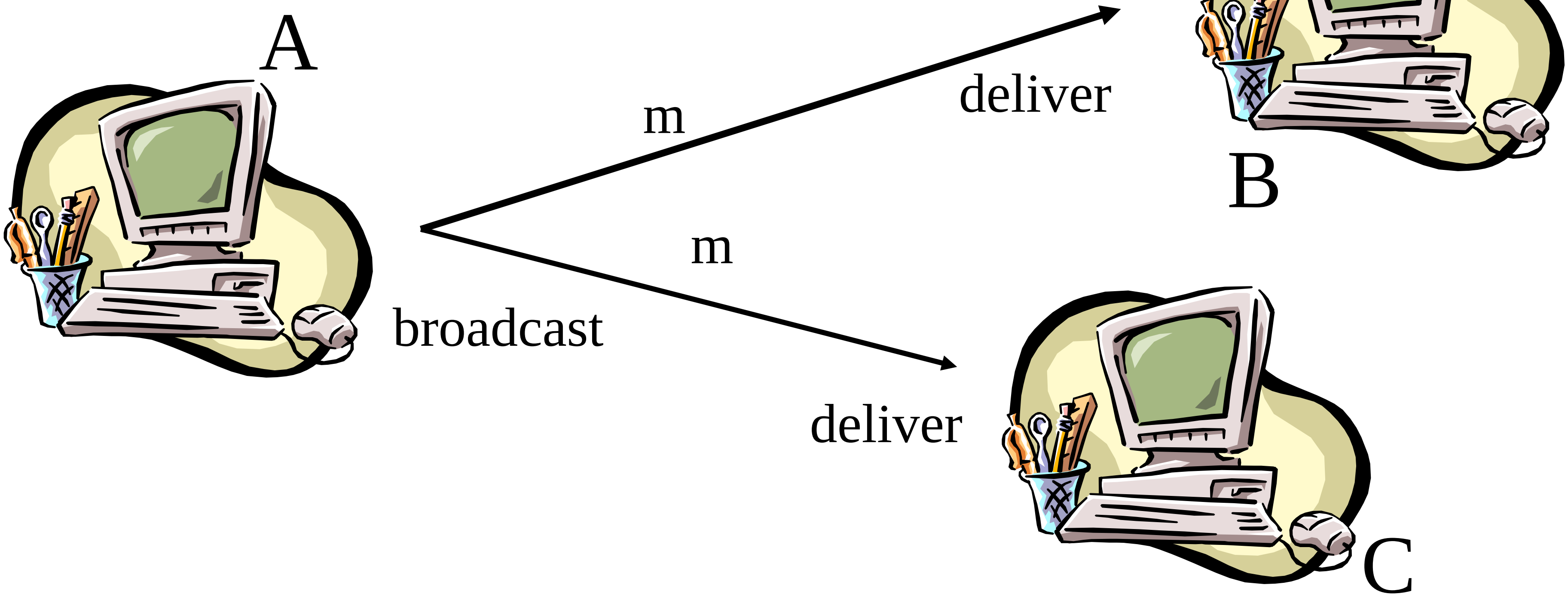
Prof Philippas Tsigas

Distributed Computing and Systems Research Group

DISTRIBUTED SYSTEMS II

FAULT-TOLERANT BROADCAST

Broadcast



Fault-Tolerant Broadcast

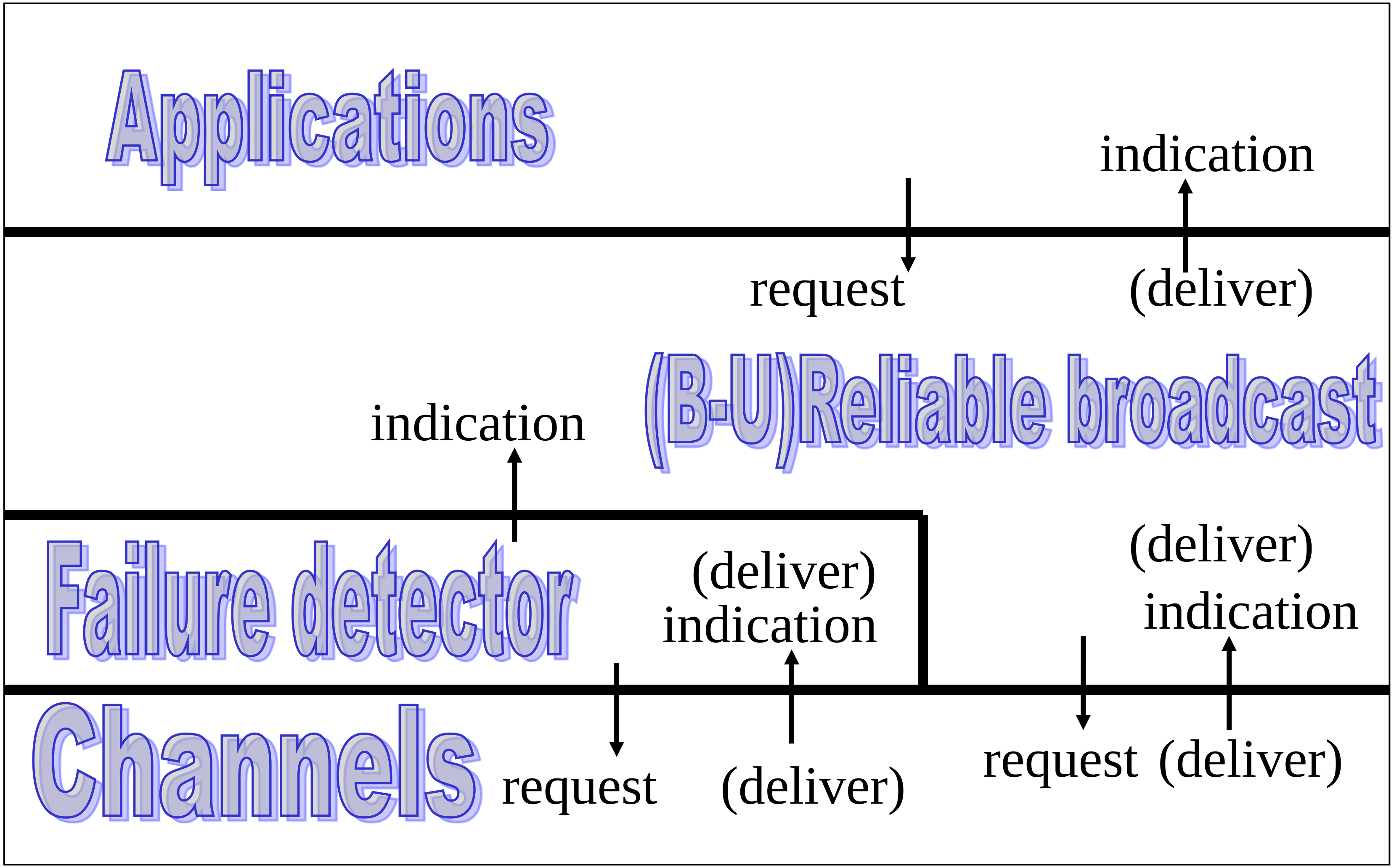
Terminology:

- *broadcast(m)* a process broadcasts a message to the others
- *deliver(m)* a process delivers a message to itself

Broadcast abstractions

Best-effort broadcast
Reliable broadcast
Uniform broadcast
…

Modules of a process



Applications

indication

request

(deliver)

indication

(B-U)Reliable broadcast

Failure detector

(deliver)

indication

(deliver)

indication

Channels

request

(deliver)

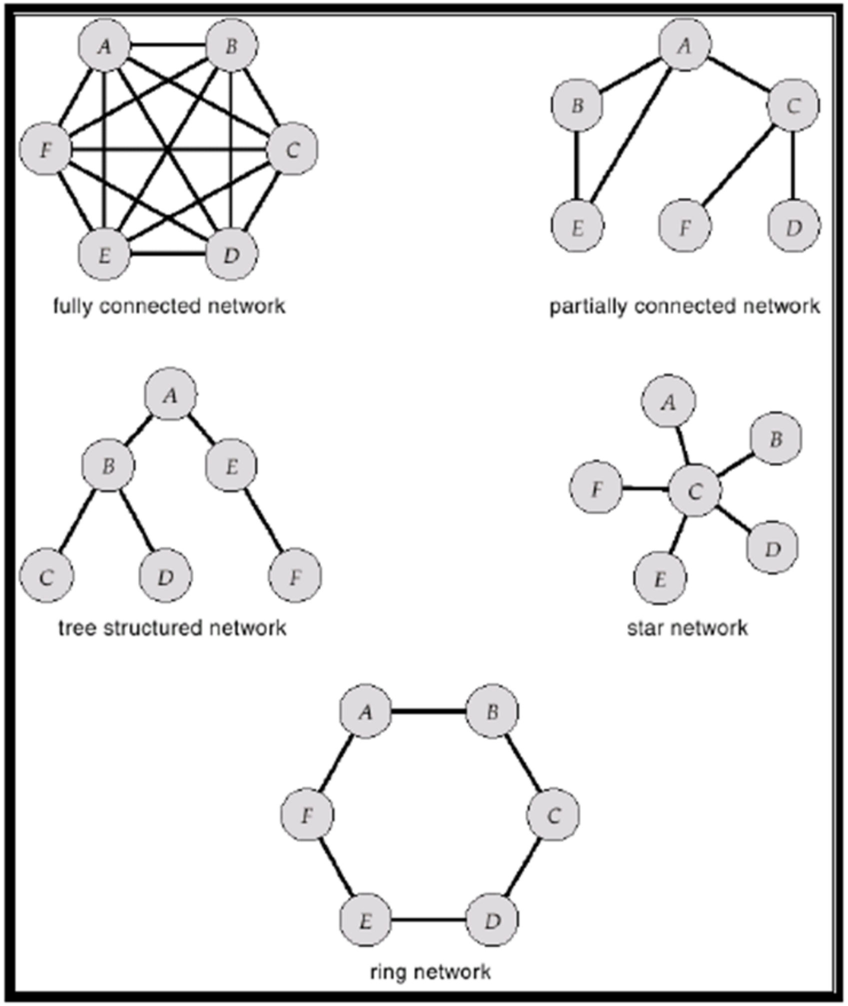
request

(deliver)

