60009

Distributed Algorithms

Imperial College London

Contents

Chapter 1

Introduction

1.1 Course Structure & Logistics

Dr Narankar Dulay

The module is taught by Dr Narankar Dulay.

Theory For weeks 2 $\rightarrow$ 10:

- Elixir (learning programming language)
- Introduction
- Reliable Broadcast
- FIFO, casual and total order Broadcast
- Consensus
- Flip Improbability Result
- Temporal Logic of Actions
- Modelling Broadcast
- Modelling Consensus

1.2 Course Resources

The course website contains all available slides and notes.

1.3 Distributed Systems

Distributed System — **Definition 1.3.1**

A set of processes connected by a network, communicating by message passing and with no shared physical clock.

- No total order on events by time (no shared clock)
- No shared memory.
- Network is logical - processes may be on the same OS process, same VM, same machine different machines communicating over a physical network.

Distributed systems must contend with the inherit uncertainty (failure, communication delay and an inconsistent view of the system's state) in communication between potentially physically independent processes (fallible machines, networks and software).

Leisle Lamport — ***Extra Fun!* 1.3.1**

A computer scientist and mathematician, credited with creating TLA (used on this course), as well as being the initial developer of latex (used for these notes).

> " There has been considerable debate over the years about what constitutes a distributed system. It would appear that the following definition has been adopted at SRC:
>
> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable. "

1.4 Distributed Algorithms

Liveness Properties — **Definition 1.4.1**

Something good happens eventually (Cannot be violated by finite computation)

Safety Properties — **Definition 1.4.2**

Nothing bad happens (Only violated by finite computations)

As liveness properties depend on computation, they can be constrained by a *fairness property*.

unconditional fairness	Every process gets its turn infinitely often.
strong fairness	Every process gets its turn infinitely often if it is enabled infinitely often.
weak fairness	Every process gets its turn infinitely often if it is continuously enabled from a particular point in the execution.

1.4.1 Key Aspects

1. **The problem** Specified in terms of the *safety* and *liveness* properties of the algorithm.
2. **Assumptions made**

Bounds on process delays	(timing assumption)
Types of process failures tolerated	(failure assumption)
Use of reliable message passing	(communication assumption)

3. **The algorithm** Expresses the solution to *the problem*, given *the assumptions*.
   - Must prove the algorithm is correct (satisfies all *safety* and *liveness* properties)
   - Time and space complexity of the algorithm

Mutual Exclusion Properties — **Example Question 1.4.1**

What are the safety, liveness and fairness properties required for mutual exclusion of processes over some critical section?

Safety	At most one process accesses the critical section.	$(s \| t) \wedge (s \neq t) \Rightarrow \neg(cs(s) \wedge cs(t))$
Liveness	Every request for the critical section is eventually granted.	$req(s) \Rightarrow (\exists t : s \preccurlyeq t \wedge cs(t))$
Fairness	Requests are granted in the order.	$req\_start(s) \wedge req\_start(t) \wedge (s \rightarrow t) \Rightarrow (next\_cs(s) \rightarrow next\_cs(t))$

Note that $\preccurlyeq$ is the *happens-before* relation.

Concensus — **Definition 1.4.3**

Processes Propose Values $\rightarrow$ Processes decide on value $\rightarrow$ Agreement Reached

Agreement Property	Two correct processes cannot decide on different values.
Validity Property	If all processes propose the same value, then the decided value is the proposed value.
Termination Property	System reaches agreement in finite time.

Consensus is impossible to solve for a fully asynchronous system, some timing assumptions are required.

It is difficult to prove the correctness of even simple distributed systems formally. By specifying an abstract model of an algorithm automatic model checkers can be used to verify properties.

1.4.2 Timing Assumptions

Asynchronous Systems — **Definition 1.4.4**

A system where process execution steps and inter-process communication take arbitrary time.

- No assumptions that processes have physical clocks.
- Sometimes useful to use *logical clocks* (used to capture a consistent ordering of events on a virtual timespan)

Synchronous Systems — **Definition 1.4.5**

A system containing assumptions on the upper bound timings for executing steps in a process.

- This means there are upper bounds for steps such as receiving messages, sending messages, arithmetic, etc.
- Easier to reason about.
- Implementation must ensure bounds are always met, this can potentially require very high bounds (so guarantee holds) which reduce performance. *Eventually synchronous models* were created to overcome this.

