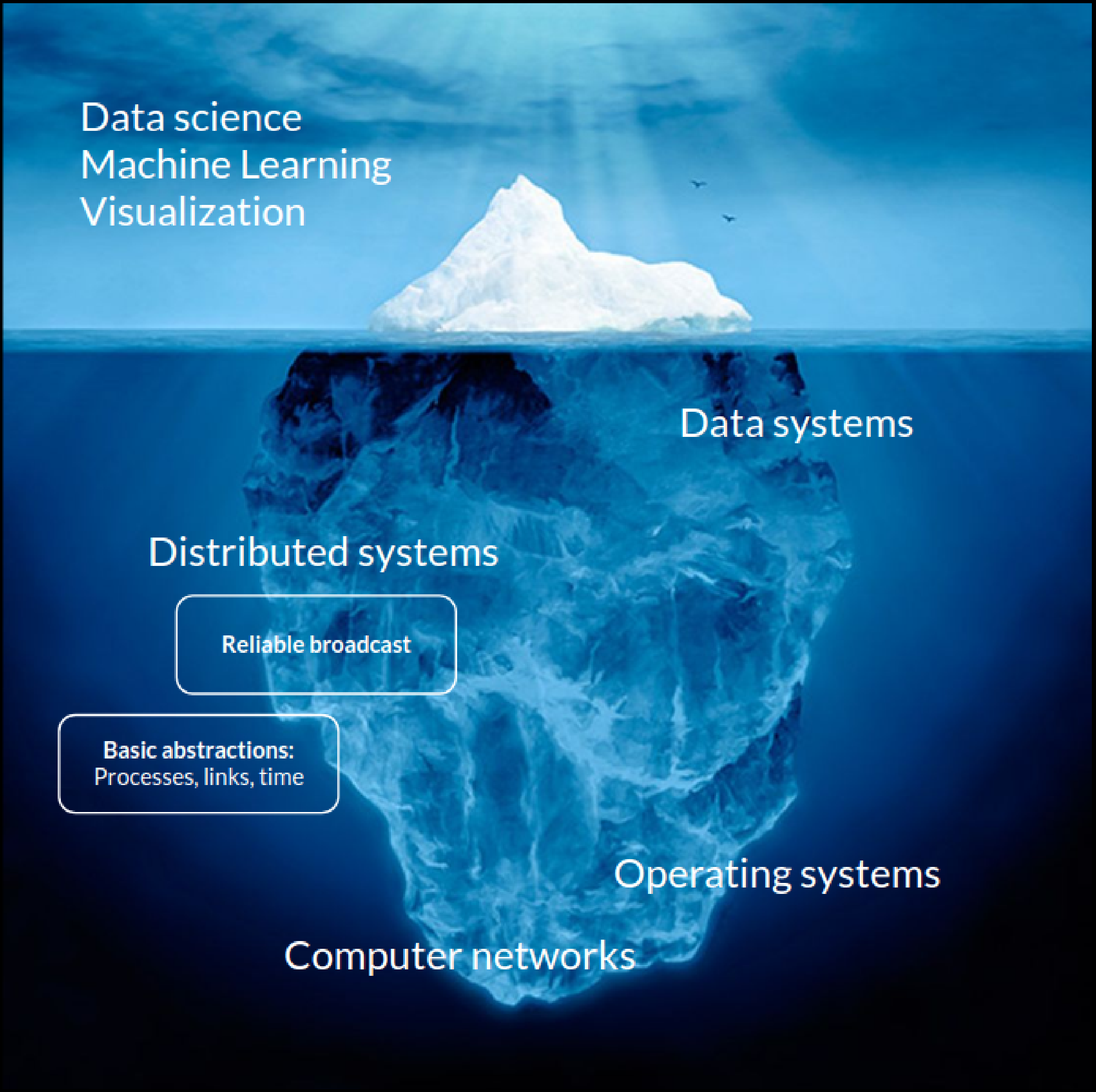
Large-scale Data Systems

Lecture 3: Reliable broadcast

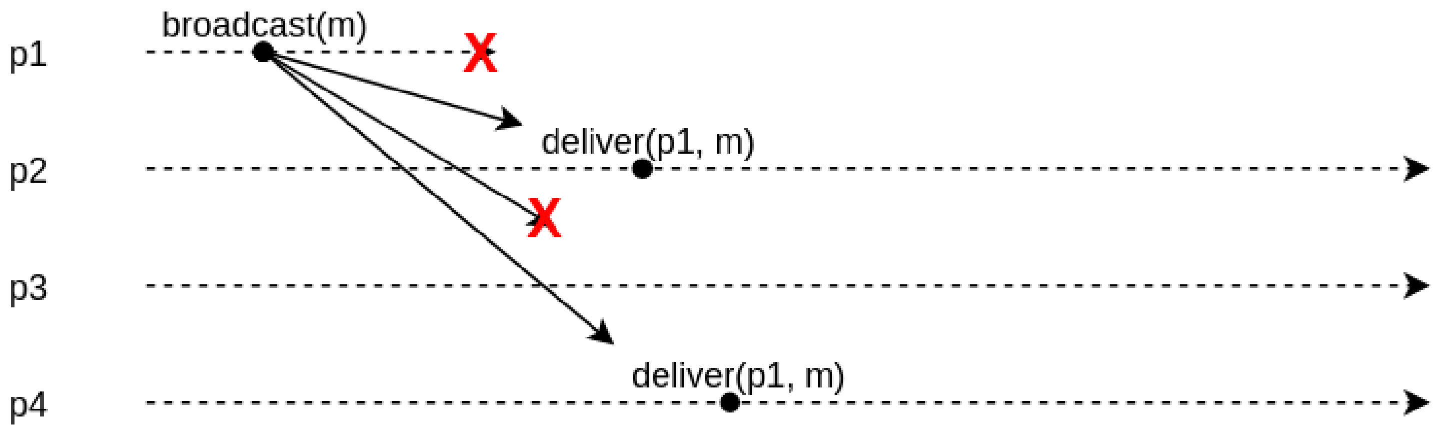
Prof. Gilles Louppe
g.louppe@uliege.be

Today

- How do you talk to multiple machines at once?
- What if some of them fail?
- Can we guarantee that correct nodes all receive the same messages?
- What about ordering?
- What about performance?


Data science
Machine Learning
Visualization
Data systems
Distributed systems
Reliable broadcast
Basic abstractions:
Processes, links, time
Operating systems
Computer networks

Unreliable broadcast



Constraints:

- The sender may fail.
- Recipients may fail.
- Packets might get lost.
- Packets may take long to travel.

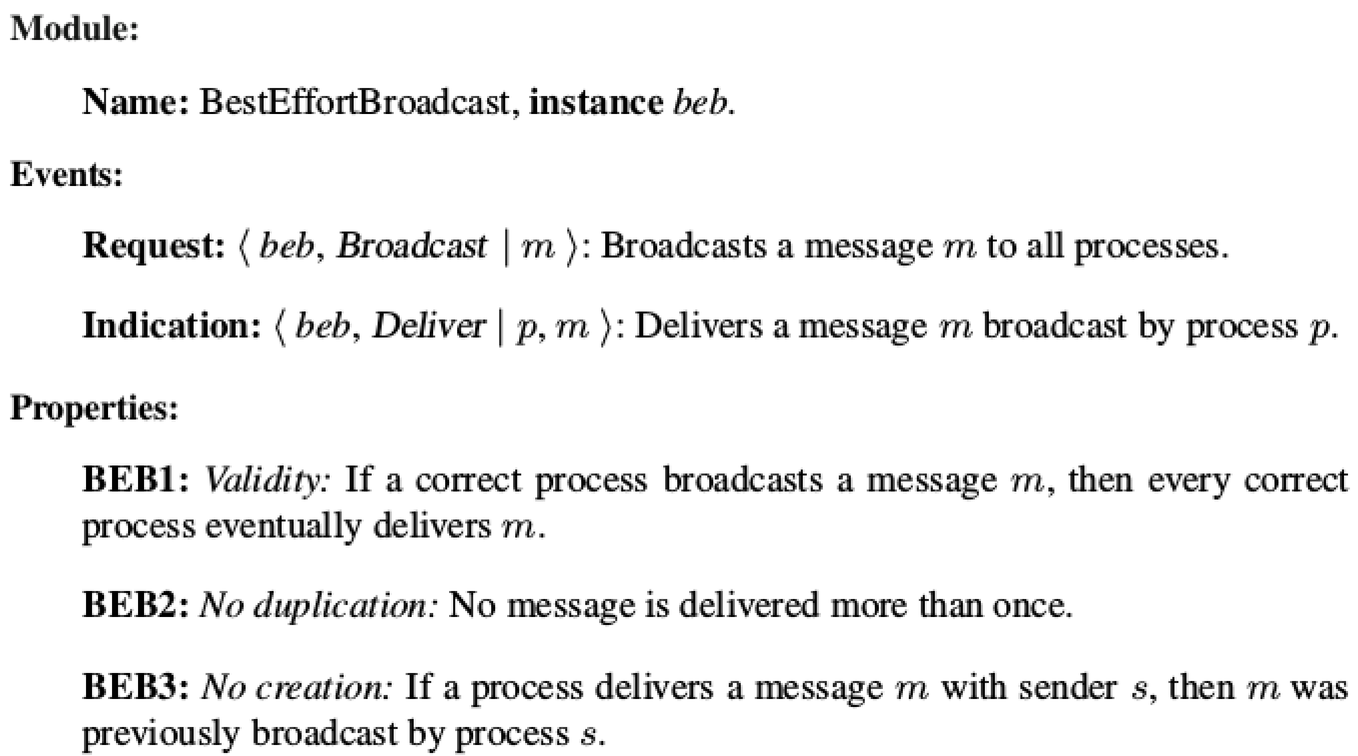
How do we define a reliable broadcast service?

Reliable broadcast abstractions

Reliable broadcast abstractions

- Best-effort broadcast
    - Guarantees reliability only if sender is correct.
- Reliable broadcast
    - Guarantees reliability independent of whether sender is correct.
- Uniform reliable broadcast
    - Also considers the behavior of failed nodes.
- Causal reliable broadcast
    - Reliable broadcast with causal delivery order.

Best-effort broadcast (*beb*)



Module:

Name: BestEffortBroadcast, **instance** *beb*.

