Causal Reliable Broadcast

Seif Haridi - Royal Institute of Technology
Peter Van Roy – Université catholique de Louvain

haridi(at)kth.se
peter.vanroy(at)uclouvain.be

Motivation

- Assume we have a chat application
  - Whatever written is reliably broadcast to group
- If you get the following output, is it ok?

```
[ali]   Are you sure? Not E?
[MrsY]  Auditorium D at Forum
[MrX]   Does anyone know where the
        lecture is today?
```

- MrX' s message caused MrsY' s message,
  - MrsY's message caused Ali's message

Motivation (2)

- Does uniform reliable broadcast remedy this? [d]
- Causal reliable broadcast solves this
  - Deliveries in causal order!
- Causality is same as happened-before relation by Lamport!

Causality Recalled

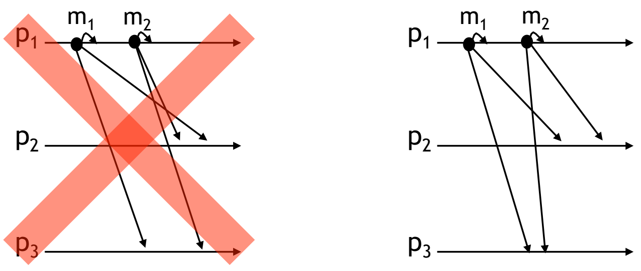
- Let m_1 and m_2 be any two messages: $m_1 \rightarrow m_2$ (m_1 causally precedes m_2) if
  - ***C1 (FIFO order)*.**
    - Some process p_i broadcasts m_1 before broadcasting m_2
  - ***C2 (Network order)*.**
    - Some process p_i delivers m_1 and later broadcasts m_2
  - ***C3 (Transitivity)*.**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

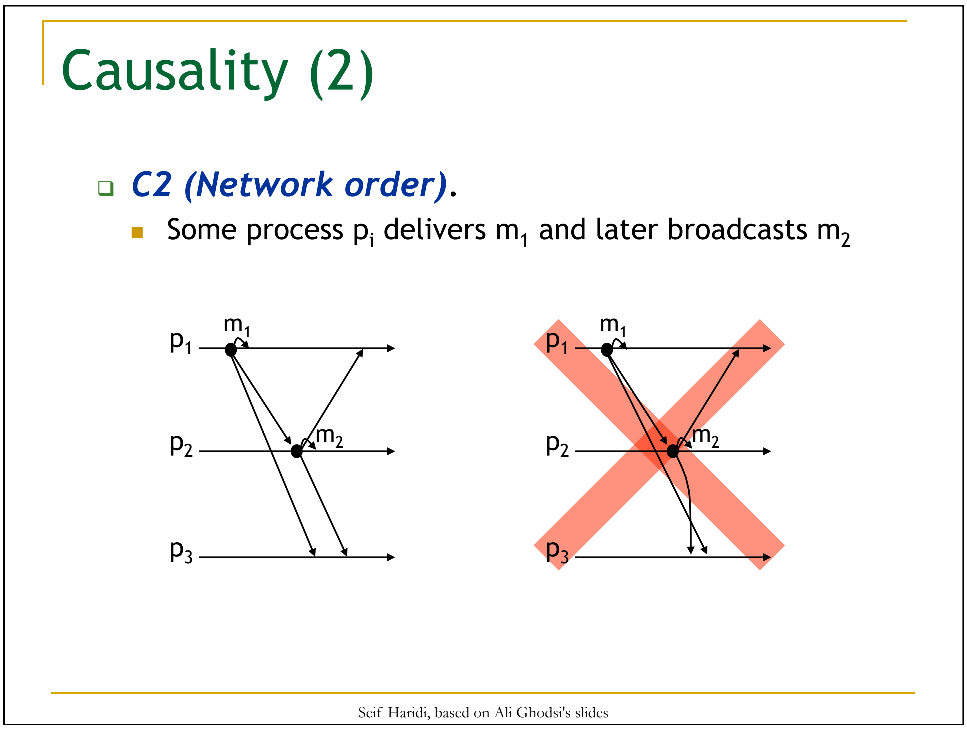
Causal Broadcast Interface

- **Module:**
  - Name: CausalOrder (co)
- ***Events***
  - Request: ⟨coBroadcast | m⟩
  - Indication: ⟨coDeliver | src, m⟩
- ***Property:***
  - ***CB***: If node p_i delivers m_1, then p_i must have delivered every message causally preceding (→) m_1 before m_1
- Is this useful? How can it be satisfied? [d]
  - It is only safety. Satisfy it by never delivering!

Causality

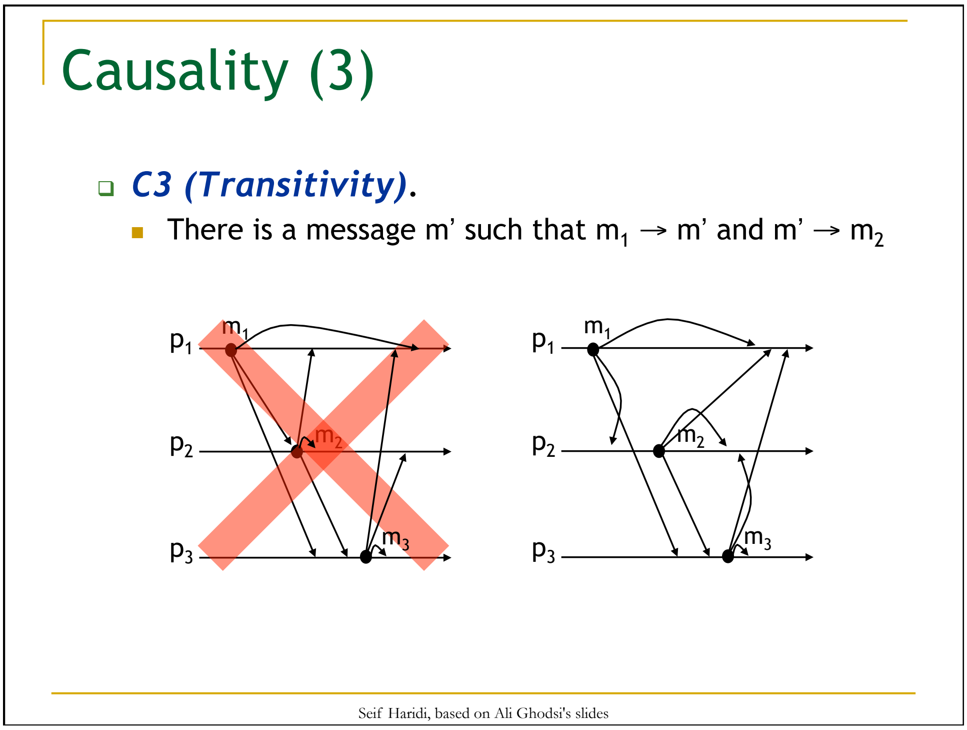
- ***C1 (FIFO order)*.**
  - Some process p_i broadcasts m_1 before broadcasting m_2





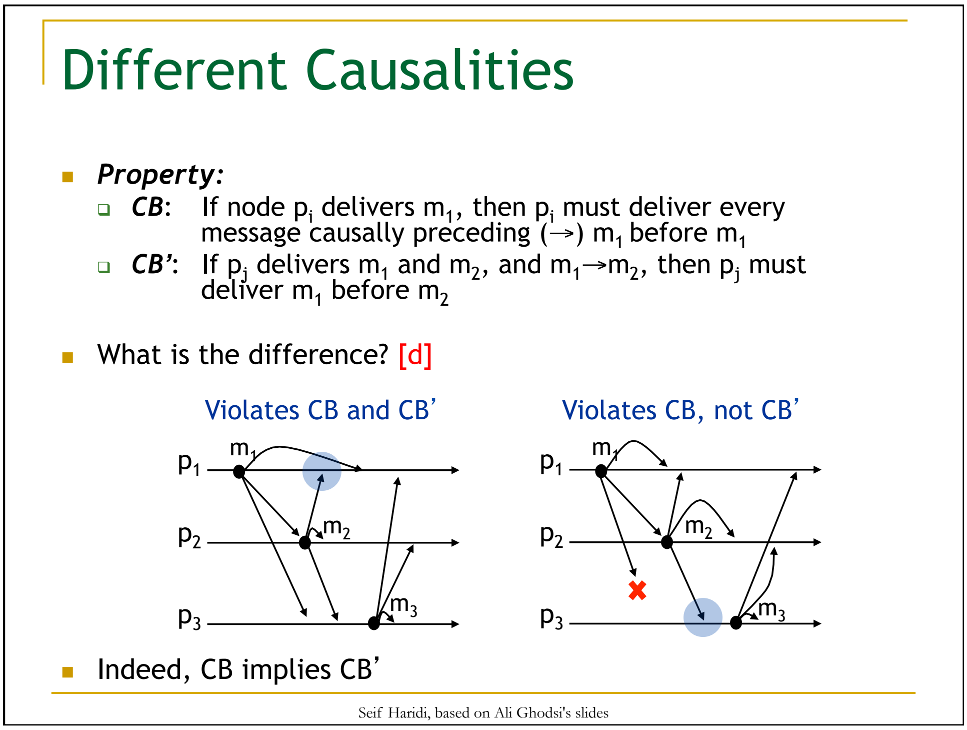
Causality (2)