Eventually Synchronous Systems — **Definition 1.4.6**

Mostly synchronous systems. Do not have to *always* meet bounds, and can have periods of asynchronicity.

1.4.3 Failure Classes

Process Failure — **Definition 1.4.7**

A process internally fails and behaves incorrectly. Process sends messages it should not, or does not send messages it should.

- Can be caused by a software bug, termination of process by user or OS, OS failure, hardware failure, cyber attack by adversary.
- The process may be slowed down to the point it cannot send messages it needs to (or meet some timing assumption)

Fail-Stop	Failure can be reliably detected by other processes.
Fail-Silent	Not Fail-Stop.
Fail-Noisy	Failure can be detected, but takes time.
Fail-Recovery	Failing process can recover from failure.

A process that is not faulty is a **Correct Process**.

Link Failure — **Definition 1.4.8**

A link allowing for processes to communicate is disconnected and remains disconnected.

A network connecting machines hosting processes may become partitioned due to a *link failure*

Byzantine Failure — **Definition 1.4.9**

Also called **Fail-Arbitrary**, a process exhibits some arbitrary behaviour (can be malicious).

Omission Failure — **Definition 1.4.10**

Send Omission	Fails to send all messages required by the algorithm.
Send Omission	Fails to properly receive all messages required.

1.4.4 Communication Assumptions

Asynchronous Message Passing

Processes continue after sending messages, they do not wait for a message to be delivered. It is possible to build a synchronous message passing abstraction from asynchronous message passing.

Reliable Message Communication

Messages are assumed to be conveyed using a reliable medium.

- All sent messages are delivered.
- No duplicate messages are created.
- All delivered messages were sent.

Network failure is still a concern (breaks assumption), so TCP is used for messages, and more reliable message passing abstractions built on top.

Message delays are bounded, as a timeout is used.

1.4.5 Complexity

Complexity can be characterised using:

- Number of messages exchanged.
- Size of messages exchanged.
- Time taken from the perspective of an external observer, or some clock on a synchronous system.
- Memory, CPU time or energy used by processes.

Chapter 2

Elixir

2.1 learning Elixir

- **Introduction To Elixir & Installation**
- **Elixir Documentation** and **Standard Library**
- **Elixir Learning Resources**
- **Devhints Exlixir Cheatsheet**
- **Elixir Quick Reference**
- **Learn Elixir in Y Minutes**

Two Sum **Example Question 2.1.1**

Write a program to provide the two indexes of numbers in a list that sum to a given target. (This is the famous leetcode problem two sum).

```
defmodule Solution do
  @spec two_sum(nums :: [integer], target :: integer) :: [integer]
  def two_sum(nums, target) do
    nums
    |> Enum.with_index()
    |> Enum.reduce_while(%{}, fn {num, idx}, acc ->
      case Map.get(acc, target - num) do
        nil ->
          {:cont, Map.put(acc, num, idx)}
        val ->
          {:halt, [idx, val]}
      end
    end)
  end
end
```

We could also write this recursively with a helper function

```
defmodule Solution do
@spec two_sum(nums :: [integer], target :: integer) :: [integer]
  def two_sum(nums, target) do
      two_sum_aux(nums, target, %{}, 0)
  end

  defp two_sum_aux([next | rest], target, prevs, index) do
      val = Map.get(prevs, target - next)
      if val != nil do
          [val, index]
      else
          two_sum_aux(rest, target, Map.put(prevs, next, index), index + 1)
```

```
      end
    end
  end
```

Add two numbers **Example Question 2.1.2**

Given The following linked list structure, write a program taking two numbers (represented in reverse as linked lists), and produce a linked list of their sum. (This is leetcode problem add two numbers)

```
# Definition for singly-linked list.
defmodule ListNode do
  @type t :: %__MODULE__{
          val: integer,
          next: ListNode.t() | nil
        }
  defstruct val: 0, next: nil
end
```

```
defmodule Solution do
  @spec add_two_numbers(l1 :: ListNode.t | nil, l2 :: ListNode.t | nil) :: ListNode.t | nil
  def add_two_numbers(l1, l2) do
    x = get_list(l1) + get_list(l2)
    if x == 0 do
        %ListNode{val: 0, next: nil}
    else
        to_list(x)
    end
  end

  defp get_list(node) do
    case node do
        %ListNode{val: v, next: n} -> v + 10 * get_list(n)
        nil -> 0
    end
  end

  defp to_list(n) do
    case n do
        0 -> nil
        i -> %ListNode{val: rem(i,10), next: to_list(div(i,10))}
    end
  end
end
```

2.2 The Elixir System

Elixir **Definition 2.2.1**

A concurrent (with actors) and functional programming language used for fault tolerant distributed systems.

