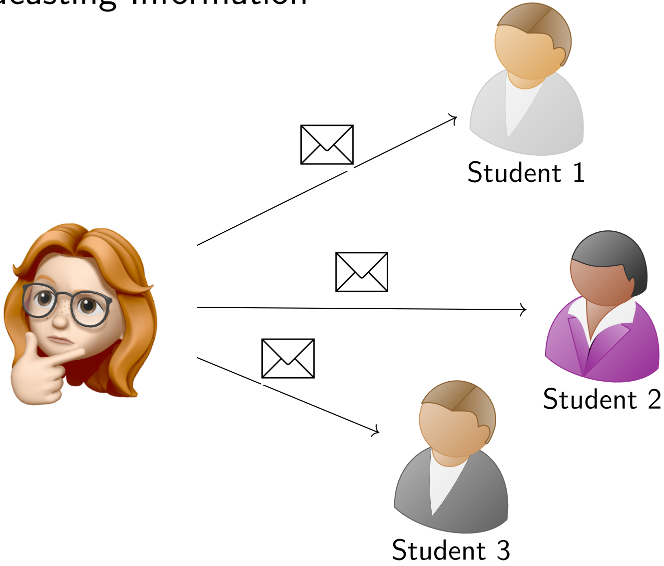
Programming Distributed Systems

Fault-tolerance in Message-Passing Distributed Systems

Annette Bieniusa

FB Informatik
TU Kaiserslautern

Broadcasting Information



The Need for Distributed Algorithms

- Distributed algorithms are at the core of any distributed systems
- Implemented as middelware between network and application
- Services beyond network protocols (e.g. TCP, UDP)
  - Group communication
  - Shared memory abstractions
  - Replicated state machines

Overview

- Formal models for specifying and analyzing distributed algorithms
- Composability of distributed algorithms
- The Broadcast Problem
  - Best-effort broadcast
  - Reliable broadcast
  - FIFO broadcast
  - Causal broadcast
  - Total-Order broadcast

Goals of this Learning Path

In this learning path, you will learn

- to formally specify safety and liveness properties of several broadcast problem
- to define fault-tolerant algorithms for Best-effort, Reliable, FIFO and Causal Broadcast in an asynchronous system with reliable channels
- to prove the correctness of these algorithms
- to use space-time diagrams to visualize executions
- to implement these algorithms in Erlang

The Broadcast Problem

Informally: A process needs to transmit a message other processes.

`broadcast(m)` $\approx$ `for each` $j \in \{1, \ldots, n\}$ `: send m to` p_j

System model

- Asynchronous system
  - no upper bound on message transfer delay
  - no failure detectors
- Static set of processes $\Pi = \{p_1, \ldots, p_n\}$
  - crash-stop fault model
- Sending and receiving messages through reliable channels (*perfect point-to-point links*)
  - no message loss / creation / modification / duplication
  - bidirectional
  - infinite capacity
- Messages are uniquely identifiable
  - e.g. tag with `<sender, seq_number>`

Only a subset $\Pi' \subseteq \Pi$ receives messages in arbitrary order at distinct, independent time instants.

What is the simplest solution that you can think of?

What is the simplest solution that you can think of?

Just go ahead and send the message to everyone, one at a time.

Specifying the Broadcast Algorithms

Wait... How do you specify an algorithm for a process again?

Specifying the Broadcast Algorithms

Wait... How do you specify an algorithm for a process again?

⇒ Deterministic I/O automaton with send/receive operations!

