*Master 2 Ingénierie/Ubinet*

# Distributed Algorithms

***Françoise Baude***

(Francesco.Bongiovanni former PhD)

Course web site : on the moodle, key M2-19-20

https://lms.univ-cotedazur.fr/course/view.php?id=3535

Oct. 2019

## Chapter 4 : Group communications (GC)



# Acknowledgement

- The slides for this lecture are based on ideas and materials from the following sources:
  - ***Introduction to Reliable Distributed Programming*** Guerraoui, Rachid, Rodrigues, Luís, 2006, 300 p., ISBN: 3-540-28845-7 (+ teaching material)
  - **ID2203 Distributed Systems Advanced Course** by Prof. Seif Haridi from KTH – Royal Institute of Technology (Sweden)
  - **CS5410/514: Fault-tolerant Distributed Computer Systems Course** by Prof. Ken Birman from Cornell University
  - **Distributed Systems : An Algorithmic Approach** by Sukumar, Ghosh, 2006, 424 p.,ISBN:1-584-88564-5 (+teaching material)

# Outline

1. Definition, motivation

2. Basic GC abstraction
   - Best Effort Bcast
   - Causal Order and Total Order Bcast

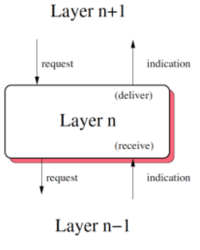
3. Reliability – small glance at reliable GC
   - Reliable Bcast
   - Reliable causal Bcast

3

# Algorithm representation

- Event-based component (or module) model
  - Nodes in the model execute programs
    - Each program consists of a set of modules (forming a software stack)

- Modules interact via events




4

# Algorithm representation (cont'd)

- Code for each component looks like this:

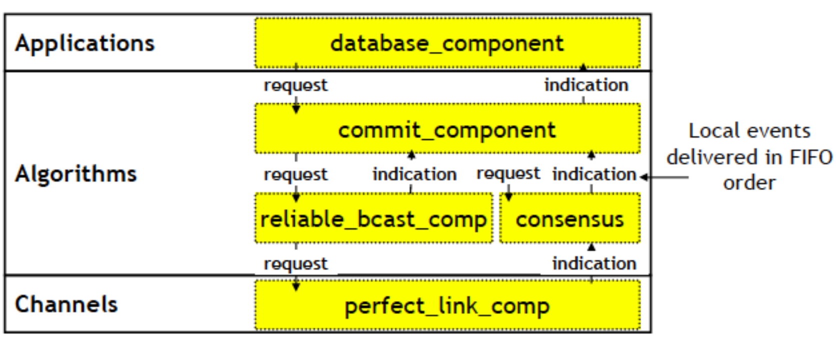
**upon event** $\langle$ *Event1* | $att_1^1$, $att_1^2$, … $\rangle$ **do**
something
**trigger** $\langle$ *Event2* | $att_2^1$,$att_2^2$, … $\rangle$; // *send some event*

- Three types of events
  - Requests
  - Indications (like a response)
  - Confirmations (like an OK or ACK)

5

# Modules on a node

- Stack of modules on a single node




6

# Channels as modules

- Channels are modules
  - Request event:
    - Send to destination some message (with data)

**trigger** <*send* | *dest*, [data1, data2, …] >

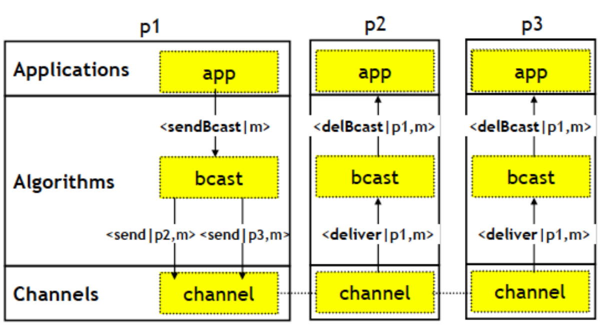
  - Indication event:
    - Deliver from source some message (with data)

**upon event** <*send* | *src*, [data1, data2, …] > do



# Example

- Application uses a Broadcast module
  - Which uses a channel module to broadcast



# A little reminder...

- Correctness in distributed systems expressed in terms of *safety* and *liveness* properties

  - Safety
    - *States that a property that is violated at time t should never be satisfied again after that time. Say another way, during the lifetime of the algorithm, only safe things happen*

    *« nothing bad will happen»*

  - Liveness
    - *States that a property should eventually hold*
    - *« eventually something good happens»*

- Lamport, L. A simple approach to specifying concurrent systems. Commun. ACM 32, 32-45 (1989).

- Lamport, L. Specifying Systems. (Addison-Wesley: 2002).



# Correctness example

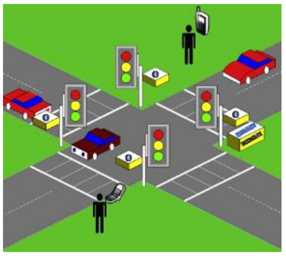
- Correctness of **YOU** in this course

  - Safety
    - You should never fail the exam

  - Liveness
    - You should eventually take the exam
    - You should eventually pass the exam !!!