- A modernized successor language to Erlang
- Runs using BEAM (Erlang's virtual machine) and hence compatible with erlang
- Has many additions over erlang (protocols, streams and metaprogramming)

Elixir Processs — **Definition 2.2.2**

A lightweight user level thread (green threads) managed by the runtime.

- Everything is a process.
- Processes are strongly isolated, when two processes interact it does not matter which nodes, or even machines they run on.
- Processes share no resources (cannot share variables), they can only interact through message passing.
- Process creation and destruction is fast.
- Processes interact by message passing.
- Processes have unique names, if a name ios known it can be used to pass messages
- Error handling is non-local.
- Processes do what they are supposed to do or fail.

Elixir Node — **Definition 2.2.3**

All elixir processes run within a node, a node can manage many processes (creation, scheduling, and garbage collection).

- A node runs as an OS process, potentially with several OS threads scheduled across several cores.
- Multiple nodes can run on a single machine (or virtual machine such as a docker container).
- A node can efficiently manage thousands to millions of elixir processes.

Communication between processes is implemented through shared memory on the same machine and TCP when over a network. However processes are not exposed to this - the same primitives are used for inter and intra node/machine communication.

2.3 Message Passing

The `send` and `receive` statements are used for message passing.

```
# send somedata (any type) to process p
send p, somedata

# Wait until a message that matches the pattern is added to the message queue
# (or a timeout occurs), then remove it (potentially skipping over messages
# that do not match)
receive do
  somepattern -> dosomething(somepattern)
  # ... some other patterns
end
```

- Each process has its own message queue.
- Messages received are appended to the message queue of the receiving process.
- The sender does not wait for the message to be appended, it continues immediately after sending.

We can implement a basic client-server system in this way. Here we are using a component-based approach (split the program into components, each asynchronously message pass), by convention each component is an elixir module, modules can be instantiated in many processes & (by convention) have a public `start()` function.

```
defmodule Cluster do
  def start do
    # Spawn two processes, with the function start
    # Server.ex and Client.ex are modules containing a public start function
    # (Assuming we have tarted a client_node and server_node)
    s = Node.spawn(:'server_node@172.19.0.2', Server, :start, [])
    c = Node.spawn(:'client_node@172.19.0.1', Client, :start, [])
```

```
    # We send the PIDs of the processes to each other, we can pattern match on
    # atoms for convenience in receiving
    send s, { :bind, c }
    send c, { :bind, s }
  end
end
```

```
defmodule Server do
  def start do
    receive do
      { :bind, c } -> next(c)
    end
 end

  # next is defined as private, here
  # recursion is used for iteration.
  # To avoid a stack overflow tail
  # recursion is required
  defp next(c) do
    receive do
      { :circle, radius } ->
        send c, { :result, 3.14 * radius
                                * radius}
      { :square, side } ->
        send c, { :result, side * side}
    end
    next(c)
  end
end
```

```
defmodule Client do
  def start do
    receive do
      { :bind, s } -> next(s)
    end
  end

  defp next(s) do
    send s, { :circle, 1.0 }
    receive do
      { :result, area } ->
        IO.puts "Area is #{area}"
    end
    Process.sleep(1000)
    next(s)
  end
end
```

Chapter 3

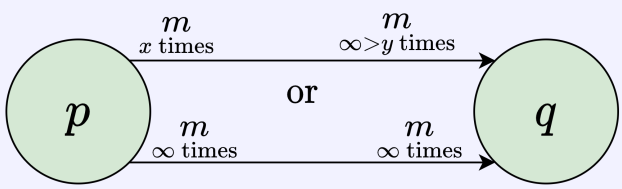
Reliable Broadcast

3.1 Links (unassessed)

A link is a mechanism defining how two processes may interact by sending and receiving messages, and what properties hold for message passing.