- Events: Messages, timers, conditions, ...
- Event-driven interface

```
Upon Event(arg1, arg2, ...) do:
  // local computation
  trigger Event(arg1', arg2',...)
```

- Correctness properties
  - Safety: Nothing bad ever happens
  - Liveness: Something good eventually happens

The Anatomy of a Broadcast Algorithm

For the broadcast algorithms:

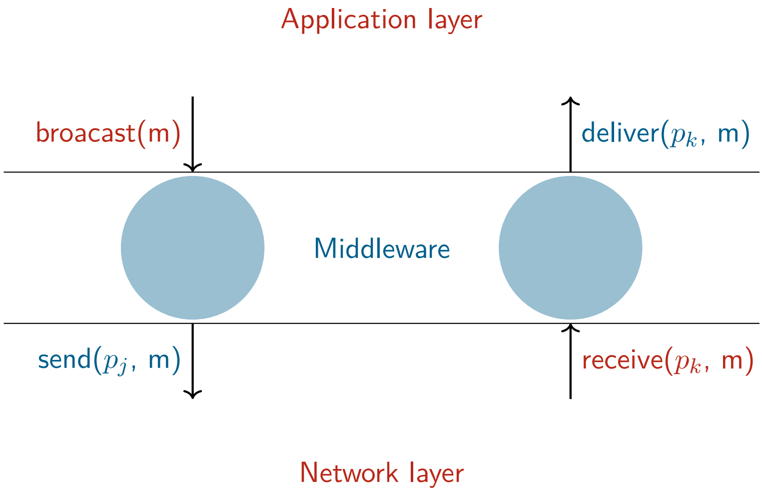
```
Upon Init            do: ...
Upon Broadcast(m)    do: ...
Upon Receive(p_k, m) do: ...
```

- You can trigger an event on another layer:

```
trigger Send(p_j, m)
trigger Deliver(p_k, m)
```

- There is a special event called `Init` for initializing the local state.
- p_j denotes the target process when sending a message
- p_k denotes the process where the message originated from

At Process p_i



Best-effort Broadcast (BEB): Specification

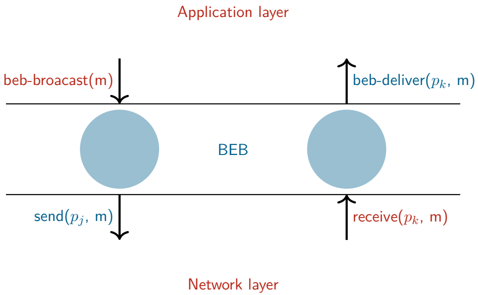
- *BEB-Validity:* If a correct process p_j beb-delivers a message m, then m has previously been beb-broadcast to p_j by some process p_i.
  - No creation, no alteration of messages
- *BEB-Integrity:* A process beb-delivers a message m at most once.
  - No duplication of messages
- *BEB-Termination:* For any two **correct** processes p_i and p_j, every message that has been beb-broadcast by p_i is eventually beb-delivered by p_j.

Best-effort Broadcast: Algorithm

Idea:

- Just go ahead and send the message to every other process.
- When you get one of these messages, you deliver it to the upper layer.
- Intuition: No guarantees if sender crashes

```
State:          --
Upon Init do:   --

Upon beb-broadcast(m) do:
  forall p_j ∈ Π:
    trigger send(p_j, m)

Upon receive(p_k, m) do:
  trigger beb-deliver(p_k, m)
```



Best-effort Broadcast: Correctness

Why does it work?

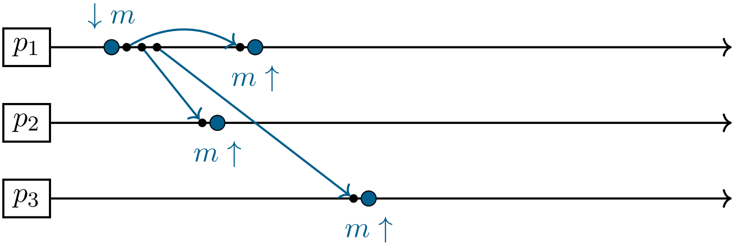
- BEB-Validity holds because Perfect-Link model guarantees no creation and there is no other way for messages to appear, only through beb-broadcast
- BEB-Integrity holds because Perfect-Link model guarantees no duplication
- BEB-Termination holds because Perfect-Link model guarantees reliable delivery

Perfect-Link Model

- **Reliable Delivery**: Considering two correct processes i and j; if i sends a message m to j, then j eventually delivers m.
- **No Duplication**: No message is delivered by a process more than once.
- **No Creation**: If a correct process j delivers a message m, then m was sent to j by some process i.