Events:

Request: $\langle$ *beb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *beb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

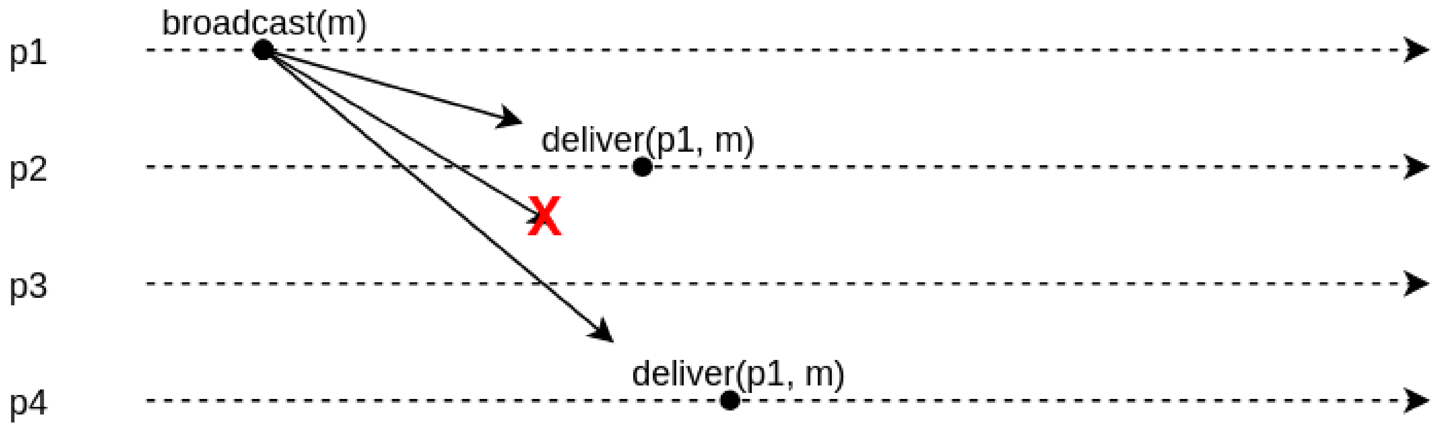
Properties:

BEB1: *Validity:* If a correct process broadcasts a message m, then every correct process eventually delivers m.

BEB2: *No duplication:* No message is delivered more than once.

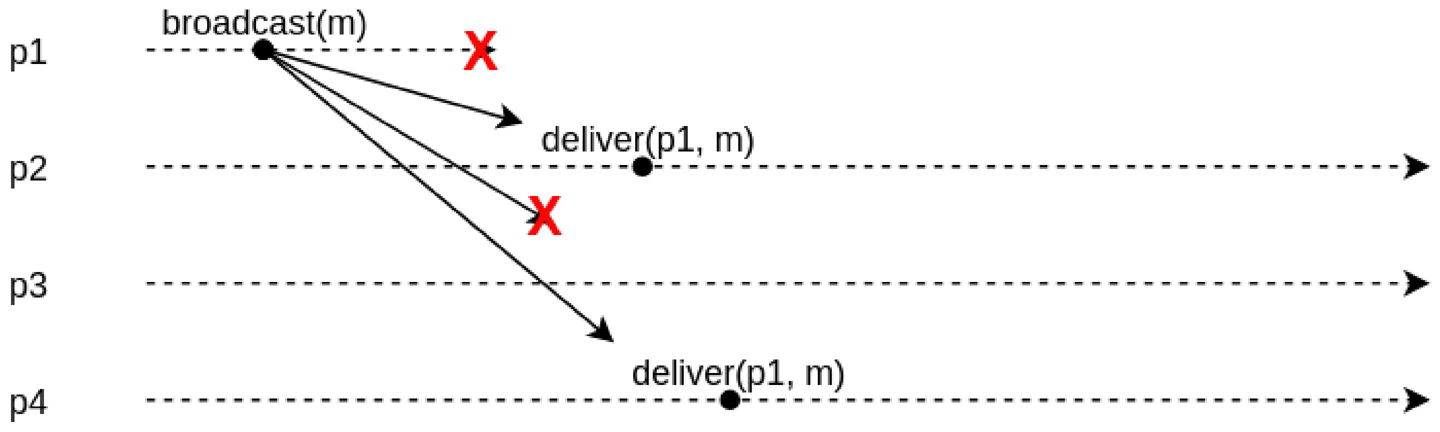
BEB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

beb example (1)



[Q] Is this allowed?

beb example (2)



[Q] Is this allowed?

Reliable broadcast (rb)

- Best-effort broadcast gives no guarantees if sender crashes.
- Reliable broadcast:
    - Same as best-effort broadcast +
    - If sender crashes, ensure all or none of the correct node deliver the message.

Module:

Name: ReliableBroadcast, **instance** *rb*.

Events:

Request: $\langle$ *rb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *rb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

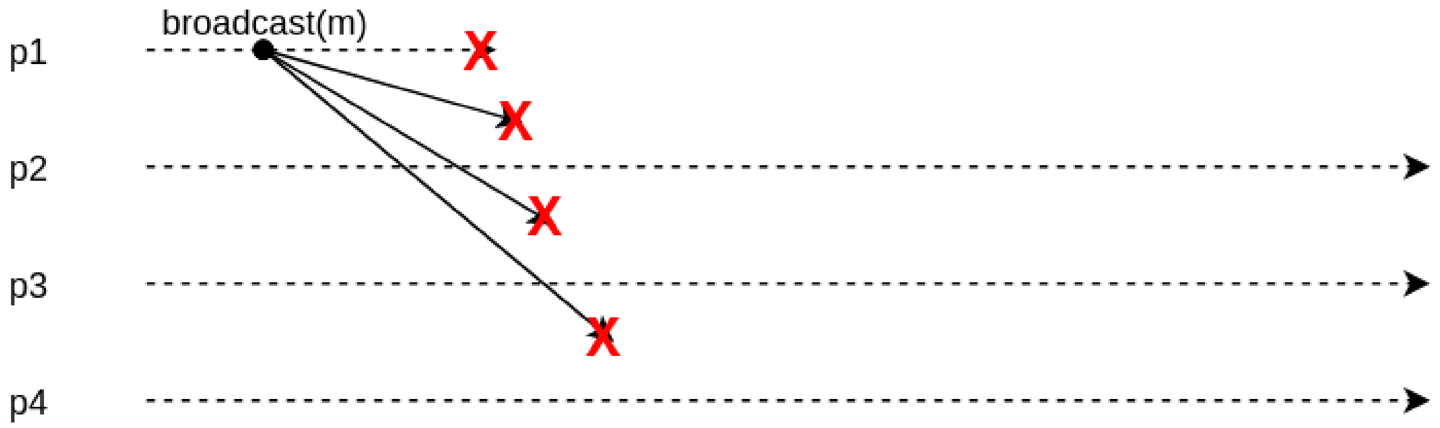
RB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

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

RB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

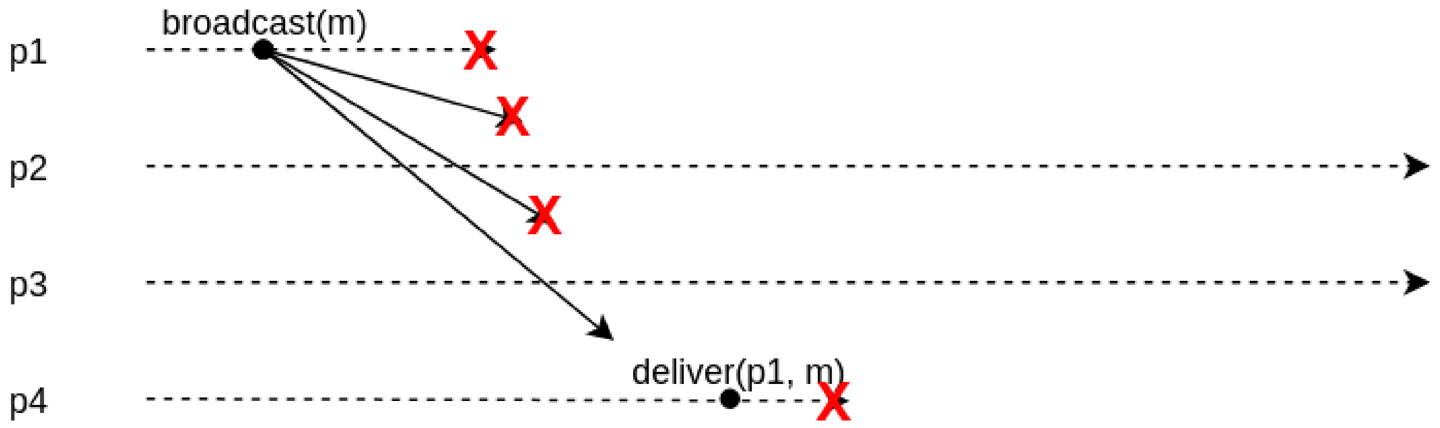
RB4: *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

rb example (1)



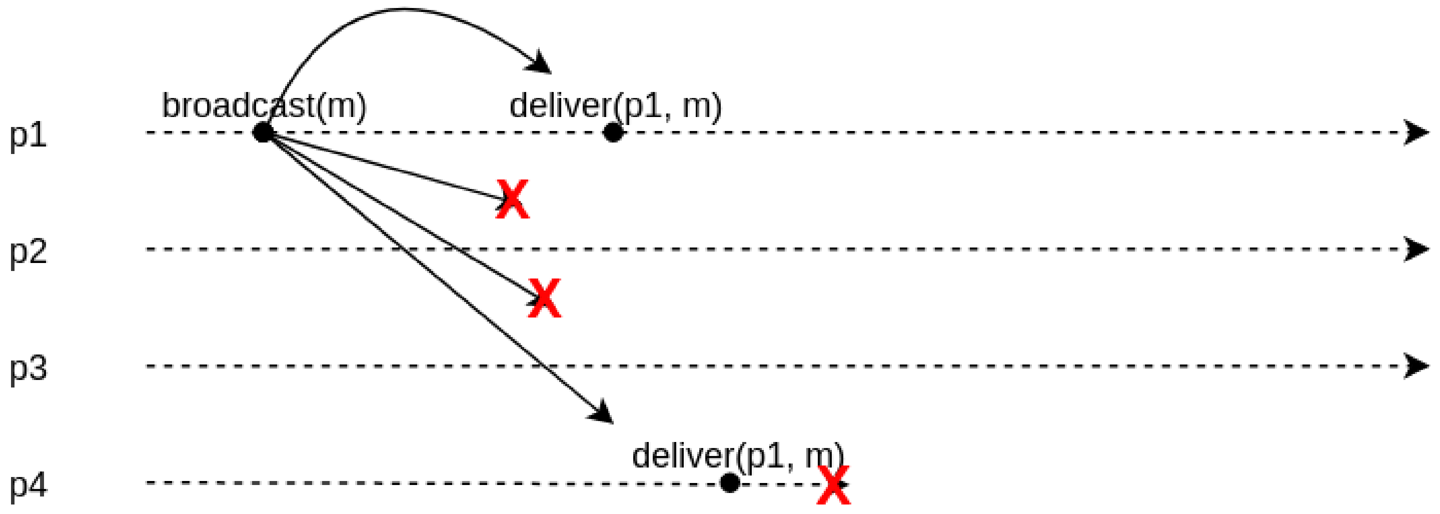
[Q] Is this allowed?

rb example (2)