## Correctness example (2)

- Correctness of a traffic light
- at intersection
  - Safety
    - Only one direction should have a green light
  - Liveness
    - Every direction should eventually get a green light



14

## Distributed algorithms abstractions

- Abstracting computers
  - => processes
- Abstracting communications
  - =>link (or channels)
- Abstracting time
  - => failure detector (*for a latter course but will be slightly introduced here*)

15

# Model and assumptions

- Specification needs to specify the model
  - Assumptions needed for the algorithm to be correct
- Model includes assumptions on:
  - Failure behavior of processes and channels
  - Timing behavior of processes and channels



# Node failures

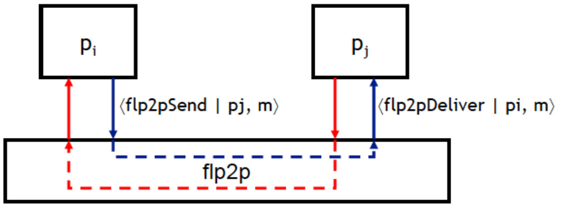
- Node may fail in 4 ways:
  - Crash-stop
  - Omissions
  - Crash-recovery
  - Byzantine / arbitrary
  - (these models are covered in details in another lecture)
- Nodes that don't fail in an execution are correct

## Channel failures modes

- Fair-Loss Links
  - Channels delivers any message sent with non-zero probability (no network partitions)
- Stubborn Links (“tétu”)
  - Channels delivers any message sent infinitely many times
- Perfect Links
  - Channels that deliver any message sent exactly once



## Fair-loss links



## Fair-loss links: Interfaces

- **Module:**
  - Name: FairLossPointToPoint (flp2p)
- **Events:**
  - Request: ⟨flp2pSend | dest, m⟩
    - Request transmission of message m to node dest
  - Indication:⟨flp2pDeliver | src, m⟩
    - Deliver message m sent by node src
- **Properties:**
  - *FL1, FL2, FL3.*

20

## Fair-loss links

- Properties
  - ***FL1. Fair-loss:*** If m is sent infinitely often by $p_i$ to $p_j$ , and neither crash, then m is delivered infinitely often by $p_j$

    remark: it could be the case that some of the infinite number of copies do not indeed arrive at their destination, but still destination receives a portion of this infinite number of copies, so, still an infinite number

  - ***FL2. Finite duplication:*** If a m is sent a finite number of times by $p_i$ to $p_j$ , then it is delivered at most a finite number of times by $p_j$
    - I.e. a message cannot be duplicated infinitely many times
  - ***FL3. No creation:*** No message is delivered unless it was sent

21

## Stubborn links: Interface

- **Module:**
  - Name: StubbornPointToPoint (sp2p)
- **Events:**
  - Request: ⟨sp2pSend | dest, m⟩
    - Request the transmission of message m to node dest
  - Indication:⟨sp2pDeliver src, m⟩
    - deliver message m sent by node src
- **Properties:**
  - *SL1, SL2*

22

## Stubborn links

- Properties
  - ***SL1. Stubborn delivery:*** if a node pi sends a message m to a correct node $p_j$, and $p_i$ does not crash, then $p_j$ delivers m an infinite number of times
  - ***SL2. No creation:*** if a message m is delivered by some node $p_j$, then m was previously sent by some node $p_i$

23

# Implementing Stubborn Links

- Implementation
  - Use the Lossy link
  - Sender stores every message it sends in *sent*
  - It periodically resends all messages in *sent*

- Correctness
  - ***SL1. Stubborn delivery***
    - If node doesn't crash, it will send every message infinitely many times. Messages will be delivered infinitely many times. Lossy link may only drop a (possibly large) fraction.
  - ***SL2. No creation***
    - Guaranteed by the Lossy link



# Algorithm (sl)

**Implements:**
StubbornPointToPointLink (sp2p).

**Uses:**
FairLossPointToPointLinks (flp2p).

**upon event** ⟨ *Init* ⟩ **do**
sent := $\emptyset$;
*startTimer* (TimeDelay);

**upon event** ⟨ *Timeout* ⟩ **do**
**forall** $(dest, m) \in$ sent **do**
**trigger** ⟨ *flp2pSend* | *dest*, *m* ⟩;
*startTimer* (TimeDelay);

**upon event** ⟨ *sp2pSend* | dest, m ⟩ **do**
**trigger** ⟨ *flp2pSend* | dest, m ⟩;
sent := sent $\cup$ {(dest,m)};

**upon event** ⟨ *flp2pDeliver* | src, m ⟩ **do**
**trigger** ⟨ *sp2pDeliver* | src, m ⟩;

## Perfect Links: Interface

- **Module:**
  - Name: PerfectPointToPoint (pp2p)
- **Events:**
  - Request: ⟨pp2pSend | dest, m⟩
    - Request the transmission of message m to node dest
  - Indication: ⟨pp2pDeliver | src, m⟩
    - deliver message m sent by node src
- **Properties:**
  - *PL1, PL2, PL3*

26

## Perfect links aka Reliable links

- Properties
  - ***PL1. Reliable Delivery:*** If neither $p_i$ nor $p_j$ crashes, then every message sent by $p_i$ to $p_j$ is eventually delivered by $p_j$
  - ***PL2. No duplication:*** Every message is delivered at most once
  - ***PL3. No creation:*** No message is delivered unless it was sent

27

# Perfect links aka Reliable links

- Question : Which one is safety/liveness/neither ?



# Perfect link implementation

- Implementation
  - Use Stubborn links
  - Receiver keeps log of all received messages in Delivered
    - Only deliver (call pp2pDeliver) messages that weren't delivered before
- Correctness
  - ***PL1. Reliable Delivery***
    - Guaranteed by Stubborn link. In fact the Stubborn link will deliver it infinite number of times
  - ***PL2. No duplication***
    - Guaranteed by our log mechanism
  - ***PL3. No creation***
    - Guaranteed by Stubborn link (and its lossy link?)

# Algorithm (pl)

**Implements:**
PerfectPointToPointLinks (pp2p).

**Uses:**
StubbornPointToPointLinks (sp2p).

**upon event** ⟨ *Init* ⟩ **do**
delivered := $\emptyset$;

**upon event** ⟨ *pp2pSend* | dest, m ⟩ **do**
**trigger** ⟨ *sp2pSend* | dest, m ⟩;

**upon event** ⟨ *sp2pDeliver* | src, m ⟩ **do**
**if** ($m \notin$ delivered) **then**
delivered := delivered $\cup$ { m };
**trigger** ⟨ *pp2pDeliver* | src, m ⟩;

30

# Default assumptions

- We assume perfect links (aka reliable) most of the lecture (unless specified otherwise)
  - Roughly, reliable links ensure messages exchanged between correct processes are delivered exactly once
- NB. Messages are uniquely identified and
  - the message identifier includes the sender's identifier
  - i.e. if “same” message sent twice, it's considered as two different messages

31

## GC – Definition

- A *group* = a collection of *users* or *objects* sharing a common interest
- Multicast, broadcast (special case of multicast)
- Purpose :
  - The purpose of introducing groups is to allow processes to deal with collections of processes as a single abstraction. Thus, a process can send a message to a group of servers without having to know how many there are or where they are.

