Reliable Broadcast

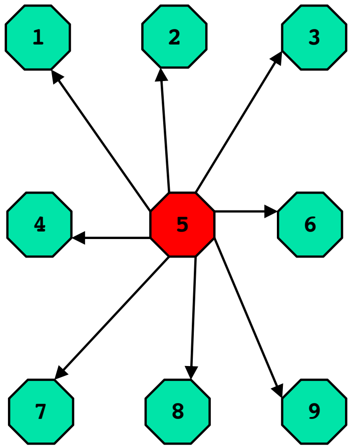
Introduction

Naranker Dulay
n.dulay@imperial.ac.uk

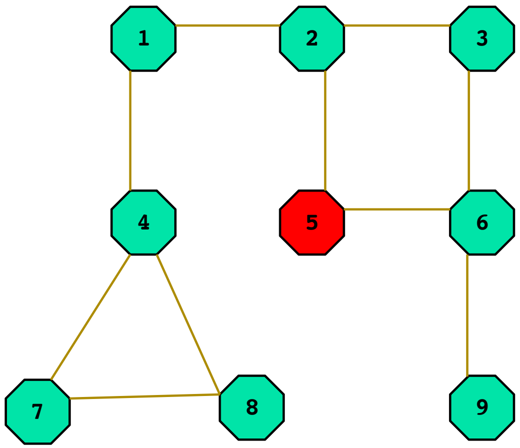
https://www.doc.ic.ac.uk/~nd/dal

Introduction

- We'll start our study of distributed algorithms with algorithms for reliable broadcast in asynchronous message-passing distributed systems that are subject to process failures.



Logical Network

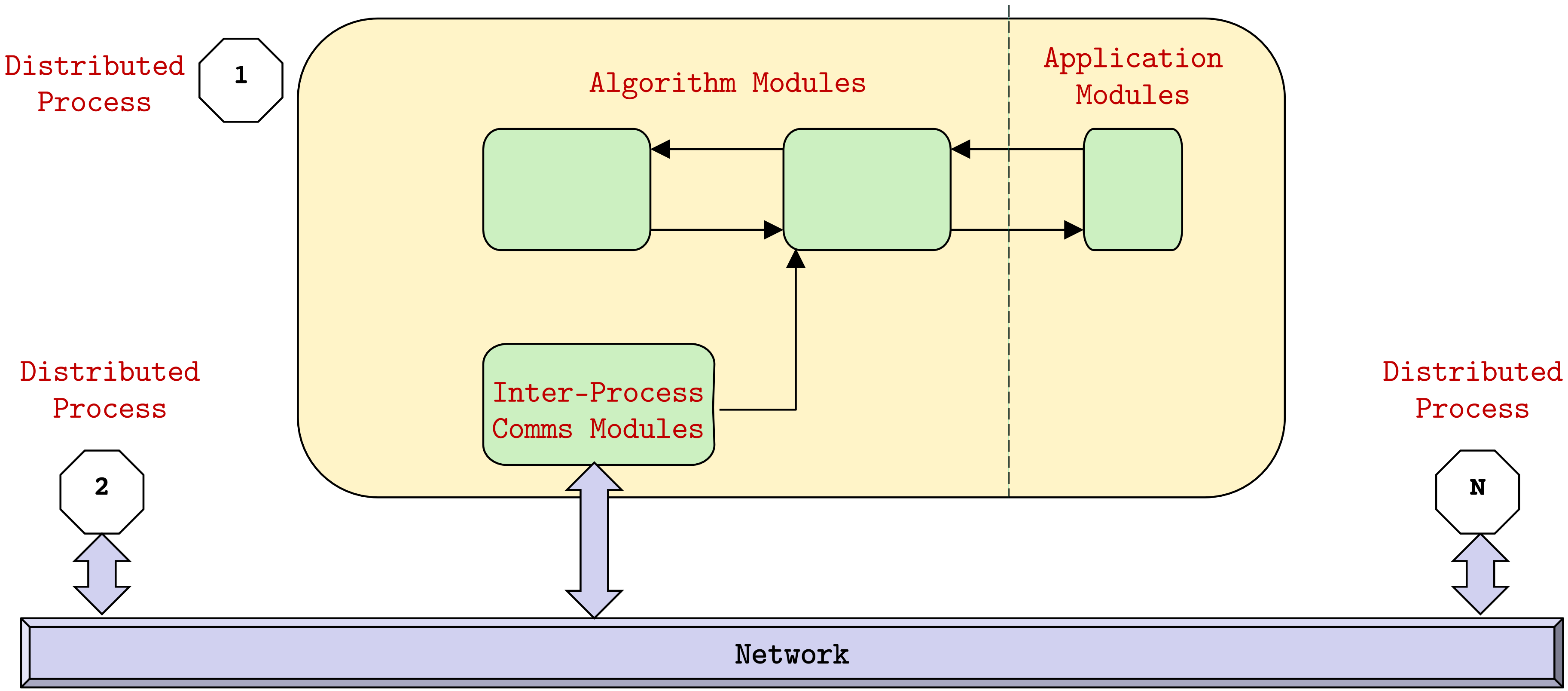


Physical Network

- Guarantee that messages are *consistently* delivered to all processes
- Agreement on the delivered messages
- No ordering among delivered messages – we'll look at ordering next time

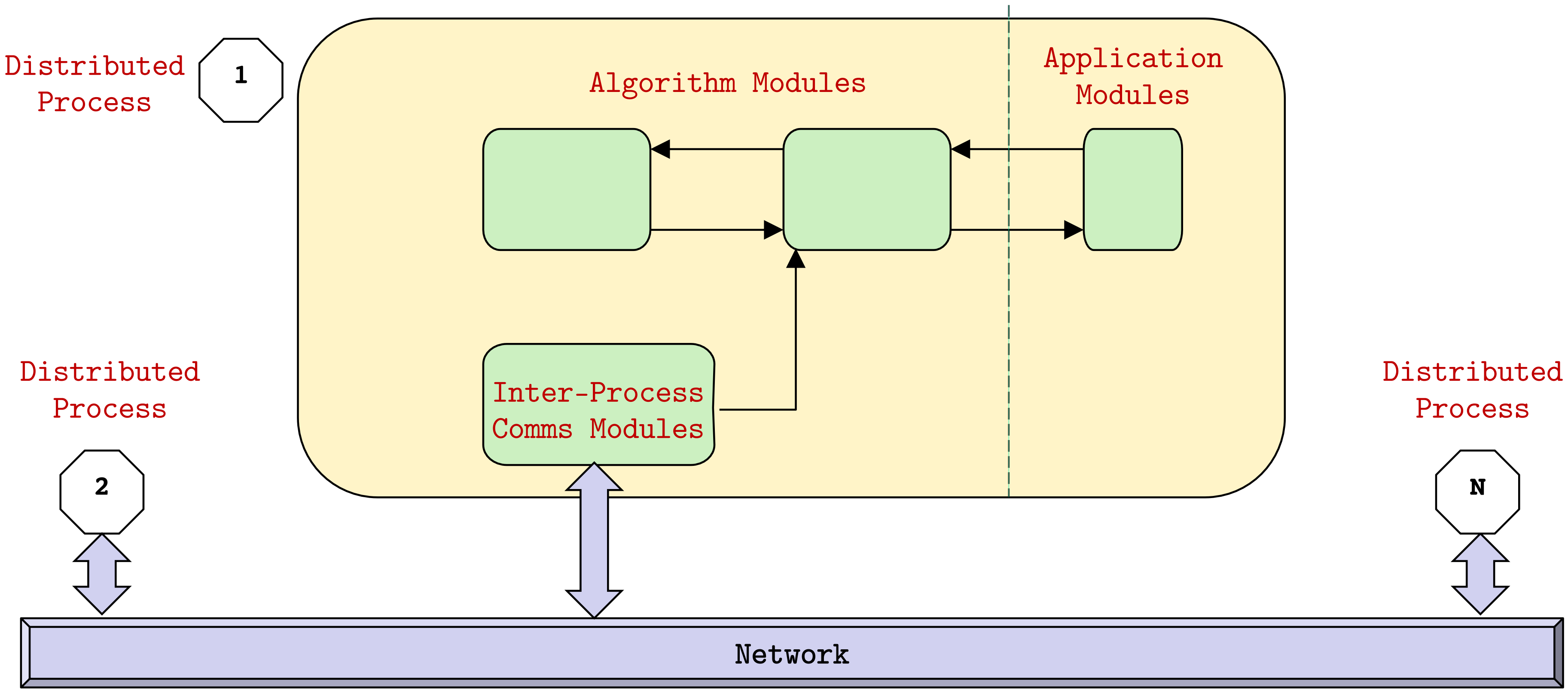
Approach

- We'll structure each distributed process (each Elixir node) as a set of interconnected modules (Elixir processes - green boxes below). Each distributed process (Elixir node) will typically have the same set of modules.



Elixir Mapping

1 Distributed System process (Yellow) = 1 Elixir node = 1 OS process (Yellow)
= Many Elixir processes (Green)



Our Assumptions

- **Asynchronous** systems: (i) No bound on message delays. (ii) No bound on the time to execute a local step in a process. (iii) The time to execute a local step is finite.

  When necessary, we'll implement and use failure detectors, logical clocks or other building blocks (abstractions), these may cause the system to no longer be classed as a (pure) asynchronous system.

- Processes will interact by **passing messages**, not via shared memory.

- We'll assume message passing is reliable. Reliable message passing can also be implemented using a modular approach – see the various "links" modules in **Cachin** for details. We'll assume that Elixir message-passing is reliable and use it instead of using the PL module.