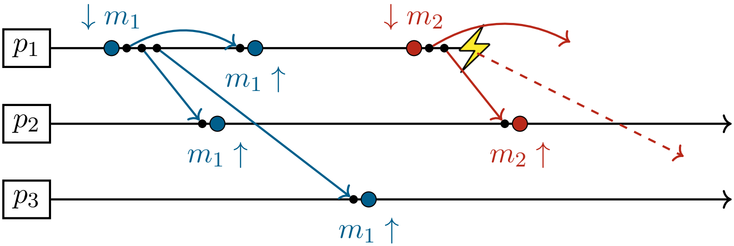
TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Visualizing Executions with Space-Time Diagrams



- $\downarrow m$ = broadcast message m
- $\uparrow m$ = deliver message m

Best-effort Broadcast: Sender crashes



Limitations of Best-effort Broadcast

What happens if a process fails while sending a message?

- If the sender crashes before being able to send the message to all processes, some process will not deliver the message.
- Even in the absence of communication failures!

Limitations of Best-effort Broadcast

What happens if a process fails while sending a message?

- If the sender crashes before being able to send the message to all processes, some process will not deliver the message.
- Even in the absence of communication failures!

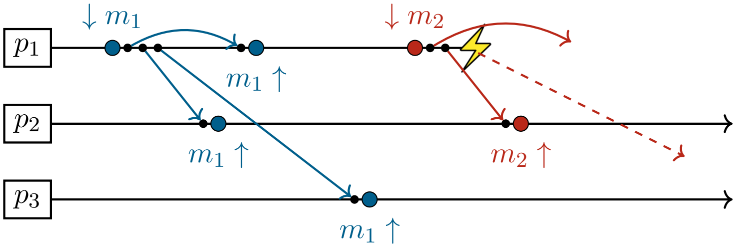
Let's try for a reliable version of broadcast!

- Guarantees that all or none of the correct nodes gets the message
- Even if sender crashes!

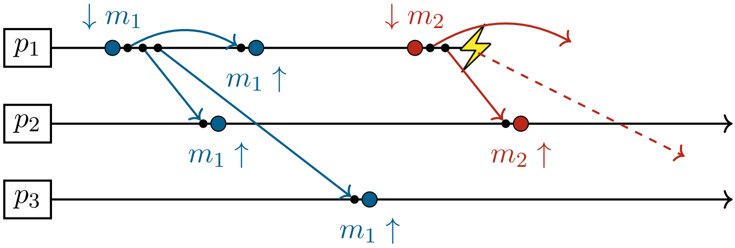
Reliable Broadcast (RB): Specification

- *RB-Validity:* If a correct process p_i rb-delivers a message m, then m has been previously rb-broadcast.
- *RB-Integrity:* A process rb-delivers a message m at most once.
- *RB-Termination-1:* If a correct process p_i rb-broadcasts message m, then p_i rb-delivers the message m.
- *RB-Termination-2:* If a correct process p_i rb-delivers a message m, then each correct process rb-delivers m.