33

## GC – Motivation

- From client-server to multi-participant systems
- Intuition:
  - distributed applications become bigger and more complex
    => interactions no longer limited to bilateral relationships
- Broadcast is useful for
  - applications where some processes subscribe to events published by other processes(e.g., stocks), and require some reliability guarantees from the broadcast service (we say sometimes quality of service–QoS) that the underlying network does not provide
- Broadcast is also useful for (database) replication
  - And is necessary in particular for a master, to maintain a global distributed state of slaves

34

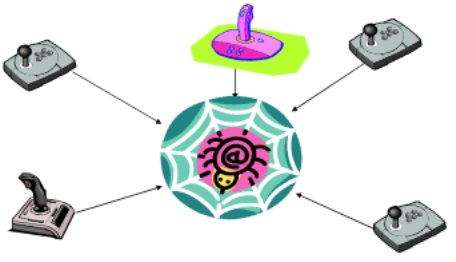
# GC – Motivation (cont'd)

- Who Needs Group Communication?
  - Highly available servers (client-server)
    - eg. cluster of J2EE servers
  - Database Replication
    - Fault tolerance by replicating the database nodes (State Machine Replication – SRM)
  - Multimedia Conferencing
    - eg Visio conf systems
  - Coordinated replicated services
    - eg. Name service for management of a cluster
      - ZooKeeper, using “Zab”: leader-based atomic broadcast protocol to maintain backup servers consistent replicated state
  - Online Games
  - …



# GC – Motivation (cont'd)

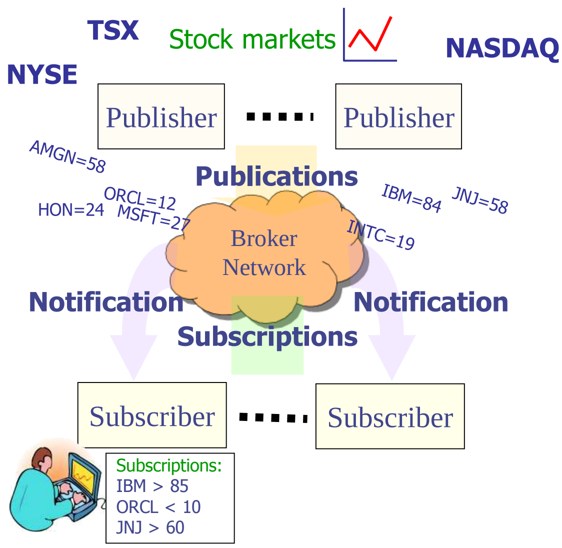
- Online game



- Fault-tolerance, order

# GC - Motivation (cont'd)

- Example: Stock Market




37

# GC - Motivation (cont'd)

| | Unreliable | Reliable |
|---|---|---|
| **Unicast** | UDP | TCP |
| **Multicast** | IP Multicast | ??? |

- IP Multicast is not activated everywhere on the Internet...

38

# Basic broadcast abstractions

39

# Best effort broadcast (beb Bcast)

- Best effort Bcast
  - **Intuition:** everything is perfect unless sender crash
  - **Module:**
    - Name: BestEffortBroadcast (beb).
  - **Events:**
    - ***Request:*** < bebBroadcast | m >: Used to broadcast message m to all processes.
    - ***Indication:* <** bebDeliver | src, m > : Used to deliver message m broadcast by process src.
  - **Properties:**
    - ***BEB1,BEB2,BEB3***

40

# Beb Bcast

- **Properties:**
  - ***BEB1:*** Best-effort validity: For any two processes $p_i$ and $p_j$,If pi and $p_j$ are correct, then every message broadcast by $p_i$ is eventually delivered by $p_j$ .
  - ***BEB2:*** No duplication: No message is delivered more than once.
  - ***BEB3:*** No creation: If a message m is delivered by some process $p_j$, then m was previously broadcast by some process $p_i$.

41

# Algorithm (beb bcast)

- Basic bcast

**Implements:**
BestEffortBroadcast (beb).

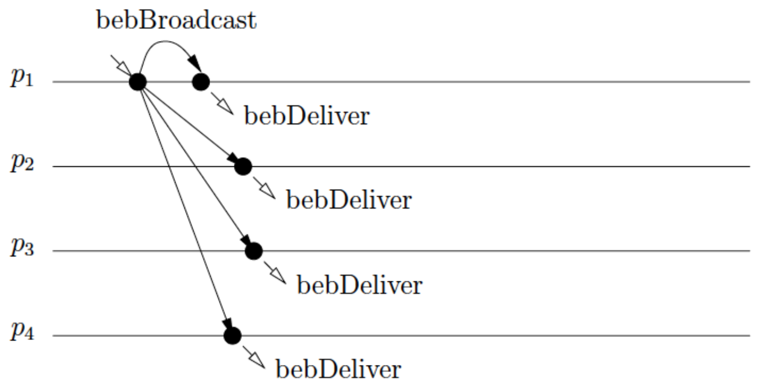
**Uses:**
PerfectPointToPointLinks (pp2p).