Broadcast

Models:

- Synchronous vs. asynchronous
- Types of process failures
- Types of communication failures
- Network topology
- Deterministic vs. randomized



Reliable Broadcast

Three conditions

- *Agreement*: all correct processes eventually deliver same set of messages
- *Validity*: set of messages delivered by correct processes includes all messages broadcasted by correct processes
- *Integrity*: each correct process *P* delivers a message from correct process *Q* at most once, and only if *Q* actually broadcasted it

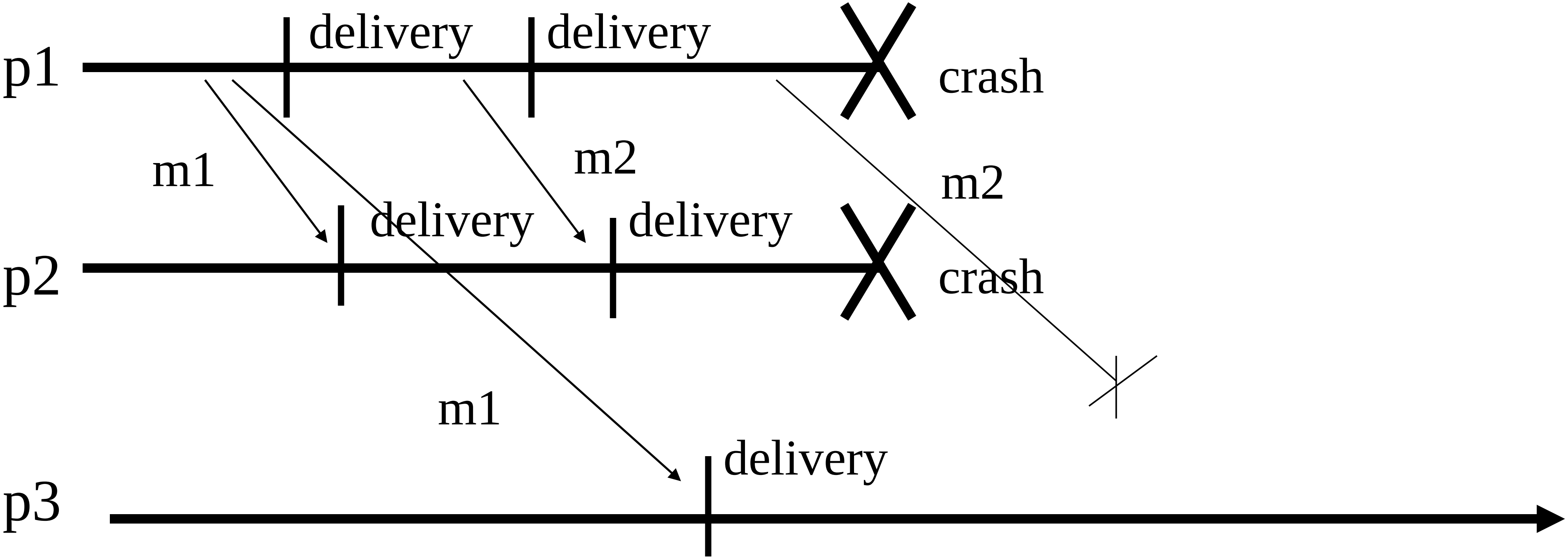
Reliable Broadcast

What about faulty processes?

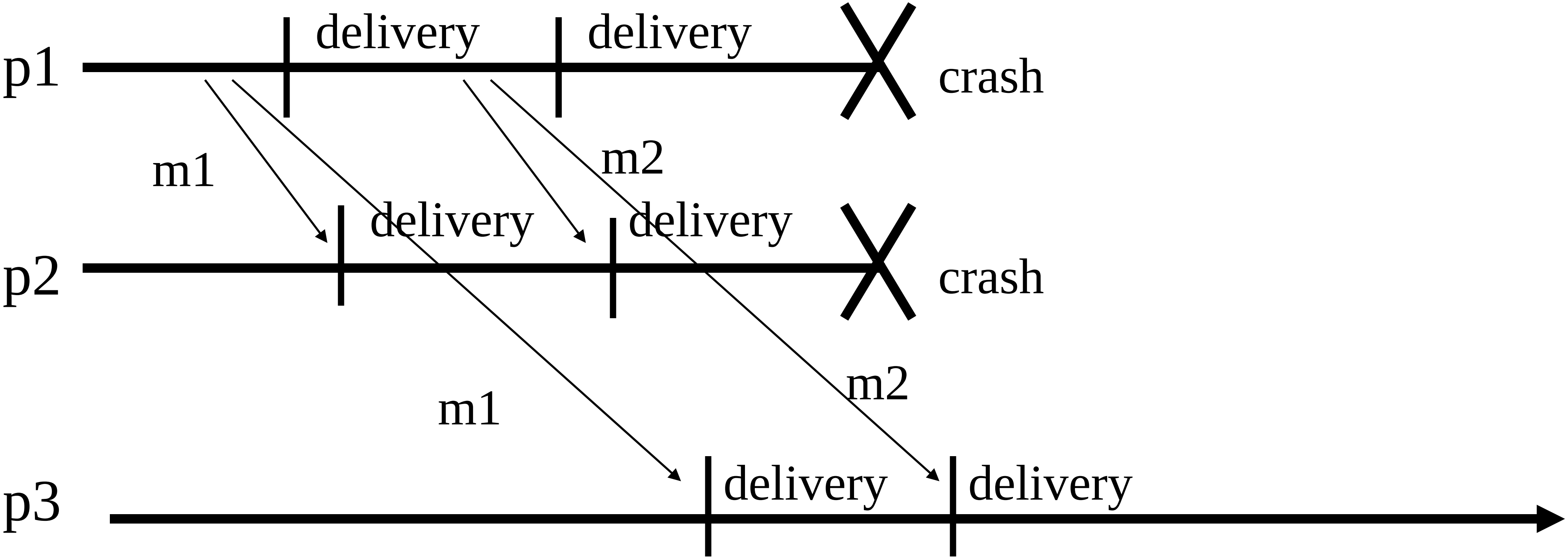
Definition: A property is uniform if faulty processes satisfy it as well.

- *Uniform agreement*:
  - If a process (correct or faulty) delivers m,
  then all correct processes eventually deliver m.
- Uniform integrity:
  - For every broadcasted message m,
  every process (correct or not) delivers m at most once, and
  only if some process has broadcasted m

Reliable broadcast



Uniform reliable broadcast



Reliable Broadcast

How can we implement Reliable Broadcast?

Model

- Asynchronous
- Benign process and link failures only
- No network partitions

Reliable Broadcast

Assume we have send(m) and receive(m) primitives

- Transmit and send messages across a link
- If P sends m to Q, and link correct, then Q eventually receives m
- For all m, Q receives m at most once from P, and
- only if P actually sent m

Reliability of one-to-one communication

- *validity*:
  - – any message in the outgoing message buffer is eventually delivered to the incoming message buffer;
- *integrity*:
  - – the message received is identical to one sent, and no messages are delivered twice.

How do we achieve validity and integrity?

