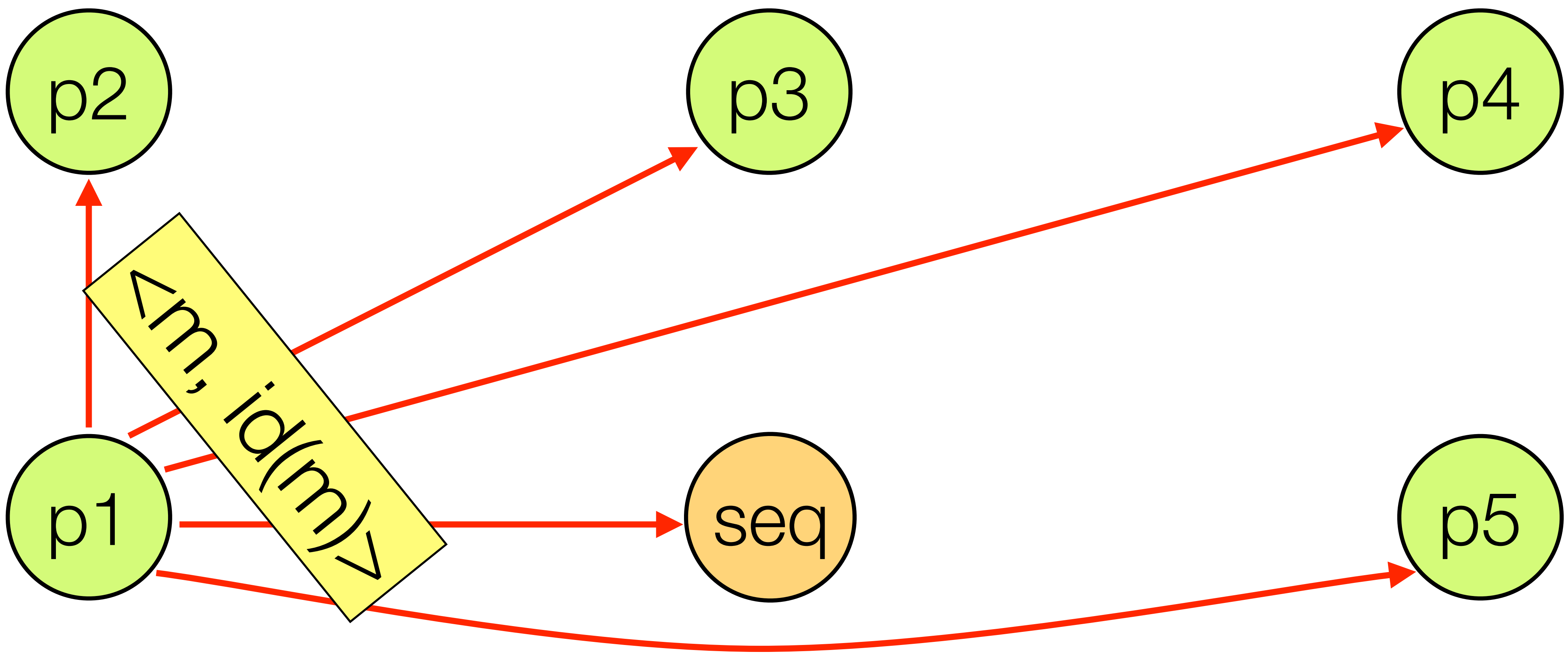
p_i has delivered any earlier message sent by p_j