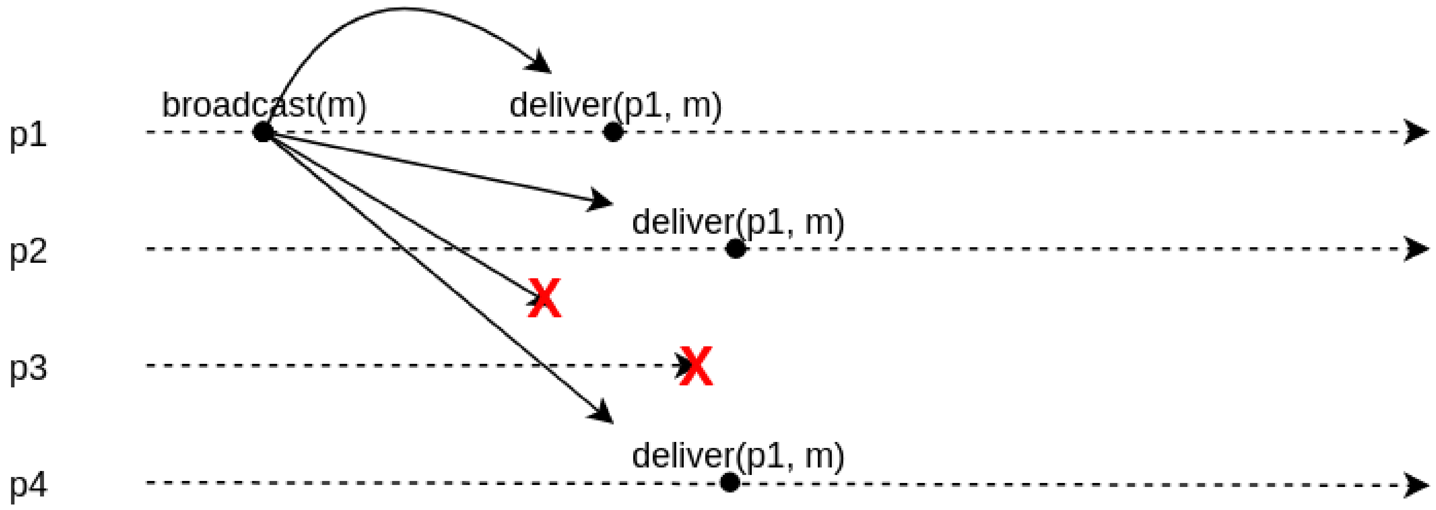
[Q] Is this allowed?

rb example (3)



[Q] Is this allowed?

rb example (4)



[Q] Is this allowed?

Uniform reliable broadcast (urb)

- Assume sender broadcasts a message
    - Sender fails
    - No correct node delivers the message
    - Failed nodes deliver the message
- Is this OK?
    - A process that delivers a message and later crashes may bring the application into a inconsistent state.
- Uniform reliable broadcast ensures that if a message is delivered, by a correct or a faulty process, then all correct processes deliver.

Module:

Name: UniformReliableBroadcast, **instance** *urb*.

Events:

Request: $\langle$ *urb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *urb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

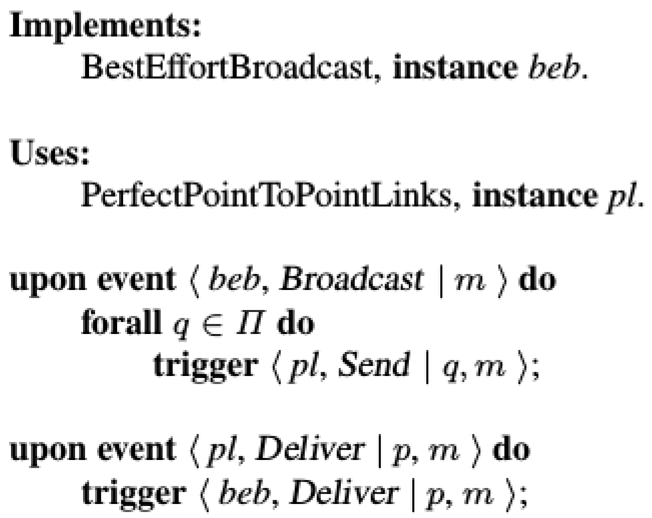
Properties:

URB1–URB3: Same as properties RB1–RB3 in (regular) reliable broadcast (Module 3.2).

URB4: *Uniform agreement:* If a message m is delivered by some process (whether correct or faulty), then m is eventually delivered by every correct process.

Implementations

Basic broadcast



```
Implements:
    BestEffortBroadcast, instance beb.

Uses:
    PerfectPointToPointLinks, instance pl.

upon event ⟨ beb, Broadcast | m ⟩ do
    forall $q \in \Pi$ do
        trigger ⟨ pl, Send | q, m ⟩;

upon event ⟨ pl, Deliver | p, m ⟩ do
    trigger ⟨ beb, Deliver | p, m ⟩;
```

Correctness:

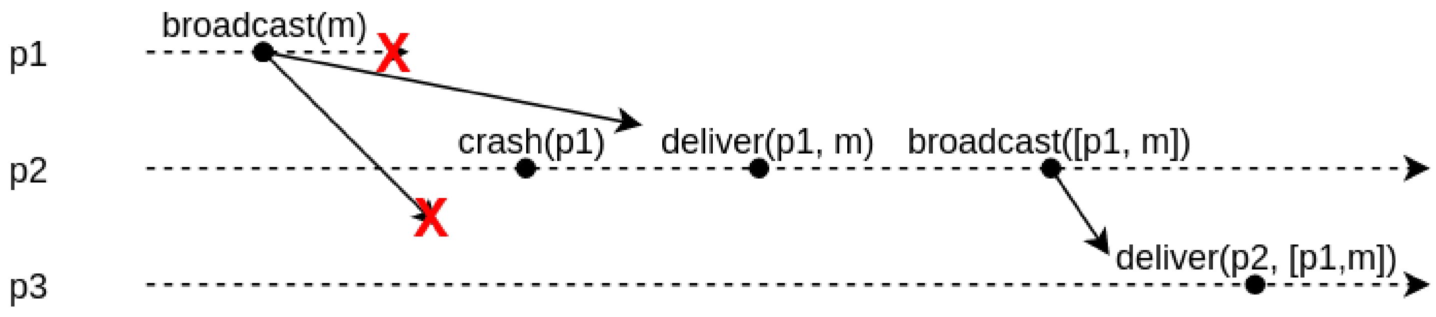
- BEB1. Validity: If a correct process p broadcasts m, then every correct process eventually delivers m.
    - If sender does not crash, every other correct node receives message by perfect channels.
- BEB2+3. No duplication + no creation
    - Guaranteed by perfect channels.

Lazy reliable broadcast

- Assume a fail-stop distributed system model.
    - i.e., crash-stop processes, perfect links and a perfect failure detector.
- To broadcast m:
    - best-effort broadcast m
    - Upon `bebDeliver`:
        - Save message
        - `rbDeliver` the message
- If sender s crashes, detect and relay messages from s to all.
    - case 1: get m from s, detect crash of s, redistribute m
    - case 2: detect crash of s, get m from s, redistribute m.
- Filter duplicate messages.

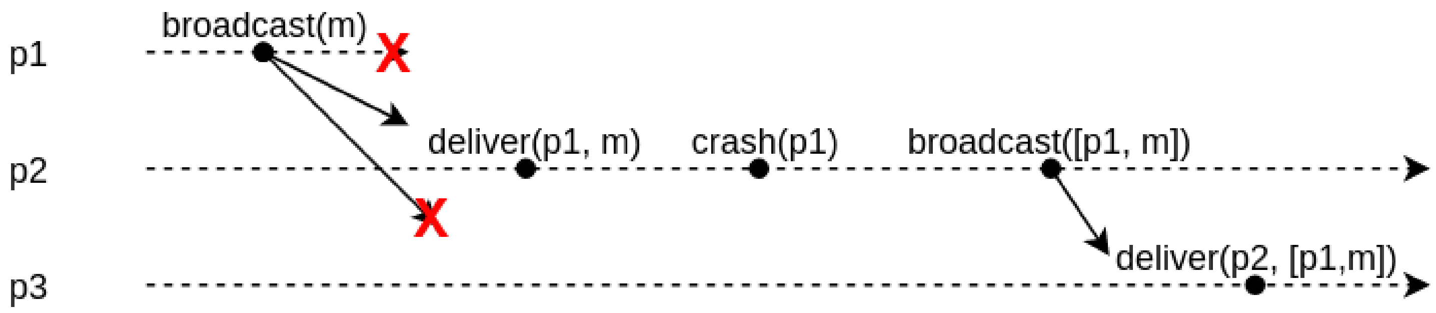
```
Implements:
    ReliableBroadcast, instance rb.

Uses:
    BestEffortBroadcast, instance beb;
    PerfectFailureDetector, instance $\mathcal{P}$.

upon event ⟨ rb, Init ⟩ do
    correct := $\Pi$;
    from[p] := $[\emptyset]^N$;

upon event ⟨ rb, Broadcast | m ⟩ do
    trigger ⟨ beb, Broadcast | [DATA, self, m] ⟩;

upon event ⟨ beb, Deliver | p, [DATA, s, m] ⟩ do
    if $m \notin$ from[s] then
        trigger ⟨ rb, Deliver | s, m ⟩;
        from[s] := from[s] $\cup \{m\}$;
        if $s \notin$ correct then
            trigger ⟨ beb, Broadcast | [DATA, s, m] ⟩;

upon event ⟨ $\mathcal{P}$, Crash | p ⟩ do
    correct := correct $\setminus \{p\}$;
    forall $m \in$ from[p] do
        trigger ⟨ beb, Broadcast | [DATA, p, m] ⟩;
```

Lazy reliable broadcast example (1)



[Q] Which case?

Lazy reliable broadcast example (2)



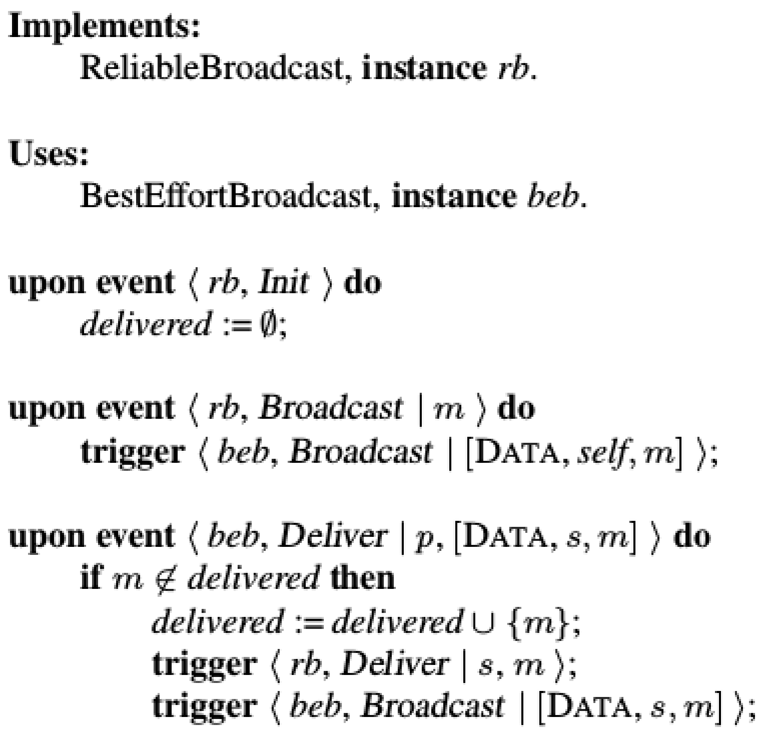
[Q] Which case?