- *C2 (Network order)*.
  - Some process p_i delivers m_1 and later broadcasts m_2



Causality (3)

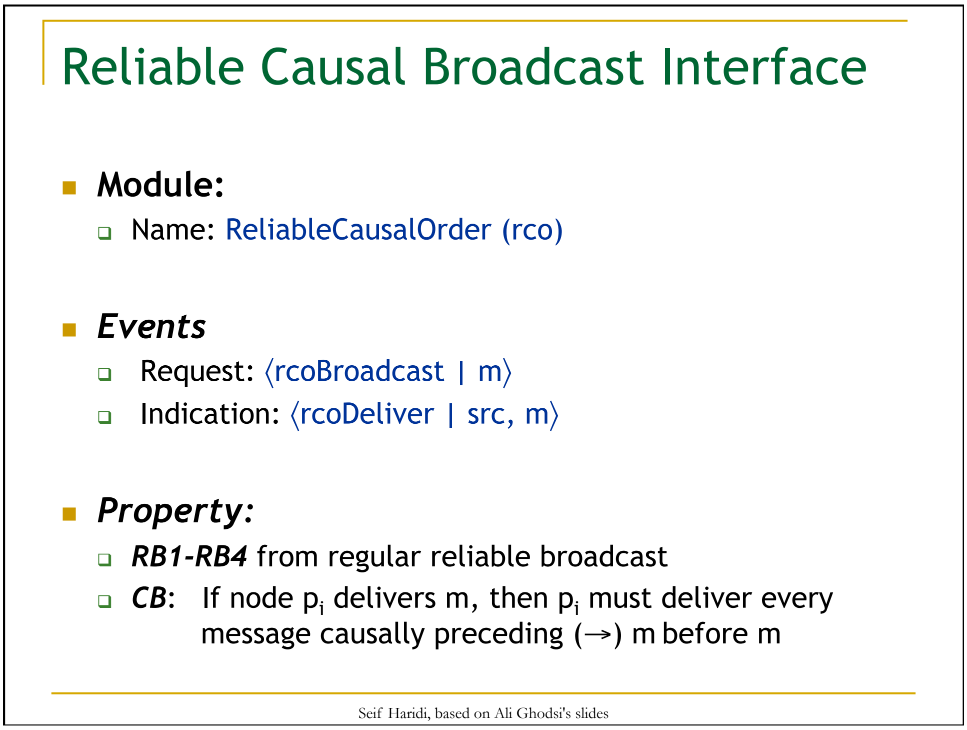
- *C3 (Transitivity)*.
  - There is a message m’ such that $m_1 \rightarrow m'$ and $m' \rightarrow m_2$



Different Causalities

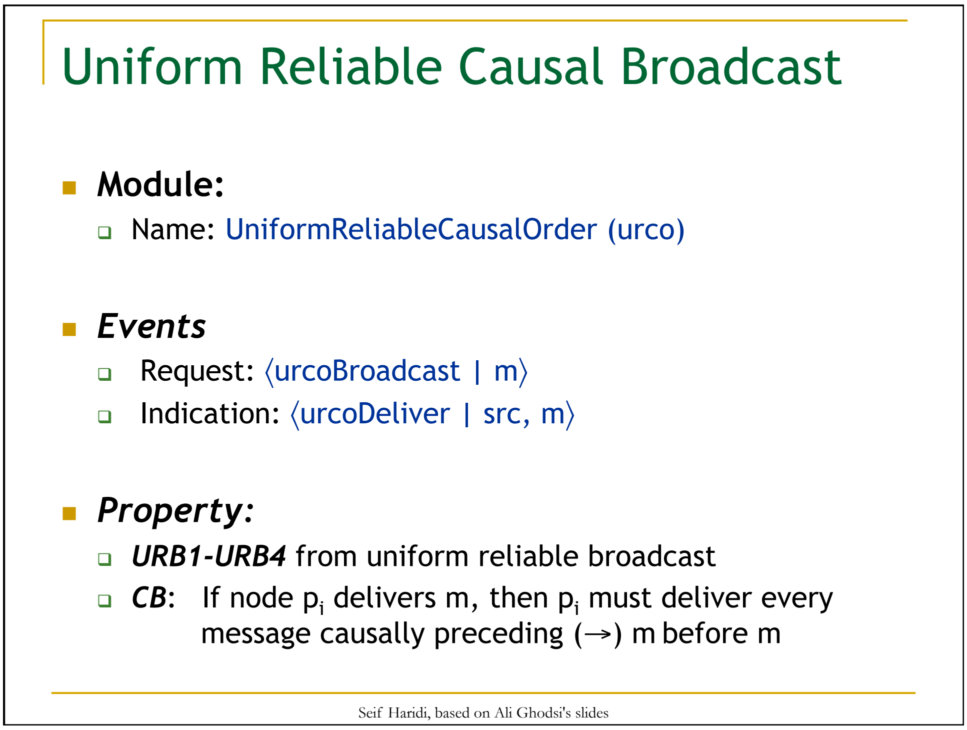
- ***Property:***
  - ***CB***: If node p_i delivers m_1, then p_i must deliver every message causally preceding (→) m_1 before m_1
  - ***CB'***: If p_j delivers m_1 and m_2, and $m_1 \rightarrow m_2$, then p_j must deliver m_1 before m_2
- What is the difference? [d]

- Indeed, CB implies CB'



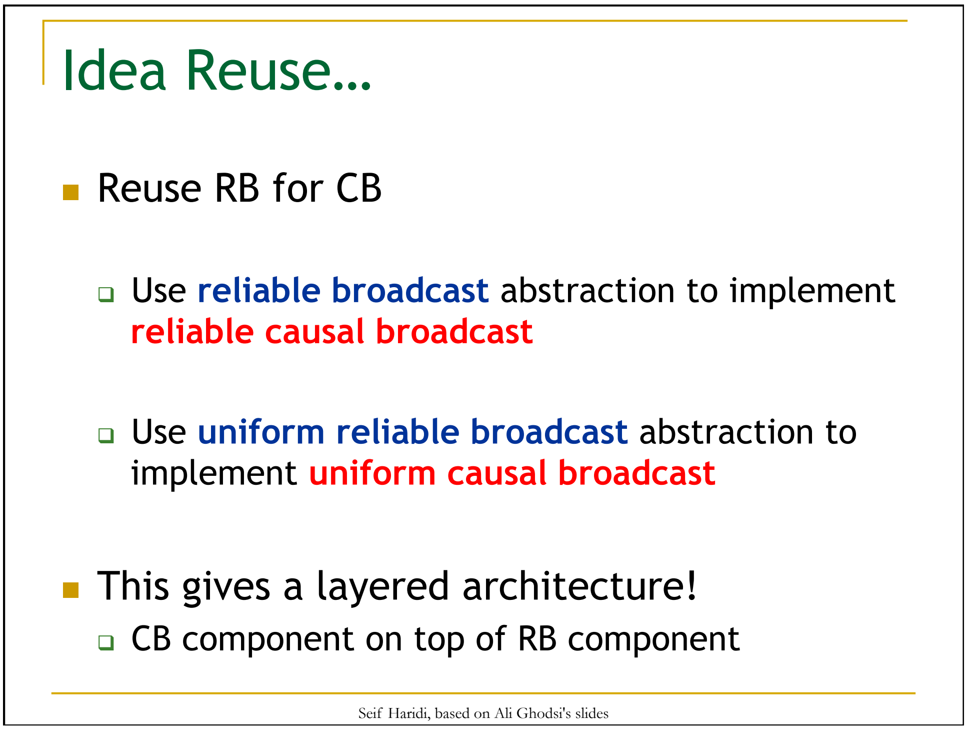
Reliable Causal Broadcast Interface

- **Module:**
  - Name: ReliableCausalOrder (rco)
- ***Events***
  - Request: ⟨rcoBroadcast | m⟩
  - Indication: ⟨rcoDeliver | src, m⟩
- ***Property:***
  - ***RB1-RB4*** from regular reliable broadcast
  - ***CB***: If node p_i delivers m, then p_i must deliver every message causally preceding (→) m before m