**upon event** ⟨ *bebBroadcast* | m ⟩ **do**
**forall** $p_i \in \Pi$ **do**
**trigger** ⟨ *pp2pSend* | $p_i, m$ ⟩;

**upon event** ⟨ *pp2pDeliver* | $p_i$, $m$ ⟩ **do**
**trigger** ⟨ *bebDeliver* | $p_i$, $m$ ⟩;

42

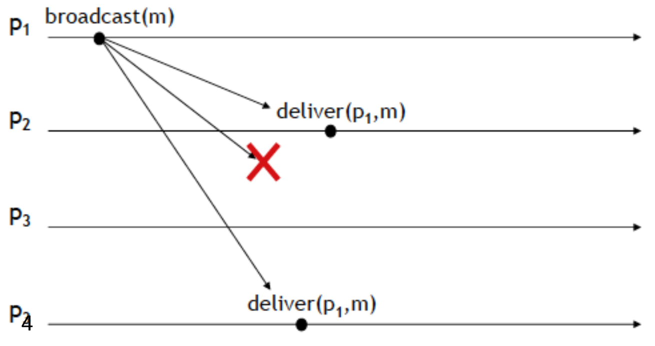
## Beb example (1)




43

## Beb example (2)

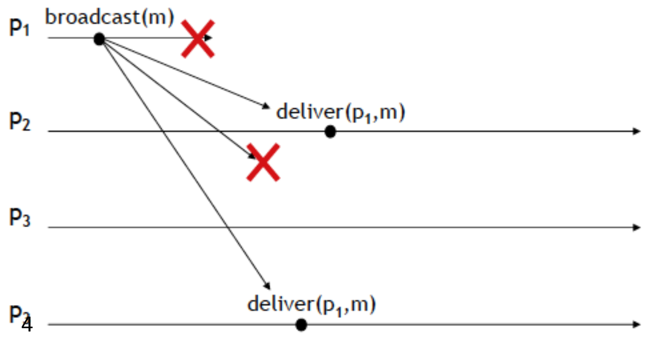
- Is this allowed ? Ie, does Beb algo could behave like that?
  - No because P1 not failed, so in the Beb mode, it should succeed to deliver m to P3 (violates BEB1)




44

# Beb example (3)

- Is this allowed ? Ie, does Beb algo could behave like that?
  - Yes because P1 is not correct, so in the Beb mode, it is not a problem that it could not deliver m to P3 (does not violate BEB1)




45

# Causal Order Bcast

- Motivation
  - Assume a chat application
    - Whatever written is broadcasted to a group
- If you get the following output, is it ok?

| | |
|---|---|
| [FBon] | Are you sure? Not in room 411 ? |
| [Mr Y] | Room 515, 5th floor at CS dept |
| [Mr X] | Does anyone know where the lecture is today? |

  - Mr X's message caused Mr Y's message
    - Mr Y's message caused FBon's message

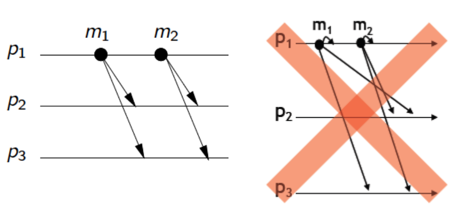
46

# Causality recalled, in context of GC

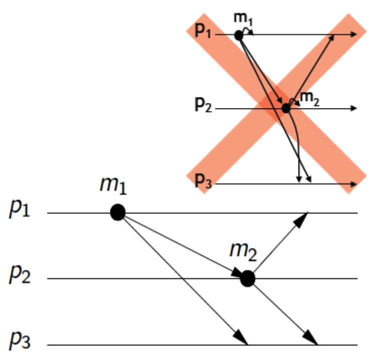
- Let $m_1$ and $m_2$ be any two messages:
  - $m_1 \rightarrow m_2$ ($m_1$ causally precedes $m_2$) if
  - ***C1 (FIFO order).***
    - Some process $p_i$ broadcasts $m_1$ before broadcasting $m_2$
  - ***C2 (Network order).***
    - Some process $p_i$ delivers $m_1$ and later broadcasts $m_2$
  - ***C3 (Transitivity).***
    - There is a message m' such that $m_1 \rightarrow$ m' and m' $\rightarrow m_2$

47

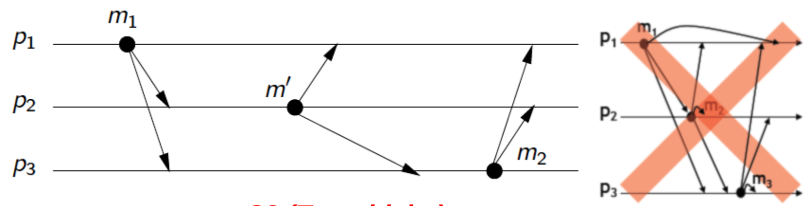
# Causal Order bcast




***C1 (FIFO order).***




***C2 (Network order).***




***C3 (Transitivity).***

48

# CO Bcast interface

- Module:
  - Name: CausalOrder (co)
- Events
  - Request: ⟨coBroadcast | m⟩
  - Indication: ⟨coDeliver | src, m⟩
- Property:
  - **CB: *causal delivery:*** If node $p_i$ delivers $m_1$, then $p_i$ must have delivered every message causally preceding ($\rightarrow$) $m_1$ before $m_1$



# Why Causal broadcast is difficult to design/implement **in the presence of faults** ?

- Exo: Come up with a simple Causal Order Broadcast algorithm using **best-effort bcast**…

Impossible!: it refers to FLP theorem =>

Because of the use of BEB: impossible to know that a message from the past is missing for ever because not transmitted by the crashed emitter process, or simply in transit.

FLP tells us: Impossible to decide / to agree (consensus); any protocol for consensus (or equivalently ordered bcast) is blocking in an asynchronous system with faults. Only way is to “handle” possible faults

So, if you deliver m, but later you receive an ancestor of m that was delayed, you violate CB property

=> Draw an illustrative example with 3 processes, and one fails

# Total Order Bcast

- Intuition:
  - Everyone delivers everything in exact same order

- For all messages $m_1$ and $m_2$ and all $p_i$ and $p_j$,
  - if both $p_i$ and $p_j$ deliver both messages, then they deliver
    them in the same order

- Difference between Causal and Total order
  - Causal enforces a global ordering for all messages that causally depend on each other
    - Such messages need to be delivered in the same order & this order must be respected causally.
  - Total ordering enforces ordering among all messages, even those that are not causally related.
- Warning!
  - Everyone delivers same order, maybe not send order! So… maybe not causal order