- We'll assume every process can **logically communicate** with every other process i.e. the communication graph is complete and abstracted away.

- We'll assume that the **number of processes is fixed and known**. We'll focus on algorithms for failure models where crashed processes do not continue.

- We'll deviate from these assumptions on occasion.

Classes of Broadcast

ONE SHOT (each message is considered separately from others)

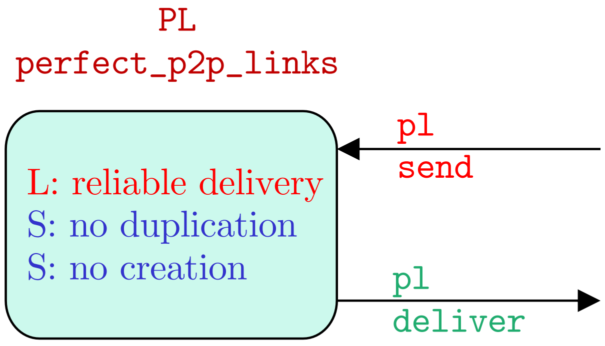
- (Unreliable) Best-Effort Broadcast (BEB)
- Regular Reliable Broadcast (RB) - we'll assume Regular if omitted.
- Uniform Reliable Broadcast (URB)

MULTI-SHOT (involve all messages that are broadcast)

- + FIFO Message Delivery
- + Causal Order Message Delivery
- + Total Order Message Delivery

We're also interested in algorithms with different failure assumptions.

- In Cachin, distributed processes communicate with each other using the Perfect Point-to-Point Links module.



- perfect_p2p_links is the name of the module.
- PL is an instance of the module. For example, created by an enclosing module.
- PL can be passed (i.e. its process-id) to other modules either as a parameter or in a message (e.g. in a **:bind** message). Module instances are implemented as Elixir processes.
- pl_send and **pl_deliver** are the names (**tags**) of messages that are sent or received. We'll typically include them as the 1st field (Elixir atom) of an Elixir tuple **`{:pl_send, ..}`**
- Text in the boxes are short names of the Safety and Liveness properties of the module.

PL : Safety and Liveness Properties

Reliable Delivery (L)

- If Alice and Bob are **correct** (non-faulty) processes then every message sent by Alice to Bob, is *eventually* delivered by Bob.

No Duplication (S)

- No message is delivered to a process more than once.

No Creation (S)

- No message is delivered unless it was sent.

Well use Elixir's send and receive primitives to implement the PL module.

*Read Cachin section 2.4 for how to implement **Perfect P2P Links** using **Stubborn P2P Links** and **Fair P2P Links** modules.*

BEB : Best effort broadcast

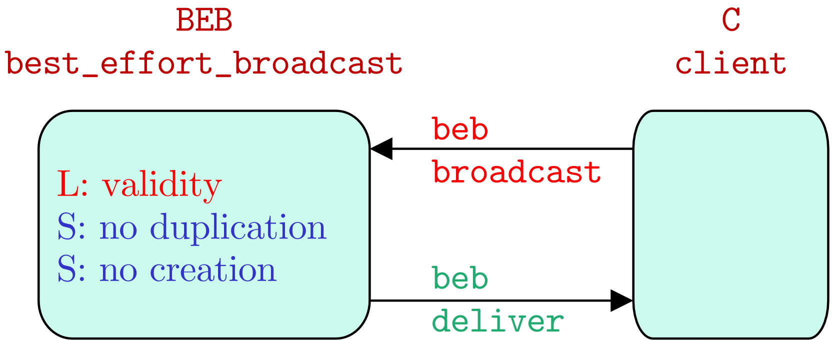
- Given a list of all distributed processes (nodes) and a message, a process could broadcast the message with multiple sends, something like:

```
for p <- processes do PL.send(p, message) end
```

- If ***message sending* is reliable** (e.g. uses PL), then for BEB, the message will be *delivered* to **every correct process. Crashed processes** may or may not have received the message.
- In an asynchronous system, messages will be received at arbitrary times.
- If the broadcasting process crashes during sending, then some arbitrary subset of processes will receive the message (there is no delivery guarantee for this).
- The broadcasting process will not know which processes received the message.

- In order to simplify code examples, like **Cachin,** we'll assume that all process-to-process messages are unique. For example, messages might include a unique process-id (node-id) plus a unique message number (or local timestamp). Such uniqueness meta-data will need to used if implementing in Elixir.

BEB : Safety and Liveness



Validity (L)

- If a correct process broadcasts a message then every correct process eventually delivers it.

No Duplication (S)

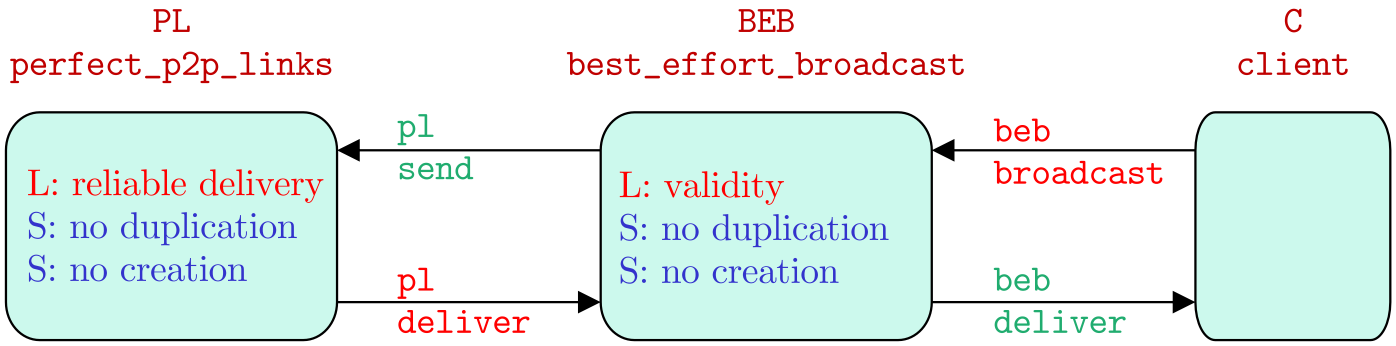
- No message is delivered to a process more than once.

No Creation (S)

- No message is delivered unless it was broadcast.

Note the similarity to the Safety & Liveness properties of the PL module

BEB : Basic broadcast (fail-silent)

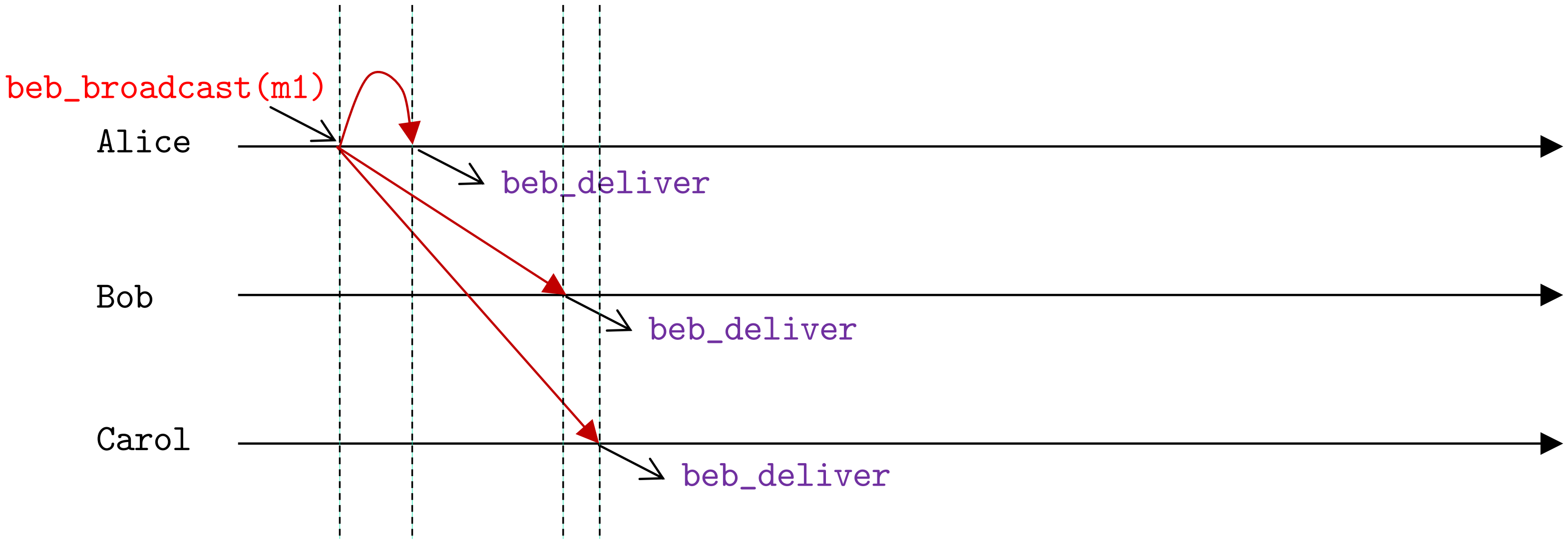


- Send message to each process using the ***perfect_p2p_links*** module (PL).
- Works because PL will ensure all correct processes will deliver the message if the sender of the message does not crash.
- Recall that an Algorithm is ***fail-silent*** if process crashes can never be reliably detected.