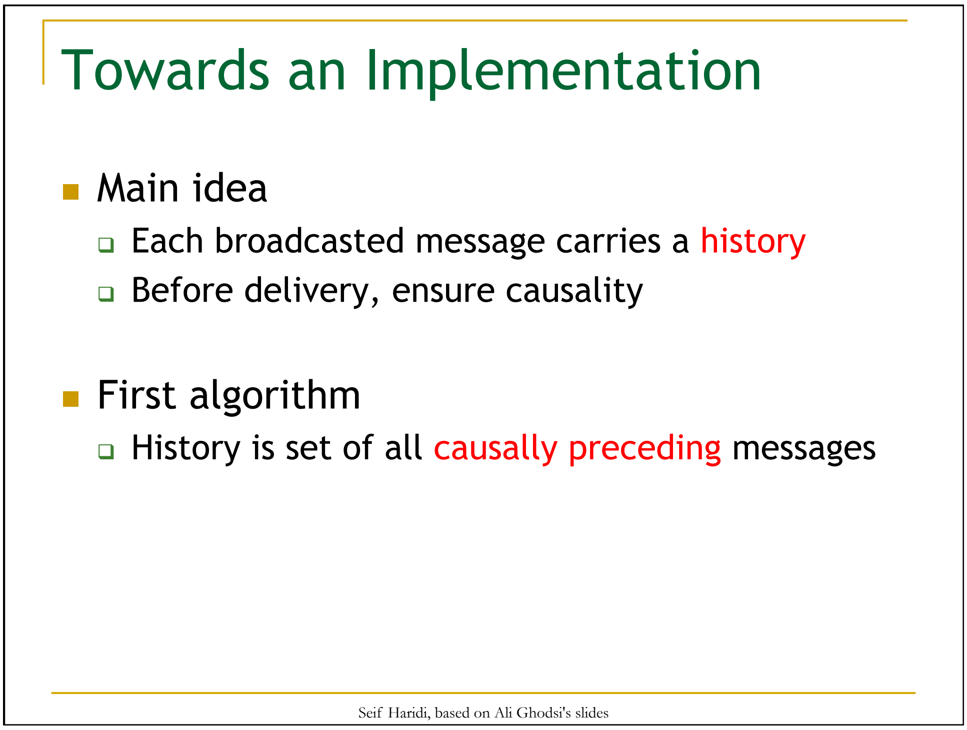
Uniform Reliable Causal Broadcast

- **Module:**
  - Name: UniformReliableCausalOrder (urco)
- ***Events***
  - Request: ⟨urcoBroadcast | m⟩
  - Indication: ⟨urcoDeliver | src, m⟩
- ***Property:***
  - ***URB1-URB4*** from uniform reliable broadcast
  - ***CB***: If node p_i delivers m, then p_i must deliver every message causally preceding (→) m before m



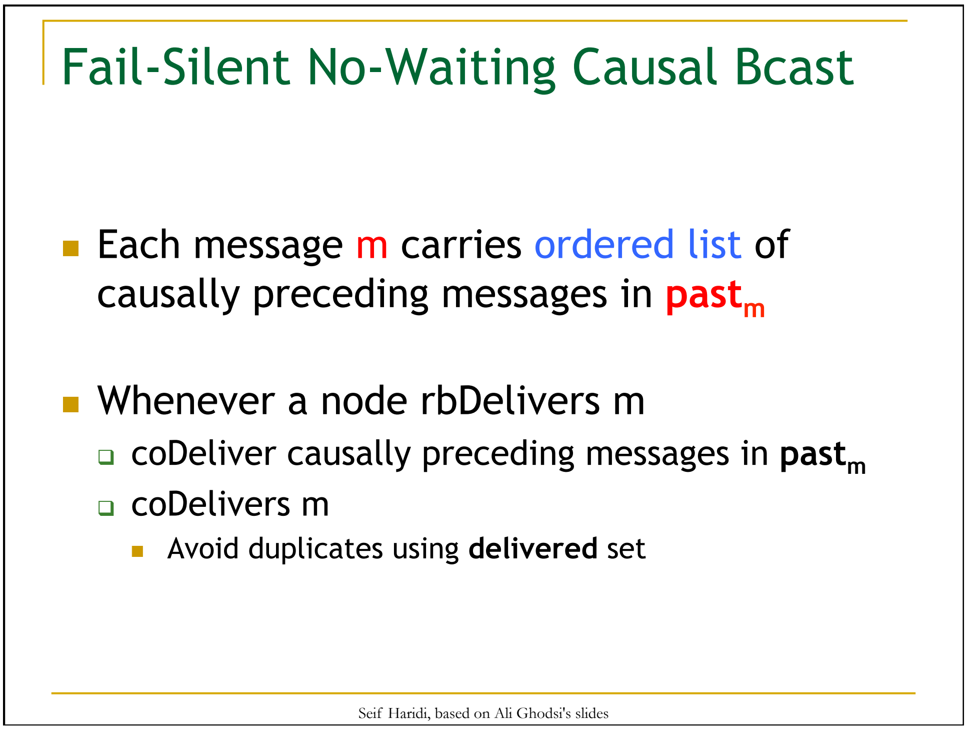
Idea Reuse…

- Reuse RB for CB
  - Use **reliable broadcast** abstraction to implement **reliable causal broadcast**
  - Use **uniform reliable broadcast** abstraction to implement **uniform causal broadcast**
- This gives a layered architecture!
  - CB component on top of RB component



Towards an Implementation

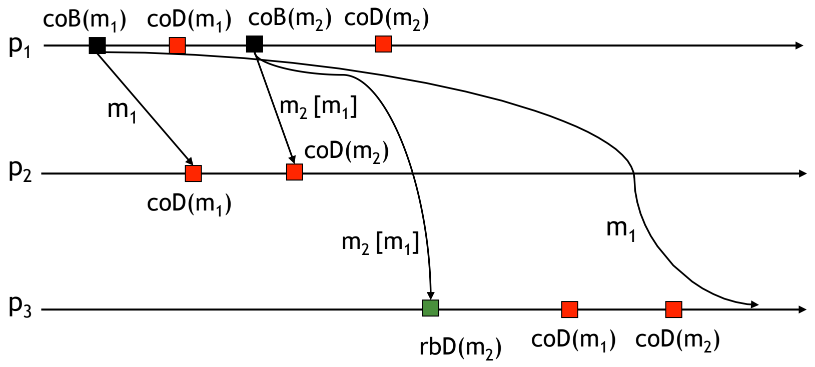
- Main idea
  - Each broadcasted message carries a history
  - Before delivery, ensure causality
- First algorithm
  - History is set of all causally preceding messages