Implementing Total Ordering

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'*

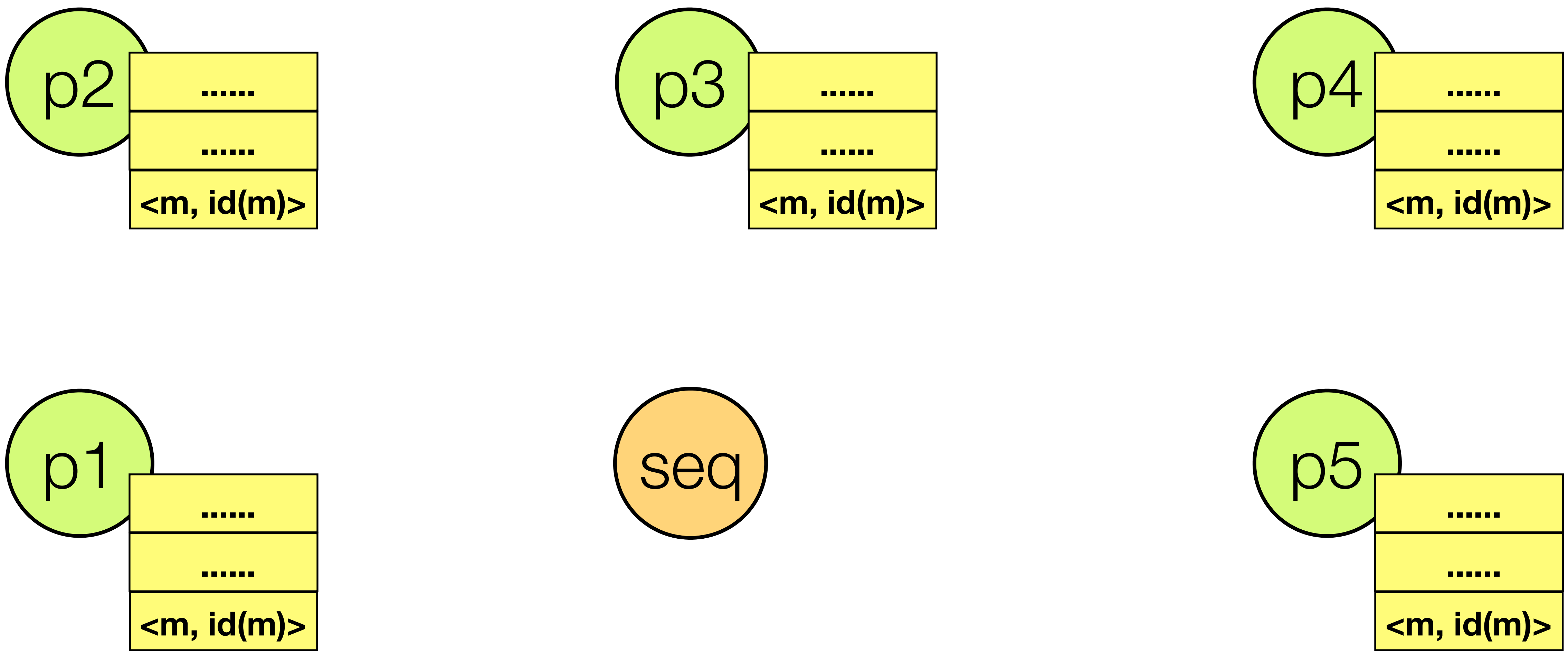
- We assume *non-overlapping groups*
- Key idea: to assign totally ordered identifiers to multicast messages so that each process makes the same ordering decision based upon these identifiers
- How: processes keep **group-specific sequence numbers** (rather than *process-specific sequence numbers* as for FIFO ordering)
- Key question: how to assign sequence numbers to messages?
- Two possible approaches: (central) sequencer or distributed agreement

Total Ordering Using a Sequencer



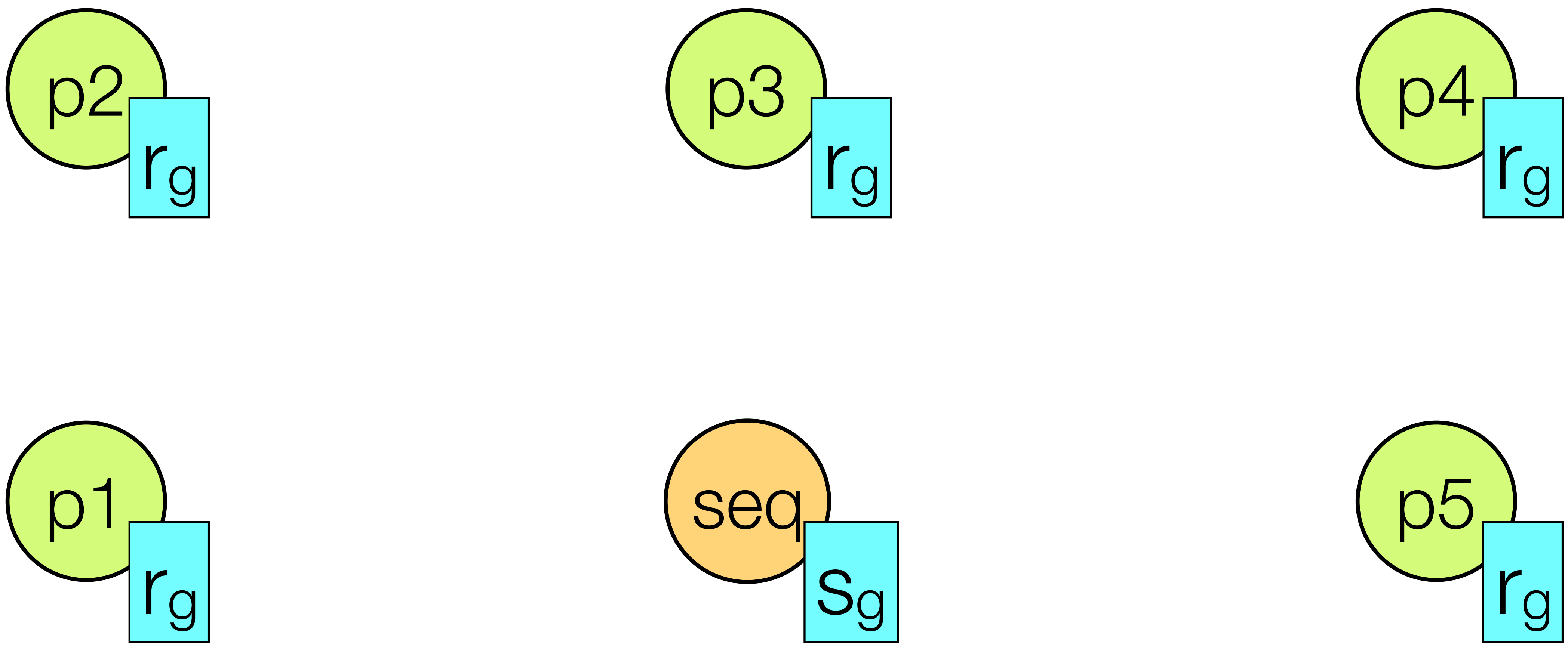
- To *TO-multicast* a message m to a group g, p1 attaches a unique identifier id(m) to it
- The messages for g are sent to the sequencer for g as well as to the members of g (the sequencer may be chosen to be a member of g)