BEB : Basic broadcast 1

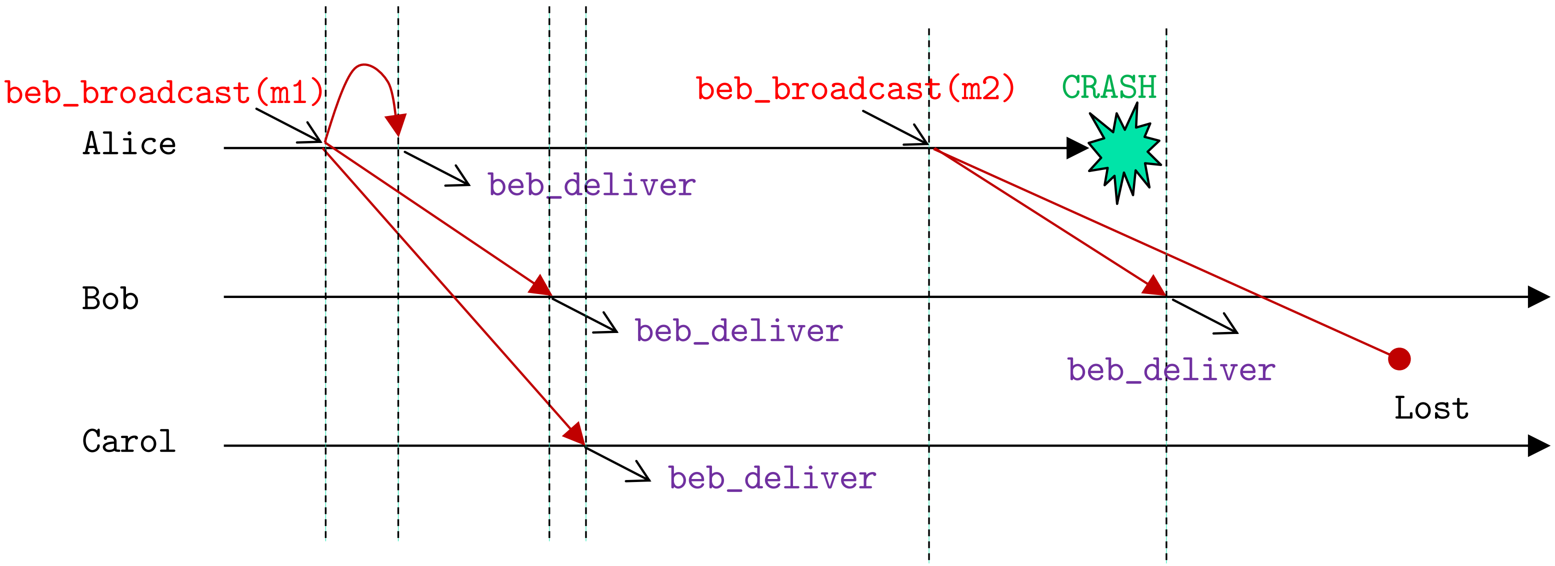
- Alice.C *beb_broadcasts* m1 which is “beb_delivered” by Alice’s, Bob’s and Carol’s BEB modules to Alice.C, Bob.C and Carol.C

- In Elixir, m1 is appended to the process queues of Alice.C, Bob.C and Carol.C



BEB : Basic broadcast 2

- Alice.C successfully *beb_broadcasts* m1 as before.
- However while *beb_broadcasting* m2 Alice crashes.
- m2 is successfully "beb_delivered" to Bob.C.
- But m2 is not "beb-delivered" to Alice (the crashed process) nor Carol (a correct process)



BEB : Basic Broadcast


```
defmodule BEB do            # basic best effort broadcast

def start(processes) do
  receive do { :bind, c, pl } ->
    %{ c: c, pl: pl, processes: processes } |> next()
  end # receive
end # start

defp next(this) do
  receive do
    { :beb_broadcast, msg } ->
       for dest <- this.processes do send this.pl, { :pl_send, dest, msg } end
    { :pl_deliver, from, msg } ->
       send this.c, { :beb_deliver, from, msg }
  end # receive
  this |> next()
end # next

end # BEB
```

BEB : Basic Broadcast

Correctness

- Recall Validity for BEB is defined as - *if a correct process broadcasts a message then every correct process eventually delivers it.*

  Validity(L) is derived from the *reliable delivery(L)* property of PL plus the fact that the broadcasting process sends the message to all processes (lines 10)

- *No duplication(S)* and *No creation(S)* are derived from the corresponding safety properties of PL.

- *No duplication(S)* also assumes that messages are unique.

Performance

- 1 broadcast step. O(N) messages, where N is the number of processes.

RB : Reliable broadcast

- For **best-effort broadcast (BEB)**, if the sending process crashes during a broadcast, then some arbitrary subset of processes will receive the message. Note: Even if the sending process sends all messages and then crashes, delivery is not guaranteed by *perfect_p2p_links* (PL).

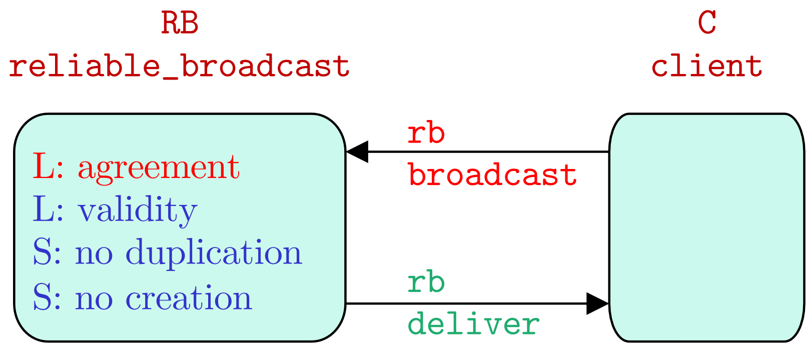
- Hence for BEB there is no delivery agreement guarantee – correct processes do not agree on the delivery of the message.

- Reliable broadcast algorithms provide a delivery agreement guarantee.

- With (regular) reliable broadcast all CORRECT processes will agree on the messages they deliver, even if the broadcasting process crashes while sending.

- Note: if the broadcasting process crashes before any message is sent, then no message is delivered. This satisfies reliable broadcast because all correct (non-faulty) processes will agree on this.

RB : Safety and Liveness



- Validity(L), No Duplication(S) and No Creation(S) properties are the same as in *Best Effort Broadcast*

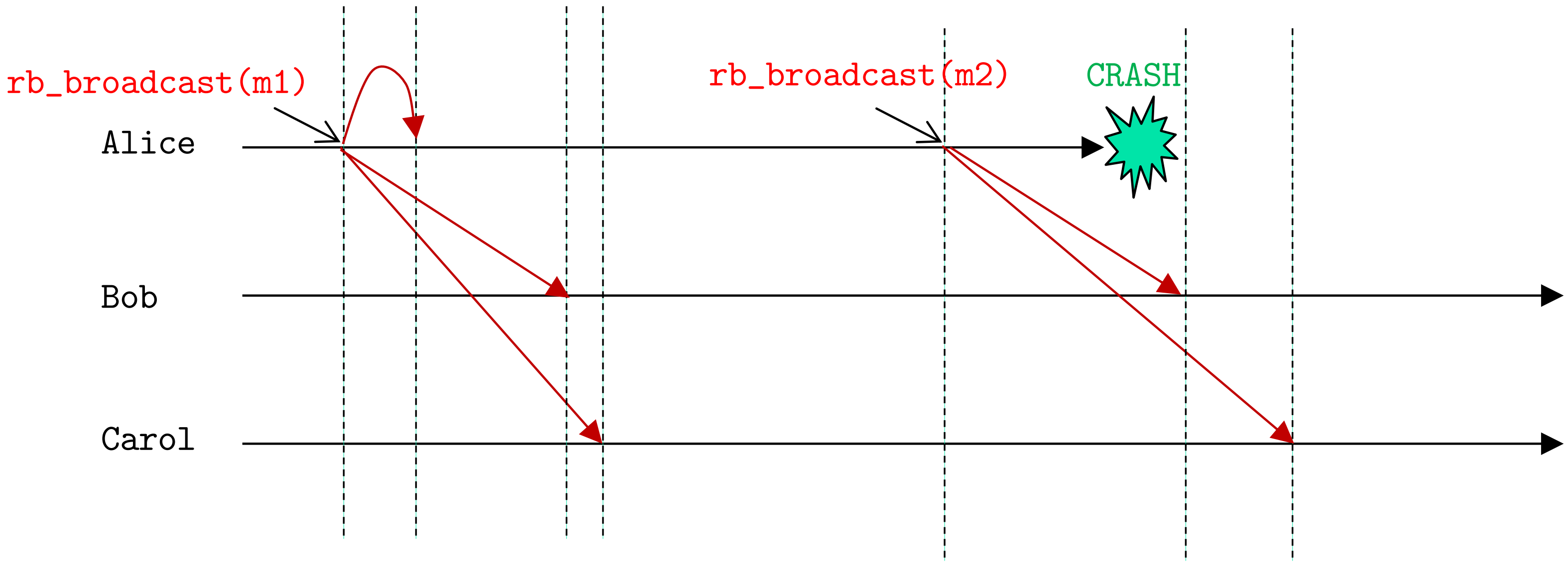
Agreement(L)

- If a correct process <u>delivers message M</u> then every correct process also delivers M
- Validity(L) and Agreement(L) together provide a Termination property for broadcasting a message.
- Only correct processes are required to deliver the message. So faulty processes could deliver messages not delivered by correct processes.

