

Distributed Systems

Eddy Caron*

ENS de Lyon

Contents

1	Algorithm for Distributed System	2
1.1	Modelization	2
1.2	Fairness	2
1.3	Network topology	2
1.4	How to write a distributed algorithm?	3
2	Communication protocols	3
2.1	Sliding window communication protocol	3
2.2	Timer-based protocol	5
3	Wave algorithm	5
3.1	Wave algorithm	5
3.2	The tree algorithm	6
3.3	The echo algorithm	7
3.4	The Polling Algorithm	8
3.5	The Traversal algorithm	8
3.6	Tarry's Graph Traversal Algorithm	8
4	Fault-tolerant system	8
5	An application: A peer-to-peer prefix tree	10
6	More Ph.D	10
7	Bully algorithm	10
8	Chang and Roberts Algorithm	12
9	I.Suzuki and T.Kasami's algorithm	12

*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 $E = (\gamma_0, \gamma_1, \dots, \gamma_n)$

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

\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 δ is reachable from γ (notation $\delta \rightsquigarrow \gamma$), if there exist a sequence $\gamma = \gamma_0, \gamma_1, \gamma_2, \dots, \gamma_k$ with $\gamma_i \rightarrow \gamma_{i+1}$ for all $0 \leq i < k$. Configuration δ 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  $\langle msg, data_{in} \rangle$   $is\_sent=0$ ;  
7 end  
8 label 2{receive  $\langle msg, i \rangle$ }  
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  datain=N;
3  datarecv = -1;
4  j=0;
5  i=0;
6  label 1{is_sent[i]==1}
7    | send ⟨msg,datain[i]⟩;
8    | is_sent[i++]=0;
9  end
10 label 2{receive ⟨msg,v⟩}
11 | datarecv[j++] = v;
12 end

```

```

    // Now, we had a procedure lost that erase messages:
1  label Lostn=⟨{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  sp is the number of words received by p from q (without missing ones);
3  lp, lq is the number of ahead values → size of the sliding window;
4  label Sp:{ap ≤ i < sp + lp}
5  | send ⟨pack,inp[i],i⟩ to q;
6  end
7  label Rp:{⟨pack,w,i⟩ ∈ Qp}
8  | receive ⟨pack, w, i⟩;
9  | if outp[i] == -1 then
10 | | outp[i] = w;
11 | | ap = max{ap, i - lq + 1};
12 | | sp = minj{outp[j] = -1};
13 | else
14 | | ignore;
15 | end
16 end
17 label Lp:{⟨pack, w, i⟩ ∈ Qp}
18 | Qp = Qp \ ⟨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.

3.2 The tree algorithm

On a tree:

- All leaves initiate the algorithm
- Each process sends exactly one message

```
1  $rec_p \leftarrow 0$  // # of received message
2  $father_p \leftarrow \text{undefined}$ ;
3 begin Initiator
4   for all  $q \in \text{Neighbourhood}(p)$  do
5     | send  $\langle \text{tok} \rangle$  to  $q$ ;
6   end
7   while  $rec_p < \# \text{Neighbourhood}(p)$  do
8     | receive  $\langle \text{tok} \rangle$ ;
9     |  $rec_p \leftarrow rec_p + 1$ ;
10  end
11  decide;
12 end
13 begin Non-initiators
14   receive  $\langle \text{tok} \rangle$  from some neighbour  $q$ ;
15    $father_p \leftarrow q$ ;
16    $rec_p \leftarrow rec_p + 1$ ;
17   for all  $q \in \text{Neighbourhood}(p) \setminus \{father_p\}$  do
18     | send  $\langle \text{tok} \rangle$  to  $q$ ;
19   end
20   while  $rec_p < \# \text{Neighbourhood}(p)$  do
21     | receive  $\langle \text{tok} \rangle$ ;
22     |  $rec_p \leftarrow rec_p + 1$ ;
23   end
24   send  $\langle \text{tok} \rangle$  to  $father_p$ 
25 end
```