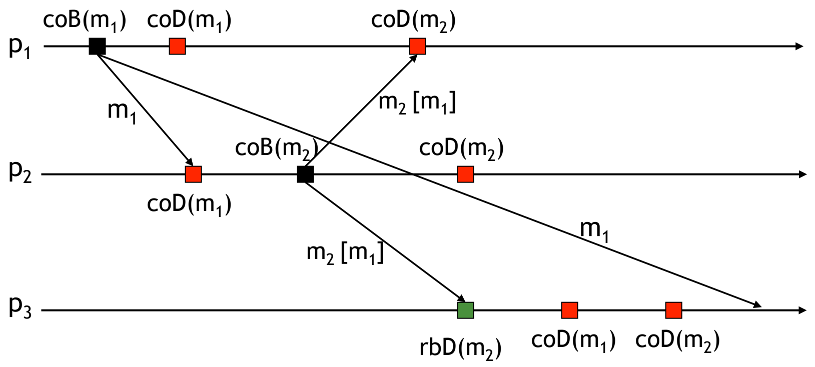
Fail-Silent No-Waiting Causal Bcast

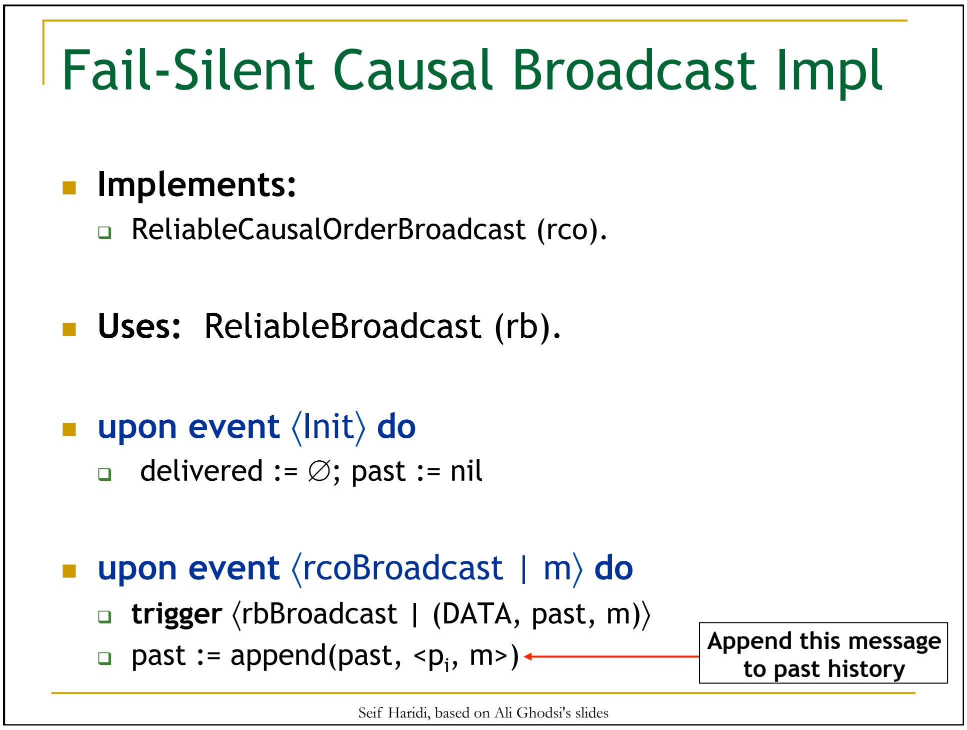
- Each message m carries ordered list of causally preceding messages in **past$_m$**
- Whenever a node rbDelivers m
  - coDeliver causally preceding messages in **past$_m$**
  - coDelivers m
    - Avoid duplicates using **delivered** set

Execution (direct override)



Execution (indirect override)





Fail-Silent Causal Broadcast Impl

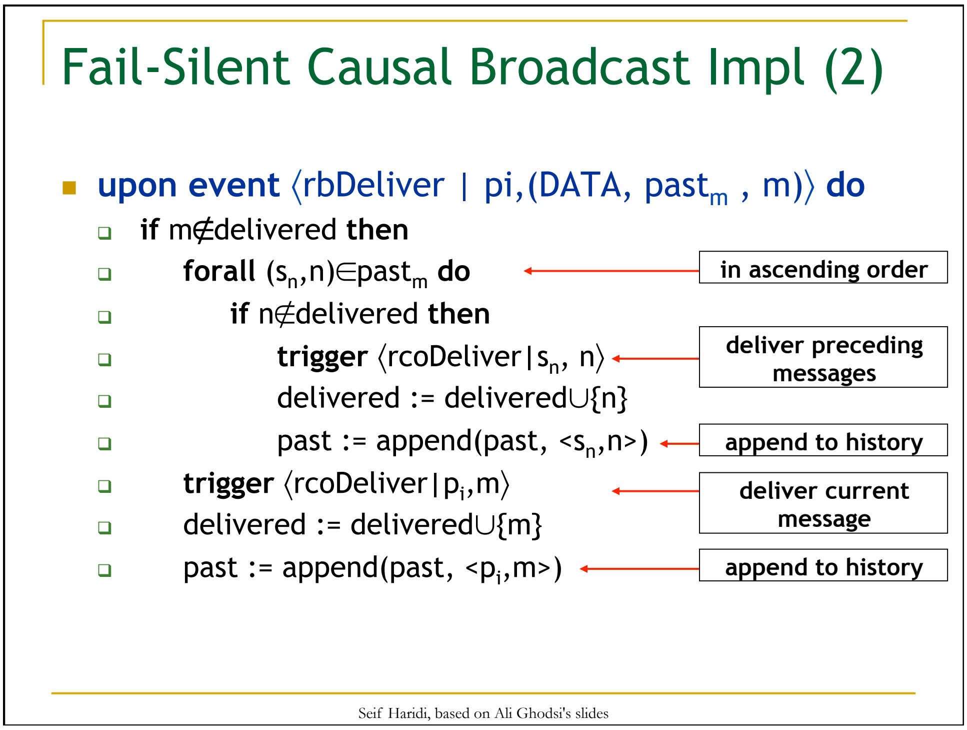
- **Implements:**
  - ReliableCausalOrderBroadcast (rco).
- **Uses:** ReliableBroadcast (rb).

```
upon event ⟨Init⟩ do
    delivered := ∅; past := nil

upon event ⟨rcoBroadcast | m⟩ do
    trigger ⟨rbBroadcast | (DATA, past, m)⟩
    past := append(past, <$p_i$, m>)
```

Append this message to past history

Seif Haridi, based on Ali Ghodsi's slides



Fail-Silent Causal Broadcast Impl (2)

```
upon event ⟨rbDeliver | pi,(DATA, $past_m$ , m)⟩ do
    if m∉delivered then
        forall ($s_n$,n)∈$past_m$ do
            if n∉delivered then
                trigger ⟨rcoDeliver|$s_n$, n⟩
                delivered := delivered∪{n}
                past := append(past, <$s_n$,n>)
        trigger ⟨rcoDeliver|$p_i$,m⟩
        delivered := delivered∪{m}
        past := append(past, <$p_i$,m>)
```