- *validity*:
  - any message in the outgoing message buffer is eventually delivered to the incoming message buffer;
- *integrity*:
  - the message received is identical to one sent, and no messages are delivered twice.

validity - by use of acknowledgements and retries

integrity

- *by* use checksums, reject duplicates (e.g. due to retries).
- If allowing for malicious users, use security techniques

Reliable Broadcast

```
R-broadcast(m)
    uniquely tag m with sender and sequence number
    send(m) to all neighbours (including self)
    end R-broadcast

R-deliver(m)
    upon receive(m) do
        if i have not already delivered m
            then if I am not the sender of m
                then send m to all neighbours
                endif
            deliver(m)
            endif
end R-deliver
```

Reliable Broadcast

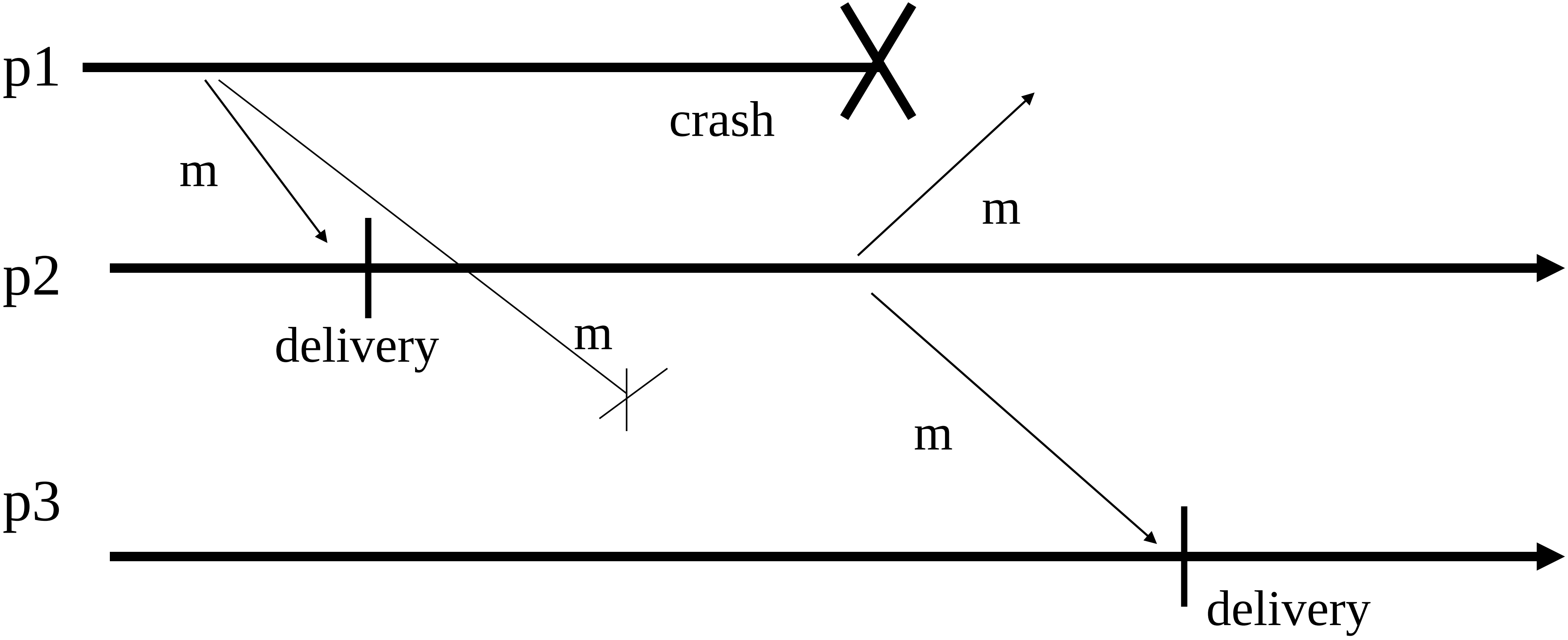
In an asynchronous system

Where every two correct processes are connected via a path that never fails,

the previous algorithm implements reliable broadcast with uniform integrity:

- For every broadcasted message m,
- every process (correct or not) delivers m at most once, and
- only if some process broadcast m.

Algorithm idea (rb)



TODO

- Prove *Agreement, Validity and Integrity*

Reliable Broadcast

In an asynchronous system

- where every two correct processes are connected via a path that never fails, and
- only receive omissions occur,
- then the algorithm satisfies uniform agreement:
  - If a process (correct or faulty) delivers m,
  - then all correct processes eventually deliver m.

TODO

- Extend the previous Proof for *Uniform Agreement*

Ordering (1)

- So far, we did not consider ordering among messages; In particular, we considered messages to be independent
- Two messages from the same process might not be delivered in the order they were broadcast

Limitations of FIFO Broadcast

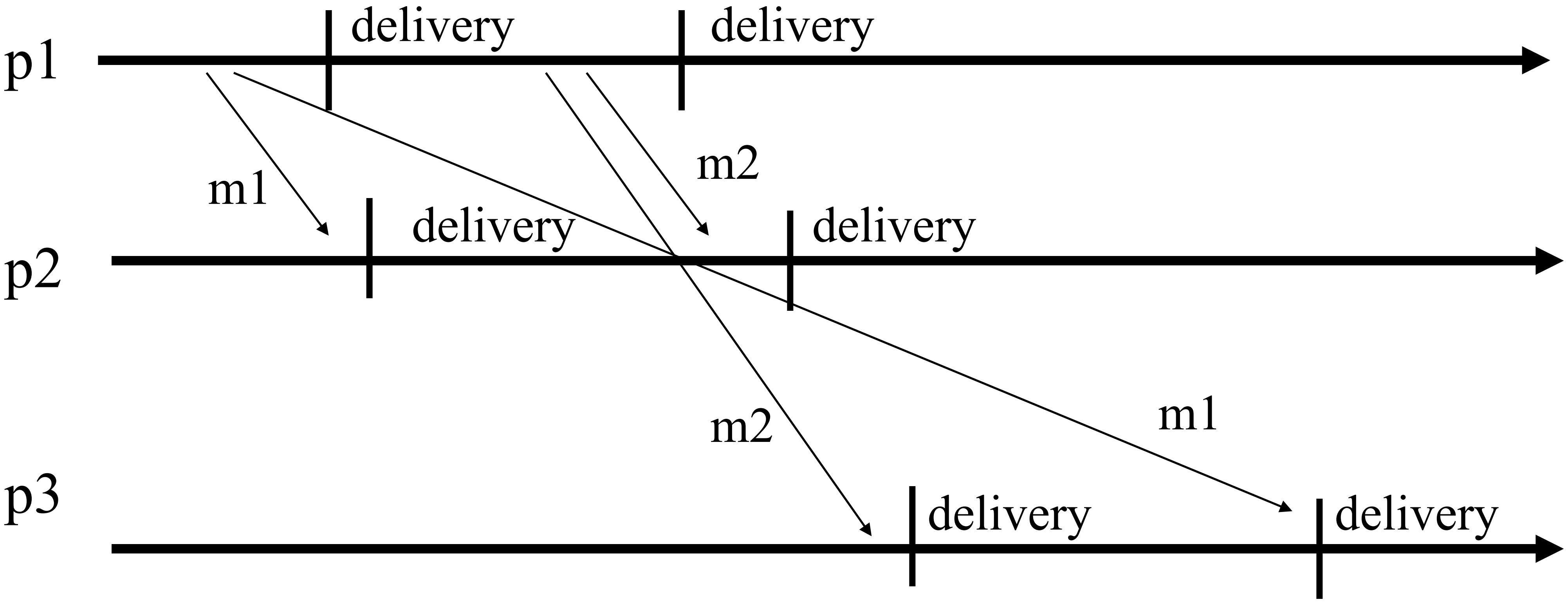
Scenario:

- User A broadcasts a message to a mailing list/Board
- B delivers that article
- B broadcasts reply
- C delivers B’s response without A´s original message
- and misinterprets the message

Intuitions

- A message m1 that causes a message m2 might be delivered by some process after m2
- Causal broadcast alleviates the need for the application to deal with such dependencies

FIFO ?


p1
p2
p3
delivery
delivery
m1
m2
delivery
delivery
m2
m1
delivery
delivery

FIFO Broadcast

- Same as reliable, plus
- All messages broadcast by same sender delivered in order sent

FIFO Broadcast

```
msgBag=0
Next[Q]=1 for all processes Q

F-broadcast(m)
    R-broadcast(m)

F-deliver(m)
    upon R-deliver(m) do
        Q := sender(m)
        msgBag := msgBagU{m}
        while (∃ m´ in msgBag : sender(m´)=Q and seq (m´) = next[Q]) do
            F-deliver(m´)
            next[Q] := next[Q]+1
            msgBag:=msqBag-{m´}
        endwhile
```