Reliable Broadcast: Scenario 1



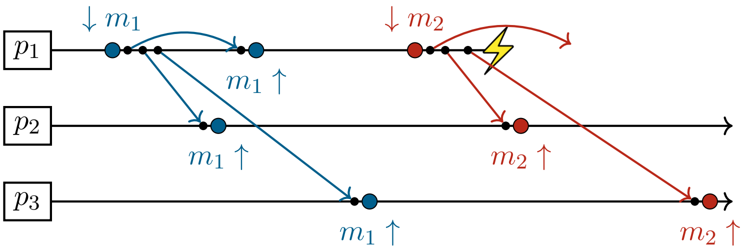
Reliable Broadcast: Scenario 1



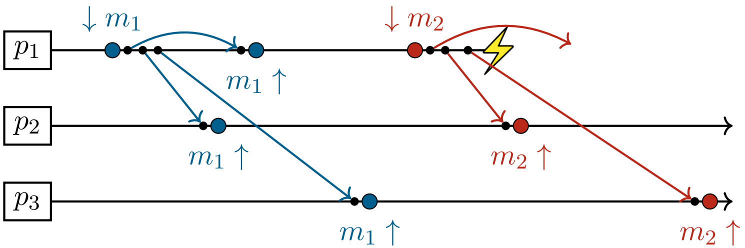
Not possible under Reliable Broadcast: RB-Termination-2 is violated!

If correct process p_2 delivers m, then correct process p_3 must also rb-deliver m.

Reliable Broadcast: Scenario 2

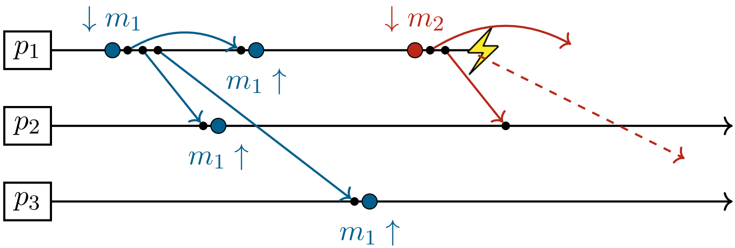


Reliable Broadcast: Scenario 2

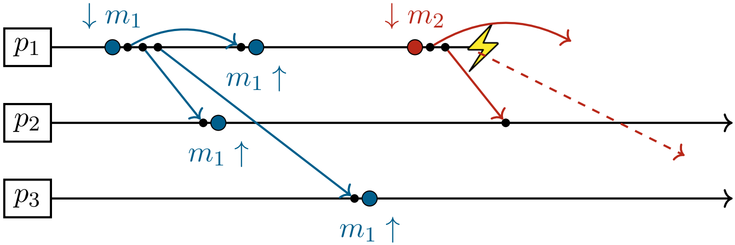


The fact that process p_1 does not deliver m_2 is not a problem, because only correct processes are required to deliver their own messages.

Reliable Broadcast: Scenario 3

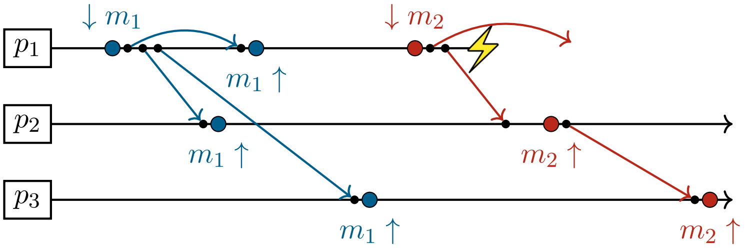


Reliable Broadcast: Scenario 3



The fact that no process delivers m_2 is not a problem, because process p_1 has crashed and no process delivers m_2.

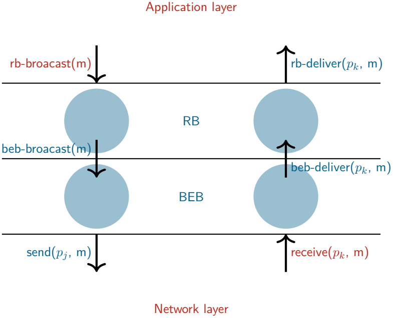
Reliable Broadcast: Idea!



TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Reliable Broadcast: Algorithm

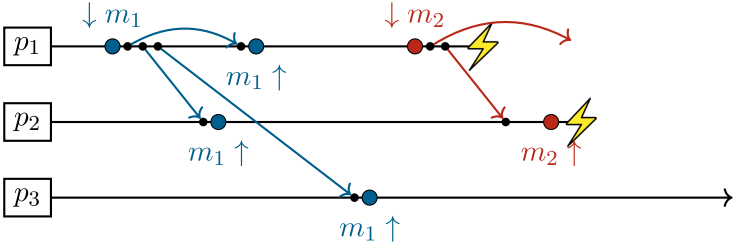
```
State:
  delivered //set of message ids
    that have already been delivered

Upon Init do:
  delivered <- ∅

Upon rb-broadcast(m) do
  m_id <- generateUniqueID(m)
  trigger beb-broadcast([m_id, m])

Upon beb-deliver(p_k, [m_id, m]) do
  if ( m_id ∉ delivered ) then
    delivered <- delivered ∪ {m_id}
    trigger rb-deliver(p_k, m)
    trigger beb-broadcast([m_id, m])
```



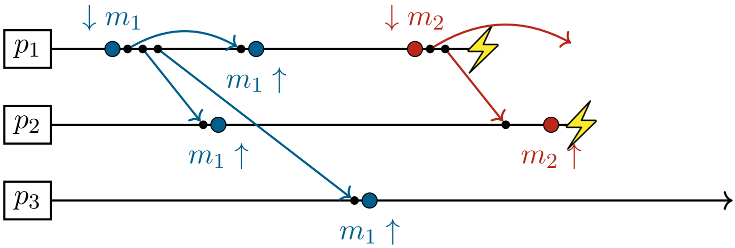
Reliable Broadcast: Correctness

- *RB-Validity:* If a correct process p_i rb-delivers a message m, then m has previously been rb-broadcast.
  - By BEB-Validity.
- *RB-Integrity:* A process rb-delivers a message m at most once.
  - By BEB-Integrity and handling the set of delivered messages.
- *RB-Termination-1:* If a correct process p_i broadcasts message m, then p_i eventually rb-delivers m.
  - By BEB-Termination and handling of the set of delivered messages.
- *RB-Termination-2:* If a correct process p_i rb-delivers a message m, then each correct process rb-delivers m.
  - After rb-delivering m, a correct process forwards m to all processes. By BEB-Termination and p_i being correct, all correct processes will eventually beb-deliver m and hence rb-deliver it.

Reliable Broadcast: Scenario 4



Reliable Broadcast: Scenario 4



The fact that m_2 has been delivered by faulty p_1 and p_2 does not imply that m_2 has to be delivered by p_3 as well. Yet, this situation is not desirable, because two processes deliver something and another one does not.

$\Rightarrow$ Interaction with external world!

Uniform Reliable Broadcast (URB): Specification

- *URB-Validity:* If a correct process p_i urb-delivers a message m, then m was urb-broadcast to p_i by some process p_j.
- *URB-Integrity:* A process p_i urb-delivers a message m at most once.
- *URB-Termination-1*: If a correct process p_i urb-broadcasts a message m, then p_i eventually urb-delivers m.
- *URB-Termination-2*: If a process p_i urb-delivers a message m, then each correct process p_j eventually urb-delivers m.

An Impossibility Result

- n: total number of processes
- t: upper bound on the number of processes that can fail
- Fail-silent system model: crash-stop + perfect point-to-point links

Theorem

There is no algorithm implementing URB under the fail-silent system model if a majority of processes can fail, i.e. if $t \geq n/2$.

Proof sketch

By contradiction.

- Assume there exists algorithm A that implements URB under the fail-silent model for $t \geq n/2$.
- Partition $\Pi = P_1 \cup P_2$ such that
  - $P_1 \cap P_2 = \emptyset$
  - $|P_1| = \lceil n/2 \rceil$ and $|P_2| = \lfloor n/2 \rfloor$ ($|P_1| \geq |P_2|$)
- Consider two executions E_1 and E_2
- Execution E_1:
  - All $p_i \in P_2$ crash initially, all processes in P_1 are correct.
  - $p_x \in P_1$ issues urb-broacast(m) using algorithm A
  - Every process in P_1 urb-delivers m

Proof sketch (2)

- Execution E_2:
  - No $p_i \in P_2$ crashes, and none of them issues urb-broadcast.
  - All processes in P_1 are correct.
  - $p_x \in P_1$ issues urb-broacast(m) using algorithm A
  - Every process in P_1 urb-delivers m and then crashes
  - Now, m is lost and can't be urb-delivered by processes in P_2, because perfect-link model requires sender and receiver to be correct for reliable delivery.
- E_1 and E_2 are indistiguishable by algorithm A.