in ascending order

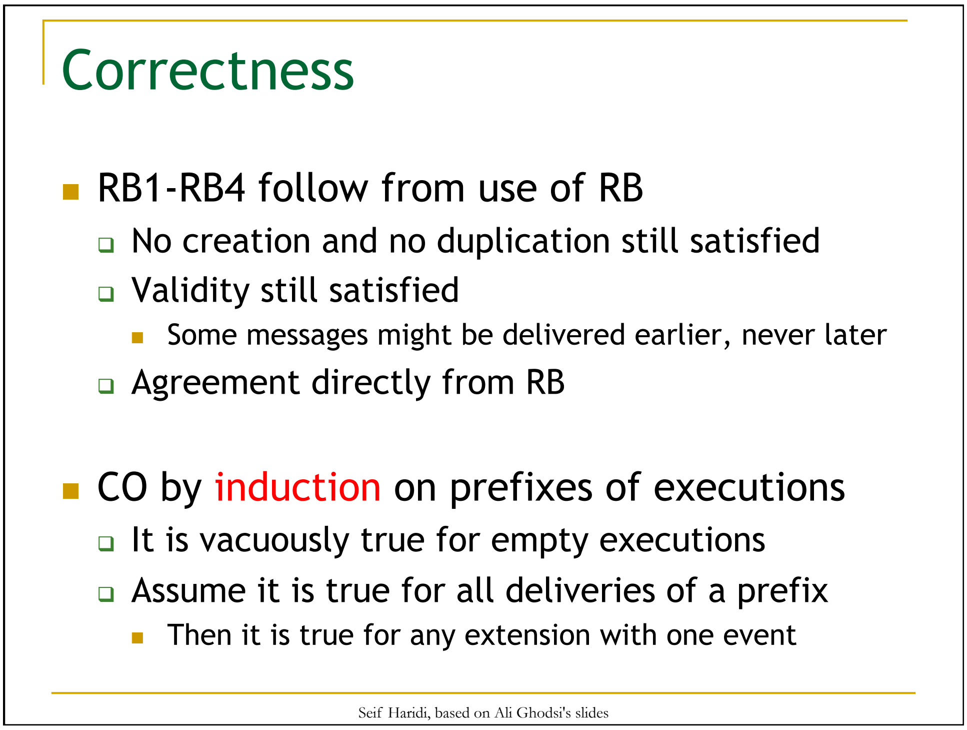
deliver preceding messages

append to history

deliver current message

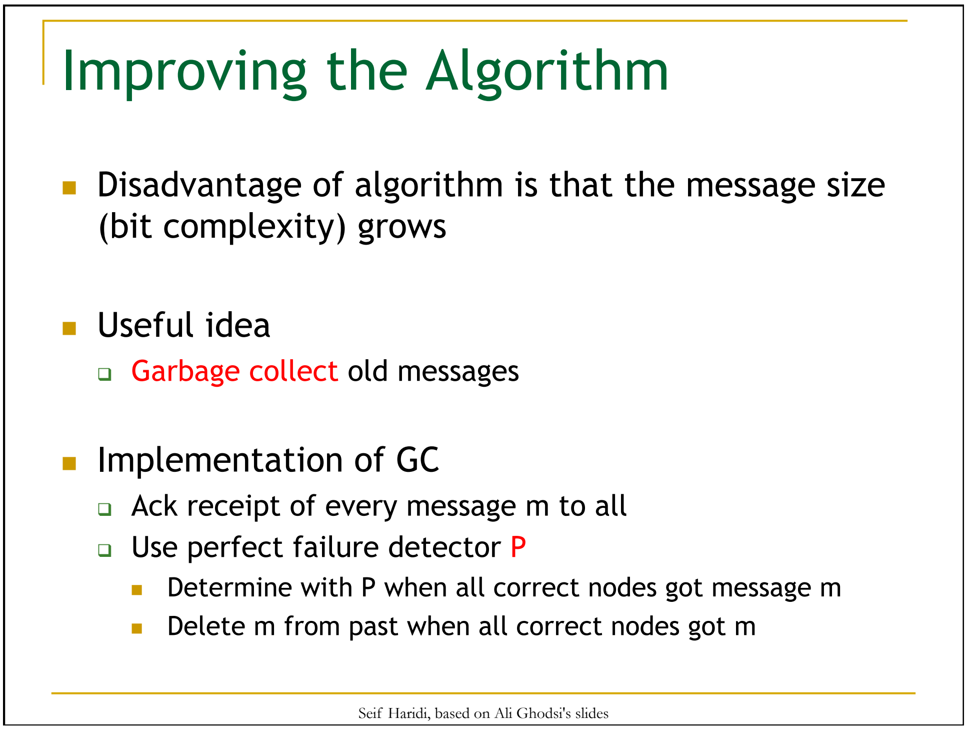
append to history

Seif Haridi, based on Ali Ghodsi's slides



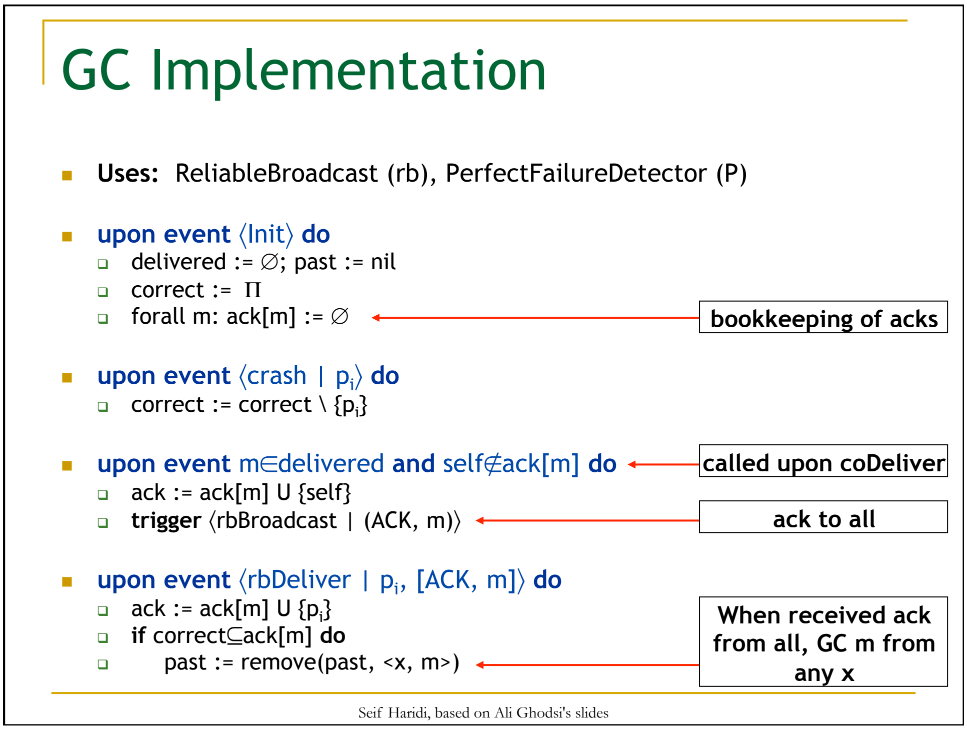
Correctness

- RB1-RB4 follow from use of RB
  - No creation and no duplication still satisfied
  - Validity still satisfied
    - Some messages might be delivered earlier, never later
  - Agreement directly from RB
- CO by induction on prefixes of executions
  - It is vacuously true for empty executions
  - Assume it is true for all deliveries of a prefix
    - Then it is true for any extension with one event



Improving the Algorithm

- Disadvantage of algorithm is that the message size (bit complexity) grows
- Useful idea
  - Garbage collect old messages
- Implementation of GC
  - Ack receipt of every message m to all
  - Use perfect failure detector P
    - Determine with P when all correct nodes got message m
    - Delete m from past when all correct nodes got m



GC Implementation

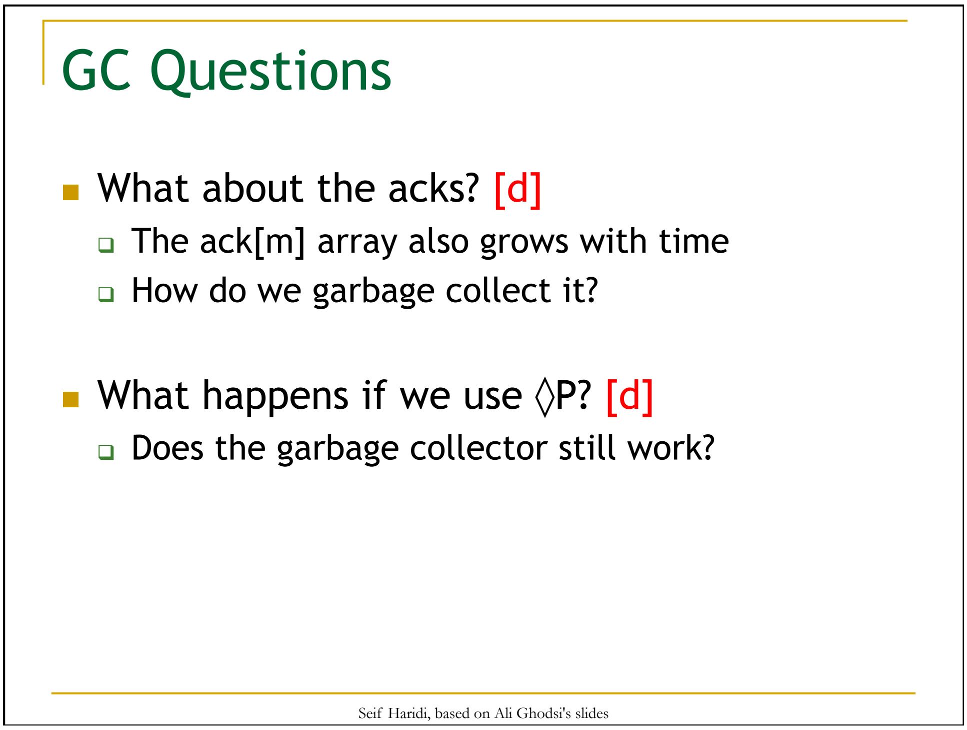
- **Uses:** ReliableBroadcast (rb), PerfectFailureDetector (P)
- **upon event** ⟨Init⟩ **do**
  - delivered := ∅; past := nil
  - correct := Π
  - forall m: ack[m] := ∅
- **upon event** ⟨crash | p_i⟩ **do**
  - correct := correct \ {p_i}
- **upon event** m∈delivered **and** self∉ack[m] **do**
  - ack := ack[m] U {self}
  - **trigger** ⟨rbBroadcast | (ACK, m)⟩
- **upon event** ⟨rbDeliver | p_i, [ACK, m]⟩ **do**
  - ack := ack[m] U {p_i}
  - **if** correct⊆ack[m] **do**
  -     past := remove(past, <x, m>)