FIFO Broadcast

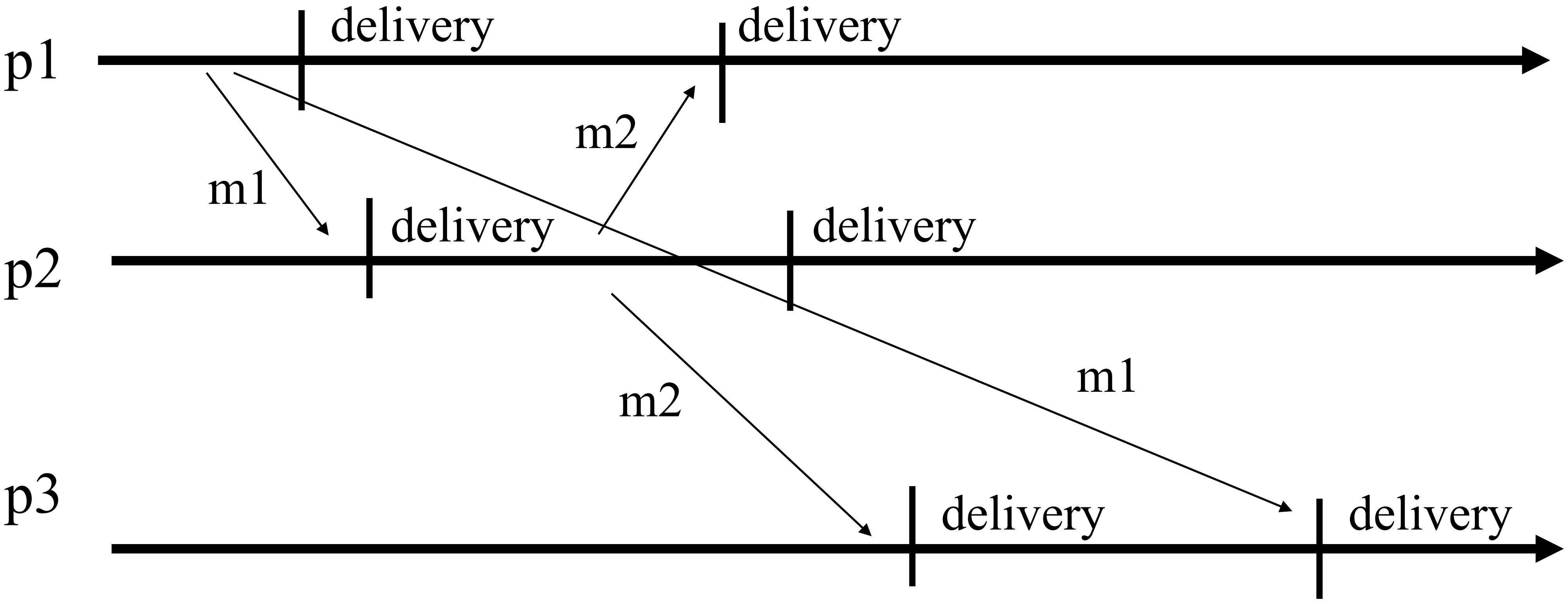
Theorem 1: Given a reliable broadcast algorithm this algorithm is uniform FIFO.

TODO: Prove it.

Theorem 2: if the reliable broadcast algorithm satisfies uniform agreement, so does this algorithm.

TODO: Prove it.

Causality ?


p1
delivery
delivery
m2
m1
p2
delivery
delivery
m2
m1
p3
delivery
delivery

Causal Broadcast

```
prevDel is
    sequence of messages that P C-delivered since its last C-broadcast

C-broadcast(m)
    F-broadcast(prevDel●m)
    prevDel:=Ø

C-deliever(m)
    upon F-deliever(m1,m2,...,ml) do
        for i in 1..l do
            if P has not previously C-delivered mi
                then C-deliver(mi)
            prevDel:=prevDel●mi
```

Causal Broadcast

Theorem 1: If the FIFO broadcast algorithm is Uniform FIFO, this is a uniform causal broadcast algorithm.

Theorem 2: if the FIFO broadcast satisfies Uniform Agreement, so does this one.

Limitation of Causal Broadcast

Causal broadcast does not impose any order on unrelated messages.

Two correct processes can deliver operations/request in different order.

Total, FIFO and causal ordering of multicast messages

Notice the 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

and the causally related messages C_1 and C_3

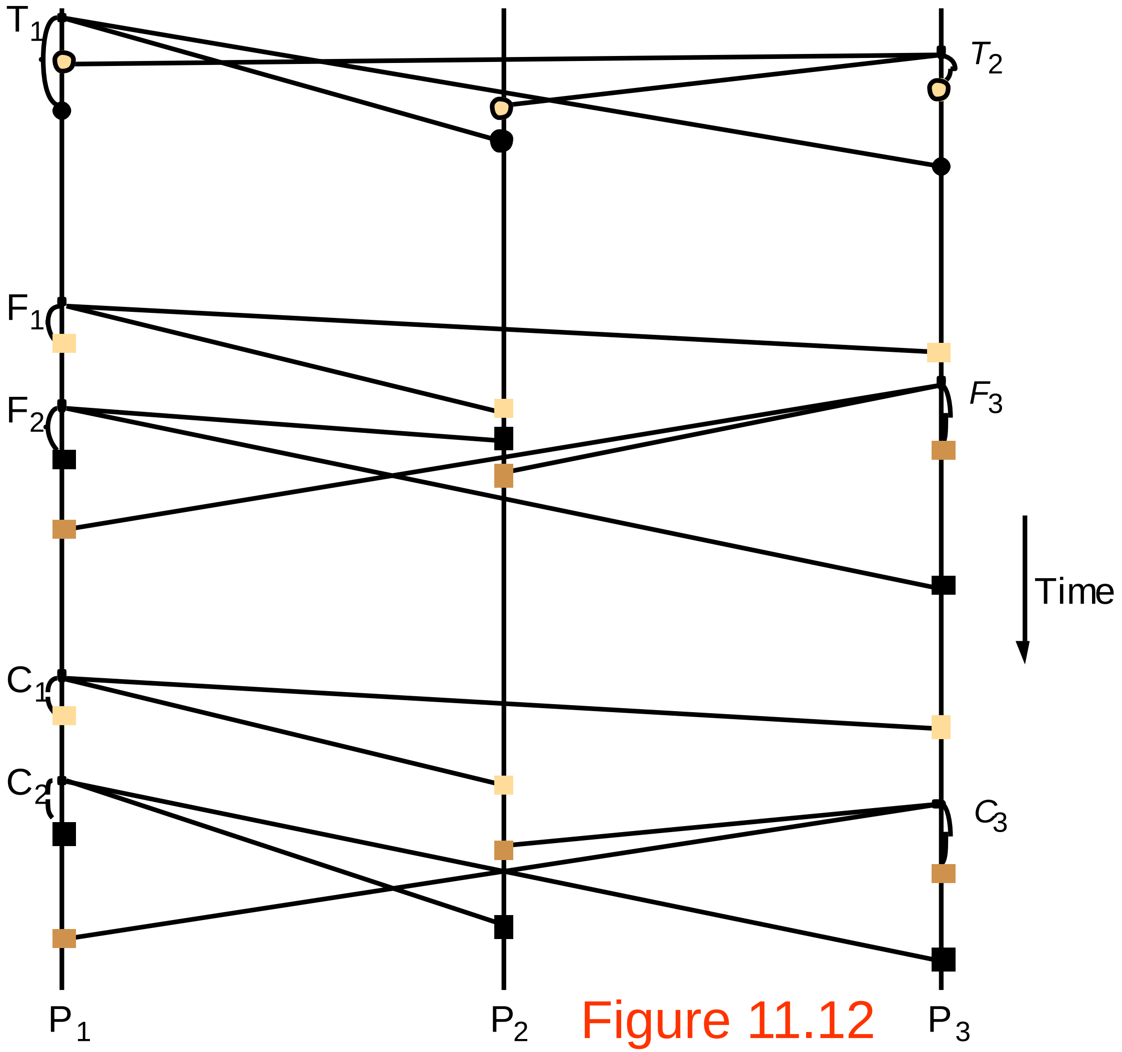


Figure 11.12

Atomic Broadcast

Requires that all correct processes deliver all messages in the same order.

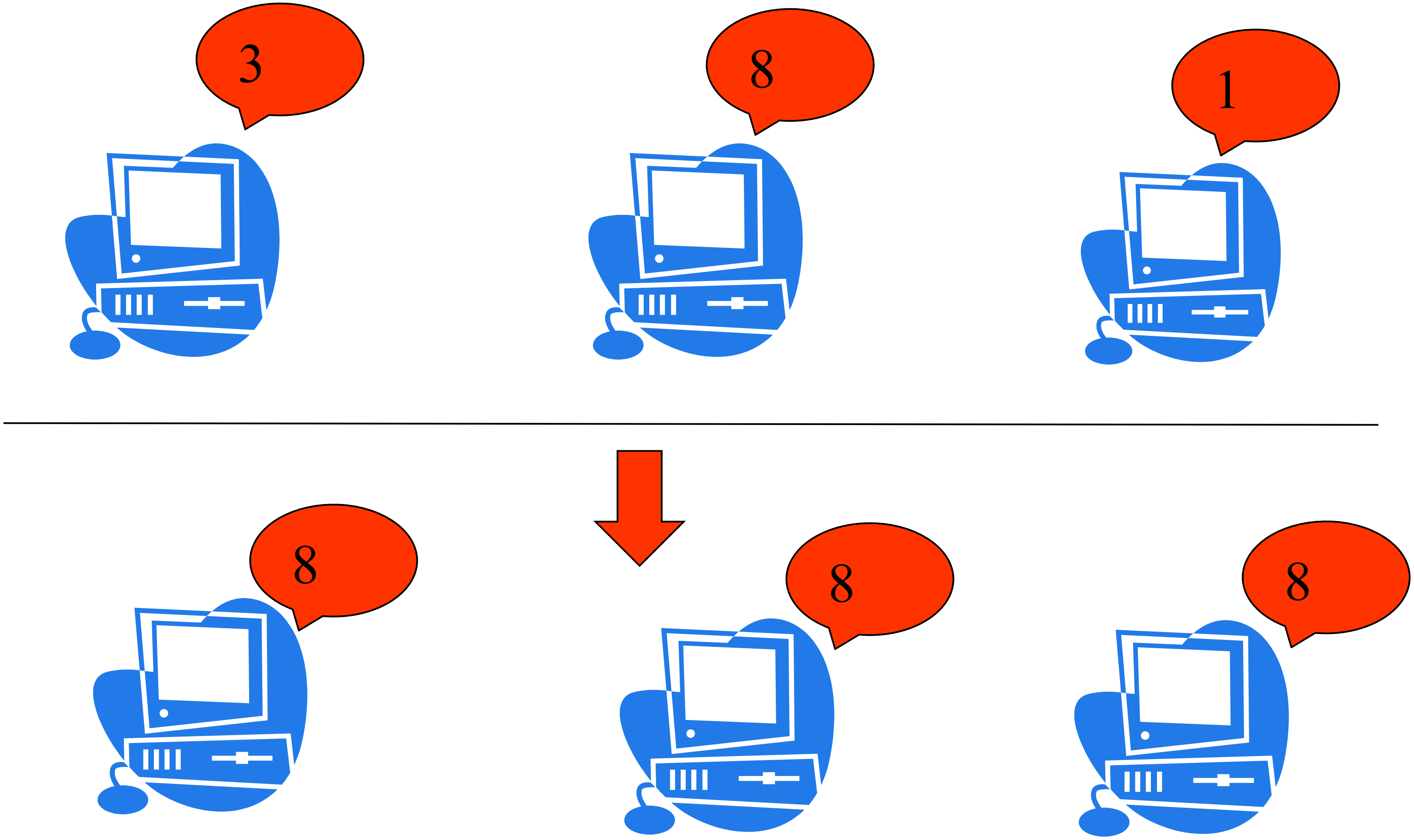
Implies that all correct processes see the same view of the world.