51

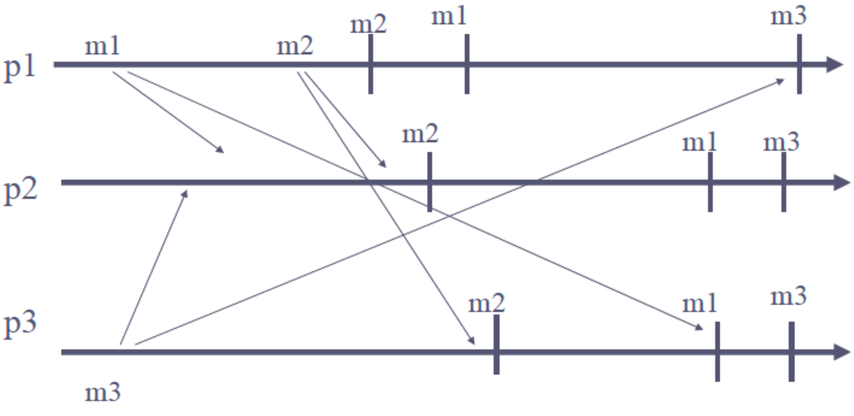
# Total Order Bcast

- Sometimes called *atomic broadcast*

  - Because the msg delivery occurs as if the broadcast were an indivisible primitive (i.e. atomic):
    - The message is delivered to all or to none of the processes and, if the message is delivered, every other message is ordered either before or after this message
- Convenient abstraction to maintain the consistency of replicas of a deterministic service
  - Exo: refer to TD1' exo4 "temperature measurement problem" !
- Two typical approaches for a total order broadcast algo.
  - Mechanism to stamp each msg to be delivered by all group members
    - Deliver msg on each group member by following total order of the stamps
  1. Use of a (centralized) sequencer, or
  2. Use logical clocks to stamp all copy of msg reception, make the max in a distributed manner (ABCAST algo.)

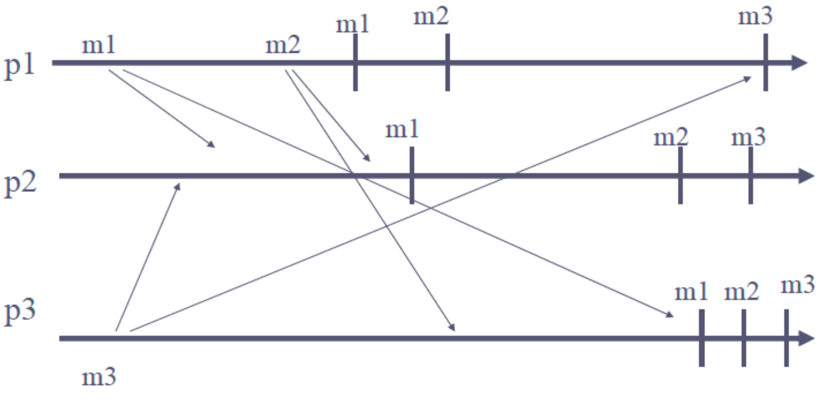
52

## Total Order Bcast




53

## Total Order Bcast




54

# Total Order Bcast (using reliable bcast see next): properties

**Module:**

**Name:** TotalOrder (to).

**Events:**

**Request:** ⟨ *toBroadcast* | $m$ ⟩: Used to broadcast message $m$ to $\Pi$.

**Indication:** ⟨ *toDeliver* | src, m ⟩: Used to deliver message $m$ sent by process *src*.

**Properties:**

**TO:** *Total order:* Let $m_1$ and $m_2$ be any two messages. Let $p_i$ and $p_j$ be any two correct processes that deliver $m_1$ and $m_2$. If $p_i$ delivers $m_1$ before $m_2$, then $p_j$ delivers $m_1$ before $m_2$.

**RB1:** *Validity:* If a correct process $p_i$ broadcasts a message $m$, then $p_i$ eventually delivers $m$.

**RB2:** *No duplication:* No message is delivered more than once.

**RB3:** *No creation:* If a message $m$ is delivered by some process $p_j$, then $m$ was previously broadcast by some process $p_i$.

**RB4:** *Agreement:* If a message $m$ is delivered by some correct process $p_i$, then $m$ is eventually delivered by every correct process $p_j$.



# A tour into reliable broadcast abstractions

## Reliable GC

- Why do we need reliable communication primitives ?
  - Network primitives are not enough...
  - Reliable applications need underlying services stronger than network protocols (TCP,UDP...)

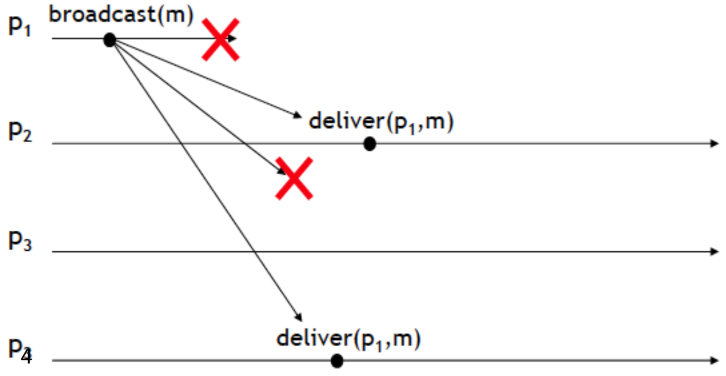


## Reliable GC (cont'd)

- Network protocols aren't enough...
  - Communications
    - Reliability guarantees (eg TCP) only offered for one-to-one communication (client-server)
    - How to do group communication?
  - High level services
    - Sometimes one-to-one communications is not enough
    - There is a need for reliable high-level services

# Reliable Bcast

- Unreliable broadcast




59

# Reliable Bcast

- BEB gives no guarantees if sender crashes

BEB1: For any 2 processes pi and pj,If pi and pj are correct, then every message broadcast by pi is eventually delivered by pj

- Reliable Broadcast Intuition
  - Same as BEB, + specific actions to give reliability property
  - If sender crashes:
    - ensure *all or none* of the correct nodes get msg

60

# Failure Detector

- But first...a word on Failure Detectors (FD)
  - Abstracting time
  - FD provide information (not necessary fully accurate) about which processes have crashed
  - Use failure detectors to encapsulate timing assumptions
    - Black box giving suspicions regarding node failures
    - Accuracy of suspicions depends on model strength
    - (Failure Detection discussed in details in another course)



# Failure Detector

- Typical Implementation of a Failure Detector
  - Periodically exchange heartbeat messages
  - Timeout based on worst case msg round trip
  - If timeout, then suspect node
  - If recv msg from suspected node, revise suspicion and increase time-out
- Two important requirements
  - 1.Completeness requirements
    - Requirements regarding actually crashed nodes
      - When do they have to be detected?
  - 2. Accuracy requirements
    - Requirements regarding actually alive nodes
      - When are they allowed to be suspected?