bookkeeping of acks

called upon coDeliver

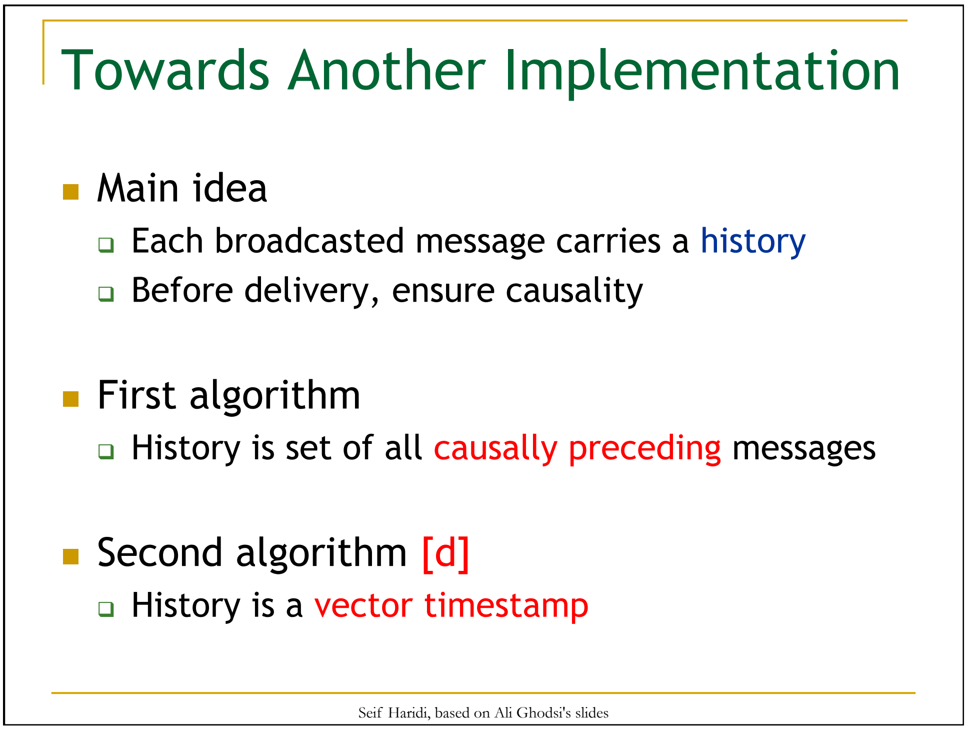
ack to all

When received ack from all, GC m from any x



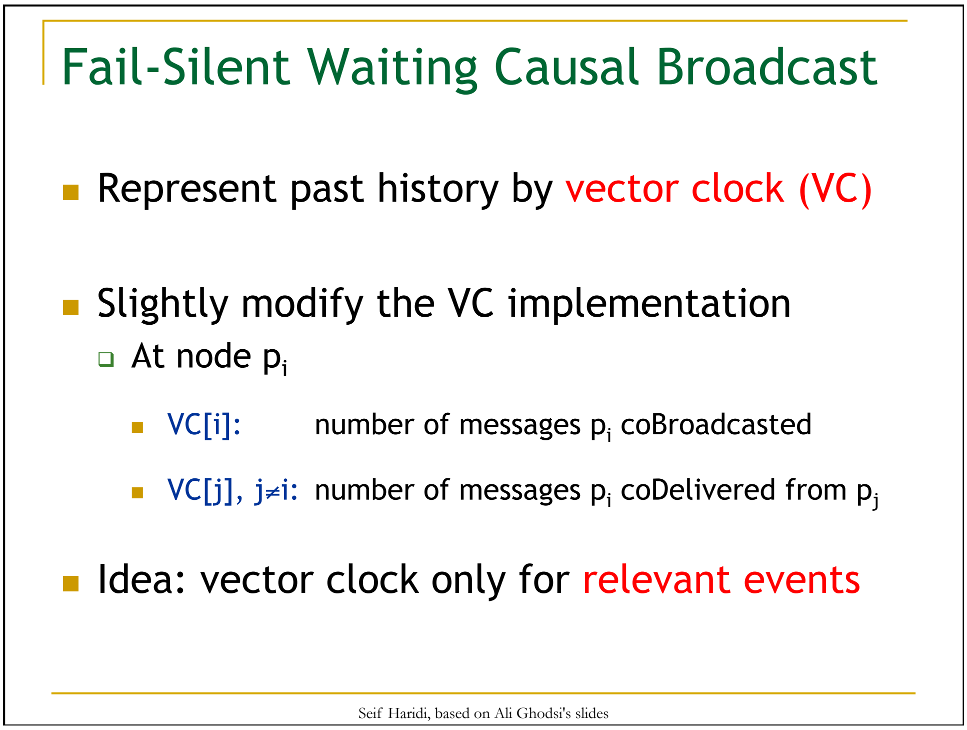
GC Questions

- What about the acks? [d]
  - The ack[m] array also grows with time
  - How do we garbage collect it?
- What happens if we use ◊P? [d]
  - Does the garbage collector still work?



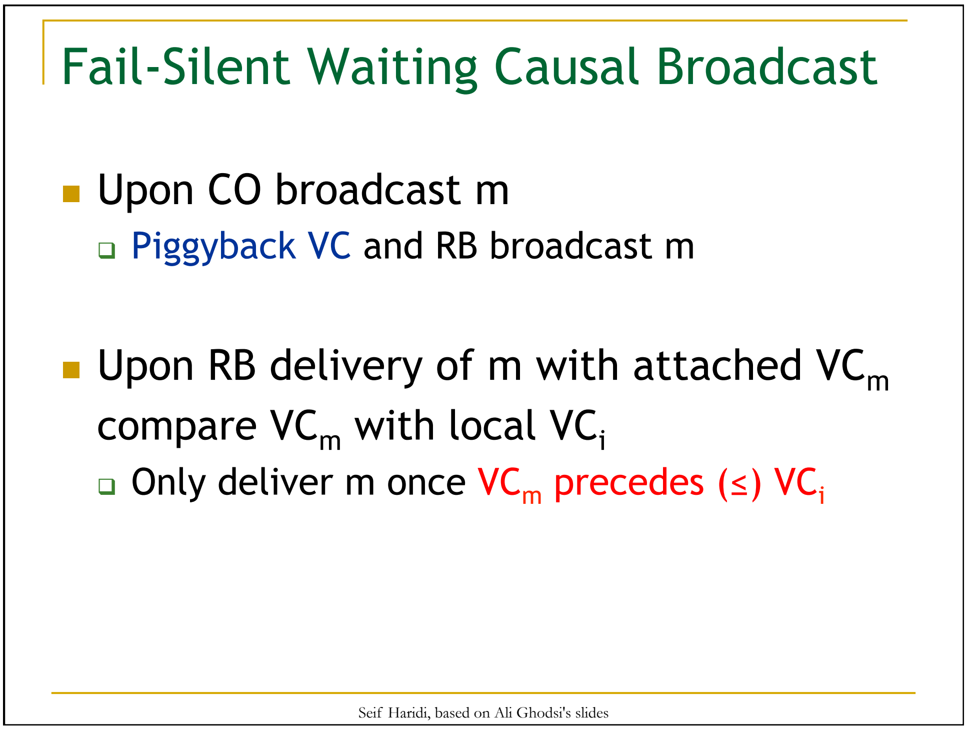
Towards Another Implementation

- Main idea
  - Each broadcasted message carries a history
  - Before delivery, ensure causality
- First algorithm
  - History is set of all causally preceding messages
- Second algorithm [d]
  - History is a vector timestamp



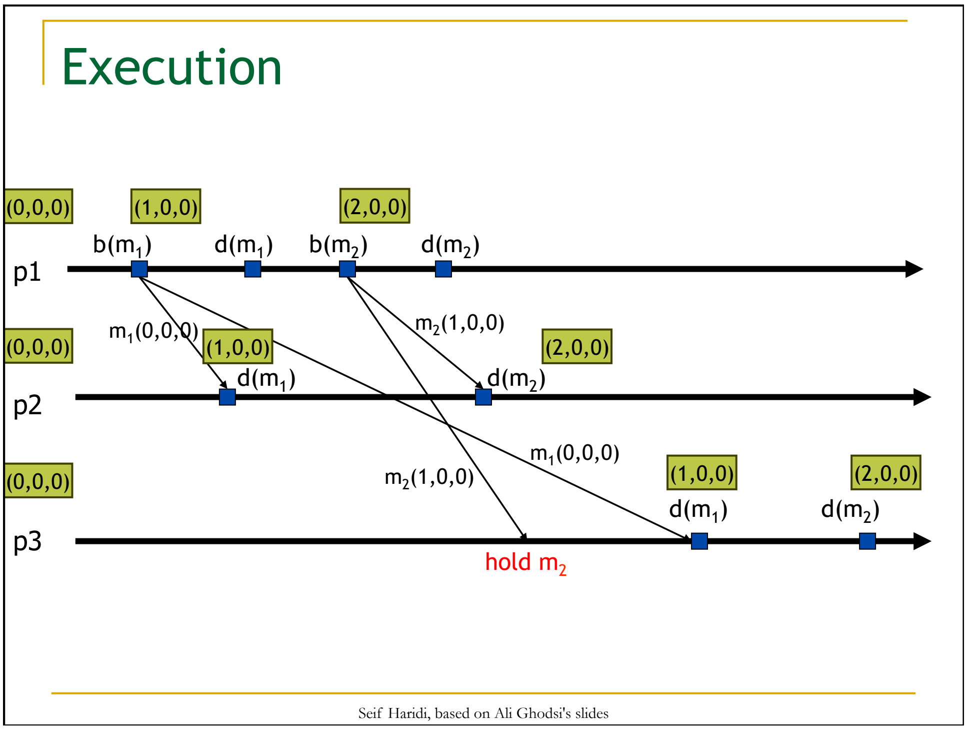
Fail-Silent Waiting Causal Broadcast

- 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≠i: number of messages p_i coDelivered from p_j
- Idea: vector clock only for relevant events