Atomic Broadcast

Theorem: Atomic broadcast is impossible in asynchronous systems.

Equivalent to consensus problem.

Review of Consensus



FLP

Theorem: Consensus is impossible in any asynchronous system if one process can halt. [Fisher, Lynch, Peterson 1985]

Atomic Broadcast

Theorem 1: Any atomic broadcast algorithm solves consensus.

- Everybody does an Atomic Broadcast
- Decides first value delivered

Theorem 2: Atomic broadcast is impossible in any asynchronous system if one process can halt.

Total ordering using a sequencer

```
1. Algorithm for group member
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-ba
    $r_g = S + 1$;

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, $s_g$>);
    $s_g := s_g + 1$;
```

A process wishing to *TO-multicast m* to *g* attaches a unique id, *id*(*m*) and sends it to the sequencer and the members.

Other processes: *B-deliver <m,i>*

put <*m*,i> in hold-back queue

B-deliver order message, get *g* and *S* and *i* from order message

wait till <*m*,*i*> in queue and $S = r_g$, *TO-deliver* *m* and set r_g to *S*+1

The *sequencer* keeps sequence number s_g for group *g*

When it *B-delivers* the message it multicasts an 'order' message to members of *g* and increments s_g.

Figure 11.14

Atomic Broadcast

Consensus is solvable in:

- Synchronous systems (we will discuss such an algorithm that works in f+1 rounds)
- Certain semi-synchronous systems

Consensus is also solvable in

- Asynchronous systems with randomization
- Asynchronous systems with failure-detectors

SLIDES FROM THE BOOK TO HAVE A LOOK AT

- Please check aslo the slides from your book.
- I appned them here.

Teaching material based on Distributed Systems: Concepts and Design, Edition 3, Addison-Wesley 2001.

Distributed Systems Course
Coordination and Agreement

Viewing: These slides must be viewed in slide show mode.

•11.4 Multicast communication

- this chapter covers other types of coordination and agreement such as mutual exclusion, elections and consensus. We will study only multicast.
- But we will study the two-phase commit protocol for transactions in Chapter 12, which is an example of consensus
- We also omit the discussion of failure detectors which is relevant to replication

Give two reasons for restricting the scope of a multicast message

- IP multicast – an implementation of group communication
  - built on top of IP (note IP packets are addressed to computers)
  - allows the sender to transmit a single IP packet to a set of computers that form a multicast group (a class D internet address with first 4 bits 1110)
  - Dynamic membership of groups. Can send to a group with or without joining it
  - To multicast, send a UDP datagram with a multicast address
  - To join, make a socket join a group *(s.joinGroup*(*group*) - Fig 4.17) enabling it to receive messages to the group
- Multicast routers
  - Local messages use local multicast capability. Routers make it efficient by choosing other routers on the way.
- Failure model
  - Omission failures ⇒ some but not all members may receive a message.
    - e.g. a recipient may drop message, or a multicast router may fail
  - IP packets may not arrive in sender order, group members can receive messages in different orders

What is meant by[the term *broadcast* ?

- Multicast communication requires coordination and agreement. The aim is for members of a group to receive copies of messages sent to the group
- Many different delivery guarantees are possible
  - e.g. agree on the set of messages received or on delivery ordering
- A process can multicast by the use of a single operation instead of a send to each member
  - For example in IP multicast *aSocket.send*(*aMessage*)
  - The single operation allows for:
    - *efficiency* I.e. send once on each link, using hardware multicast when available, e.g. multicast from a computer in London to two in Beijing
    - *delivery guarantees* e.g. can't make a guarantee if multicast is implemented as multiple sends and the sender fails. Can also do ordering

●

System model

- The system consists of a collection of processes which can communicate *reliably* over 1-1 channels
- Processes fail only by crashing (no arbitrary failures)
- Processes are members of groups - which are the destinations of multicast messages
- 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*) is called to get a multicast message delivered. It is different from *receive* as it may be delayed to allow for ordering or reliability.
- 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

•

Does IP multicast support open and closed groups?

- Closed groups
  - only members can send to group, a member delivers to itself
  - they are useful for coordination of groups of cooperating servers
- Open
  - they are useful for notification of events to groups of interested processes

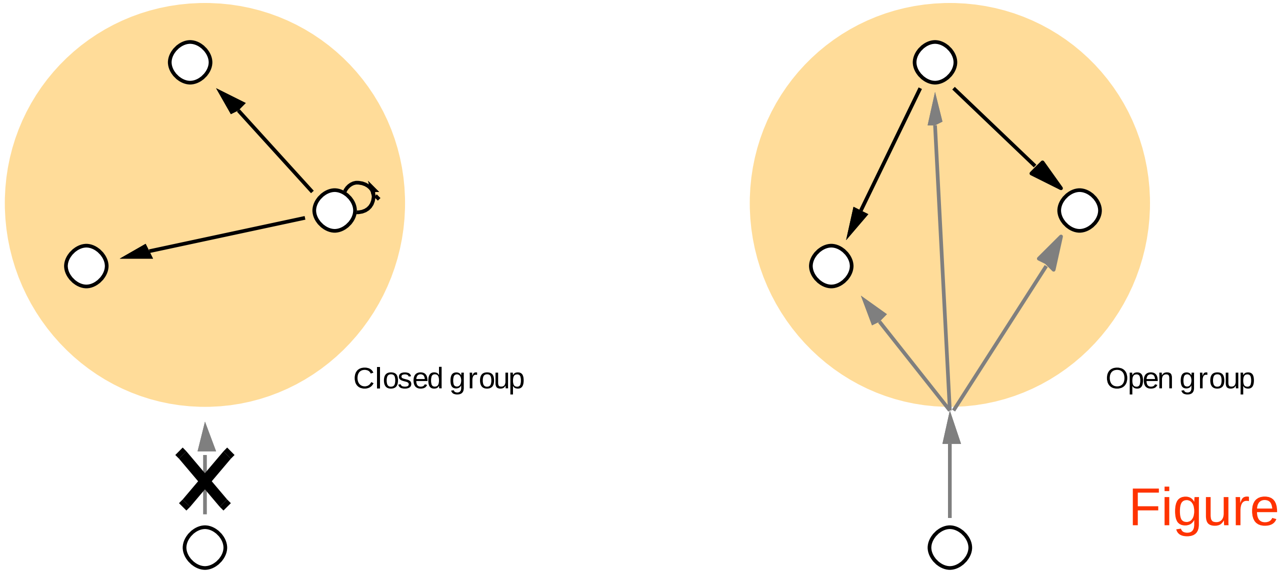


Figure 11.9

How do we achieve integrity?

- The term *reliable 1-1 communication* is defined in terms of *validity* and *integrity* as follows:
- *validity*:
  - any message in the outgoing message buffer is eventually delivered to the incoming message buffer;
- *integrity*:
  - the message received is identical to one sent, and no messages are delivered twice.

validity - by use of acknowledgements and retries

integrity

- *by* use checksums, reject duplicates (e.g. due to retries).
- If allowing for malicious users, use security techniques

•

What are ack-implosions?

