# Distributed Systems

Eddy Caron*
ENS de Lyon

## Contents



*hadriencroubois.com

**Final grade** 2/3 Final exam + 1/3 mid-term exam + 1/3 (project + partial)

We will use the programming language *Erlang* (1987 $\rightsquigarrow$ 1998)
- Concurrent, real time, distributed

$\Rightarrow$ Use BEAM

Variables can be integers, float, PID, functions, tuples, maps, ... and atoms (specific to Erlang).
There are built-in functions for message passing
Lists are not strongly typed, tuples are like python's
In Erlang, there is no overwriting of the variables (cannot affect them a new value).

# 1 Algorithm for Distributed System

## 1.1 Modelization

**Definition 1** (Transition relation)**.** *Let $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ be a transition system. An execution of $S$ is a* maximal sequence $E = (\gamma_0, \gamma_1, ..., \gamma_n)$

**Definition 2** (Terminal configuration)**.** *A* terminal configuration *is a configuration $\gamma$ for which there is no $\delta$ such that $\gamma \rightarrow \delta$. Note that a sequence $E = (\gamma_0, \gamma_1, \gamma_2, ...)$*

$\rightarrow$ this notion is very tricky in parallel programming, as it is difficult to know whether all the executions are finished or not.

**Definition 3.** *A configuration $\delta$ is* reachable *from $\delta$ (notation $\delta \rightsquigarrow \gamma$), if there exist a sequence $\gamma = \gamma_0, \gamma_1, \gamma_2, ..., \gamma_k$ with $\gamma_i \rightarrow \gamma_{i+1}$ for all $0 \leq i < k$. Configuration $\delta$ is reachable if it is reachable from an initial configuration.*

**Definition 4** (System with Asynchronous Message Passing)**.** *Cf APPD course*

## 1.2 Fairness

**Definition 5.** *An execution is* weakly fair *if no event is applicable in infinitely many consecutive configurations without occurring in the execution*

**Definition 6.** *An execution is* strongly fair *if no event is applicable in infinitely many configuration without occurring in the execution.*

## 1.3 Network topology

Many topology exists, including :

- Ring
- Tree
- Hypercube
- Star
- Clique

You can "change the topology" of your network to adapt and algorithm, for example taking a star sub-graph of a clique.

### 1.4 How to write a distributed algorithm?

- A specific code for a specific node (or family of node)
  - The sender code and the receiver code
  - The initial code and the non-initial code
- One code for all
  - The same code is executed on each node
  - A requirement can switch to the right code for a node

Use label to separated the code between initiators, non-initiator or reception of a specific message (for example stopping the execution).

**Warning** The execution flow is not clear: there is no assumption about the label selected for the execution (it is randomly chosen), so the algorithm must run for all the chosen labels for al the processors in the current state.

It is useful to try to find an incorrect workflow to test whether the algorithm is correct.

No assumptions must be made on the execution time (especially linking synchronization with the size of the code)!

## 2 Communication protocols

### 2.1 Sliding window communication protocol

**Example** Write an algorithm to exchange informations between both process

```
1  Var:
2  data_in=N;
3  data_recv = −1;
4  is_sent = -1;
5  label 1{is_sent==1}
6  |  send ⟨msg,data_in⟩ is_sent=0;
7  end
8  label 2{receive ⟨msg,i⟩}
9  |  data_recv=i;
10 end
```

The number of exchanged message can be a good measure of the performance of the algorithm.

```
  // Now, we want to send an array of data:
1 Var:
2 data_in=N;
3 data_recv = −1;
4 j=0;
5 i=0;
6 label 1{is_sent[i]==1}
7 |  send ⟨msg,data_in[i]⟩;
8 |  is_sent[i++]=0;
9 end
10 label 2{receive ⟨msg,v⟩}
11 |  data_recv[j++]=v;
12 end
```

```
  // Now, we had a procedure lost that erase messages:
1 label Lost_n={⟨msg⟩}
2 |  kill(msg)
3 end
```

```
  // We can do it with using 0 acknowledgements, by using the data you need to send as
     an acknowledgement:
1 Var:
2 s_p is the number of words received by p from q (without missing ones);
3 l_p, l_q is the number of ahead values → size of the sliding window;
4 label S_p:{a_p ≤ i < s_p + l_p}
5 |  send ⟨pack,in_p[i], i⟩ to q;
6 end
7 label R_p:{⟨pack,w, i⟩∈ Q_p}
8 |  receive ⟨pack, w, i⟩;
9 |  if out_p[i] == −1 then
10 |  |  out_p[i] = w;
11 |  |  a_p = max{a_p, i − l_q + 1};
12 |  |  s_p = min_j{out_p[j] = −1};
13 |  else
14 |  |  ignore;
15 |  end
16 end
17 label L_p:{⟨pack, w, i⟩∈ Q_p
18 |  Q_p = Q_p\⟨pack, w, i⟩;
19 end
```

### 2.2 Timer-based protocol

Create channel, and see whether it is opened or not (See slides for details).

# 3 Wave algorithm

## 3.1 Wave algorithm

We need to solve broadcast, global synchronisation, trigger events, parallel computing, data-parallelism.
For that, we use message passing.
Requirements:

- Termination
- Decision
- Dependence

List of aspects:

- Centralized: one initiator
- Decentralized: $n$ initiators
- Network: topology, undirected links, connected, *Asynchronous* on *Synchronous*

Initial knowledge:

- Name
- Name of its neighbour

Most of these algorithm send "empty messages" (simple token).

The ring algorithm is a wave algorithm
On a tree:

```
   // We assume that the initiator is not a leaf
 1 label Initiator:
 2 |   Send ⟨tok⟩ to the children;
 3 |   Receive ⟨tok⟩ from the children;
 4 |   Decide;
 5 end
 6 label Non-Initiator:
 7 |   Receive ⟨tok⟩;
 8 |   Send to all the children ⟨tok⟩;
 9 |   Receive from all the children ⟨tok⟩;
10 |   Send ⟨tok⟩ to the father;
11 end
```