RB : Reliable broadcast 1

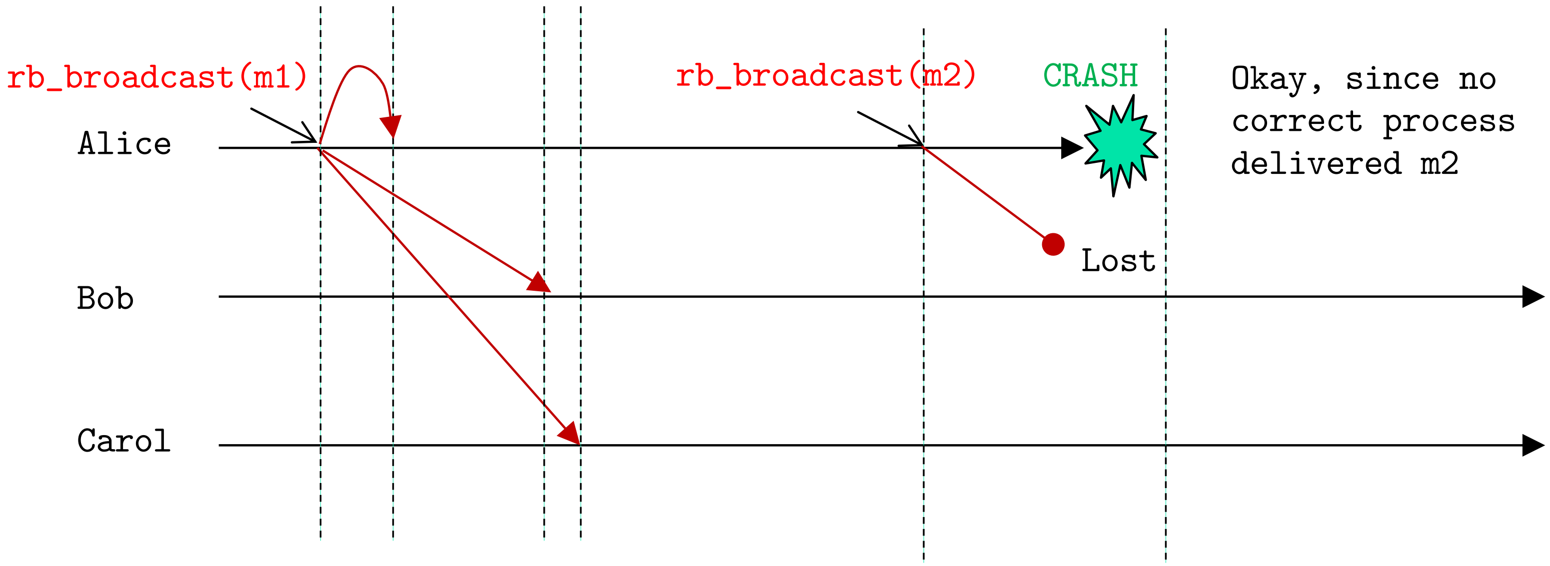
- Alice.C *rb_broadcasts* m1, which is "rb_delivered" by Alice's, Bob's and Carol's RB modules to Alice.C, Bob.C and Carol.C

- While *rb_broadcasting* m2, Alice crashes. m2 is successfully "rb_delivered" to Bob.C and Carol.C m2 is not "rb_delivered" to Alice.C
- This satisfies the agreement property of reliable broadcast, since both correct processes deliver m2.



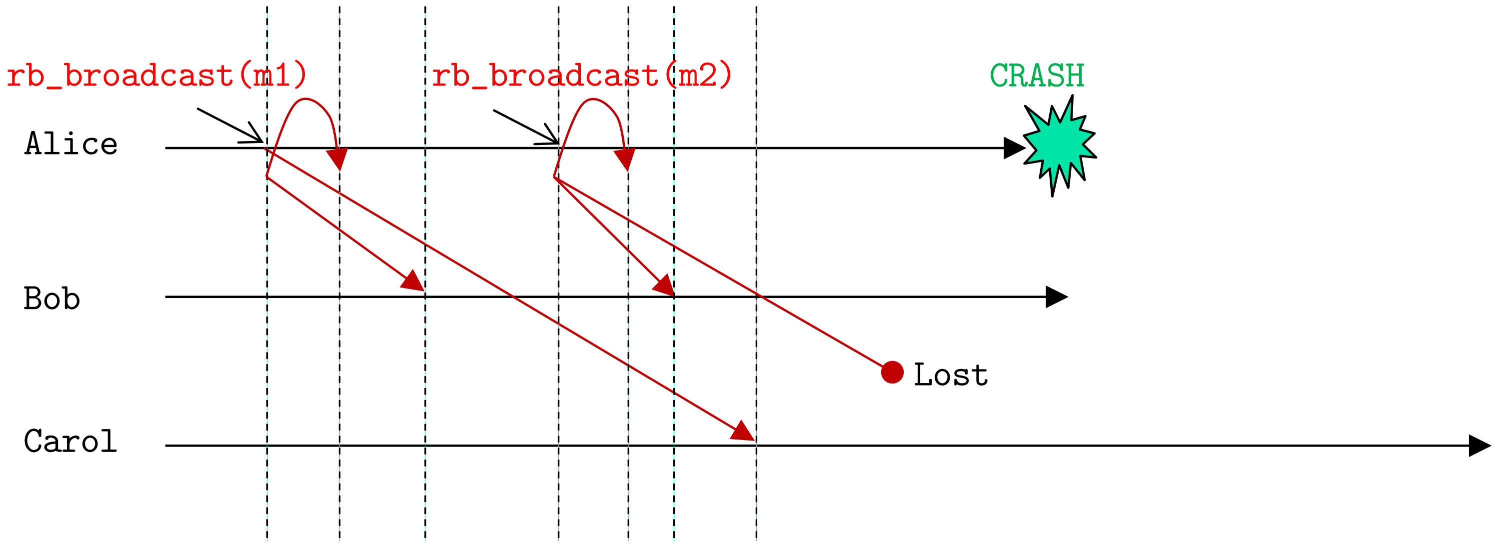
RB : Reliable broadcast 2

- While *rb_broadcasting* m2, Alice crashes.
- m2 is not “rb_delivered” to Bob.C or Carol.C.
- This also satisfies the agreement property of reliable broadcast, since neither correct process delivers m2.



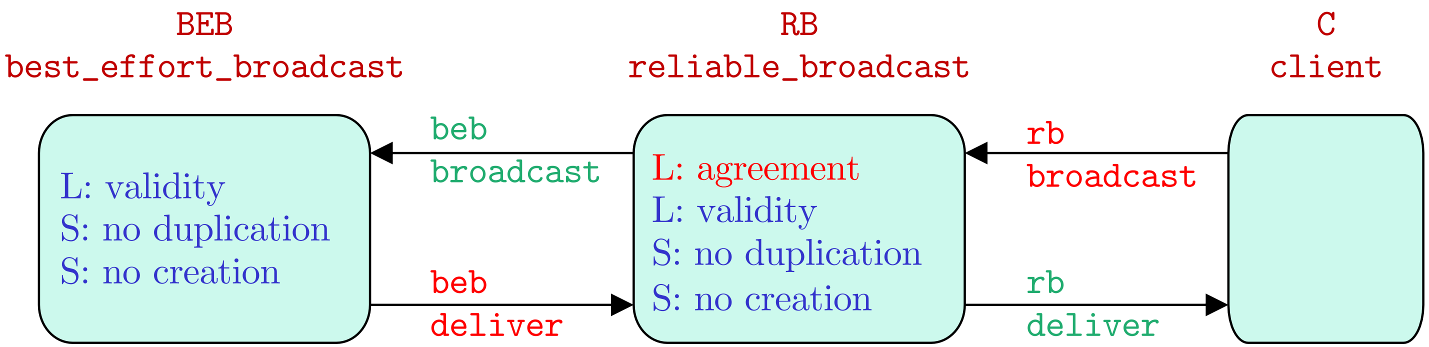
RB : Reliable broadcast 3

- While *rb_broadcasting* m2, Alice crashes.
- m2 is successfully "rb_delivered" to Alice.C and Bob.C only.
- m2 is not "rb_delivered" to Carol.C
- This does not satisfy the agreement property of reliable broadcast since only one correct process delivered m2 (Bob), the other correct process (Carol) did not.