Fair Loss Link **Definition 3.1.1**

A weak link abstraction from which other links (e.g stubborn) can be built.



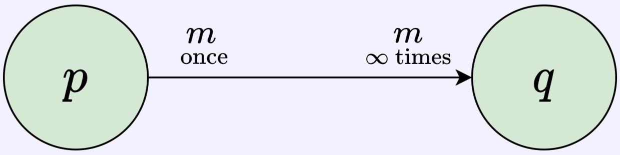
$\langle fll, Send \mid q, m\rangle$
Send m to process q

$\langle fll, Receive \mid p, m\rangle$
recieve m from process p

Fair-Loss	Liveness	Correct process p infinitely sends message m to correct process q $\Rightarrow$ q receives m from p infinitely many times.
Finite Duplication	Liveness	Correct process p sends message m a finite number of times to q $\Rightarrow$ m cannot be received infinitely many times from p.
No Creation	Safety	Some process q receives a message m with sender $p \Rightarrow p$ previously sent m to q.

Stubborn Link — Definition 3.1.2

A link guaranteeing messages are received infinitely many times.



$\langle sl, Send \mid q, m \rangle$
Send m to process q

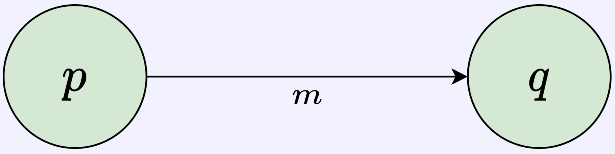
$\langle sl, Receive \mid p, m \rangle$
recieve m from process p

Stubborn Delivery	Liveness	Correct process p sends message m to correct process $q \Rightarrow q$ receives m from p infinitely many times.
No Creation	Safety	Some process q receives a message m with sender $p \Rightarrow p$ previously sent m to q.

No change in mind — Example Question 3.1.1

Implement stubborn links with elixir using the fair loss link.

Perfect Point-to-Point Link — Definition 3.1.3



$\langle pl, Send \mid q, m \rangle$
Send m to process q

$\langle pl, Receive \mid p, m \rangle$
recieve m from process p

- Also called *reliable message passing*

Reliable Delivery	Liveness	Correct process p sends m to correct process $q \Rightarrow q$ will eventually receive m.
No Duplication	Safety	No message is received by a process more than once.
No Creation	Safety	Some process q receives a message m with sender $p \Rightarrow p$ previously sent m to q.

3.2 Failure Detection

A failure detector provides a process with a list of *suspected processes*.

- Failure detectors make, and encapsulate some timing assumptions in order to determine which processes are suspect.
- They are not fully accurate, and their specification allows for this.

Perfect Failure Detector — **Definition 3.2.1**

A failure detector that is never incorrect / is entirely accurate.

- Never changes its view on failure → once detected as crashed it cannot be *unsuspected*.
- Often represented as $\mathcal{P}$

Strong Completeness	Liveness	Eventually every process that crashes is permanently detected as crashed by every correct process.
Strong Accuracy	Safety	p detected $\Rightarrow$ p has crashed. No process is suspected before it crashed.

We can implement a failure detector using timeouts and a heartbeat.

- Perfect links used to send requests for heartbeat.
- If reply is not received before timeout, the process is suspected to have crashed.
- **perfect links** are only reliable for correct processes.
- Timeout period has to be long enough to send the heartbeat to all processes and for the receiving processes to respond.

```
#
defmodule Perfect_Failure_Detector do
  def start do
    receive do
      { :bind, c, pl, processes, delay } ->
        # Send the first heartbeat request
        heartbeat_requests(delay)

        next(c, pl, processes, delay, processes, MapSet.new())
    end
  end

  defp next(c, pl, processes, delay, alive, crashed) do
    receive do
      # Send heartbeat requests over perfect link
      { :pl_deliver, from, :heartbeat_request } ->
        send pl, { :pl_send, from, :heartbeat_reply }
        next(c, pl, processes, delay, alive, crashed)

      # Receive heartbeat responses over perfect links
      { :pl_deliver, from, :heartbeat_reply } ->
        next(c, pl, processes, delay, MapSet.put(alive, from), crashed)

      # Timeout period expired
      # 1. Get all previously alive processes that did not respond (these have crashed)
      # 2. Send crashed to each
      :timeout ->
        newly_crashed =
          for p <- processes, p not in alive and p not in crashed, into: MapSet.new do p end

        # Inform process p of all newly crashed processes
        for p <- newly_crashed do send c, { :pfd_crash, p } end

        # Send new heartbeat requests over perfect links
        for p <- alive do send pl, { :pl_send, p, :heartbeat_request } end

        heartbeat_requests(delay)

        # Loop (empty set of alive, union set of old and newly crashed)
        next(c, pl, processes, delay, MapSet.new(), Mapset.union(crashed, newly_crashed))
```

```
    end
  end

  defp heartbeat_requests(delay) do
    # after delay milliseconds, timeout will be received by this process
    Process.send_after(self(), :timeout, delay)
  end
end
```