Correctness of lazy reliable broadcast

- RB1-RB3
  - Satisfied with best-effort broadcast.
- RB4. Agreement: If a message m is delivered by some correct process, then m is eventually delivered by every correct process.
  - When correct p_j delivers m broadcast by p_i
    - if p_i is correct, BEB ensures correct delivery
    - if p_i crashes,
      - p_j detects this (because of completeness of the PFD)
      - p_j uses BEB to ensure (BEB1) every correct node gets m.

Eager reliable broadcast

- What happens if we use instead an eventually perfect failure detector?
    - Only affects performance, not correctness.
- Can we modify Lazy RB to not use a perfect failure detector?
    - Assume all nodes have failed.
    - BEB broadcast all received messages.



```
Implements:
    ReliableBroadcast, instance rb.

Uses:
    BestEffortBroadcast, instance beb.

upon event ⟨ rb, Init ⟩ do
    delivered := ∅;

upon event ⟨ rb, Broadcast | $m$ ⟩ do
    trigger ⟨ beb, Broadcast | [DATA, self, $m$] ⟩;

upon event ⟨ beb, Deliver | $p$, [DATA, $s$, $m$] ⟩ do
    if $m \notin$ delivered then
        delivered := delivered $\cup \{m\}$;
        trigger ⟨ rb, Deliver | $s$, $m$ ⟩;
        trigger ⟨ beb, Broadcast | [DATA, $s$, $m$] ⟩;
```

[Q] Show that eager reliable broadcast is correct.

Uniformity

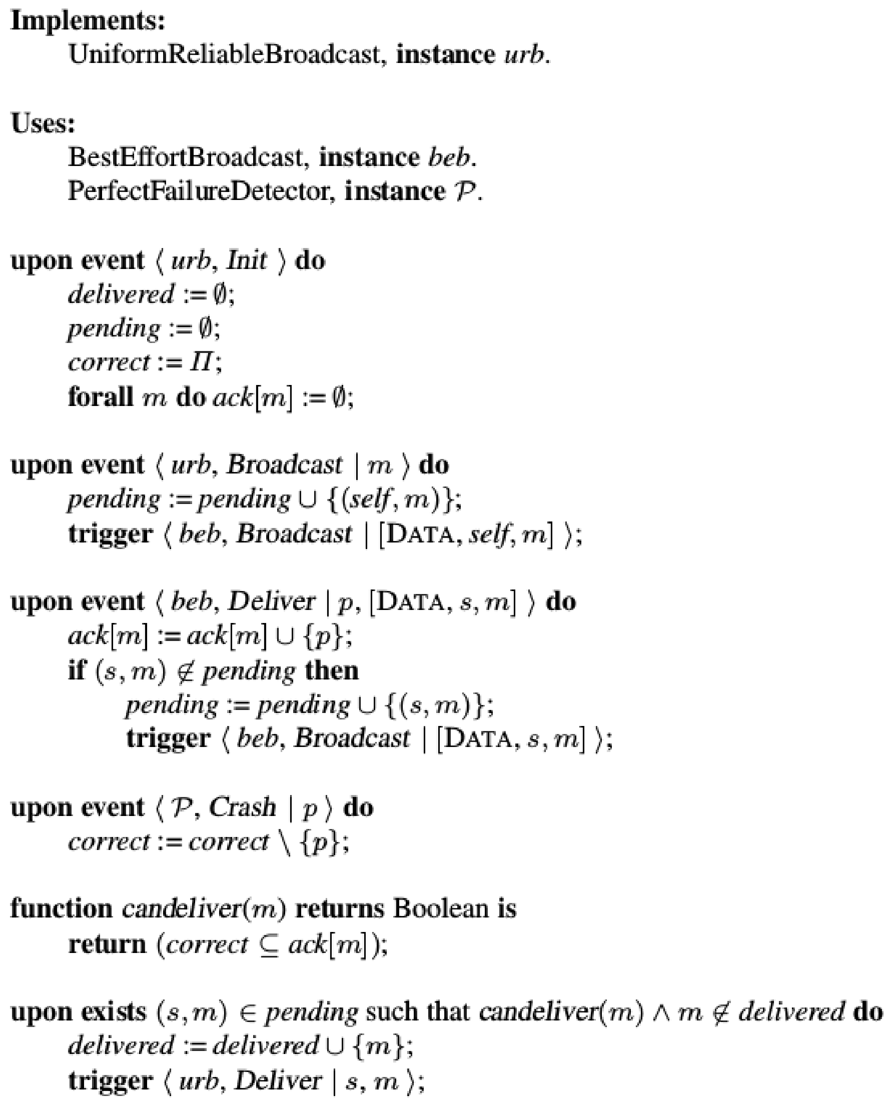
Neither Lazy reliable broadcast nor Eager reliable broadcast ensure uniform agreement.

E.g., sender p immediately RB delivers and crashes. Only p delivered the message.

Strategy for uniform agreement

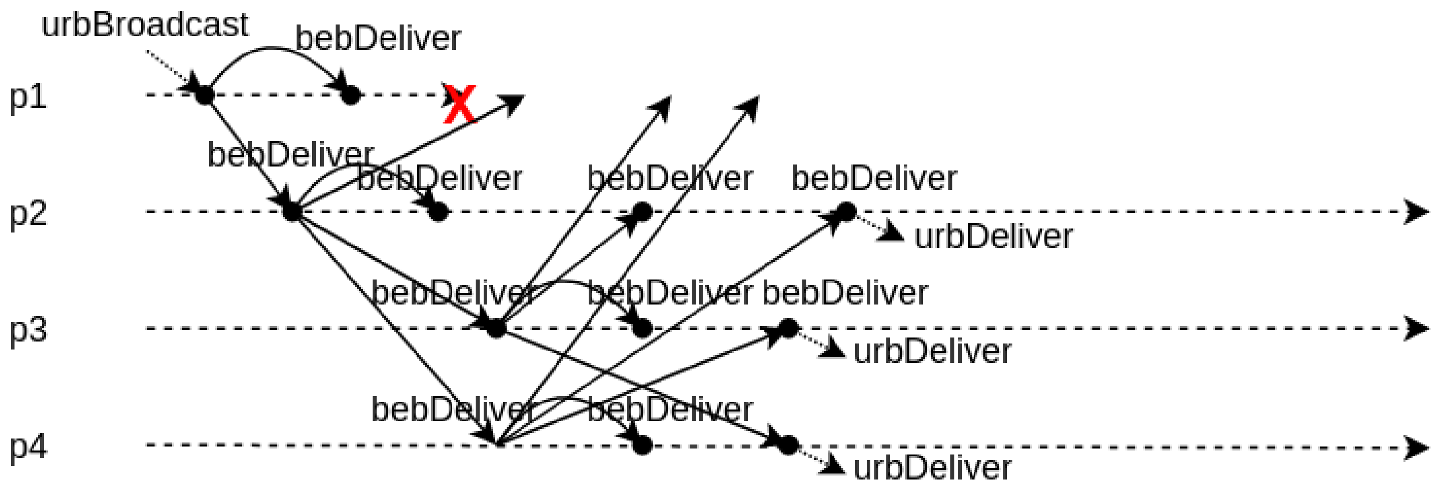
- Before delivering a message, we need to ensure all correct nodes have received it.
- Messages are pending until all correct nodes get it.
    - Collect acknowledgements from nodes that got the message.
- Deliver once all correct nodes acked.

All-ack uniform reliable broadcast



```
Implements:
    UniformReliableBroadcast, instance urb.

Uses:
    BestEffortBroadcast, instance beb.
    PerfectFailureDetector, instance $\mathcal{P}$.

upon event ⟨ urb, Init ⟩ do
    delivered := $\emptyset$;
    pending := $\emptyset$;
    correct := $\Pi$;
    forall $m$ do ack[$m$] := $\emptyset$;

upon event ⟨ urb, Broadcast | $m$ ⟩ do
    pending := pending $\cup$ {(self, $m$)};
    trigger ⟨ beb, Broadcast | [DATA, self, $m$] ⟩;

upon event ⟨ beb, Deliver | $p$, [DATA, $s$, $m$] ⟩ do
    ack[$m$] := ack[$m$] $\cup$ {$p$};
    if ($s, m$) $\notin$ pending then
        pending := pending $\cup$ {($s, m$)};
        trigger ⟨ beb, Broadcast | [DATA, $s$, $m$] ⟩;

upon event ⟨ $\mathcal{P}$, Crash | $p$ ⟩ do
    correct := correct $\setminus$ {$p$};

function candeliver($m$) returns Boolean is
    return (correct $\subseteq$ ack[$m$]);

upon exists ($s, m$) $\in$ pending such that candeliver($m$) $\wedge$ $m \notin$ delivered do
    delivered := delivered $\cup$ {$m$};
    trigger ⟨ urb, Deliver | $s$, $m$ ⟩;
```

Example


urbBroadcast
bebDeliver
p1
X
bebDeliver
bebDeliver
bebDeliver
bebDeliver
p2
urbDeliver
bebDeliver
bebDeliver
bebDeliver
p3
urbDeliver
bebDeliver
bebDeliver
p4
urbDeliver

Correctness of All-ack URB

Lemma. If a correct node p BEB delivers m, then p eventually URB delivers m.

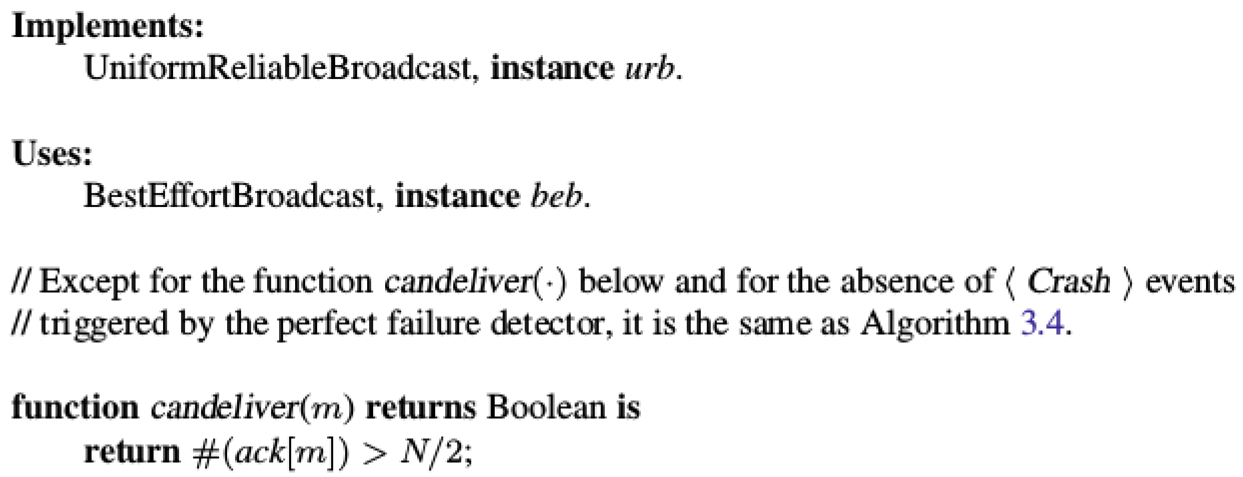
Proof:

- A correct node p BEB broadcasts m as soon as it gets m.
- By BEB1, every correct node gets m and BEB broadcasts m.
- Therefore p BEB delivers from every correct node by BEB1.
- By completeness of the perfect failure detector, p will not wait for dead nodes forever.
    - `canDeliver` becomes true and p URB delivers m.