- A correct process will eventually deliver the message provided the **multicaster does not crash**
  - note that IP multicast does not give this guarantee
- The primitives are called *B-multicast* and *B-deliver*
- A straightforward but ineffective method of implementation:
  - use a reliable 1-1 *send* (i.e. with integrity and validity as above)

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

    On *receive* (*m*) at *p*: *B-deliver* (*m*) at *p*
- Problem
  - if the number of processes is large, the protocol will suffer from *ack-implosion*

A practical implementation of Basic Multicast may be achieved over IP multicast (on next slide, but not shown)

11.4.2 Reliable multicast

- The protocol is correct even if the multicaster crashes
- it satisfies criteria for *validity, integrity* and *agreement*
- it provides operations *R-multicast* and *R-deliver*
- *Integrity* - a correct process, *p* delivers *m* at most once. Also *p* ε *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 correct processes in *group*(*m*) will eventually deliver *m*

integrity as for 1-1 communication

validity - simplify by choosing sender as the one process

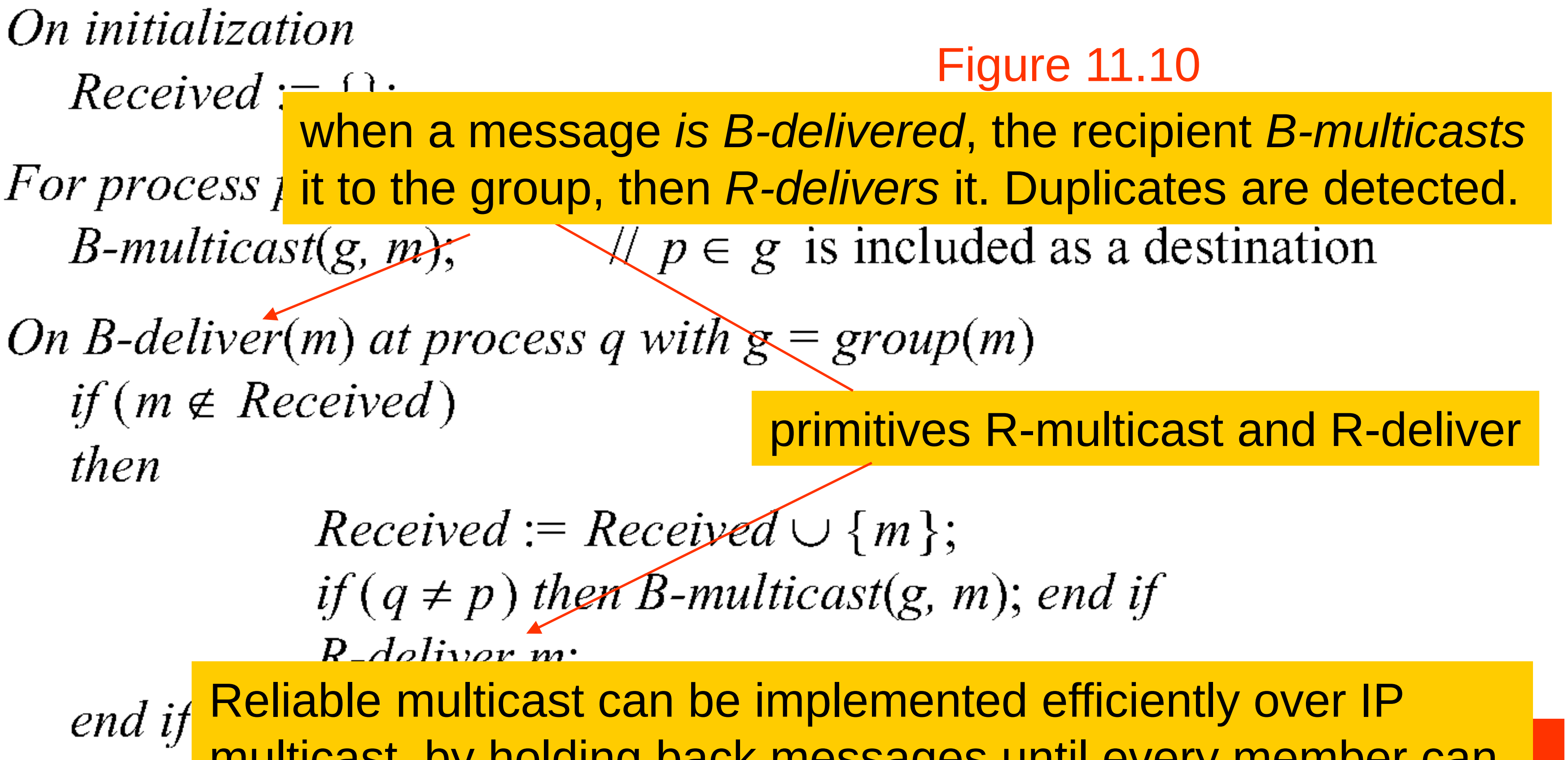
agreement - all or nothing - atomicity, even if multicaster crashes

-

Agreement - every correct process *B-multicasts* the message to the others. If *p* does not *R-deliver* then this is because it didn't *B-deliver* - because no others did either.