This implementation meets the properties of a *perfect failure detector* as:

Strong Completeness	If a process crashes it will no longer reply to heartbeat messages, hence by *perfect links* **no-creation** property, no correct process will receive a heartbeat. So every correct process will detect a crash.
Strong Accuracy	A process can only miss the timeout if it has crashed under out timing assumption.

Eventually Perfect Failure Detector — **Definition 3.2.2**

A failure detector that is not entirely accurate.

- Can restore processes (no longer suspected).
- Often represented as $\Diamond\mathcal{P}$

Strong Completeness	Liveness	Eventually every process that crashes is permanently detected as crashed by every correct process.
Eventual Strong Accuracy	Liveness	Eventually no correct process is suspected by any other correct process

3.3 Best Effort Broadcast

Best Effort Broadcast — **Definition 3.3.1**

A non-reliable, single-shot broadcast.

- Only reliable if the broadcasting process is correct during broadcast (if crashing during broadcast only some messages may be delivered, and processes may disagree on delivery)
- No delivery agreement guarantee (correct processes may disagree on delivery)
- Uses *Perfect Point-to-Point Link* and inherits properties from it.

Validity	Liveness	If a correct process broadcasts a message then every correct process eventually receives it.
No Duplication	Safety	No message is received by a process more than once.
No Creation	Safety	No broadcast is delivered unless it was broadcast.

We can implement this in elixir using the send and receive primitives as *Perfect Point-to-Point Link*.

```
# Broadcast using perfect point-to-point links
# processes <- the list of processes in the broadcast space
# pl        <- the perfect links process to use
# c         <- the object broadcasting & being delivered
defmodule Best_Effort_Broadcast do
  def start(processes) do
    receive do {:bind, pl, c} -> next(processes, pl, c)
  end

  defp next(processes, pl, c) do
    receive do
      {:beb_broadcast, msg} ->
```

```
    for dest <- processes do
      send pl, {:pl_send, dest, msg}
    end
   {:pl_deliver, src, msg} ->
    send c, {:beb_deliver, src, msg}
  end
  next (processes, pl, c)
 end
end
```

3.4 Reliable Broadcast

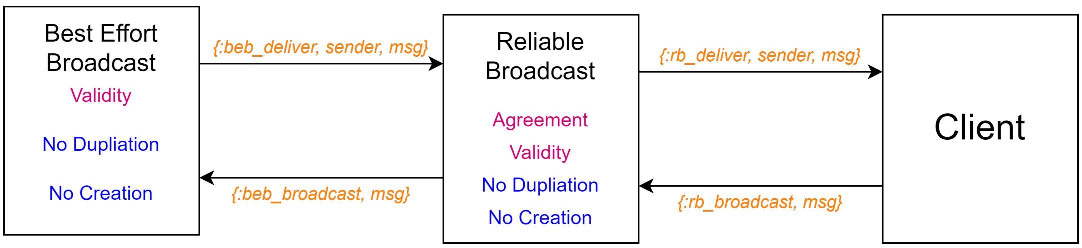
Reliable Broadcast **Definition 3.4.1**

Adds a delivery guarantee to *best effort broadcast*

Agreement	Liveness	If a correct process delivers message m then all correct processes deliver m

All Properties from Best Effort Broadcast

The combination of **Validity** and **Agreement** form a *termination property* (system reaches agreement in finite time).