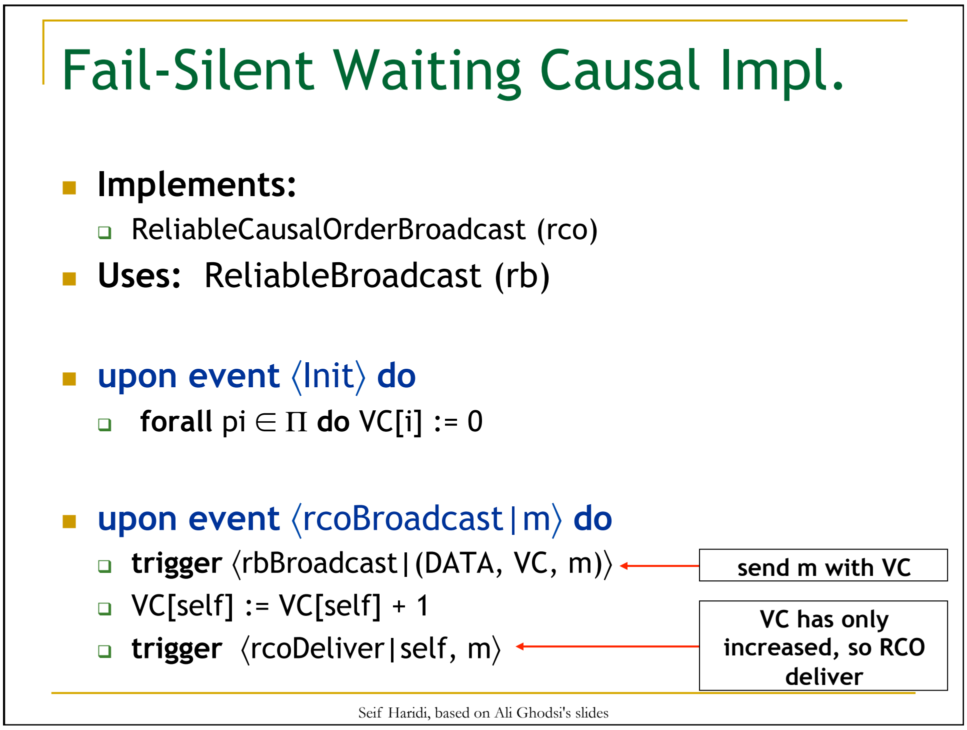


Fail-Silent Waiting Causal Broadcast

- 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



Fail-Silent Waiting Causal Impl.

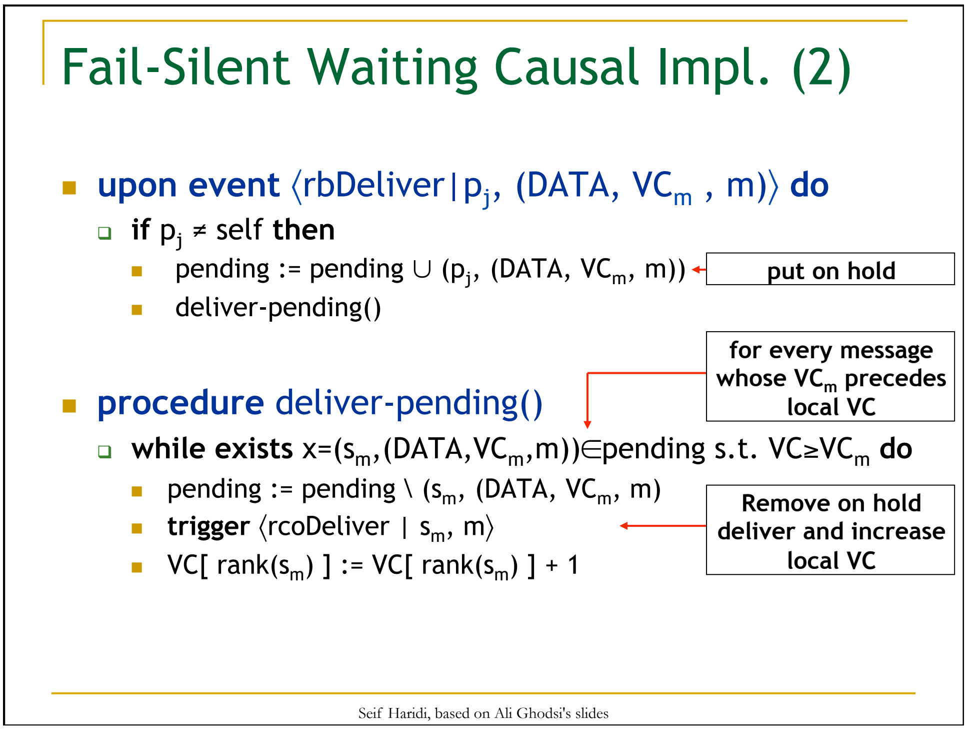
- **Implements:**
  - ReliableCausalOrderBroadcast (rco)
- **Uses:** ReliableBroadcast (rb)

- **upon event** ⟨Init⟩ **do**
  - **forall** pi $\in \Pi$ **do** VC[i] := 0

- **upon event** ⟨rcoBroadcast | m⟩ **do**
  - **trigger** ⟨rbBroadcast | (DATA, VC, m)⟩
  - VC[self] := VC[self] + 1
  - **trigger** ⟨rcoDeliver | self, m⟩

send m with VC

VC has only increased, so RCO deliver



Fail-Silent Waiting Causal Impl. (2)

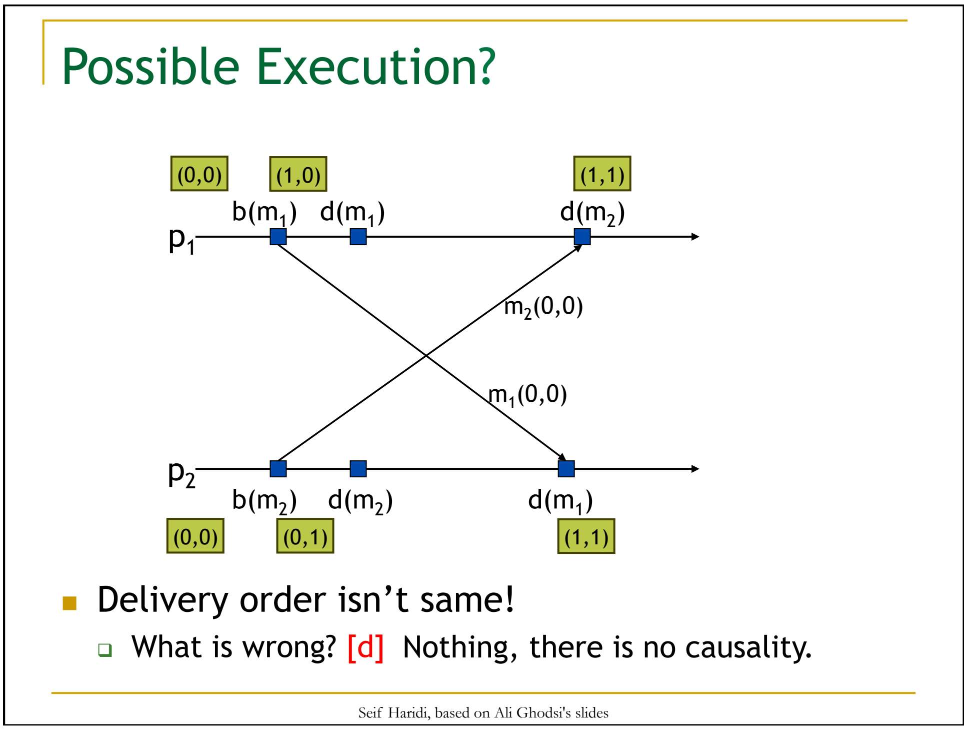
- **upon event** ⟨rbDeliver | p_j, (DATA, VC_m , m)⟩ **do**
  - **if** $p_j \neq$ self **then**
    - pending := pending ∪ (p_j, (DATA, VC_m, m))
    - deliver-pending()

put on hold

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⟩
    - VC[rank(s_m)] := VC[rank(s_m)] + 1