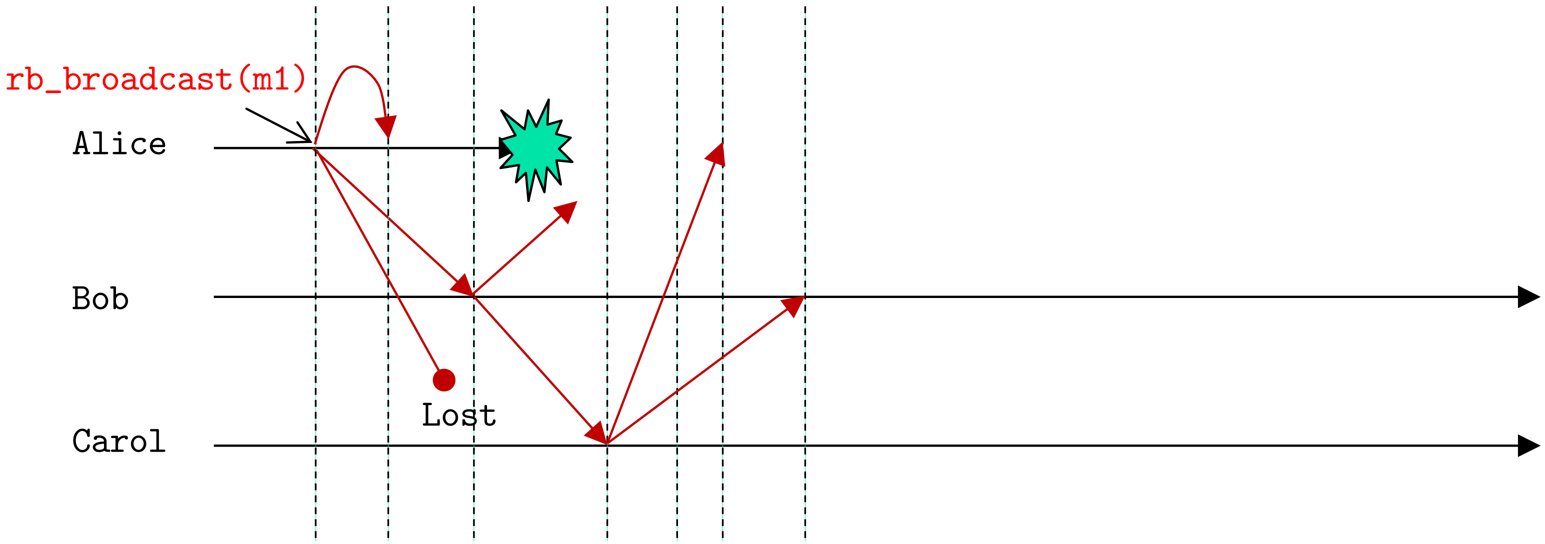
RB : Eager Reliable Broadcast (fail-silent)

- Every process re-broadcasts every message it delivers.
- So, if the broadcasting process crashes, the message will be forwarded by other processes using Best Effort Broadcast.



RB : Eager Reliable broadcast

- While *rb_broadcasting* m1, Alice crashes.
- m1 is successfully "rb_delivered" to Bob.C from Alice's broadcast of m1 and to Carol.C from Bob's re-broadcast of m1. The Alice to Carol message was lost or corrupted.
- m1 is also "rb_delivered" to Alice.C before Alice crashes, but this does not matter since we only care that the correct processes (Bob and Carol) are in agreement about the delivery or non-delivery of messages.



RB : Eager Reliable Broadcast


```
defmodule RB do                  # eager_reliable_broadcast
def start do
  receive do { :bind, c, beb } ->  %{ c: c, beb: beb, delivered: empty_set() } |> next() end
end # start

defp next(this) do
  receive do
    { :rb_broadcast, msg } ->
       send this.beb, { :beb_broadcast, { :rb_data, nodeID(), msg } }
       this |> next()
    { :beb_deliver, from, { :rb_data, sender, msg } = data } ->
       if msg in this.delivered do   # msg already delivered
            this |> next()
       else send this.c,   { :rb_deliver, sender, msg }
            send this.beb, { :beb_broadcast, data }
            this |> delivered_put(msg) |> next()
       end # if
  end # receive
end # next
end # RB
```

delivered is the set of rb_delivered messages

RB : Eager Reliable Broadcast

Correctness

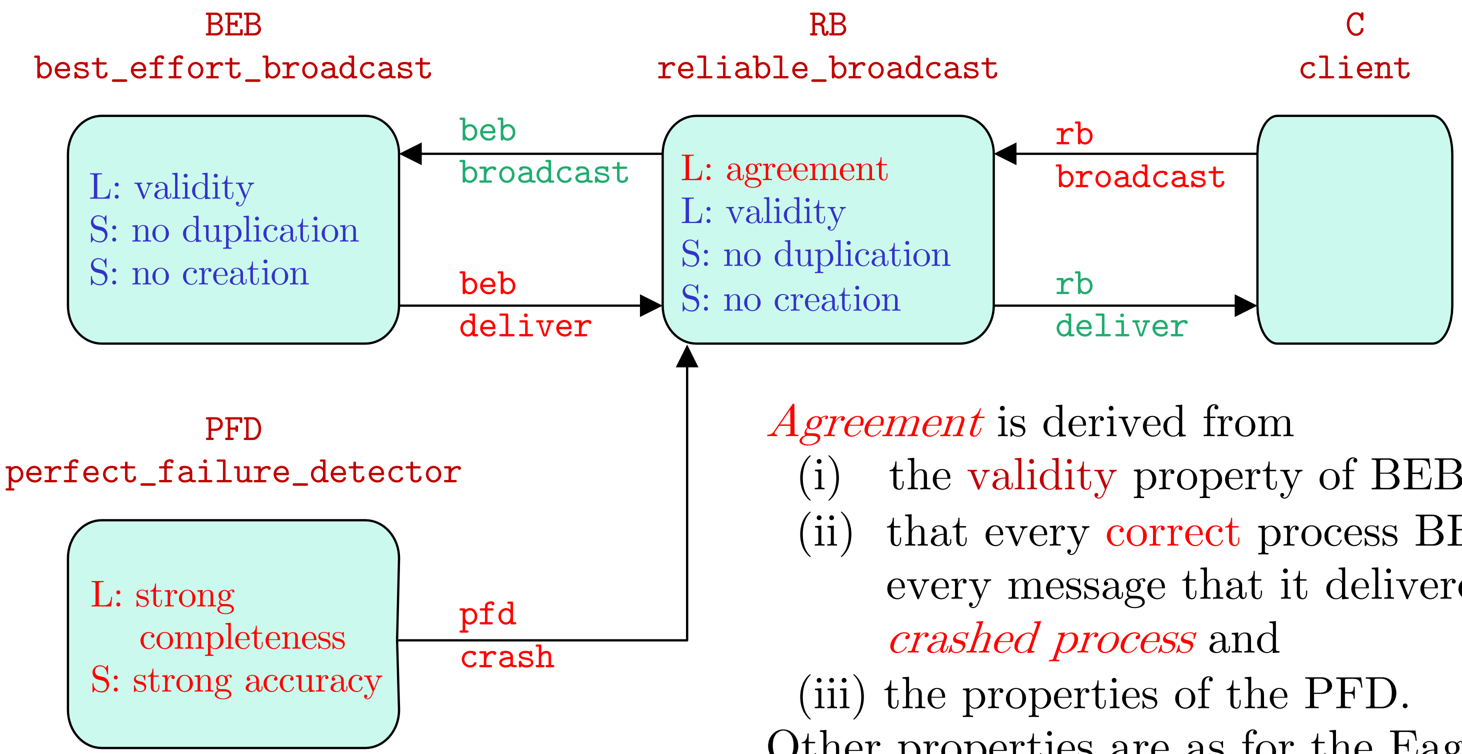
- *Agreement(L)* is derived from the *validity(L)* property of BEB and the fact that every correct process immediately BEB-broadcasts every message it delivers (line 15).
- *No creation(S)* and *validity(L)* are derived from corresponding properties of BEB.
- *No duplication(S)* because the algorithm keeps track of all messages that have been delivered (lines 11 & 16) and the assumption that messages are unique.

Performance

- O(N) BEB broadcasts, O(N^2) messages.

RB: Lazy Reliable broadcast (fail-stop)

- Uses best-effort-broadcast, but includes a failure detector module/algorithm to detect processes that have failed (&stopped).



Agreement is derived from

(i) the validity property of BEB,

(ii) that every correct process BEB-broadcasts every message that it delivered from a *crashed process* and

(iii) the properties of the PFD.

Other properties are as for the Eager RB algorithm.

RB : Lazy Reliable Broadcast 1


```
defmodule RB do                    # lazy reliable broadcast

def start do
  receive do
    { :bind, c, beb, processes } ->
      %{ c: c, beb: beb, correct: processes, delivered: init_map(processes, empty_set) } |> next()
  end # receive
end # start

defp next(this) do
  receive do
    { :rb_broadcast, msg } ->
      send this.beb, { :beb_broadcast, { :rb_data, nodeID(), msg } }
      this |> next()

    { :pfd_crash, crashedP } ->
      for msg <- this.delivered[crashedP] do
        send this.beb, { :beb_broadcast, { :rb_data, crashedP, msg } }
      end # for
      this |> correct_delete(crashedP) |> next()      # continues on next slide
```

Broadcast crashedP's delivered msgs

RB : Lazy Reliable Broadcast 2

```
    { :beb_deliver, from, { :rb_data, sender, msg } = data } ->
       if msg in this.delivered[sender] do
          this |> next()
       else
          send this.c, { :rb_deliver, sender, msg }

          if sender not in this.correct do
             send this.beb, { :beb_broadcast, data }
          end # if

          this |> delivered_put(sender, msg) |> next
        end # if
    end # receive
end # next

end # RB
```

```
slow delivery of message from crashed process
i.e. crash was detected before beb delivery of msg
```

```
add msg to the set of messages received from sender
```

Failure Detectors

Perfect failure detector P

- Provides processes with a list of *suspected* processes (detected processes) that have crashed.
- Makes timing assumptions (i.e. system is no longer asynchronous)
- Never changes its view – suspected processes remain suspected forever.

Eventually perfect failure detector $\Diamond P$

- May make mistakes but will eventually accurately detect a crashed process.