- URB1. Validity: If a correct process p broadcasts m, then p delivers m
    - If sender is correct, it will BEB delivers m by validity (BEB1)
    - By the lemma, it will therefore eventually URB delivers m.
- URB2. No duplication
    - Guaranteed because of the `delivered` set.
- URB3. No creation
    - Ensured from best-effort broadcast.
- URB4. Uniform agreement: If a message m is delivered by some process (correct or faulty), then m is eventually delivered by every correct process
    - Assume some node (possibly failed) URB delivers m.
        - Then `canDeliver` was true, and by accuracy of the failure detector, every correct node has BEB delivered m.
    - By the lemma, each of the nodes that BEB delivered m will URB deliver m.

urb for fail-silent

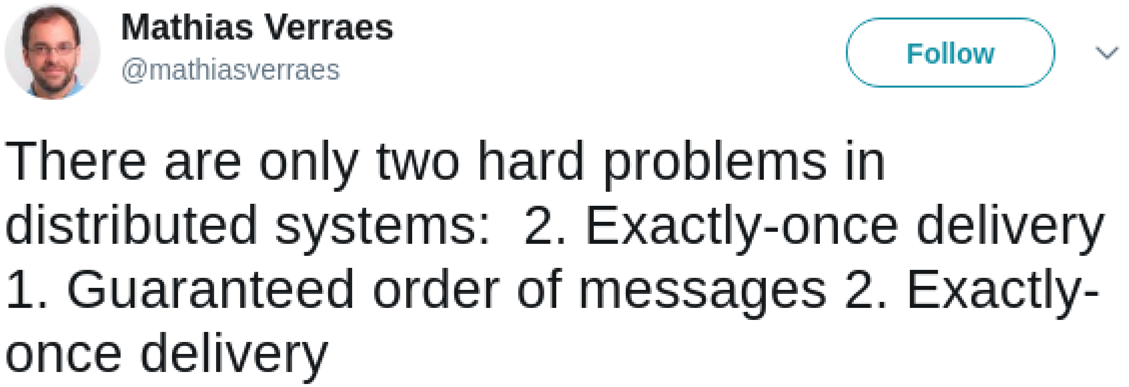
- All-ack URB requires a perfect failure detector (fail-stop).
- Can we implement URB in fail-silent, without a perfect failure detector?
- Yes, provided a majority of nodes are correct.



```
Implements:
    UniformReliableBroadcast, instance urb.

Uses:
    BestEffortBroadcast, instance beb.

// Except for the function candeliver(·) below and for the absence of ⟨ Crash ⟩ events
// triggered by the perfect failure detector, it is the same as Algorithm 3.4.

function candeliver($m$) returns Boolean is
    return $\#(ack[m]) > N/2$;
```

[Q] Show that this variant is correct.

Causal reliable broadcast


Mathias Verraes
@mathiasverraes

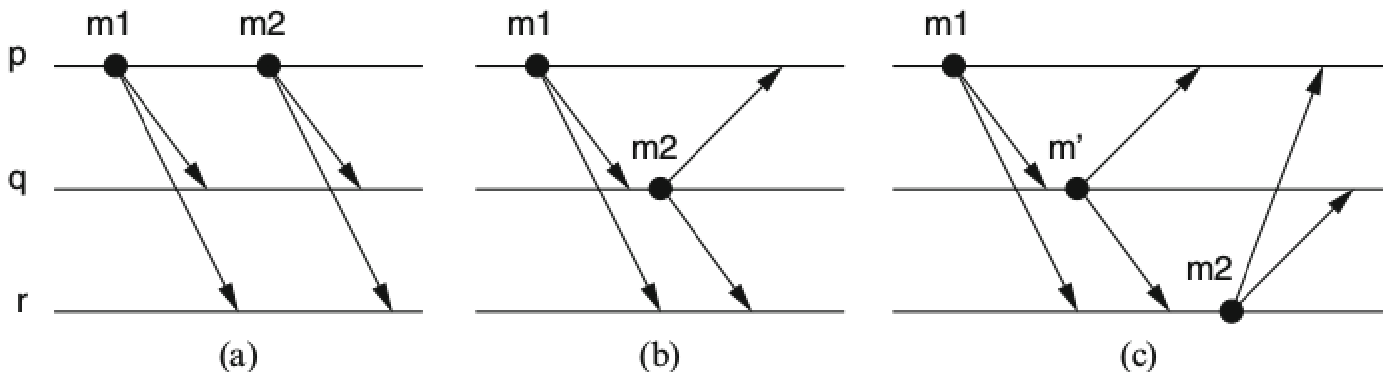
There are only two hard problems in distributed systems: 2. Exactly-once delivery 1. Guaranteed order of messages 2. Exactly-once delivery

Reliable broadcast:

- Exactly-once delivery: guaranteed by the properties of RB.
- Order of message? Not guaranteed!

[Q] Does uniform reliable broadcast remedy this?

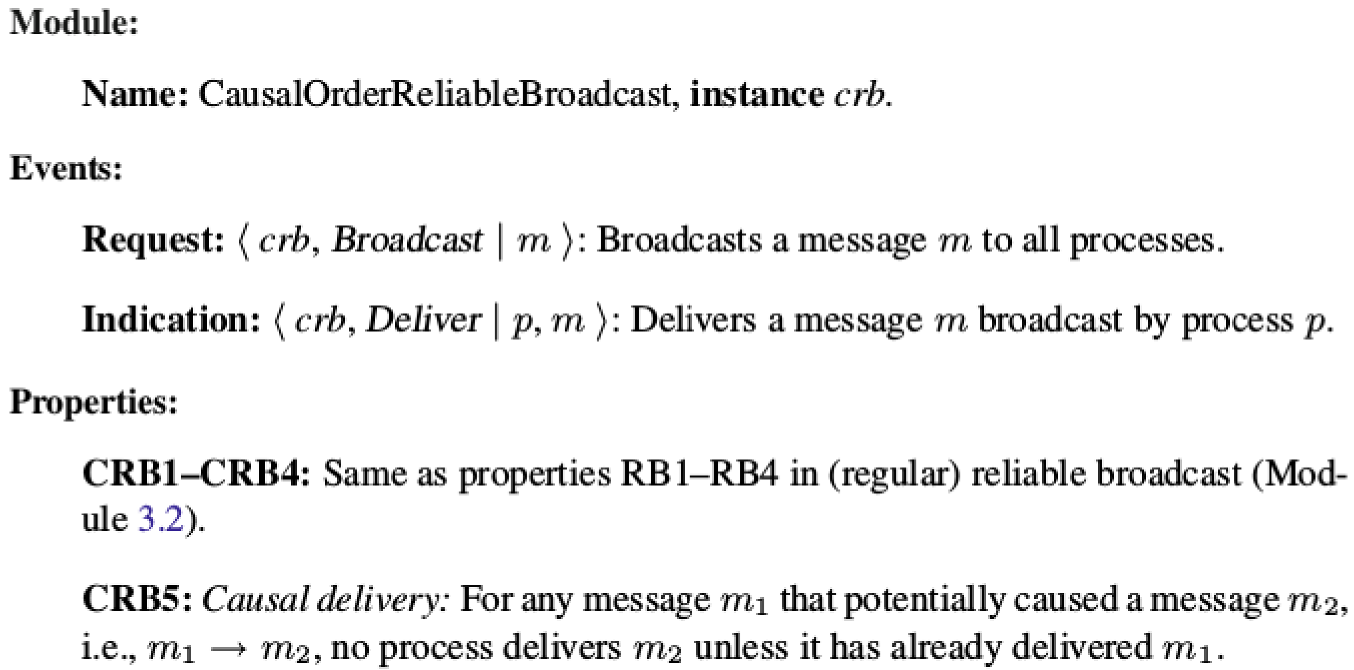
Causal order of messages



A message m_1 may have caused another message m_2, denoted $m_1 \rightarrow m_2$ if any of the following relations apply:

- (a) some process p broadcasts m_1 before it broadcasts m_2;
- (b) some process p delivers m_1 and subsequently broadcasts m_2; or
- (c) there exists some message m' such that $m_1 \rightarrow m'$ and $m' \rightarrow m_2$.

Causal broadcast (crb)



Module:

Name: CausalOrderReliableBroadcast, **instance** *crb*.

Events:

Request: ⟨ *crb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *crb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

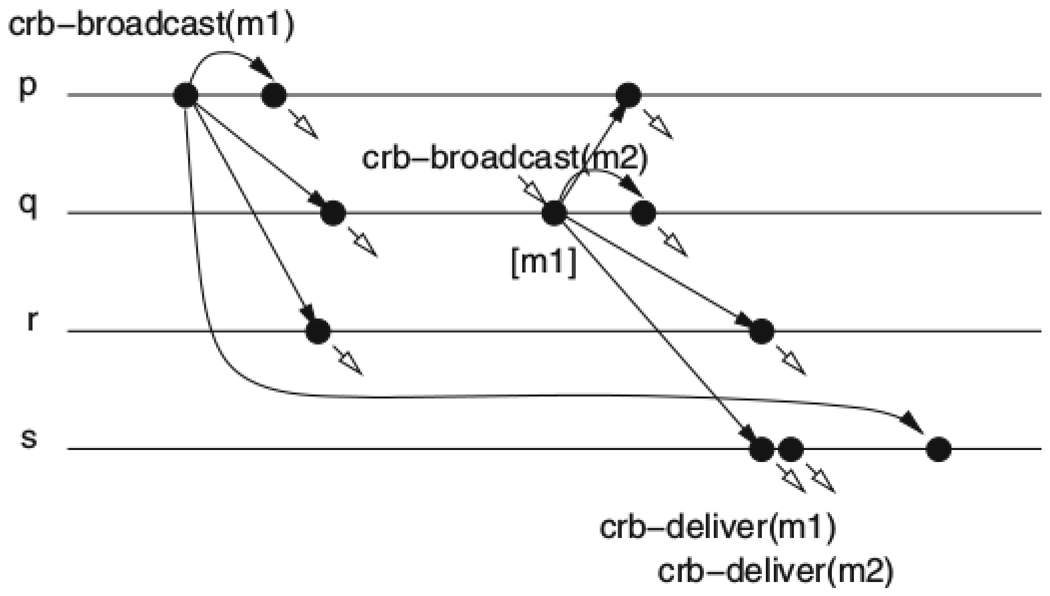
CRB1–CRB4: Same as properties RB1–RB4 in (regular) reliable broadcast (Module 3.2).