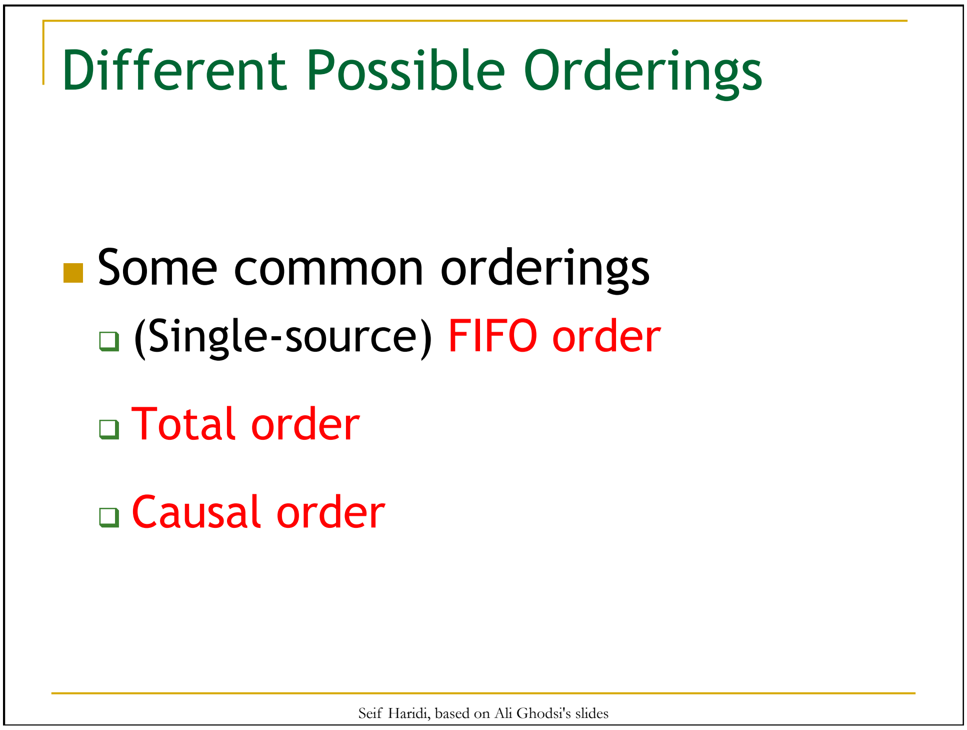
Remove on hold deliver and increase local VC



Possible Execution?

- Delivery order isn't same!
  - What is wrong? [d] Nothing, there is no causality.

Seif Haridi, based on Ali Ghodsi's slides



Different Possible Orderings

- Some common orderings
  - (Single-source) FIFO order
  - Total order
  - Causal order

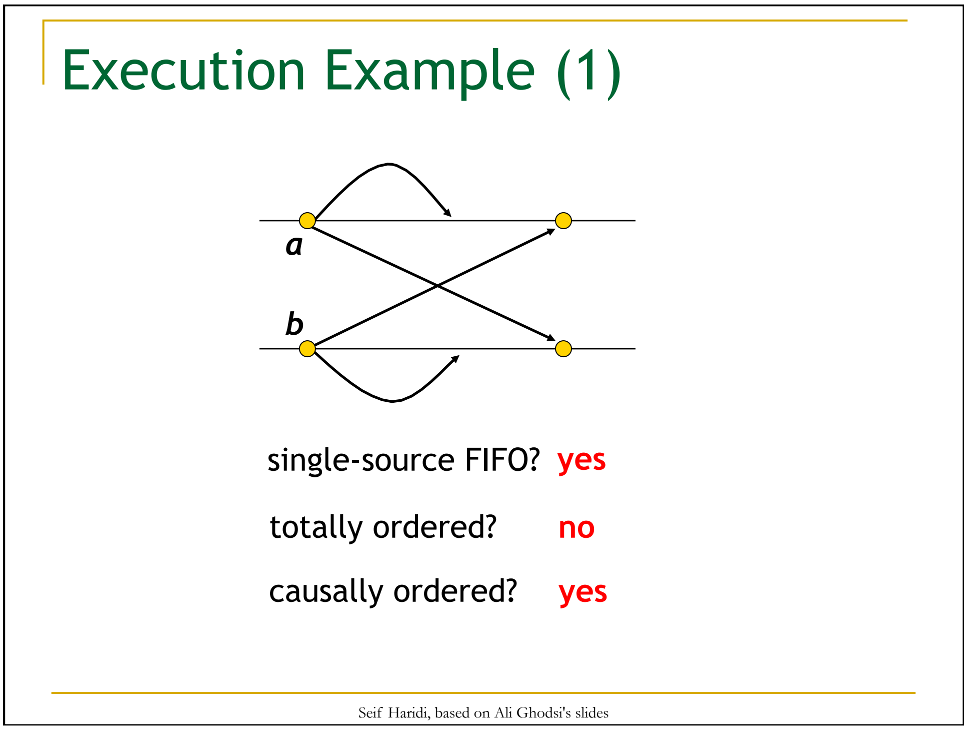
Seif Haridi, based on Ali Ghodsi's slides

Single-Source FIFO Order

- Intuitively
  - Msgs from same node delivered in order sent
- For all messages m_1 and m_2 and all p_i and p_j,
  - if p_i broadcasts m_1 before m_2, and if p_j delivers m_1 and m_2, then p_j delivers m_1 before m_2
- Caveat
  - This formulation doesn't require delivery of both messages

Total Order

- Intuitively
  - 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
- Caveats
  - This formulation doesn't require delivery of both messages
  - Everyone delivers in the *same order*, this might not be the *send order*!

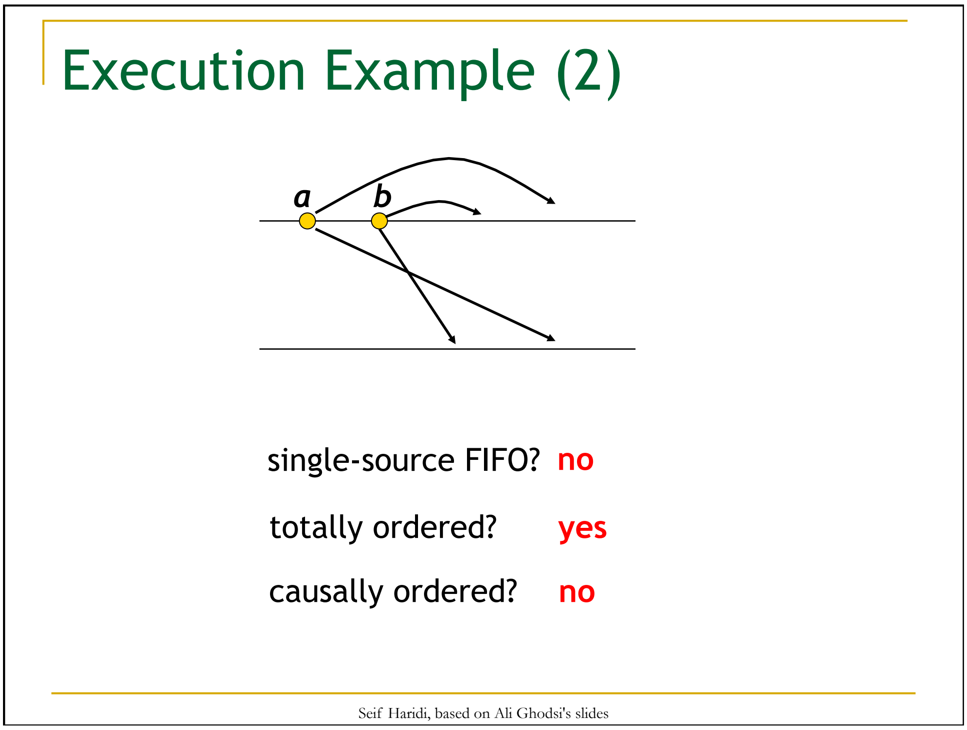


Execution Example (1)

single-source FIFO? yes

totally ordered? no

causally ordered? yes



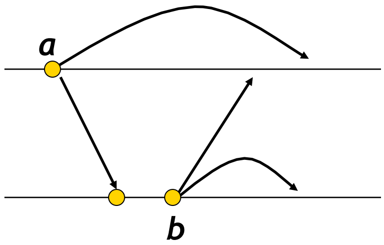
Execution Example (2)

single-source FIFO? no

totally ordered? yes

causally ordered? no

Execution Example (3)



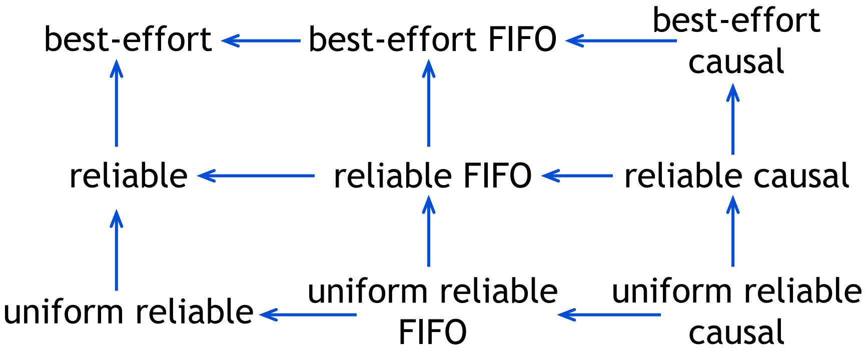
single-source FIFO? yes

totally ordered? no

causally ordered? no

Hierarchy of Orderings

- Stronger implies weaker ordering (→)



- Where does total order fit? [d]