Strong completeness(L)

- Every process that crashes will eventually be permanently suspected by every correct process

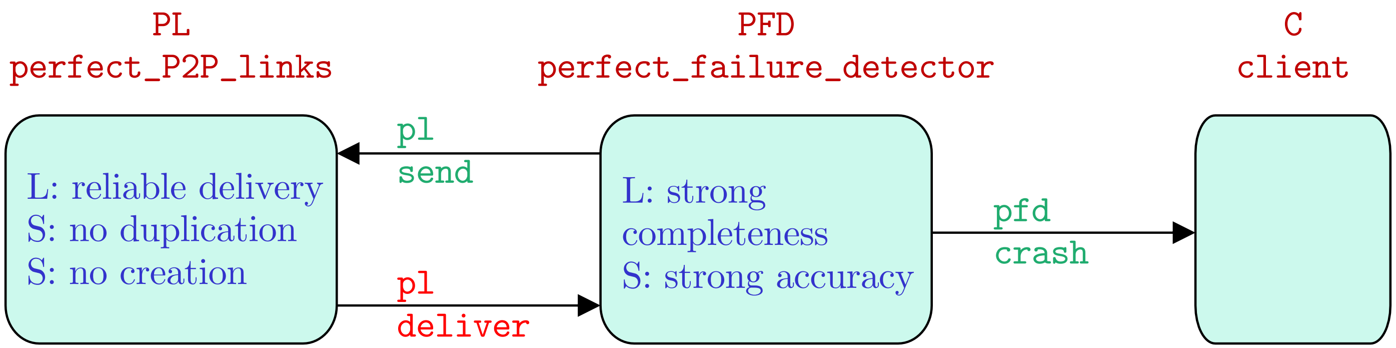
Strong accuracy(S)

- No process is suspected before it crashes

Eventually strong accuracy(L)

- Eventually no correct process is suspected

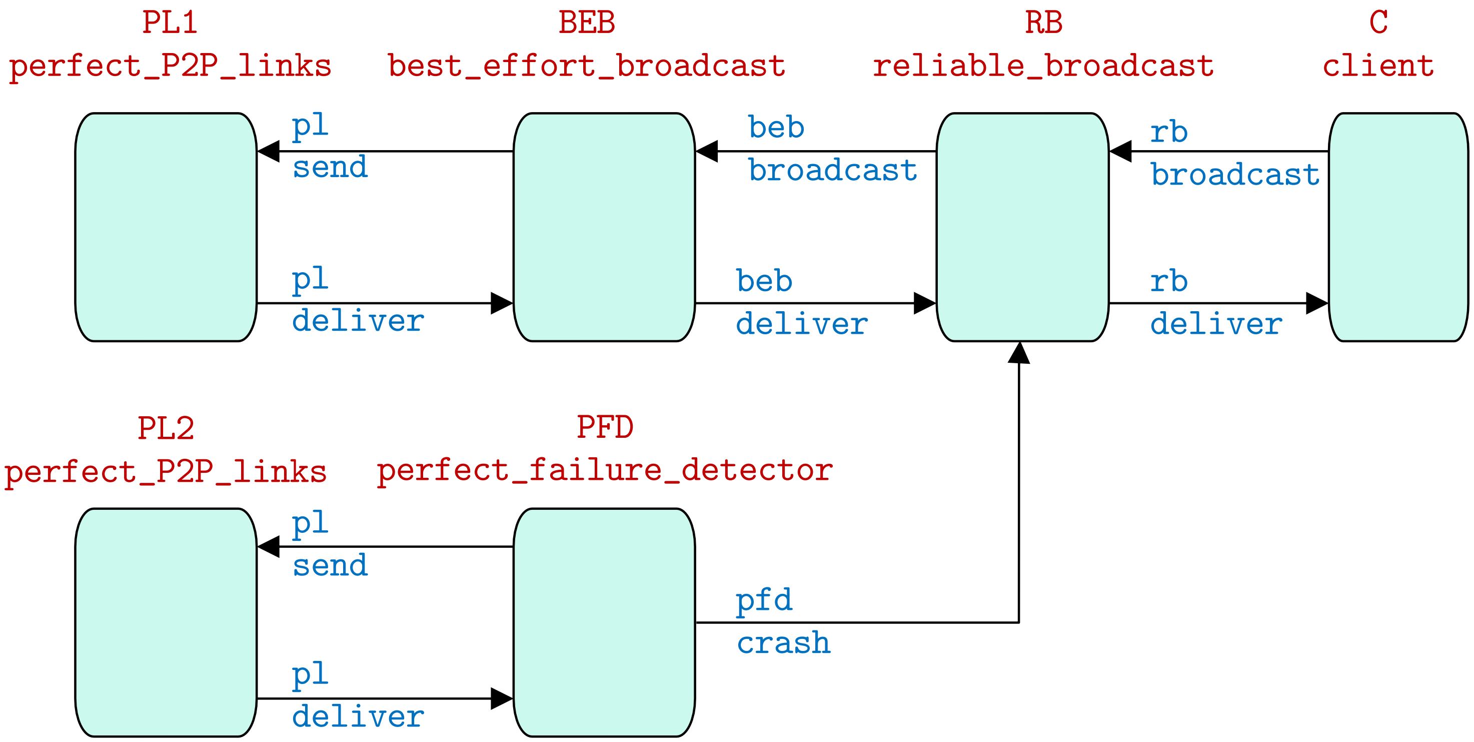
PFD: Exclude on timeout



- Uses PL to exchange *heartbeat* messages (request and reply) and uses a timeout period for replies in order to suspect/detect a crashed process.
- Recall PL performs reliable sending for correct processes.
- The timeout period needs to be large enough to send a heartbeat message to all processes, processing at the receiving processes and getting replies back (Synchrony assumption).

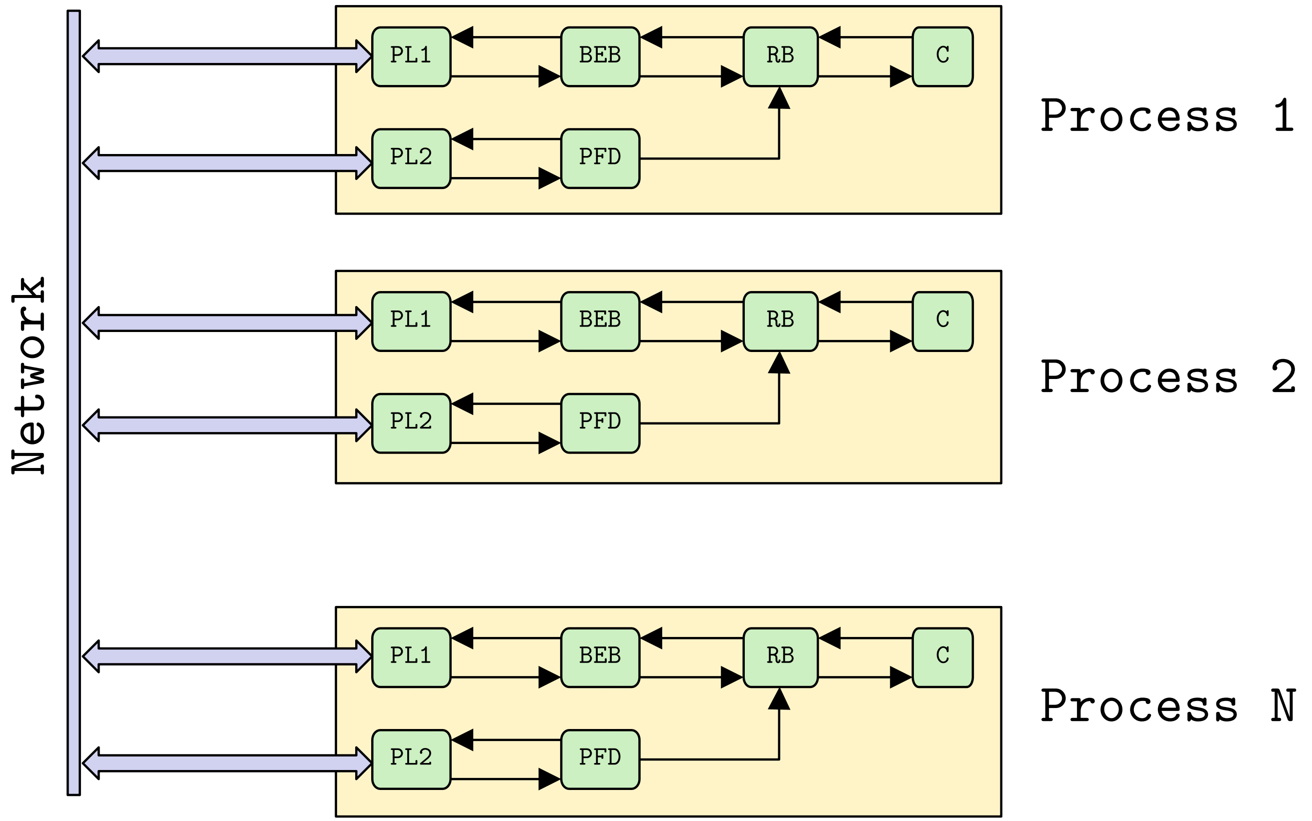
Process configuration

- Putting all the modules for Lazy Reliable Broadcast together, we have in ***each distributed process*** (i.e. in each *Elixir node/OS process*) the following configuration of **modules** (*Elixir processes*)