# Perfect Failure Detector

**Module:**

**Name:** PerfectFailureDetector ($\mathcal{P}$).

**Events:**

**Indication:** $\langle$ *crash* | $p_i$ $\rangle$: Used to notify that process $p_i$ has crashed.

**Properties:**

**PFD1:** *Strong completeness:* Eventually every process that crashes is permanently detected by every correct process.

**PFD2:** *Strong accuracy:* If a process $p$ is detected by any process, then $p$ has crashed.

63

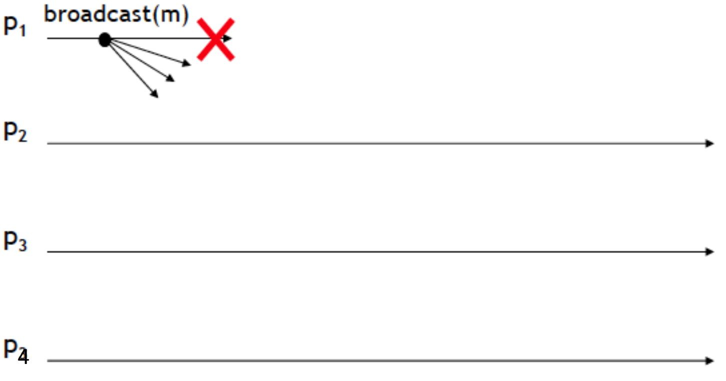
# Reliable Bcast

- Module:
  - Name: (regular)ReliableBroadcast (rb).
- Events:
  - Request: <rbBroadcast | m >: Used to broadcast message m.
  - Indication: < rbDeliver | src,m> : Used to deliver message m broadcast by process src.
- Properties:
  - ***RB1=Beb1:*** *Validity*: If a correct process $p_i$ broadcasts a message m, then $p_i$ eventually delivers m.
  - ***RB2=Beb2:*** *No duplication*: No message is delivered more than once.
  - ***RB3=Beb3:*** *No creation*: If a message m is delivered by some process $p_j$, then m was previously broadcast by some process $p_i$.
  - ***RB4:*** *Agreement*: If a message m is delivered by some correct process $p_i$,then m is eventually delivered by every correct process $p_j$ .

64

# Reliable Bcast – example 1

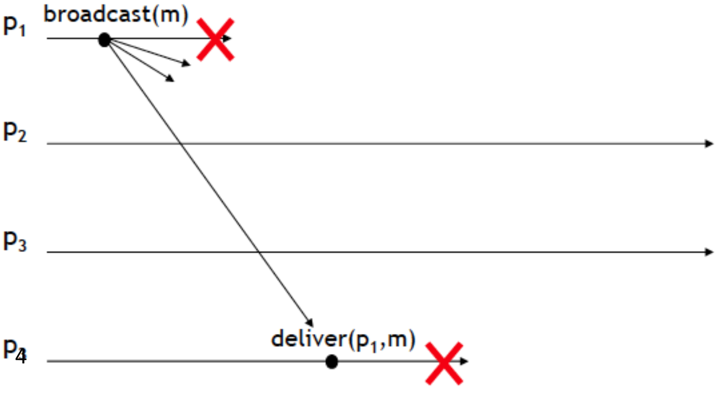
- Is this allowed ?





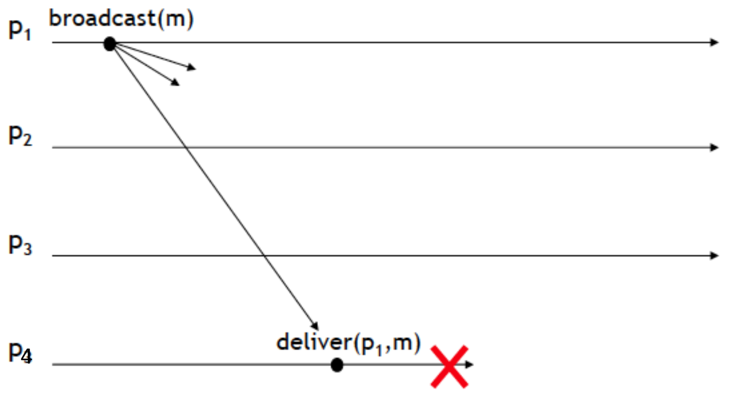

# Reliable Bcast – example 2

- Is this allowed ?



# Reliable Bcast – example 3

- Is this allowed ?






# Algorithm (RBcast)

**Implements:**
ReliableBroadcast (rb).

**Uses:**
BestEffortBroadcast (beb).
PerfectFailureDetector ($\mathcal{P}$).

**upon event** ⟨ *Init* ⟩ **do**
  delivered := $\emptyset$;
  correct := $\Pi$;
  **forall** $p_i \in \Pi$ **do**
    from$[p_i]$ := $\emptyset$;