Uniform Reliable Broadcast for $t < n/2$: Algorithm

```
State:
  delivered //set of message ids that have already been delivered
  pending // set of messages to be delivered
  ack // map m_id to received acknowledgments

Upon Init do:
  delivered, pending <- ∅
  ∀m_id: ack[m_id] = ∅

Upon urb-broadcast(m) do
  m_id <- generateUniqueID(m)
  pending <- pending ∪ {m_id}
  trigger beb-broadcast([self, m_id, m])
```

Uniform Reliable Broadcast for $t < n/2$: Algorithm (2)

```
Upon beb-deliver(p_k, [p_j, m_id, m]) do
  ack[m_id] <- ack[m_id] ∪ {k}
  if ( (p_j, m_id, m) ∉ pending ) then
    pending <- pending ∪ (p_j, m_id, m)
    trigger beb-broadcast([p_j, m_id, m])

Upon exists (p_j, m_id, m) ∈ pending
    with ack[m_id] > n/2 and m_id ∉ delivered
  delivered <- delivered ∪ m_id
  trigger urb-deliver(p_j, m)
```

Uniform Reliable Broadcast: Correctness

- Assume majority of correct processes ($t < n/2$)
- If a process urb-delivers, it got acknowledgement from majority
- In this majority, at least one process p must be correct
- p ensures that all correct processes beb-deliver m
- These correct processes (majority!) will ack and urb-deliver the message

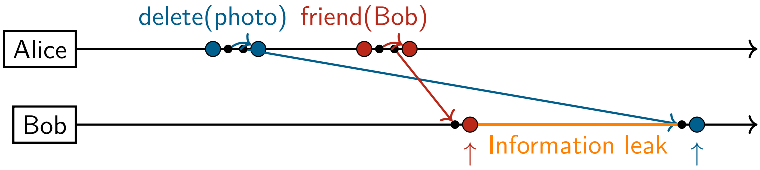
Resilience

- Defined by maximum number of faulty processes an algorithm can handle
- Algorithm for URB under fail-silent model has resilience $< n/2$

Problem: Message ordering

- Given the asynchronous nature of distributed systems, messages may be delivered in *any* order.
- Some services, such as replication, need messages to be delivered in a consistent manner, otherwise replicas may diverge.

FIFO Order

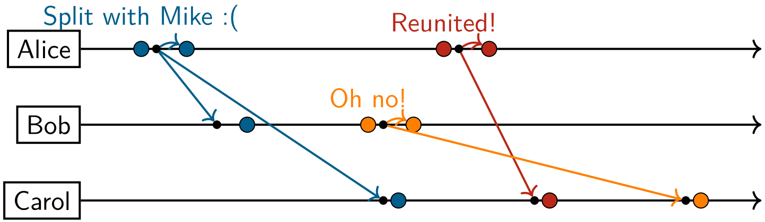


FIFO Property

If a process p broadcasts a message m before the same process broadcasts another message m', then no correct process q delivers m' unless it has previously delivered m.

$$broadcast_p(m) \rightarrow broadcast_p(m') \Rightarrow deliver_q(m) \rightarrow deliver_q(m')$$

Causal Order



Causality Property

If the broadcast of a message m happens-before the broadcast of some message m', then no correct process delivers m' unless it has previously delivered m.

$$broadcast_p(m) \rightarrow broadcast_q(m') \Rightarrow deliver_r(m) \rightarrow deliver_r(m')$$