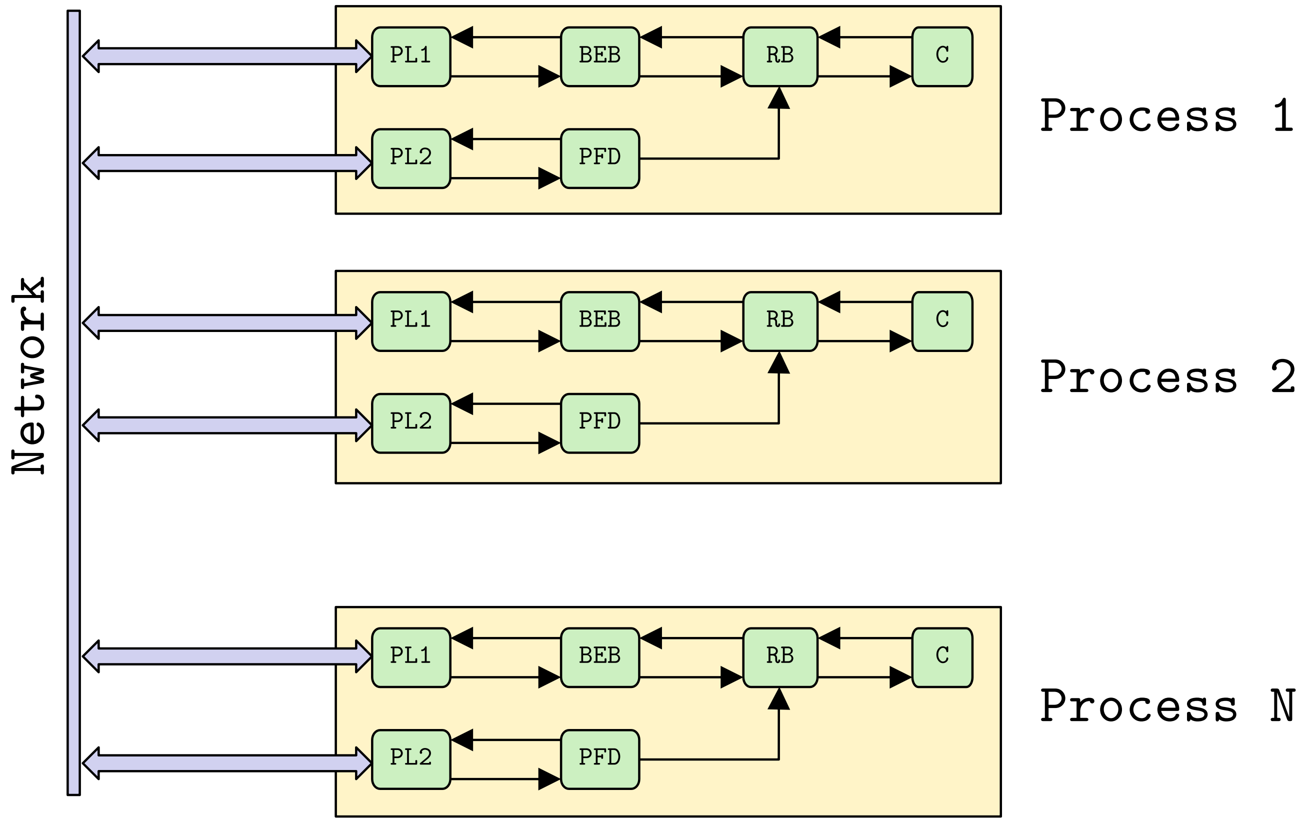
Distributed system configuration

- Our distributed processes are connected by a *logical* network giving us the following configuration of distributed processes.



Reliable broadcast message delivery

- Module P2.C would receive a broadcast message from P1.C via the path:
  P1.C → P1.RB → P1.BEB → P1.PL1 → Network →
  P2.PL1 → P2.BEB → P2.RB → P2.C



PFD: Exclude on timeout 1

```
defmodule PFD do                # perfect failure detector exclude on timeout

def start do
  receive do
    { :bind, c, pl, delay, processes } ->
      Process.send_after(self(), :timeout, delay)
      %{ c: c, pl: pl, delay: delay, processes: processes, alive: processes, crashed: empty_set() }
        |> next
  end # receive
end # start

defp next(this) do
  receive do
    { :pl_deliver, from, :heartbeat_request } ->
       send this.pl, { :pl_send, from, :heartbeat_reply }
       this |> next()

    { :pl_deliver, from, :heartbeat_reply } ->
        this |> alive_put(from) |> next()

# continued on next slide
```

PFD: Exclude on timeout 2

```
  :timeout ->
     newly_crashed =
       for p <- this.processes, p not in this.alive and p not in this.crashed, into: empty_set do p end
     for p <- newly_crashed do send this.c, { :pfd_crash, p } end
     for p <- this.alive do send pl, { :pl_send, p, :heartbeat_request } end
     Process.send_after(self(), :timeout, this.delay)
     this |> alive(empty_set) |> crashed_union(newly_crashed) |> next()
  end # receive
end # next
```

DIY timer c.f. Elixir's Process.send_after() function

```
defmodule OurTimer do
  def start_timer(delay) do
      spawn(OurTimer, :start_timer/2, [ delay, self() ])
  end # start_timer/1

  defp start_timer(delay, caller) do
      Process.sleep(delay)
      send caller, :timeout end
  end # start_timer/2
end # OurTimer
```

PFD: Safety and Liveness

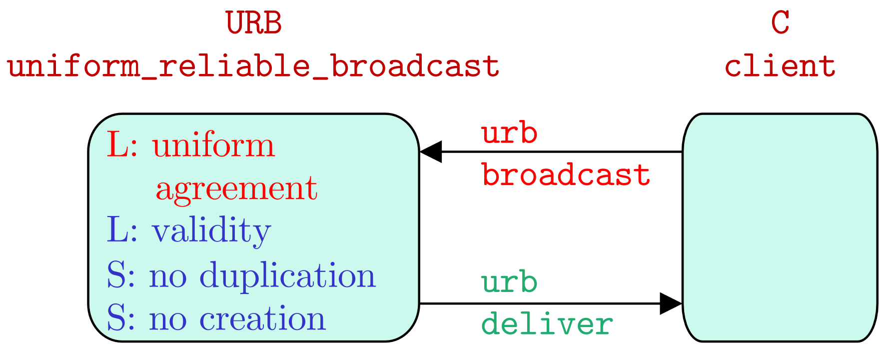
Strong completeness(L)

- *Every process that crashes will eventually be permanently suspected by every correct process*
- If a process crashes it stops replying to heartbeat messages, and no process will deliver its reply.
- PL ensures that no message is delivered unless sent.
- Every correct process will thus detect the crash.

Strong accuracy(S)

- *No process is suspected before it crashes*
- A process is suspected only if no heartbeat reply is delivered from it before the timeout.
- This can only happen if the process has crashed under our timing assumption – i.e. the reply is delivered before timeout

URB : Uniform Reliable Broadcast



Validity(L), No Duplication(S) and No Creation(S) properties are the same as *best effort broadcast* and *regular reliable broadcast*

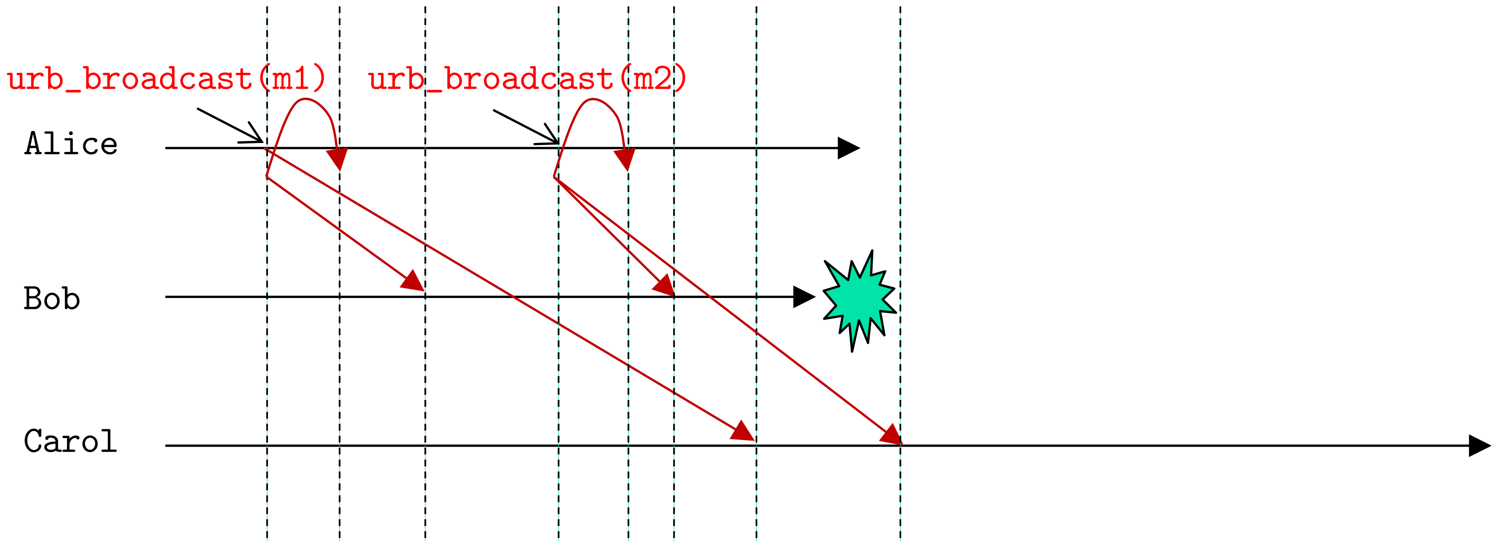
Uniform Agreement(L)

- If a ~~correct~~ process delivers message M then every correct process will also deliver M.
- Implies the set of messages delivered by a faulty process is always a subset of messages delivered by a correct process (stronger guarantee)

Why might uniform reliable broadcast be needed?