- processes c

to R-multicast a message, a process B-multicasts it to processes in the group including itself



```
On initialization
    Received := {};

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

Figure 11.10

when a message *is B-delivered*, the recipient *B-multicasts* it to the group, then *R-delivers* it. Duplicates are detected.

primitives R-multicast and R-deliver

Reliable multicast can be implemented efficiently over IP multicast by holding back messages until every member can receive them. We skip that.

Reliable multicast over IP multicast (page 440)

- This protocol assumes groups are closed. It uses:
  - piggybacked acknowledgement messages
  - negative acknowledgements when messages are missed

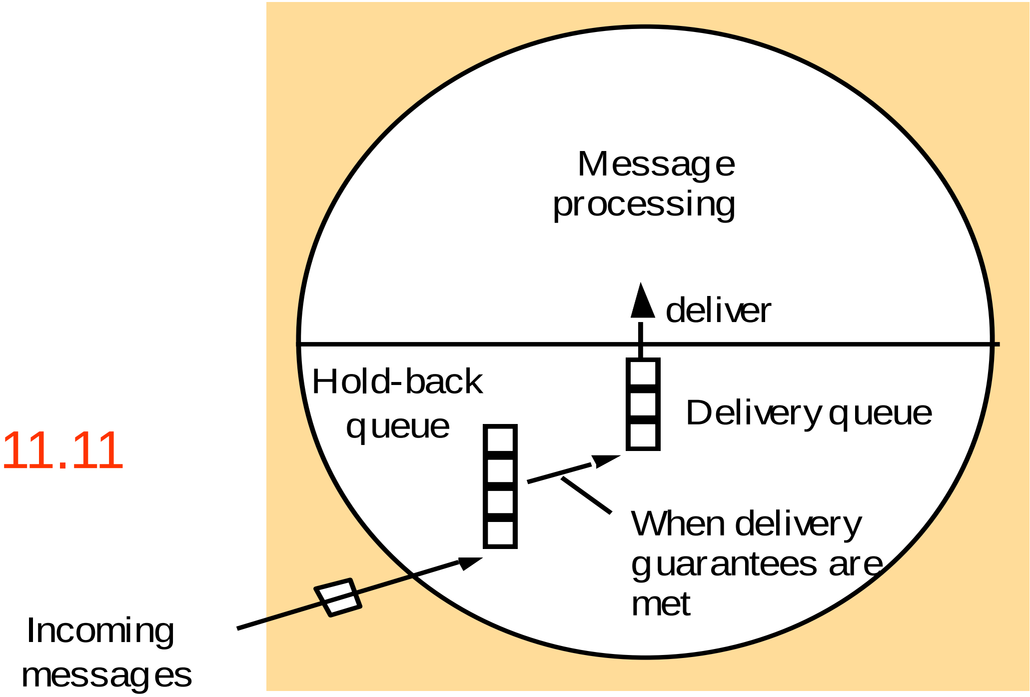
the piggybacked values in a message allow recipients to learn about messages they have not yet received

  - R^q_g sequence number of latest message received from process q to g
- For process p to *R-multicast* message m to group g
  - piggyback S^p_g and +ve acks for messages received in the form <q, $R^q g$ >
  - IP multicasts the message to g, increments S^p_g by 1
- A process on receipt by 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
  - If $S \leq R^p_g$ discard the message
  - If $S> R^p_g + 1$ or if $R<R^q_g$ (for enclosed ack <q,R>)
    - then it has missed messages and requests them with negative acknowledgements
    - puts new message in hold-back queue for later delivery

The hold-back queue for arriving multicast messages

- The hold back queue is not necessary for reliability as in the implementation using IP muilticast, but it simplifies the protocol, allowing sequence numbers to represent sets of messages. Hold-back queues are also used for ordering protocols.

Figure 11.11



Reliability properties of reliable multicast over IP

- *Integrity* - duplicate messages detected and rejected.
  IP multicast uses checksums to reject corrupt messages
- *Validity* - due to IP multicast in which sender delivers to itself
- *Agreement* - processes can detect missing messages. They must keep copies of messages they have delivered so that they can re-transmit them to others.
- discarding of copies of messages that are no longer needed :
  - when piggybacked acknowledgements arrive, note which processes have received messages. When all processes in *g* have the message, discard it.
  - problem of a process that stops sending - use ‘heartbeat’ messages.
- This protocol has been implemented in a practical way in Psynch and Trans (refs. on p442)

11.4.3 Ordered multicast

- The basic multicast algorithm delivers messages to processes in an arbitrary order. A variety of orderings may be implemented:
- 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'* .
- 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'*.
- Ordering is expensive in delivery latency and bandwidth consumption

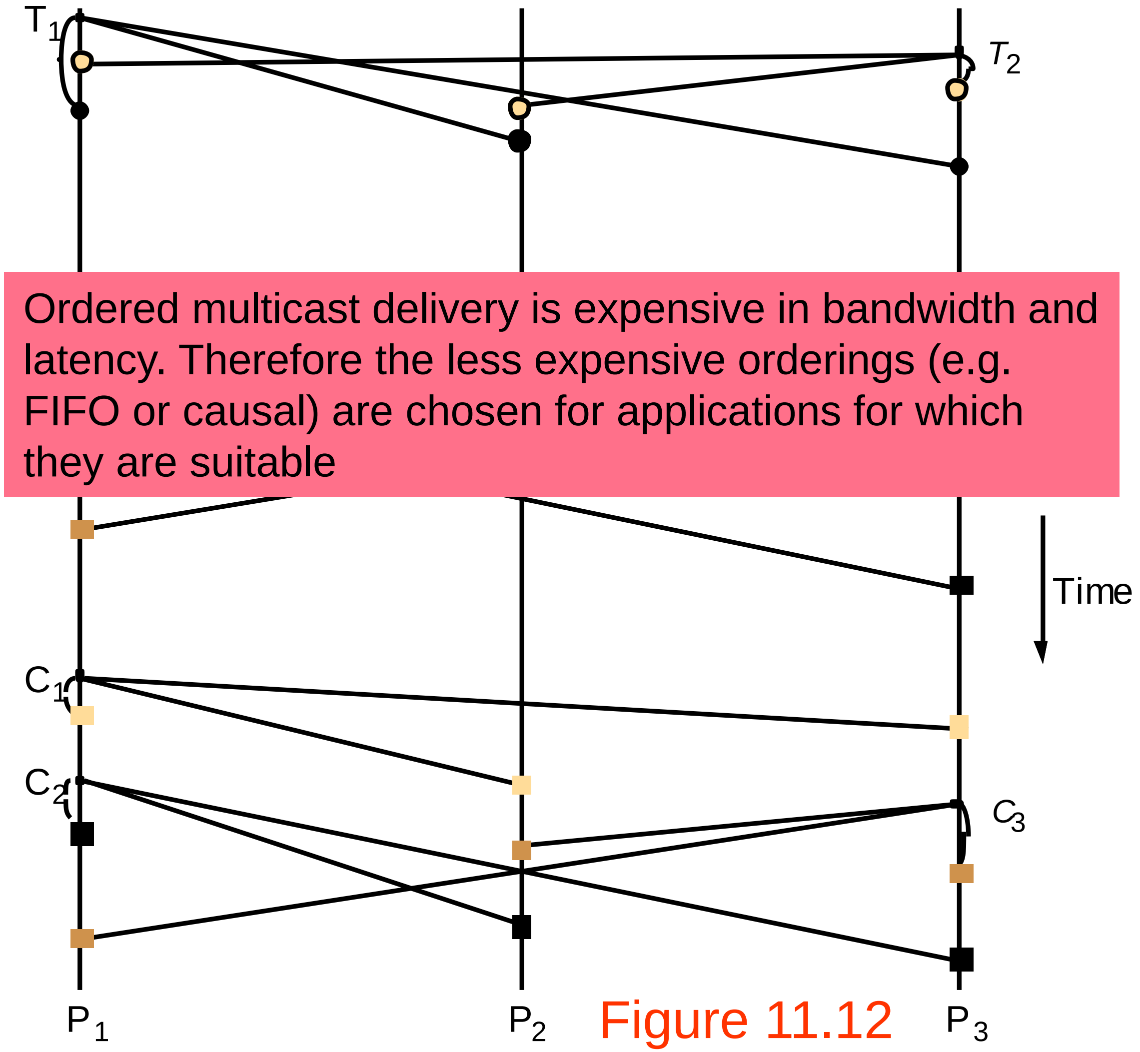
Total, FIFO and causal ordering of multicast messages

Notice the 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

Note the FIFO-related messages F_1 and F_2

and the causally related messages C_1 and C_3

these definitions do not imply reliability, but we can define *atomic multicast* - reliable and totally ordered.

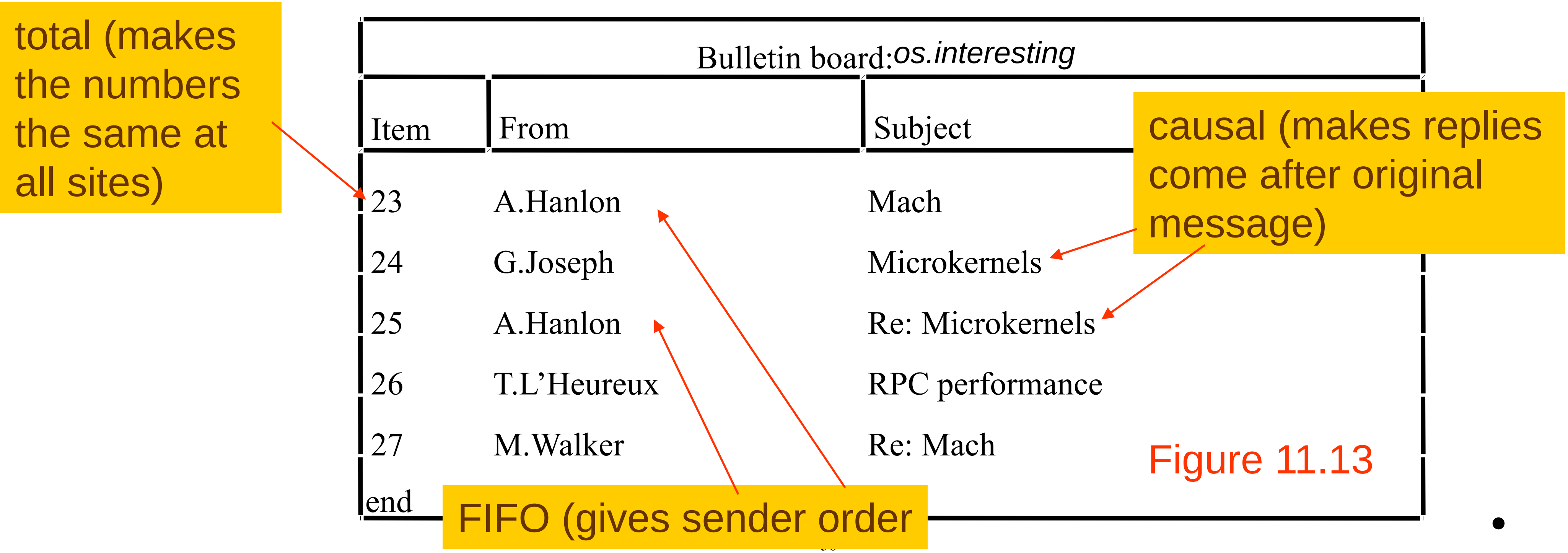


Ordered multicast delivery is expensive in bandwidth and latency. Therefore the less expensive orderings (e.g. FIFO or causal) are chosen for applications for which they are suitable

Figure 11.12

Display from a bulletin board program

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



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		

total (makes the numbers the same at all sites)

causal (makes replies come after original message)

FIFO (gives sender order

Figure 11.13

Implementation of FIFO ordering over basic multicast

- We discuss FIFO ordered multicast with operations *FO-multicast* and *FO-deliver* for non-overlapping groups. It can be implemented on top of any basic multicast
- Each process p holds:
  - S^p_g a count of messages sent by *p* to *g* and
  - R^q_g the sequence number of the latest message to *g* that *p* delivered from *q*
- For *p* to *FO-multicast* a message to *g*, it piggybacks S^p_g on the message, *B-multicasts* it and increments S^p_g by 1
- On receipt of a message from *q* with sequence number *S*, *p* checks whether $S = R^q_g + 1$. If so, it *FO-delivers* it.
- if $S > R^q_g + 1$ then *p* places message in hold-back queue until intervening messages have been delivered. (note that *B-multicast* does eventually deliver messages unless the sender crashes)

Implementation of totally ordered multicast

- The general approach is to attach *totally ordered identifiers* to multicast messages
  - each receiving process makes ordering decisions based on the identifiers
  - similar to the FIFO algorithm, but processes keep group specific sequence numbers
  - operations *TO-multicast* and *TO-deliver*
- we present two approaches to implementing total ordered multicast over basic multicast
  1. using a sequencer (only for non-overlapping groups)
  2. the processes in a group collectively agree on a sequence number for each message

Total ordering using a sequencer

```
1. Algorithm for group membe
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-ba
    $r_g = S + 1$;
```

A process wishing to *TO-multicast m* to *g* attaches a unique id, *id*(*m*) and sends it to the sequencer and the members.

Other processes: *B-deliver <m,i>*

put <*m*,i> in hold-back queue

B-deliver order message, get *g* and *S* and *i* from order message

wait till <*m,i*> in queue and $S = r_g$, *TO-deliver* *m* and set r_g to $S+1$