CRB5: *Causal delivery:* For any message m_1 that potentially caused a message m_2, i.e., $m_1 \rightarrow m_2$, no process delivers m_2 unless it has already delivered m_1.

No-waiting causal broadcast

```
Implements:
    CausalOrderReliableBroadcast, instance crb.

Uses:
    ReliableBroadcast, instance rb.

upon event ⟨ crb, Init ⟩ do
    delivered := ∅;
    past := [];

upon event ⟨ crb, Broadcast | m ⟩ do
    trigger ⟨ rb, Broadcast | [DATA, past, m] ⟩;
    append(past, (self, m));

upon event ⟨ rb, Deliver | p, [DATA, mpast, m] ⟩ do
    if m ∉ delivered then
        forall (s, n) ∈ mpast do                                  // by the order in the list
            if n ∉ delivered then
                trigger ⟨ crb, Deliver | s, n ⟩;
                delivered := delivered ∪ {n};
                if (s, n) ∉ past then
                    append(past, (s, n));
        trigger ⟨ crb, Deliver | p, m ⟩;
        delivered := delivered ∪ {m};
        if (p, m) ∉ past then
            append(past, (p, m));
```

No-waiting CB example



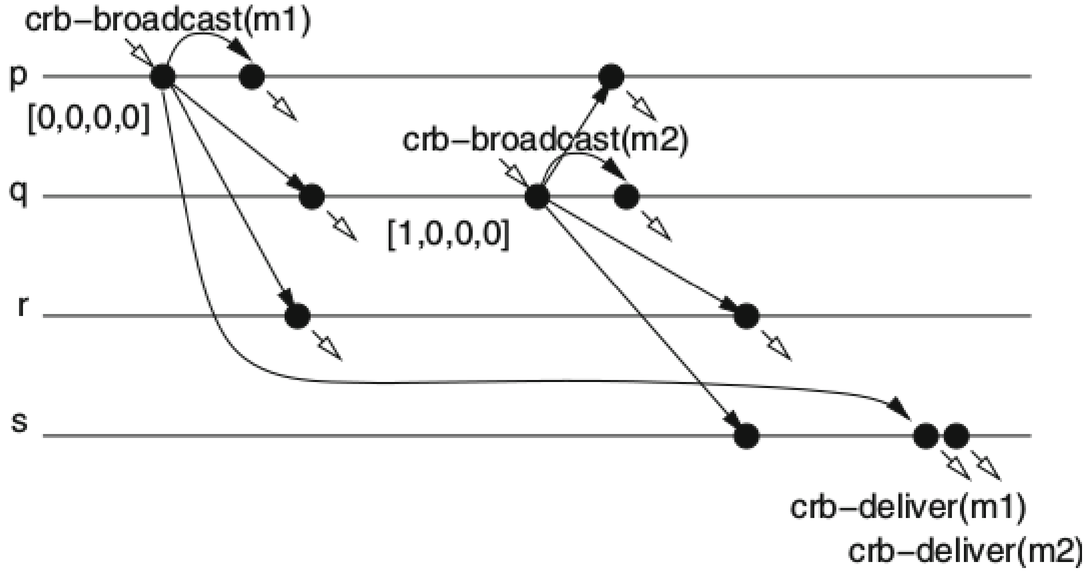
- The size of the message grows with time, as messages include their list of causally preceding messages `mpast`.
- Solution 1: Garbage collect old messages by sending acknowledgements of delivery to all nodes and purging messages that have been acknowledged from all.
- Solution 2: History is a vector timestamp!

Waiting causal broadcast

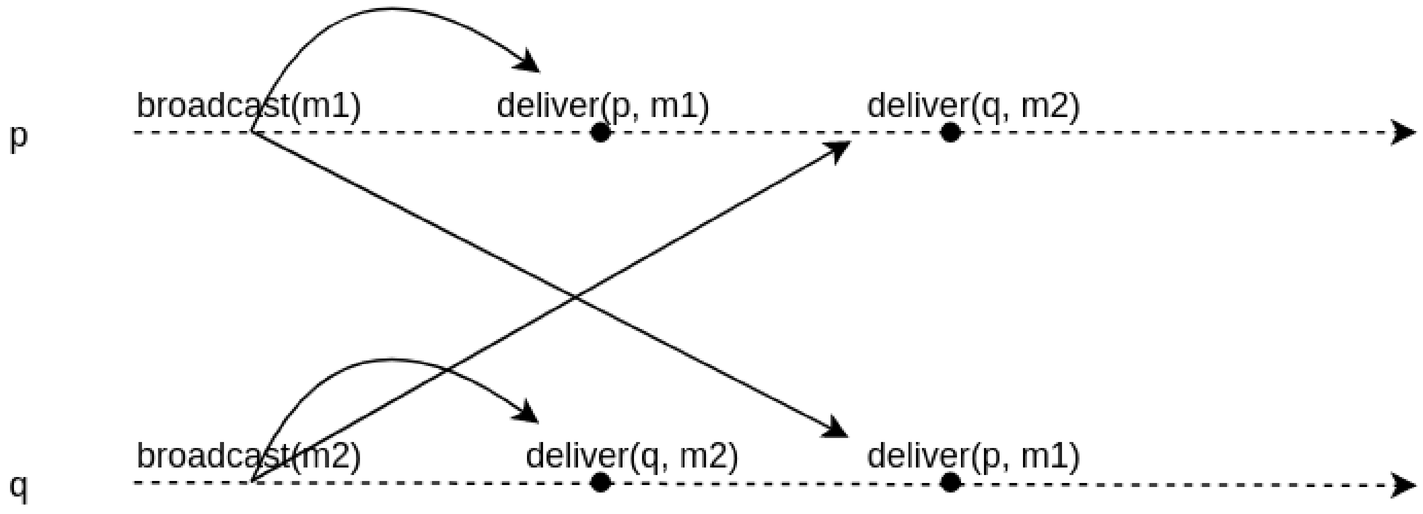
```
Implements:
    CausalOrderReliableBroadcast, instance crb.

Uses:
    ReliableBroadcast, instance rb.

upon event ⟨ crb, Init ⟩ do
    $V := [0]^N$;
    $lsn := 0$;
    $pending := \emptyset$;

upon event ⟨ crb, Broadcast | m ⟩ do
    $W := V$;
    $W[\mathrm{rank}(self)] := lsn$;
    $lsn := lsn + 1$;
    trigger ⟨ rb, Broadcast | [DATA, W, m] ⟩;

upon event ⟨ rb, Deliver | p, [DATA, W, m] ⟩ do
    $pending := pending \cup \{(p, W, m)\}$;
    while exists $(p', W', m') \in pending$ such that $W' \leq V$ do
        $pending := pending \setminus \{(p', W', m')\}$;
        $V[\mathrm{rank}(p')] := V[\mathrm{rank}(p')] + 1$;
        trigger ⟨ crb, Deliver | p', m' ⟩;
```

[Q] Show the correctness of the algorithm.

Waiting CB example



Possible execution?



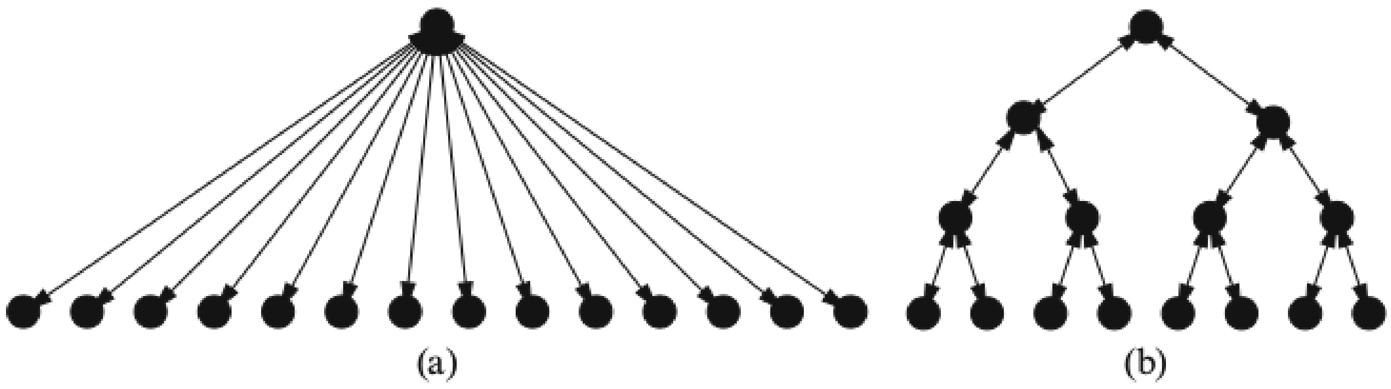
[Q] Is this a valid execution? the order of delivery is not the same.

Probabilistic broadcast

(a.k.a. epidemic broadcast or gossiping.)

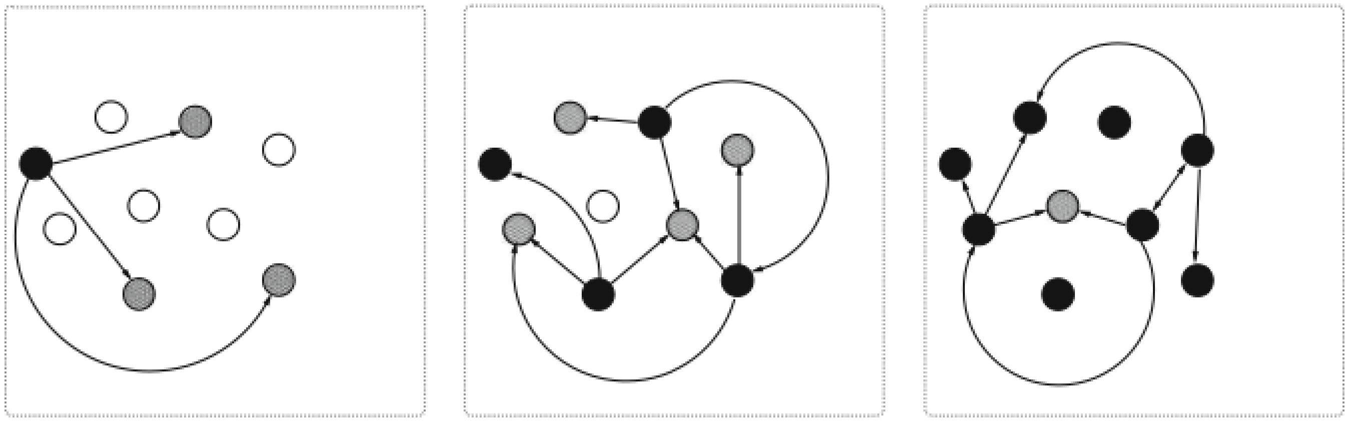
Scalability of reliable broadcast

- In order to broadcast a message, the sender needs
    - to send messages to all other processes,
    - to collect some form of acknowledgement.
    - $O(N^2)$ are exchanged in total.
        - If N is large, this can become overwhelming for the system.
- Bandwidth, memory or processing resources may limit the number of messages/acknowledgements that may be sent/collected.
- Hierarchical schemes reduce the total number of messages.
    - This reduces the load of each process.
    - But increases the latency and fragility of the system.