Total Ordering Using a Sequencer



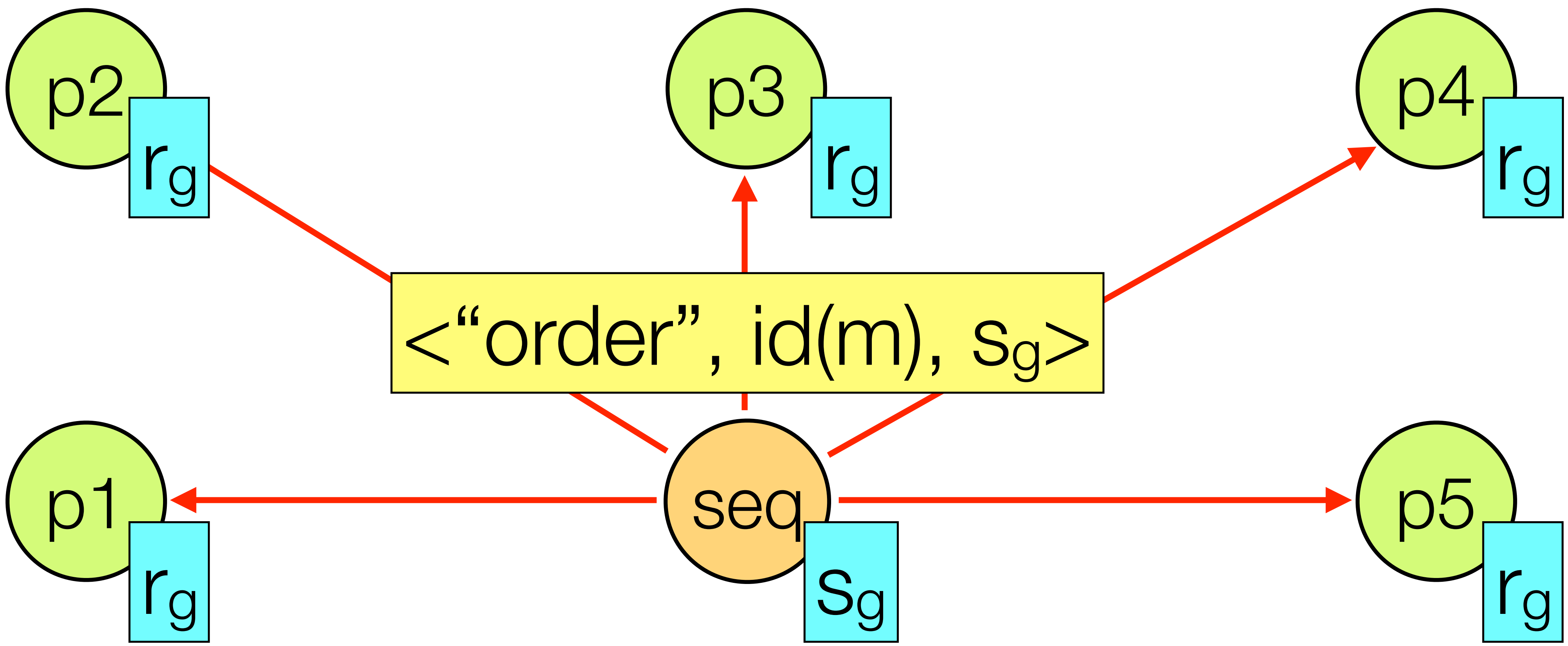
- On *B-deliver(<m, id(m)>)* a process (but NOT THE SEQUENCER) places the message <m, id(m)> in its hold-back queue