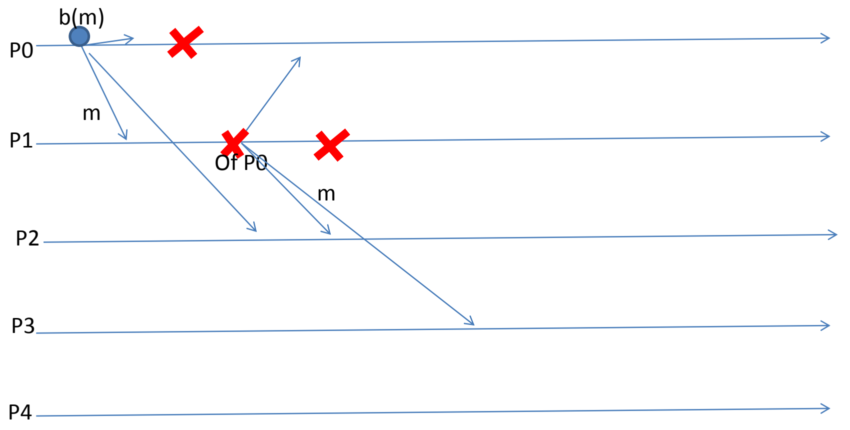
**upon event** ⟨ *rbBroadcast* | $m$ ⟩ **do**
  **trigger** ⟨ *bebBroadcast* | [DATA, self, $m$] ⟩;

**upon event** ⟨ *bebDeliver* | $p_i$, [DATA, $s_m$, $m$] ⟩ **do**
  **if** ($m \notin$ delivered) **then**
    delivered := delivered $\cup \{m\}$
    **trigger** ⟨ *rbDeliver* | $s_m$, $m$ ⟩;
    from$[p_i]$ := from$[p_i] \cup \{(s_m, m)\}$
    **if** ($p_i \notin$ *correct*) **then**
      **trigger** ⟨ *bebBroadcast* | [DATA, $s_m$, $m$] ⟩;

**upon event** ⟨ *crash* | $p_i$ ⟩ **do**
  correct := correct $\setminus \{p_i\}$
  **forall** $(s_m, m) \in$ from$[p_i]$ **do**
    **trigger** ⟨ *bebBroadcast* | [DATA, $s_m$, $m$] ⟩;

- Fail-stop model

# Initial Time-space diagram for exercice 1: continue the simulation




# Reliable Bcast plus uniformity

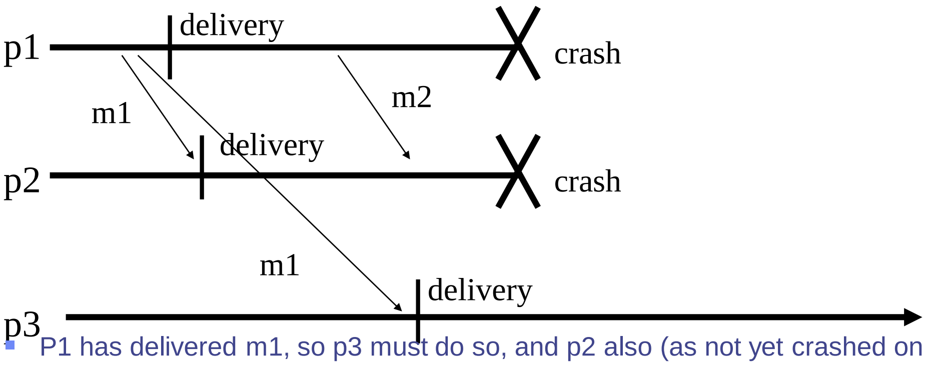
- BEB gives no guarantees if sender crashes

  BEB1: For any 2 processes pi and pj,If pi and pj are correct, then every message broadcast by pi is eventually delivered by pj

- Reliable Broadcast Intuition
  - Same as BEB, + specific actions to give reliability property
  - If sender crashes:
    - ensure *all or none* of the correct nodes get msg
    - What about the msg delivery on the sender and other processes even if crashed ?
      - A *property* (e.g. agreement) is *uniform* if faulty processes satisfy it as well
      - In practice: uniformity ensures that effect of an action (here, msg delivery) visible from the external world (not easily "cancellable") is the same on every (remaining correct) node. (e.g. bank account credit on a replica, even if fails afterwards, must be propagated to all correct replica!)
    - *URB. Uniform Agreement:* For any message m, if a process delivers m, then every correct process delivers m.
      - Then the Reliable bcast algo is also Uniform -URB
    - NB: In this course, we only focus on (non uniform) bcast algorithms

70

# Uniform reliable broadcast




- P1 has delivered m1, so p3 must do so, and p2 also (as not yet crashed on the picture!)
- P1 has not delivered m2, so p3 is not obliged to deliver m2, nor p2
- Assume no comm failure: on R-bcast, not only forward m to all but also RB-deliver m

# Reliable Causal Order Bcast

**Module:**

**Name:** ReliableCausalOrder (rco).

**Events:**

⟨ *rcoBroadcast* | *m* ⟩ and ⟨ *rcoDeliver* | src, m ⟩: with the same meaning and interface as the causal order interface.

**Properties:**

**RB1**–**RB4** from reliable broadcast and **CB** from causal order broadcast.

**CB:** If node $p_i$ delivers $m_1$, then $p_i$ must have delivered every message causally preceding ($\rightarrow$) $m_1$ before $m_1$

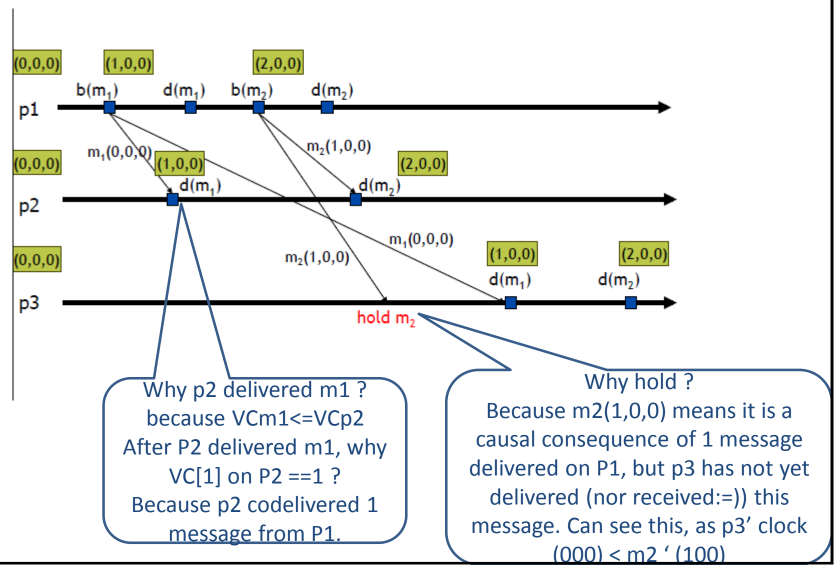
- Main idea
  - Each broadcasted message carries a history
  - Before delivery, ensure causality