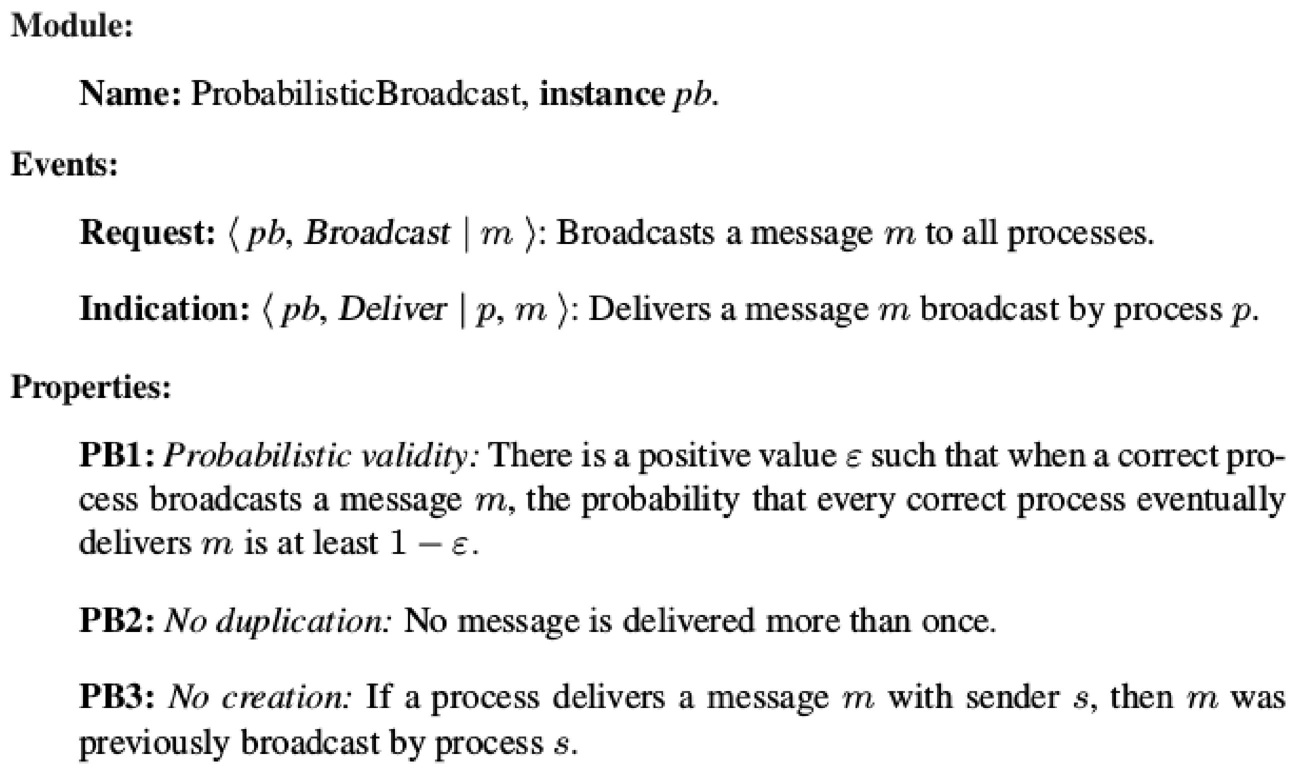
Epidemic dissemination

- Nodes infect each other through messages sent in rounds.
    - The fanout k determines the number of messages sent by each node.
    - Recipients are drawn at random (e.g., uniformly).
    - The number of rounds is limited to R.
- Total number of messages is usually less than $O(N^2)$.
- No node is overloaded.



(a) round 1 (b) round 2 (c) round 3

Probabilistic broadcast (*pb*)



Module:

Name: ProbabilisticBroadcast, **instance** *pb*.

Events:

Request: $\langle$ *pb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *pb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

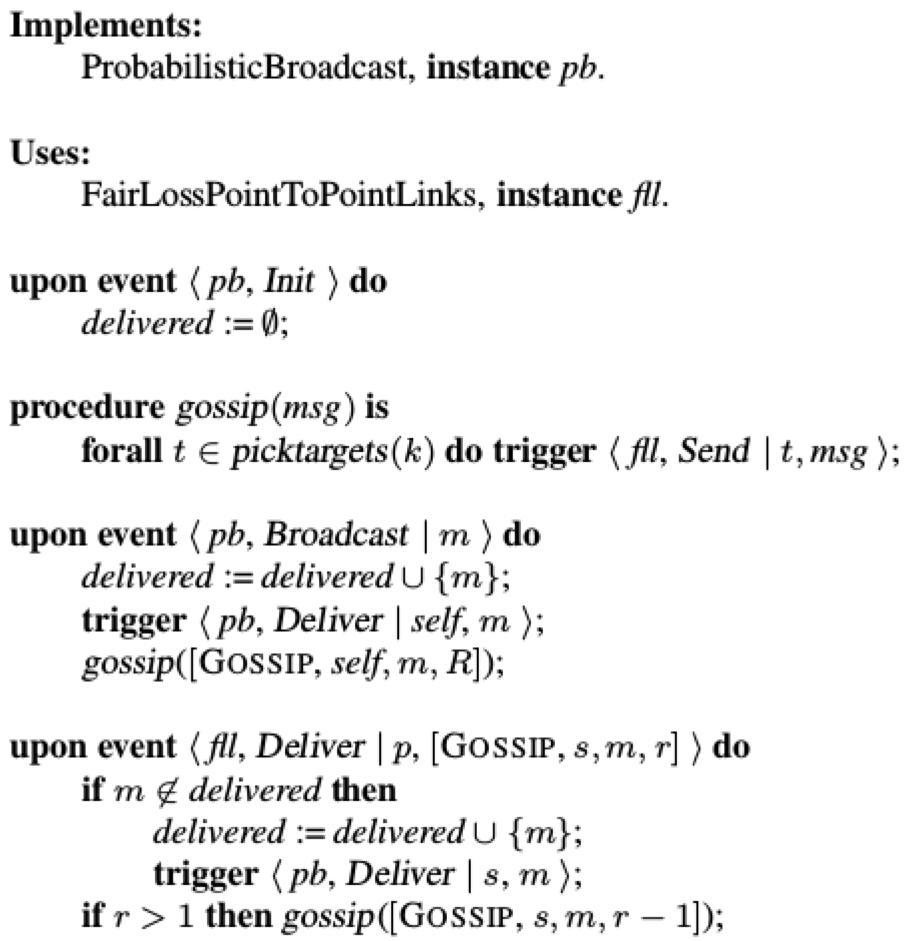
Properties:

PB1: *Probabilistic validity:* There is a positive value ε such that when a correct process broadcasts a message m, the probability that every correct process eventually delivers m is at least $1 - \varepsilon$.

PB2: *No duplication:* No message is delivered more than once.

PB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

Eager probabilistic broadcast



```
Implements:
    ProbabilisticBroadcast, instance pb.

Uses:
    FairLossPointToPointLinks, instance fll.

upon event ⟨ pb, Init ⟩ do
    delivered := ∅;

procedure gossip(msg) is
    forall $t \in$ picktargets(k) do trigger ⟨ fll, Send | t, msg ⟩;

upon event ⟨ pb, Broadcast | m ⟩ do
    delivered := delivered ∪ {m};
    trigger ⟨ pb, Deliver | self, m ⟩;
    gossip([GOSSIP, self, m, R]);

upon event ⟨ fll, Deliver | p, [GOSSIP, s, m, r] ⟩ do
    if $m \notin$ delivered then
        delivered := delivered ∪ {m};
        trigger ⟨ pb, Deliver | s, m ⟩;
    if $r > 1$ then gossip([GOSSIP, s, m, $r - 1$]);
```

The mathematics of epidemics

Assume a virus using a distributed system to propagate, with human hosts as nodes.

Setup

- Initial population of N individuals.
- At any time t,
  - $S(t) =$ the number of susceptible individuals,
  - $I(t) =$ the number of infected individuals.
- $I(0) = 1$
- $S(0) = N - 1$
- $S(t) + I(t) = N$ for all t.

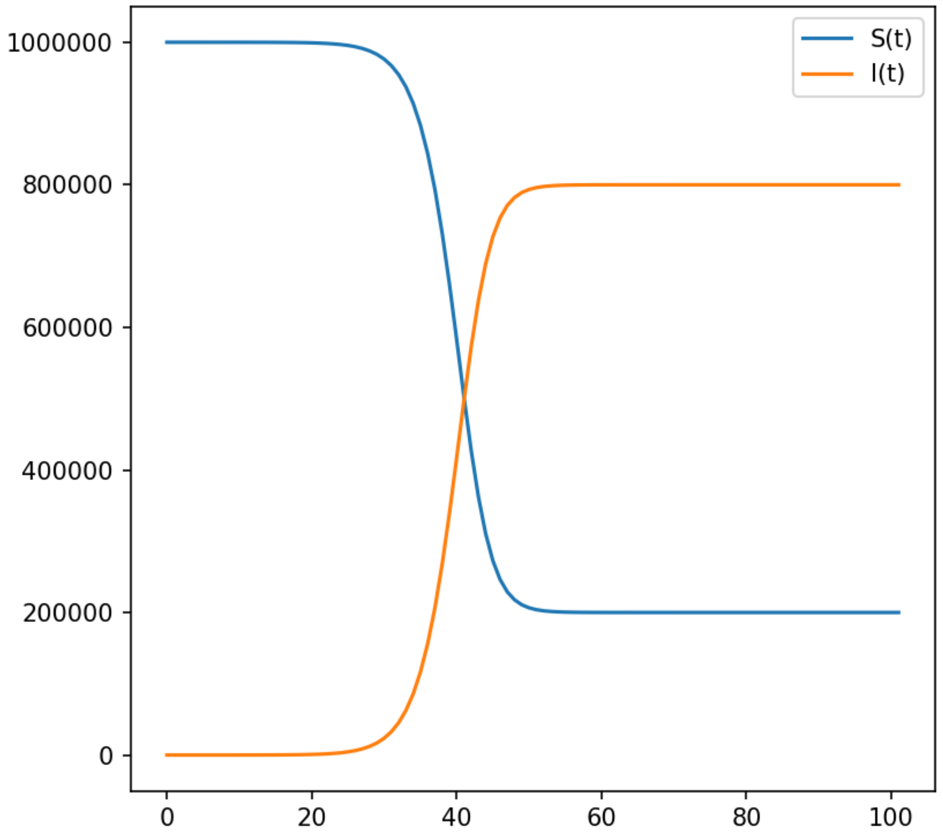
The expected dynamics of the SIS model is given as follows:

$$S(t+1) = S(t) - \frac{\alpha \Delta t}{N} S(t) I(t) + \gamma \Delta t I(t)$$

$$I(t+1) = I(t) + \frac{\alpha \Delta t}{N} S(t) I(t) - \gamma \Delta t I(t)$$

where

- α is the contact rate with whom infected individuals make contact per unit of time.
- $\frac{S(t)}{N}$ is the proportion of contacts with susceptible individuals for each infected individual.
- γ is the probability for an infected individual to recover and switch to the pool of susceptibles.



$$N = 1000000, \alpha = 5, \gamma = 0.5, \Delta t = 0.1$$

In eager reliable broadcast,

- $\alpha = k$
  - An infected node selects k nodes among N to send its messages.
- $\gamma = 1$
  - An infected node immediately recovers.