3.4.1 Eagre Reliable Broadcast

Eagre Reliable Broadcast **Definition 3.4.2**

A *reliable broadcast* where every process re-broadcasts every message it delivers.

- If the broadcasting process crashes, and only some correct processes deliver the message, then re-broadcast ensures eventually all will receive.
- This broadcast is *fail-silent*
- Very inefficient to implement, broadcast to n processes results in $O(n^2)$ messages.
- **Validity** property combined with retransmission provides **agreement**.

All Properties from Reliable Broadcast

```
# Eagre reliable broadcast implemented using Best Effort Broadcast
# beb    <- the best effort broadcast process
# client <- the object broadcasting & being delivered
defmodule Eagre_Reliable_Broadcast do

 def start do
  receive do { :bind, client, beb } -> next(client, beb, MapSet.new) end
 end

 defp next(client, beb, delivered) do
  receive do
   { :rb_broadcast, msg } ->
    send beb, { :beb_broadcast, { :rb_data, our_id(), msg } }
```

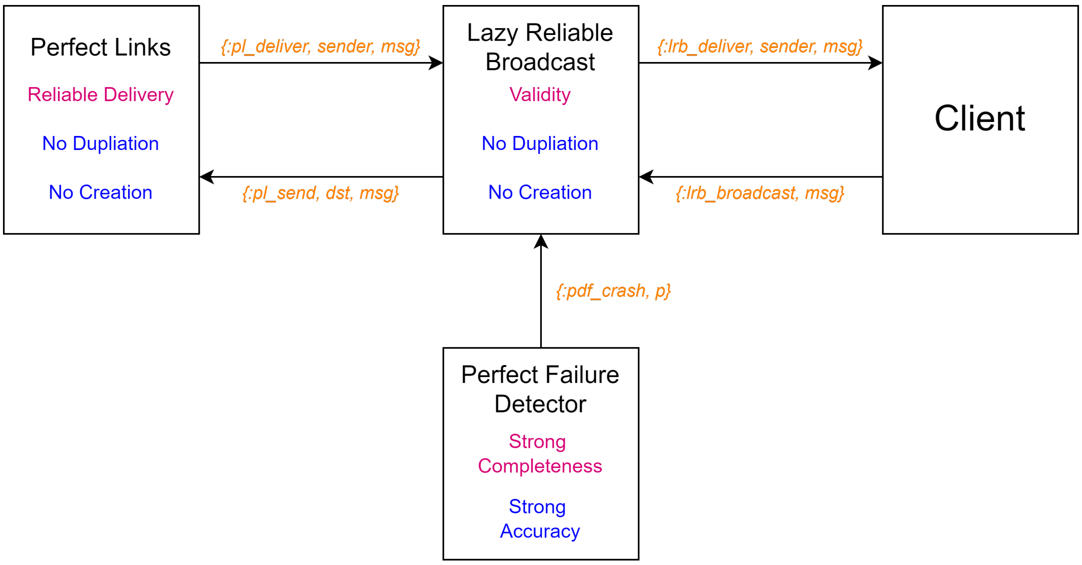
```
      next(client, beb, delivered)
    { :beb_deliver, from, { :rb_data, sender, msg } = rb_m } ->
      if msg in delivered do
        # Message was already delivered, so can be ignored
        next(client, beb, delivered)
      else
        # Message is new, so add to delivered, deliver to c & rebroadcast
        send client, { :rb_deliver, sender, msg }
        send beb, { :beb_broadcast, rb_m }
        next(client, beb, MapSet.put(delivered, msg))
      end
    end
  end

end
```

3.4.2 Lazy Reliable Broadcast

Lazy Reliable Broadcast **Definition 3.4.3**

A *reliable broadcast* using *Best Effort Broadcast* with a *Failure Detector* to enforce agreement.