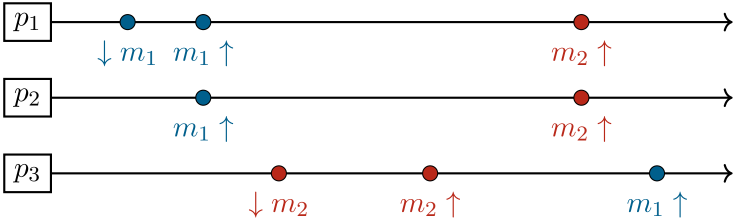
Total Order

Total Order Property

If correct processes p and q both deliver messages m, m', then p delivers m before m' if and only if q delivers m before m'.

$$deliver_p(m) \rightarrow deliver_p(m') \Rightarrow deliver_q(m) \rightarrow deliver_q(m')$$

Message ordering: Quizzzzz



Is this allowed under FIFO Order, Causal Order, Total Order?

(Reliable) FIFO Broadcast (FIFO): Specification

- All properties from reliable broadcast
- *FIFO delivery*: If a process fifo-broadcasts m and later m', then no process fifo-delivers m' unless it already delivered m.

FIFO-Broadcast: Algorithm

```
State:
  next    // array mapping process id to seq numer
  seq     // sequence numbers for broadcast messages
  pending // messages to be delivered
Upon Init do:
  next <- [0, ..., 0]
  seq <- 0
  pending <- ∅
Upon fifo-broadcast(m) do
  m_id <- seq++ // generate next seq number
  trigger rb-broadcast([m_id , m])

Upon rb-deliver(p_k, [m_id, m]) do
  if  m_id = next[p_k] then
    trigger fifo-deliver(p_k, m)
    next[p_k]++
    while exists (p_k, n_id, n) ∈ pending with n_id = next[p_k] do
      trigger fifo-deliver(p_k, n)
      next[p_k]++
      pending <- pending \ {(p_k, n_id, n)}
  else pending <- pending ∪  {(p_k, m_id, m)}
```

(Reliable) Causal Broadcast (RCO): Specification

- All properties from reliable broadcast
- *Causal delivery:* No process p_i delivers a message m' unless p_i has already delivered every message m such that $m \rightarrow m'$.

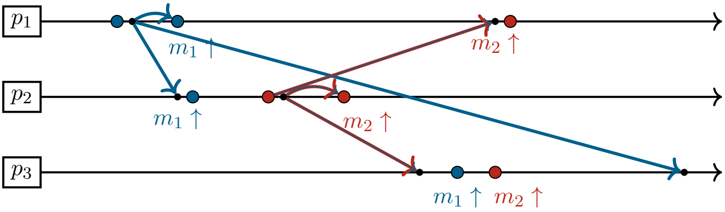
Idea

- Each messages carries $\texttt{past}_m$, an ordered list of messages that causally precede m
- When a process rb-delivers m,
  - it co-delivers first all causally preceding messages in $\texttt{past}_m$
  - it co-delivers m
  - avoiding duplicates using `delivered`

Causal Broadcast (RCO): Algorithm 1 (No-waiting)

```
State:
  delivered //set of messages ids that were already rco-delivered
  past      // ordered list
Upon Init do:
  delivered <- ∅
  past      <- []
Upon rco-broadcast(m) do
  m_id <- generateUniqueID(m)
  trigger rb-broadcast([m_id , past, m])
  past <- past ++ [(self, m_id, m)] // append at the end
Upon rb-deliver(p_k, [m_id, past_m, m]) do
  if ( m_id ∉ delivered ) then
    for (p_j, n_id, n) : past_m do // from old to recent
      if (n_id ∉ delivered ) then
        trigger rco-deliver(p_j, n)
        delivered <- delivered ∪ {n_id}
        if (p_j, n_id, n) ∉ past then
            past <- past ++ [(p_j, n_id, n)]
    trigger rco-deliver(p_k, m)
    delivered <- delivered ∪ {m_id}
    if (p_k, m_id, m) ∉ past then
        past <- past ++ [(p_k, m_id, m)]
```