3.3 The echo algorithm

```
1  $rec_p \leftarrow 0$  // # of received message
2  $father_p \leftarrow \text{undefined}$ ;
3 begin Initiator
4   for all  $q \in \text{Neighbourhood}(p)$  do
5     send  $\langle \text{tok} \rangle$  to  $q$ ;
6   end
7   while  $rec_p < \# \text{Neighbourhood}(p)$  do
8     receive  $\langle \text{tok} \rangle$ ;
9      $rec_p \leftarrow rec_p + 1$ ;
10  end
11  decide;
12 end
13 begin Non-initiators
14   receive  $\langle \text{tok} \rangle$  from some neighbour  $q$ ;
15    $father_p \leftarrow q$ ;
16    $rec_p \leftarrow rec_p + 1$ ;
17   for all  $q \in \text{Neighbourhood}(p) \setminus \{father_p\}$  do
18     send  $\langle \text{tok} \rangle$  to  $q$ ;
19   end
20   while  $rec_p < \# \text{Neighbourhood}(p)$  do
21     receive  $\langle \text{tok} \rangle$ ;
22      $rec_p \leftarrow rec_p + 1$ ;
23   end
24   send  $\langle \text{tok} \rangle$  to  $father_p$ ;
25 end
```

3.4 The Polling Algorithm

Cf slides

3.5 The Traversal algorithm

They are a subset of wave algorithms. All events are totally ordered by the causality relation, and the last event occurs in the same process as the first event.

Definition 7 (f-traversal algorithm). *An algorithm is a f-traversal algorithm for a class of network if:*

- *It is a traversal algorithm*
- *In each computation,*

Example The ring algorithm is a traversal algorithm.

3.6 Tarry's Graph Traversal Algorithm

We start from a connected graph, and apply two rules:

- A process never use the same communication channel twice
- A process will send to its father a message only if there is no other choice in respect to the first rule

// TODO

4 Fault-tolerant system

There are different types of faults:

- Fail-stop: some processes stop and terminated unexpectedly
- Crash: Freeze, node unavailable
- Oversight: Some messages are lost, some information are missing
- Byzantine: No idea of what can occur: sometimes a fault, a wrong message. Unpredictable fault in your system.

They have different durations, degree of detection and degree of correction.

One can implement an algorithm tolerant to a given fault class, where a specific fault is not a “bad” action.

Two parameters are important for our algorithms: *safety* and *liveness*.

Safety E sends a message if and only if the previous sent message was received by R and if each message sent by R has been received.

Liveness Every message sent by E will be received (one day).

Example In synchronous mode:


```

1 label Sender
2   label init
3     get(m);
4     send(m);
5   end
6   label rcv(ack
7     )
8   end
9     get(m);
10    send(m);
11 end
12 label Receiver
13   label rcv(m
14     )
15   end
16     dlv(m);
17     send(ack);
18 end

```

```

// With correction
1 label Sender
2   label init
3     get(m);
4     send(seq,m);
5   end
6   label rcv(ack) and (ack=seq)
7     seq=seq  $\oplus$  1;
8     get(m);
9     send(seq,m);
10  end
11  label rcv(ack) and (ack $\neq$ seq)
12    "nothing";
13  end
14 end
15 label Receiver
16   label rcv(seq,m) and (ack $\neq$ seq)
17     dlv(m);
18     ack:=seq;
19     send(seq);
20   end
21   label rcv(seq,m) and (ack=seq)
22     send(seq);
23   end
24 end

```

Two Generals' problem Two army are fighting, the blue and the red. The red are on both side, and they need to coordinates to attack at the same time, but the messengers must pass inside the blue army.
⇒ there is no solution!

Proof. If we consider the most efficient protocol, and we kill the last messenger. Then it should still work, so we get a better protocol: contradiction! □

Fundamental limits

- algorithm requires an infinity of messages
- we never attack

We will thus consider than at least one copy of each message arrives.