Total Ordering Using a Sequencer



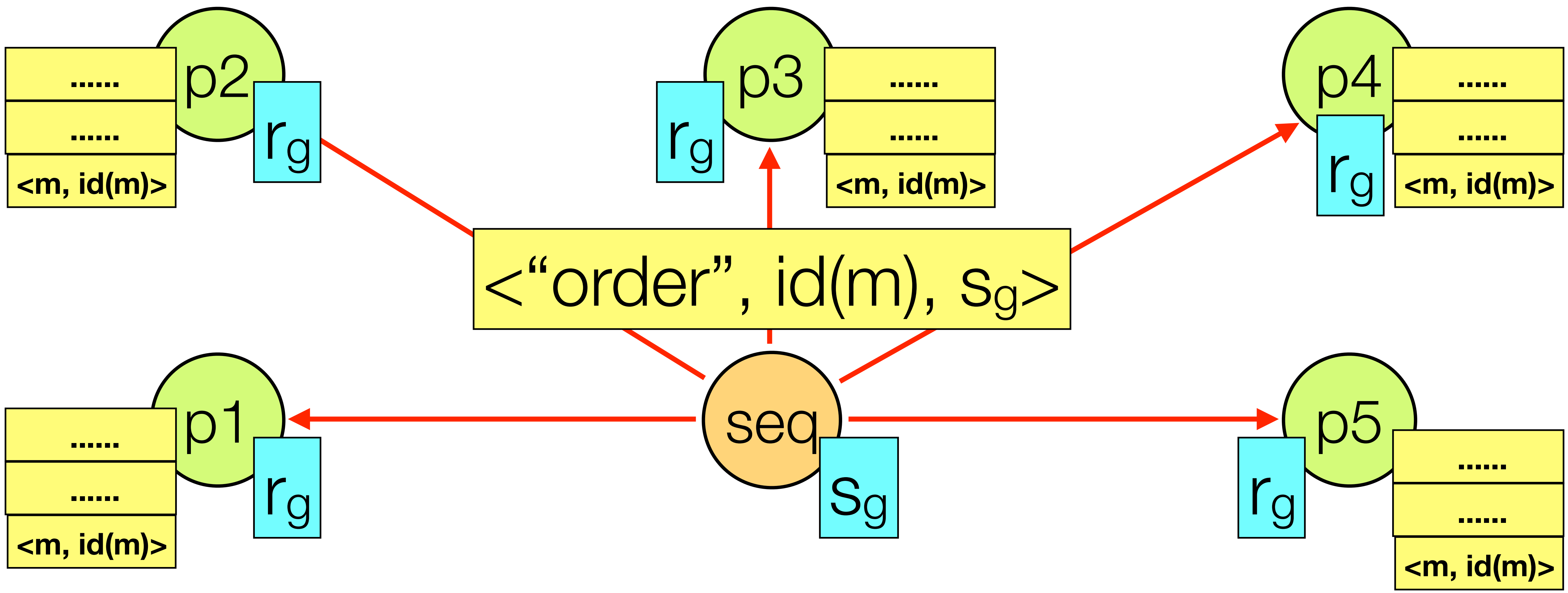
- The sequencer maintains a group-specific sequence number s_g, which it uses to assign increasing and consecutive sequence numbers to the messages that it *B-delivers*
- Processes have their local group-specific sequence number r_g

Total Ordering Using a Sequencer



- On *B-deliver(<m, id(m)>)* the sequencer announces the sequence numbers by *B-multicasting* "order" messages to g

Total Ordering Using a Sequencer



- A message will remain in a hold-back queue indefinitely until it can be *TO-delivered* according to the corresponding sequence number (**$s_g = r_g$**)

Total Ordering Using a Sequencer - Algorithm

- 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>);
$s_g := s_g + 1$;

N.B.: since the sequence numbers are well defined by the sequencer, the criterion of total ordering is met.

- 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)
$r_g = S + 1$;

Total Ordering Using Distributed Agreement

- The obvious problem with a sequencer-based approach is that the sequencer may become a bottleneck and is a critical point of failure

- Practical algorithms exist that address this problem (ask me if interested)

- Approach NOT based on a sequencer:

    - Key Idea: the processes collectively agree on the assignment of sequence numbers to messages in a distributed fashion

Total Ordering Using Distributed Agreement