Causal Broadcast: Scenario 1



Causal Broadcast - Algorithm 1: Correctness

- Validity follows directly from rb-broadcast
- Integrity follows from rb-broadcast and the check before rco-delivering messages from $\texttt{past\_}m$
- Termination follows directly from rb-broadcast and the fact that no waiting occurs
  - Every message is rco-delivered once rb-delivered
- Causal delivery
  - Each message m carries its causal past
  - Causal past is in order delivered before m
  - Proof by induction on trace prefix
    - Initial state
    - For every delivery

Remarks

- Message from causal past of m are delivered before message m (*causal delivery*)
- Message id's could be reused for rb-broadcast
- Size of messages grows linearly with every message that is broadcast since it includes the complete causal past

Idea: Garbage collect the causal past

- If we know when a process fails (i.e., under the fail-stop model), we can remove messages from the causal past.
- When a process rb-delivers a message m, it rb-broadcasts an acknowledgement message to all other processes.
- When an acknowledgement for message m has been rb-delivered by all correct processes, m is removed from $past$
- N^2 additional ack messages for each application message
- Typically, acknowledgements are grouped and processed in batch mode

$\Rightarrow$ Requires still unbounded messages sizes

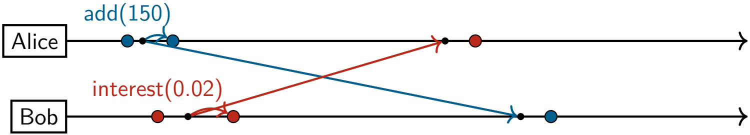
Causal Broadcast (RCO): Algorithm 2 [1]

```
State:
  pending //set of messages that cannot be delivered yet
  VC // vector clock
Upon Init do:
  pending <- ∅
  forall p_i ∈ Π do: VC[p_i] <- 0

Upon rco-broadcast(m) do
  trigger rco-deliver(self, m)
  trigger rb-broadcast(VC, m)
  VC[self] <- VC[self] + 1

Upon rb-deliver(p_k, VC_m, m) do
  if ( p_k ≠ self ) then
    pending <- pending ∪ {(p_k, VC_m, m)}
    while exists (q, VC_{m_q}, m_q) ∈ pending with VC ≥ VC_{m_q} do
        pending <- pending \ {(q, VC_{m_q}, m_q)}
        trigger rco-Deliver(q, m_q)
        VC[q] <- VC[q] + 1
```

Limitations of Causal Broadcast

Example: Replicated database handling bank accounts

- Initially, account A holds 1000 Euro.
- User deposits 150 Euro, triggers broadcast of message $m_1 =$ `'add 150 Euro to A'`
- Concurrently, bank initiates broadcast of message $m_2 =$ `'add 2% interest to A'`
- Diverging state because processes can observe messages in different order



Outlook: Total-order broadcast (aka Atomic Broadcast)

- All processes deliver their messages in the same order
- Replicated services
  - Multiple processes execute the same sequence of commands
  - Replicated State Machines (RSM)
- Impossibile under our assumed system model

Summary

- Composability of distributed algorithms
- Correctness proofs based on properties of underlying level + algorithmic properties
- Different variants of solution to the Broadcast Problem
  - Best-effort broadcast
    - Reliable only if sender is correct
  - Reliable broadcast
    - Reliable independent of whether sender is correct
  - Uniform reliable broadcast
    - Considers also behavior of failed nodes
  - FIFO broadcast
    - Reliable broadcast with FIFO delivery order
  - Causal broadcast
    - Reliable broadcast with causal delivery order
  - Total-order broadcast
    - Reliable and same order of delivery at all nodes

Further reading I

[1] Michel Raynal, André Schiper und Sam Toueg. „The Causal Ordering Abstraction and a Simple Way to Implement it“. In: *Inf. Process. Lett.* 39.6 (1991), S. 343–350. DOI: 10.1016/0020-0190(91)90008-6. URL: https://doi.org/10.1016/0020-0190(91)90008-6.