Theorem 1 (Fisher, Lynch and Paterson). *[1985] Consensus is impossible with an asynchronous network if a process can stop.*

Self-stabilization

- Wrong messages in channels
- Wrong values in variables

5 An application: A peer-to-peer prefix tree

Work from the Ph.D of Cédric Tedeschi.

- Based on a prefix tree for the indexing system.
- Using DHT and Clustering, build a ring between peers.
- The load balancing is handled by a heuristic: *Max Local Throughput*
- Fault tolerance via the *snap-stabilizing protocol*

6 More Ph.D

See slides (self-stabilisation of DIET and termination detection).

7 Bully algorithm

Efficiency $O(n^2)$ in worst case → costly

Problem It is complicated to find the best time-out

```

1 begin Init:
2   for all  $i$  do
3      $\text{ask\_elect}_i = \text{false};$ 
4      $\text{coordinator}_i = \text{max};$ 
5   end
6 end
7 begin R1:
8   // I want to be a leader
9    $\text{ask\_elect}_i = \text{true};$ 
10  for all  $j > i$  do
11     $\text{send}(p_j, \text{ELECT});$ 
12     $\text{start\_timeout}(T);$ 
13  end
14 begin R2:
15    $\text{receive}(p_j, \text{ELECT});$ 
16   if not( $\text{ask\_elect}_i$ ) then
17     for  $k > i$  do
18        $\text{send}(p_k, \text{ELECT});$ 
19     end
20   end
21    $\text{send}(p_j, \text{ACK});$ 
22 end
23 begin R3:
24    $\text{receive ACK};$ 
25   wait for receive ELECTED from  $p$ ;
26 end
27 begin R4:
28    $\text{receive}(p_j, \text{ELECTED});$ 
29    $\text{ask\_elect}_i = \text{false};$ 
30    $\text{coordinator}_i = j;$ 
31 end
32 Function  $\text{start\_timeout}(T):$ 
33    $\text{receive}(p_j, \text{ACK});$ 
34    $\text{send}(\text{all}, \text{ELECTED});$ 

```

8 Chang and Roberts Algorithm

To reduce the number of messages, we can use a topology: a virtual *ring*. Multiple elections can be in progress simultaneously.

```
1 begin Candidat
2   | candidati=true;
3   | send(succ[i],ELECT,i);
4 end
5 begin R1: receive {pi, ELECT, j}
6   | switch i, j do
7     | label j > i
8     |   | send(succ[i], ELECT, i);
9     | end
10    | label j < i
11    |   | if not(candidati then
12    |     |   | candidati=true send(succ[i], ELECT, i)
13    |     | end
14    |   | end
15    |   | label j = i
16    |     | send(all, ELECTED, i)
17    |   | end
18  | end
19 end
```

Complexity $O(n \log n)$ in average, $O(n^2)$ in worst case

9 I.Suzuki and T.Kasami's algorithm

```

1 begin Init
2   RN[i] = number of the last request received from  $p_i$  to get the token;
3   LN[i] = number of the last critical section (CS) for  $p_i$ ;
4   queue = request queue for unsatisfied request for the token
5 end
6 begin Request for CS
7   // on  $p_i$ 
8   if not token_is_here then
9     RN[i]++;
10    broadcast(REQ, i RN[i]);
11    wait(token_is_here);
12    cs_inprogress = true;
13    enter in CS;
14 end
15 begin Exit of CS
16   cs_inprogress = false;
17   LN[i] = RN[i] for all  $i$  s.t.  $p_j \notin$  queue do
18     if  $RN[j] == LN[j] + 1$  then
19       enter_queue(j);
20     end
21     if not(empty_queue) then
22       k = exit_queue();
23       send (k, TOKEN);
24     end
25   end
26 end
27 begin receive { REQ, j, n}
28   RN[j] = max ( RN[j], n);
29   if token_is_here and not cs_inprogress then
30     if  $RN[j] == LN[j] + 1$  then
31       send(j, TOKEN);
32       token_is_here = false;
33     end
34   end
35 end
36 begin receive TOKEN
37   token_is_here = true;
38   cs_inprogress = true;
39   enter in CS;
40 end

```