- Uses a *perfect failure detector.*
- When a process is detected to have crashed, all broadcasts delivered from the process are rebroadcasted
- Agreement is derived from the **validity** of *best effort broadcast*, that every correct process broadcasts every message delivered from a crashed process and the properties of the *perfect failure detector.*

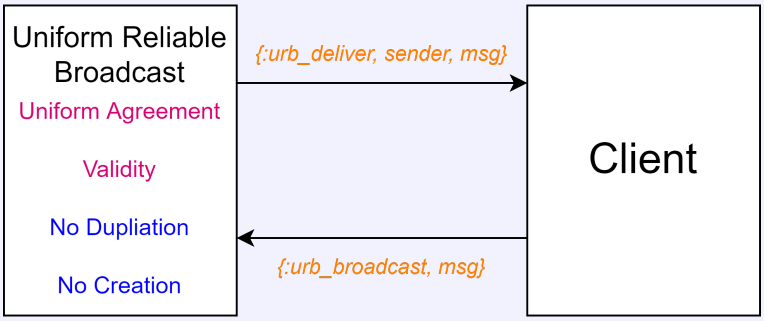



```
# Lazy Reliable Broadcast implemented using best effort broadcast
# beb    <- the best effort broadcast process
# client <- the object broadcasting & being delivered
defmodule Lazy_Reliable_Broadcast do
  def start do
    receive do
      { :bind, processes, client, beb } ->
        delivered = Map.new(processes, fn p -> {p, MapSet.new} end)
        next(client, beb, processes, delivered)
    end
  end

  defp next(client, beb, correct, delivered) do
    receive do
```

```
      { :rb_broadcast, msg } ->
        # broadcast a message with our id
        send beb, { :beb_broadcast, { :rb_data, our_id(), msg } }
        next(client, beb, correct, delivered)

      { :pfd_crash, crashedP } ->
        # Failure detector has detected a crashed process
        # For each message delivered by the crashed process,
        # rebroadcast (from them)
        for msg <- delivered[crashedP] do
          send beb, { :beb_broadcast, { :rb_data, CrashedP, msg } }
        end
        next(c, beb, MapSet.delete(correct, crashedP), delivered) # cont

      { :beb_deliver, from, { :rb_data, sender, msg } = rb_m } ->
        # A message is delivered, if already received do nothing,
        # otherwise record the delivered message,
        if msg in delivered[sender] do
          next(c, beb, correct, delivered)
        else
          send c, { :rb_deliver, sender, msg }
          # add msg to the set of messages received from sender
          sender_msgs = MapSet.put(delivered[sender], msg)
          delivered = Map.put(delivered, sender, sender_msgs)

          # Due to transmission delay, the sender may have crashed
          # before this message is delivered, so we must check rebroadcast
          # if this is the case.
          if sender not in correct do
            send beb, { :beb_broadcast, rb_m }
          end

          next(c, beb, correct, delivered)
      end
    end
  end

end
```

3.5 Uniform Reliable Broadcast

Uniform Reliable Broadcast **Definition 3.5.1**



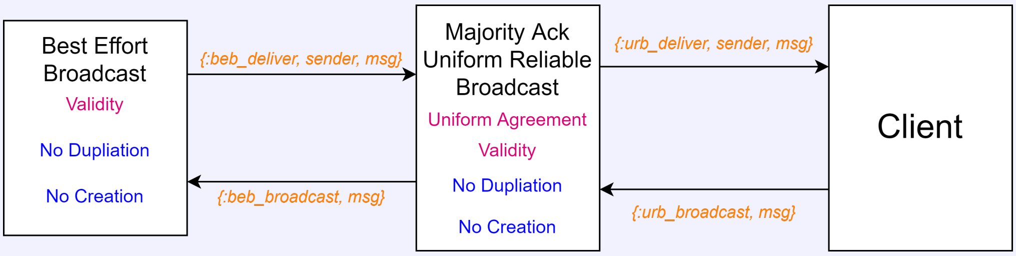
Uniform Agreement Liveness If a non-correct process delivers message m then all correct processes deliver m

All Properties from Best Effort Broadcast

- Implies that faulty processes deliver a subset of messages delivered to correct processes (stronger than **agreement** - only for correct processes).
- Avoids any scenario where a crashed process broadcasts and only a crashed process delivers (correct processes miss message).

Majority Ack Uniform Reliable Broadcast **Definition 3.5.2**

A *uniform reliable broadcast* implementation that assumes a majority of processes are correct.



- *Fail-silent* and does not use a *failure detector*.
- If n processes may crash, then $2n+1$ processes are needed with at least $n+1$ (majority) being correct

UNFINISHED!!!

Chapter 4

Credit

Image Credit

Content

Based on the distributed algorithms course taught by Prof Naranka Dulay.

These notes were written by Oliver Killane.