Probabilistic validity

At time t, the probability of not receiving a message is

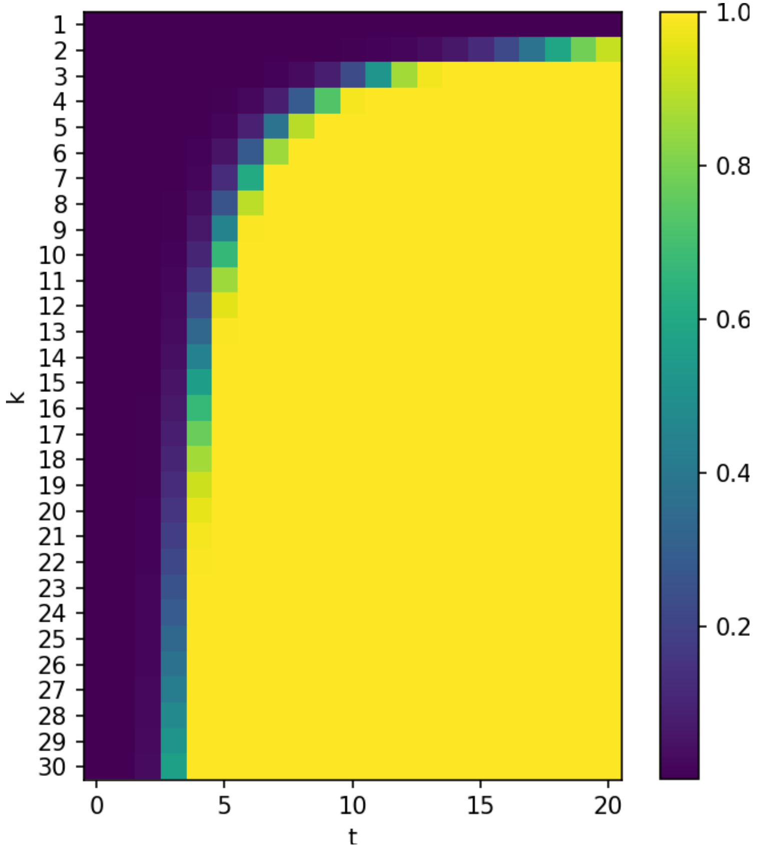
$$(1 - \frac{k}{N})^{i(t)}$$

Therefore the probability of having received of one or more gossip messages up to time t, that is to have PB-delivered, is

$$p(\text{delivery}) = 1 - (1 - \frac{k}{N})^{\sum_{t_i=0}^{t} i(t_i)}$$

[Q] What if nodes fail? if packets are loss?

$p(\text{delivery}|k, t)$



$$N = 1000000, \gamma = 1.0$$

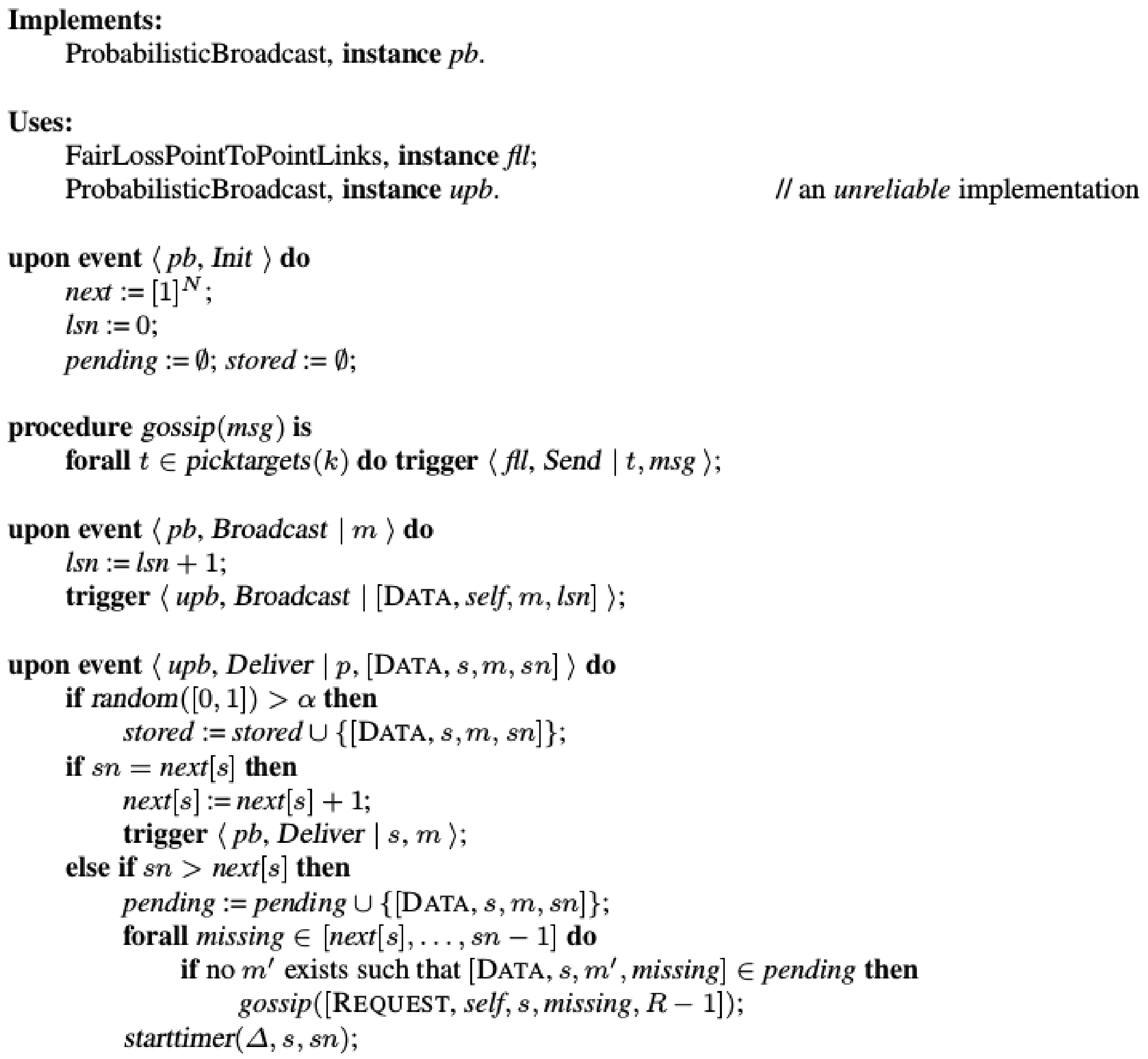
From this plot, we observe that:

- Within only a few rounds (low latency), a large fraction of nodes receive the message (reliability)
- Each node has transmitted no more than kR messages (lightweight).

Lazy Probabilistic broadcast

- Eager probabilistic broadcast consumes considerable resources and causes many redundant transmissions.
    - in particular as r gets larger and almost all nodes have received the message once.
- Assume a stream of messages to be broadcast.
- Broadcast messages in two phases:
    - Phase 1 (data dissemination): run probabilistic broadcast with a large probability ϵ that reliable delivery fails. That is, assume a constant fraction of nodes obtain the message (e.g., $\frac{1}{2}$).
    - Phase 2 (recovery): upon delivery, detect omissions through sequence numbers and initiate retransmissions with gossip.

Phase 1: data dissemination



```
Implements:
    ProbabilisticBroadcast, instance pb.

Uses:
    FairLossPointToPointLinks, instance fll;
    ProbabilisticBroadcast, instance upb.                    // an unreliable implementation

upon event ⟨ pb, Init ⟩ do
    next := $[1]^N$;
    lsn := 0;
    pending := ∅; stored := ∅;

procedure gossip(msg) is
    forall $t \in$ picktargets($k$) do trigger ⟨ fll, Send | $t$, msg ⟩;

upon event ⟨ pb, Broadcast | $m$ ⟩ do
    lsn := lsn + 1;
    trigger ⟨ upb, Broadcast | [DATA, self, $m$, lsn] ⟩;

upon event ⟨ upb, Deliver | $p$, [DATA, $s$, $m$, $sn$] ⟩ do
    if random([0, 1]) > $\alpha$ then
        stored := stored ∪ {[DATA, $s$, $m$, $sn$]};
    if $sn$ = next[$s$] then
        next[$s$] := next[$s$] + 1;
        trigger ⟨ pb, Deliver | $s$, $m$ ⟩;
    else if $sn$ > next[$s$] then
        pending := pending ∪ {[DATA, $s$, $m$, $sn$]};
        forall missing ∈ [next[$s$], ..., $sn - 1$] do
            if no $m'$ exists such that [DATA, $s$, $m'$, missing] ∈ pending then
                gossip([REQUEST, self, $s$, missing, $R - 1$]);
        starttimer($\Delta$, $s$, $sn$);
```

Phase 2: recovery

```
upon event ⟨ $fll$, Deliver | $p$, [REQUEST, $q$, $s$, $sn$, $r$] ⟩ do
    if exists $m$ such that [DATA, $s$, $m$, $sn$] $\in$ $stored$ then
        trigger ⟨ $fll$, Send | $q$, [DATA, $s$, $m$, $sn$] ⟩;
    else if $r > 0$ then
        $gossip$([REQUEST, $q$, $s$, $sn$, $r - 1$]);

upon event ⟨ $fll$, Deliver | $p$, [DATA, $s$, $m$, $sn$] ⟩ do
    $pending := pending \cup \{[\text{DATA}, s, m, sn]\}$;

upon exists [DATA, $s$, $x$, $sn$] $\in$ $pending$ such that $sn = next[s]$ do
    $next[s] := next[s] + 1$;
    $pending := pending \setminus \{[\text{DATA}, s, x, sn]\}$;
    trigger ⟨ $pb$, Deliver | $s$, $x$ ⟩;

upon event ⟨ Timeout | $s$, $sn$ ⟩ do
    if $sn > next[s]$ then
        $next[s] := sn + 1$;
```

Summary

- Reliable multicast enable group communication, while ensuring validity and (uniform) agreement.
- Causal broadcast extends reliable broadcast with causal ordering guarantees.
- Probabilistic broadcast enable low-latency, reliable and lightweight group communication.

The end.

References

- Allen, Linda JS. "Some discrete-time SI, SIR, and SIS epidemic models." Mathematical biosciences 124.1 (1994): 83-105.