72

# Reliable Causal Order Bcast Algo with Vector Clock

- Represent past history by vector clock (VC)
- Slightly modify the VC implementation
  - At node $p_i$
    - VC[i]: number of messages $p_i$ coBroadcasted
    - VC[j], $j \neq i$: number of messages $p_i$ coDelivered from $p_j$
- Upon CO broadcast m
  - Piggyback VC and RB broadcast m
- Upon RB delivery of m with attached $VC_m$ compare $VC_m$ with local $VC_i$
  - Only deliver m once $VC_m$ precedes ($\leq$) $VC_i$

# Execution



# Reliable Causal Order Bcast Algo with Vector Clock (contd)

- **Implements:**
  - ReliableCausalOrderBroadcast (rco)
- **Uses:** ReliableBroadcast (rb)

- **upon event** ⟨Init⟩ **do**
  - **forall** pi ∈ Π **do** VC[i] := 0

- **upon event** ⟨rcoBroadcast | m⟩ **do**
  - **trigger** ⟨rbBroadcast | (DATA, VC, m)⟩ send m with VC
  - VC[self] := VC[self] + 1
  - **trigger** ⟨rcoDeliver | self, m⟩ VC has only increased, so RCO deliver

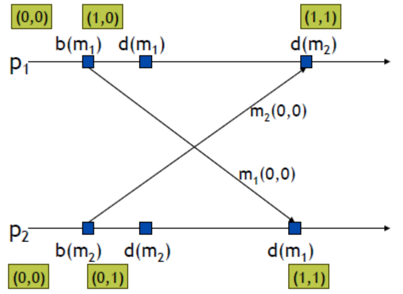
# Reliable Causal Order Bcast Algo with Vector Clock (contd)

- **upon event** ⟨rbDeliver | $p_j$, (DATA, $VC_m$ , m)⟩ **do**
  - **if** $p_j$ ≠ self **then**
    - pending := pending ∪ ($p_j$, (DATA, $VC_m$, m)) put on hold
    - deliver-pending()

for every message whose $VC_m$ precedes local VC

- **procedure** deliver-pending()
  - **while exists** x=($s_m$,(DATA,$VC_m$,m))∈pending s.t. VC≥$VC_m$ **do**
    - pending := pending \ ($s_m$, (DATA, $VC_m$, m)
    - **trigger** ⟨rcoDeliver | $s_m$, m⟩ Remove on hold deliver and increase local VC
    - VC[ rank($s_m$) ] := VC[ rank($s_m$) ] + 1

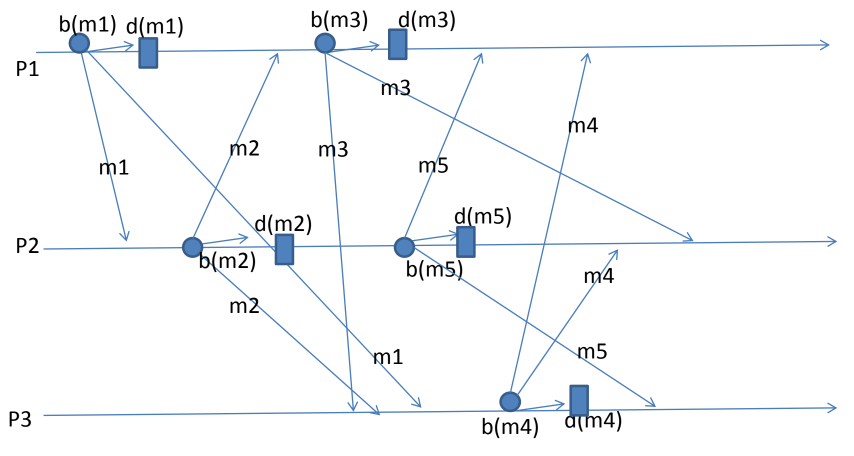
# Possible Execution




- Delivery order isn't same!

Nothing wrong , there is no causality.

# Time-space diagram for exercise 2




Additional questions:

1. comment on d(m3) on P2 has VCtimestamp==(2,2,0)
2. comment on m4 timestamp holding VCtimestamp==(2,1,0)

## Consensus vs. Group Comm ?

**Decide of a value among propositions, eg updating replicated variable A**

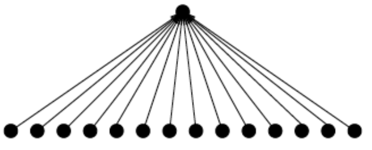
- All non failed nodes must agree the same
- To avoid node failures, need to suspect node crash
- Decision value per value
- Ordering among decided values can be
  - Total
    - Eg RSM any sequence A:=5;read A; A:=7
  - And FIFO (eg as in Zookeeper' AB: primary order FIFO)

**Decide to deliver m, a msg broadcasted, on each node**

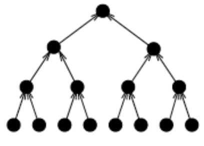
- All correct nodes must deliver the message
- Same: use PFD to be informed about nodes crash
- Decision msg per msg
- Ordering among delivered messages can be
  - FIFO
  - Causal
  - **Total**

## Scalability issues of Reliable Bcast

- The *Ack implosion* problem



- One way to circumvent it: using an *ack tree*



- Other mechanisms can also be used to circumvent this limitation
  - Epidemic dissemination (or *probabilistic broadcast*)

80

## Group Communication Systems

- Jgroups (www.jgroups.org)
- Appia (appia.di.fc.ul.pt)
- ISIS
- Horus
- Quicksilver
- Ensemble
- Joram, a “JMS compliant” library, from OW2
- ...