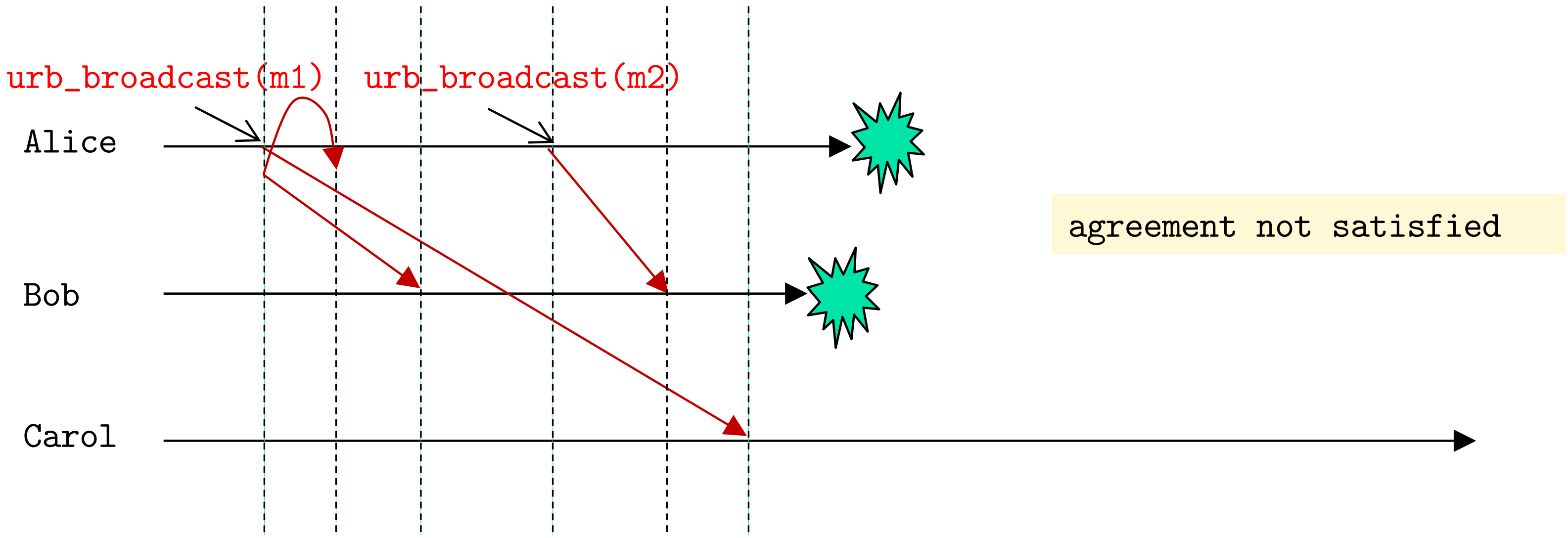
URB : Uniform reliable broadcast 1

- While *urb_broadcasting* m2, Bob crashes.
- m2 is successfully "urb_delivered" to Alice.C and to Carol.C.
- This satisfies the URB agreement property since m2 is "urb_delivered" to both correct processes.

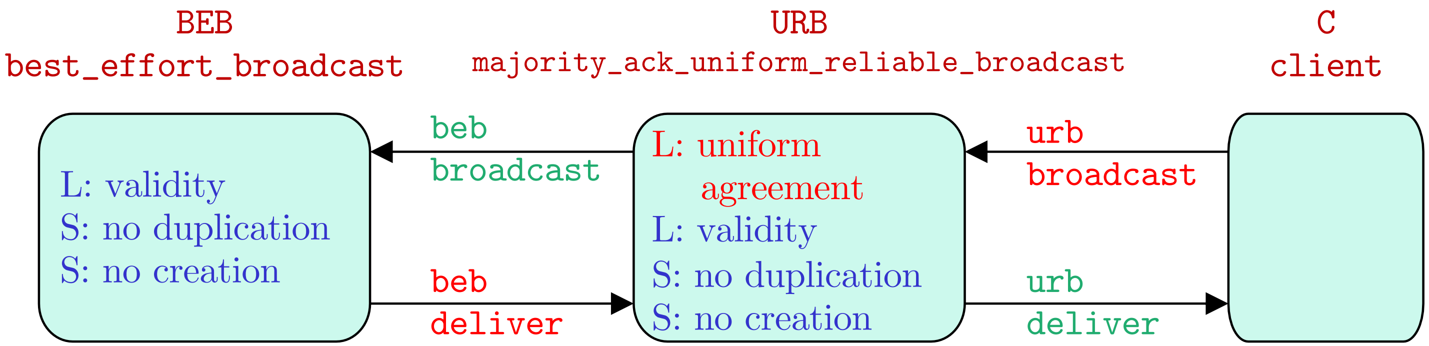


URB : Uniform reliable broadcast 2

- While *urb_broadcasting* m2, Alice and Bob crash
- m2 is successfully "urb_delivered" to Bob.C
- m2 was not "urb_delivered" to Carol.C.
- This does not satisfy the URB agreement property since m2 was not "urb_delivered" to Carol (a correct process) but was to Bob (a crashed process)



URB : Majority-Ack Uniform Reliable Broadcast



- **urb-deliver's** message only after the message has been **beb-deliv**ered by a majority (quorum) of correct processes.
- **Fail-silent algorithm** where process crashes are not reliably detected.
- Does not use a failure detector.
- Rather **assumes that the majority of processes are correct**. If *f* processes might crash then we need at least *2f+1* processes, i.e. we need to have a majority of at least *f+1* correct processes.

URB : Majority-Ack Uniform Reliable Broadcast 1 p84-85

```
defmodule URB do      # majority-ack urb

def start do
  receive do
    { :bind, c, beb, n_processes } ->
      %{ c: c, beb: beb, n_processes: n_processes, delivered: empty_set(), pending: empty_set(),
         bebd: %{}) } |> next()
    end # receive
end # start

defp next(this) do
  receive do
    { :urb_broadcast, msg } ->
       send this.beb, { :beb_broadcast, { :urb_data, nodeID(), msg } }
       send self(), :can_deliver    # asynchronously check if we can deliver any messages
       this |> pending_put({nodeID(), msg}) |> next()

    # continued on next slide
```

```
delivered - messages that been urb_delivered
pending   - messages that have been beb_broadcast but need to be urb-delivered
bebd      - foreach message, the set of processes that have beb-delivered it (seen it)
```

URB : Majority-Ack Uniform Reliable Broadcast 2

```
{ :beb_deliver, from, { :urb_data, sender, msg } = urb_m } ->
    Add 'from' to bebd[msg] i.e. processs 'from' has beb'd msg
    msg_pset = Map.get(this.bebd, msg, empty_set() )     # empty set if new msg
    this = this |> bebd.put(msg, MapSet.put(msg_pset, from))
    send self(), :can_deliver      # asynchronously check

    if { sender, msg } in this.pending do   # msg has already been beb-broadcast and is pending urb-delivery
      this |> next()
    else                                    # new msg so beb_broadcast once and add to pending
      send this.beb, { :beb_broadcast, urb_m }
      this |> pending_put({sender, msg }) |> next()
    end # if

% continued on next slide
```

URB : Majority-Ack Uniform Reliable Broadcast 3

```
  :can_deliver ->
```

Deliver pending messages to client only if not already delivered and only if message was beb_delivered by a majority of processes

```
    new_delivered_msgs =
      for { sender, msg } <- this.pending,
        msg not in this.delivered and MapSet.size(this.bebd[msg]) > this.n_processes div 2
        into: empty_set()
      do
        send this.c, { :urb_deliver, sender, msg }
        msg
      end # for

    this |> delivered_union(new_delivered_msgs) |> next()
  end  # receive
end # next

end # URB
```

URB : Majority-Ack Uniform Reliable Broadcast 1

Observe: For correct process P and message M

- If P `beb_delivers` M, then eventually P `urb_delivers` M (in `:can_deliver`)
- If P `beb_broadcasts` M then all correct processes `beb_broadcast` M (S2-L8)
- Given the majority assumption, P will eventually `beb_deliver` M from the majority of processes and will then `urb_deliver` M. (1)

Correctness

- *No creation(S)* follows from corresponding safety property of BEB.
- *No duplication(S)* because algorithm keeps track of messages that have been `urb_delivered` (S3-L10 and S3-L4)

Performance

- Best case: 2 steps.
- 1st step requires N messages. 2nd re-broadcast step requires N(N-1) messages, i.e. we need N+N(N-1) messages.

URB : Majority-Ack Uniform Reliable Broadcast 2

Observe: For correct process P and message M

- If P `beb_delivers` M, then eventually P `urb_delivers` M (in :`can_deliver`)
- If P `beb_broadcasts` M then all correct procesess `beb_broadcast` M
- Given the majority assumption, P will eventually `beb_deliver` M from the majority of processes and will then `urb_deliver` M. (1)

Correctness

Validity(L) because

- P `urb_broadcasts` M -> P `beb_broadcasts` M -> P eventually `beb_delivers` M -> P `urb_delivers` M using (1)

Uniform agreement(L) because if Q is any process that `urb_delivers` M

- Implies Q `beb_delivered` M from the majority of processes (which we assume is a correct majority) so at least 1 correct process must have `beb_broadcast` M.
- Hence all correct processes will eventually `beb_deliver` M (validity of BEB) and will also eventually `urb_deliver` M.

Halt

As distributed systems become more complex, there is often a need for a group of processes to communicate in a reliable manner.

One of the most useful abstractions for this is reliable broadcast. However, there are many variants, whose behaviours are interesting although quite subtle at times.

The choice of which variant is needed depends on the application, some can tolerate weak forms like regular reliable broadcast (or even, best-effort broadcast), others require stronger guarantees.

To be continued ☺

Reading: Chapter 3 of Cachin and Chapter 3 of Raynal.