```
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, $s_g$>);
    $s_g := s_g + 1$;
```

The *sequencer* keeps sequence number s_g for group *g*

When it *B-delivers* the message it multicasts an 'order' message to members of *g* and increments s_g.

Figure 11.14

Members that do not multicast send heartbeat messages (with a sequence number)

- Since sequence numbers are defined by a sequencer, we have total ordering.
- Like B-multicast, if the sender does not crash, all members receive the message

What are the potential problems with using a single sequencer?

Kaashoek’s protocol uses hardware-based multicast

The sender transmits one message to sequencer, then

the sequencer multicasts the sequence number and the message

but IP multicast is not as reliable as B-multicast so the sequencer stores messages in its history buffer for retransmission on request

members notice messages are missing by inspecting sequence numbers

The ISIS algorithm for total ordering

- this protocol is for open or closed groups

1. the process *P1 B-multicasts* a message to members of the group

2. the receiving processes propose numbers and return them to the sender

3. the sender uses the proposed numbers to generate an agreed number

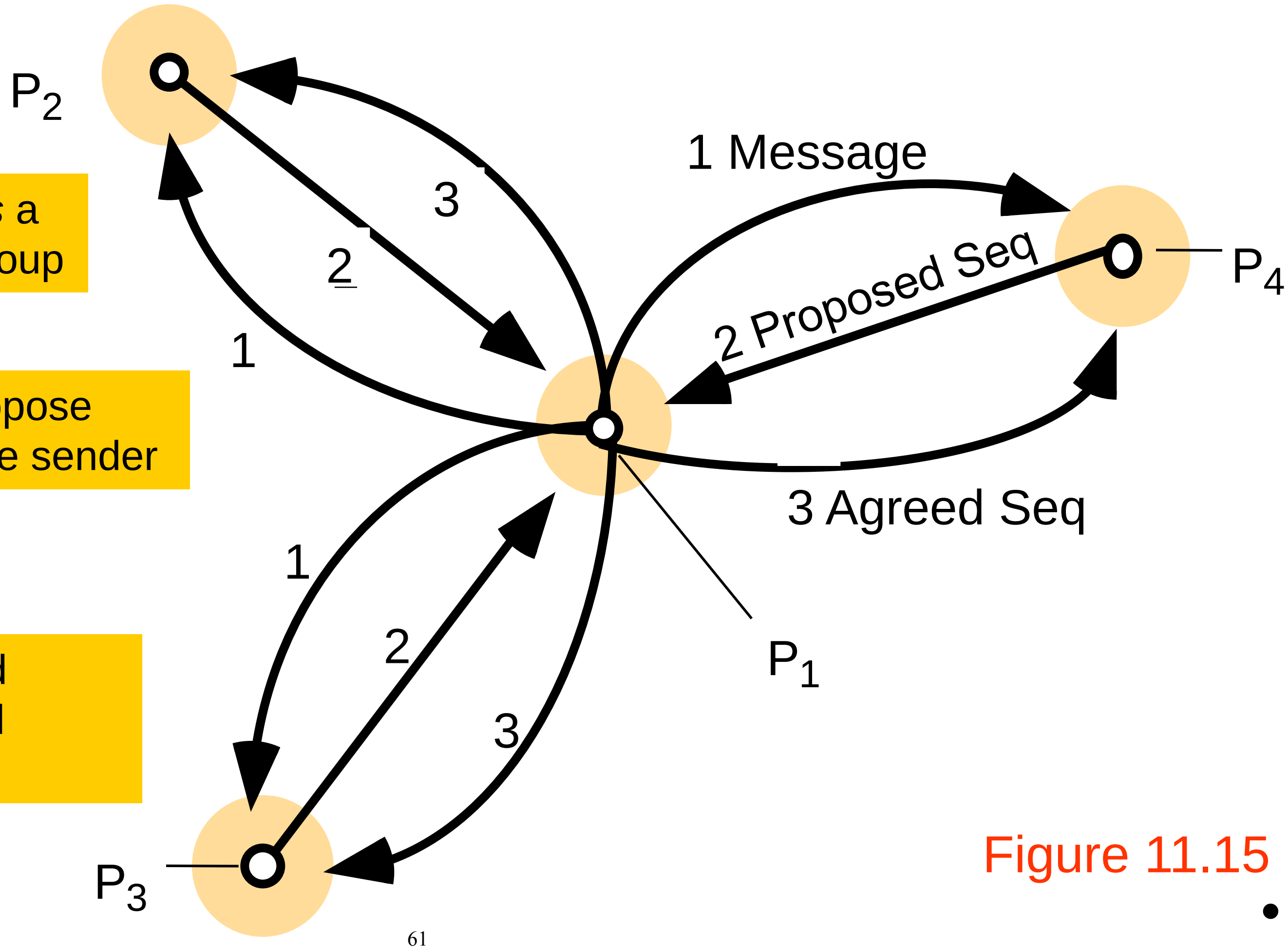


Figure 11.15

ISIS total ordering - agreement of sequence numbers

- Each process, *q* keeps:
  - A^q_g the largest agreed sequence number it has seen and
  - P^q_g its own largest proposed sequence number
- 1. Process *p B-multicasts* <*m*, *i*> to *g*, where *i* is a unique identifier for *m*.
- 2. Each process *q* replies to the sender *p* with a proposal for the message's agreed sequence number of
  - $P^q_g := \mathrm{Max}(A^q_g, P^q_g) + 1$.
  - assigns the proposed sequence number to the message and places it in its hold-back queue
- 3. *p* collects all the proposed sequence numbers and selects the largest as the next agreed sequence number, *a*.
  It *B-multicasts* <*i*, *a*> to *g*. Recipients set $A^q_g := \mathrm{Max}(A^q_g, a)$, attach *a* to the message and re-order hold-back queue.

Discussion of ordering in ISIS protocol

proof of total ordering on page 448

- Hold-back queue
- ordered with the message with the smallest sequence number at the front of the queue
- when the agreed number is added to a message, the queue is re-ordered
- when the message at the front has an agreed id, it is transferred to the delivery queue
  - even if agreed, those not at the front of the queue are not transferred
- every process agrees on the same order and delivers messages in that order, therefore we have total ordering.
- Latency
  - 3 messages are sent in sequence, therefore it has a higher latency than sequencer method
  - this ordering may not be causal or FIFO

•

Causally ordered multicast

- We present an algorithm of Birman 1991 for causally ordered multicast in non-overlapping, closed groups. It uses the *happened before* relation (on multicast messages only)
  - that is, ordering imposed by one-to-one messages is not taken into account
- It uses vector timestamps - that count the number of multicast messages from each process that happened before the next message to be multicast

Causal ordering using vector timestamps

each process has its own vector timestamp

```
Algorithm for group member $p_i$

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

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

On B-deliver($<V_j^g, m>$) from
    place $<V_j^g, m>$ in hold-bac
    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
    $V_i^g[j] := V_i^g[j] + 1$;
```

To *CO-multicast m* to *g*, a process adds 1 to its entry in the vector timestamp and *B-multicasts m* and the vector timestamp

When a process *B-delivers m*, it places it in a hold-back queue until messages earlier in the causal ordering have been delivered:-

a) earlier messages from same sender have been delivered

b) any messages that the sender had delivered when it sent the multicast message have been delivered

then it CO-delivers the message and updates its timestamp

Figure 11.16

Note: a process can immediately *CO-deliver* to itself its own messages (not shown)

Comments

- after delivering a message from p_j, process p_i updates its vector timestamp
  - by adding 1 to the *j*th element of its timestamp
- compare the vector clock rule where $V_i[j] := \max(V_i[j], t[j])$ for $j=1, 2, \ldots N$
  - in this algorithm we know that only the *j*th element will increase
- for an outline of the proof see page 449
- if we use *R-multicast* instead of *B-multicast* then the protocol is reliable as well as causally ordered.
- If we combine it with the sequencer algorithm we get total and causal ordering

Comments on multicast protocols

- we need to have protocols for overlapping groups because applications do need to subscribe to several groups
- definitions of ‘global FIFO ordering’ etc on page 450 and some references to papers on them
- multicast in synchronous and asynchronous systems
  - all of our algorithms do work in both
- reliable and totally ordered multicast
  - can be implemented in a synchronous system
  - but is impossible in an asynchronous system (reasons discussed in consensus section - paper by Fischer et al.)

Summary

- Multicast communication can specify requirements for reliability and ordering, in terms of integrity, validity and agreement
- B-multicast
  - a correct process will eventually deliver a message provided the multicaster does not crash
- reliable multicast
  - in which the correct processes agree on the set of messages to be delivered;
  - we showed two implementations: over B-multicast and IP multicast
- delivery ordering
  - FIFO, total and causal delivery ordering.
  - FIFO ordering by means of senders' sequence numbers
  - total ordering by means of a sequencer or by agreement of sequence numbers between processes in a group
  - causal ordering by means of vector timestamps
- the hold-back queue is a useful component in implementing multicast protocols