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 γ for which there is no δ 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 δ is* reachable *from δ (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 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

```
Var:
data_in=N;
data_recv = −1;
is_sent = -1;
label 1{is_sent==1}
    send ⟨msg,data_in⟩ is_sent=0;
end
label 2{receive ⟨msg,i⟩}
    data_recv=i;
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:
Var:
data_in=N;
data_recv = −1;
j=0;
i=0;
label 1{is_sent[i]==1}
    send ⟨msg,data_in[i]⟩;
    is_sent[i++]=0;
end
label 2{receive ⟨msg,v⟩}
    data_recv[j++]=v;
end
```

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

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

```
rec_p ← 0 // # of received message
father_p ← undefined;
begin Initiator
    for all q ∈ Neighbourhood(p) do
        send ⟨tok⟩ to q;
    end
    while  rec_p < #Neighbourhood(p) do
        receive ⟨tok⟩;
        rec_p ← rec_p + 1;
    end
    decide;
end
begin Non-initiators
    receive ⟨tok⟩ from some neighbour q;
    father_p ← q;
    rec_p ← rec_p + 1;
    for all q ∈ Neighbourhood(p) ∖ {father_p} do
        send ⟨tok⟩ to q;
    end
    while rec_p < #Neighbourhood(p) do
        receive ⟨tok⟩;
        rec_p ← rec_p + 1;
    end
    send ⟨tok⟩ to father_p
end
```

3.3 The echo algorithm

```
rec_p ← 0 // # of received message
father_p ← undefined;
begin Initiator
    for all q ∈ Neighbourhood(p) do
        send ⟨tok⟩ to q;
    end
    while rec_p < #Neighbourhood(p) do
        receive⟨tok⟩;
        rec_p ← rec_p + 1;
    end
    decide;
end
begin Non-initiators
    receive ⟨tok⟩ from some neighbour q;
    father_p ← q;
    rec_p ← rec_p + 1;
    for all q ∈ Neighbourhood(p) ∖ {father_p} do
        send ⟨tok⟩ to q;
    end
    while rec_p < #Neighbourhood(p) do
        receive ⟨tok⟩;
        rec_p ← rec_p + 1;
    end
    send ⟨tok⟩ to father_p;
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 cass 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 unexpectingly
- 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. Unpredictible 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 id 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 send by R has been received.

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

Example In synchronous mode:

```
label Sender
    label init
        get(m);
        send(m);
    end
    label rcv(ack
        )
    end
    get(m);
    send(m);
end
label Receiver
    label rcv(m
        )
    end
    dlv(m);
    send(ack);
end
```

```
// With correction
label Sender
    label init
        get(m);
        send(seq,m);
    end
    label recv(ack) and (ack=seq)
        seq=seq ⊕1;
        get(m);
        send(seq,m);
    end
    label rcv(ack) and (ack≠seq)
        "nothing";
    end
end
label Receiver
    label rcv(seq,m) and (ack≠seq)
        dlv(m);
        ack:=seq;
        send(seq);
    end
    label rcv(seq,m) and (ack=seq)
        send(seq);
    end
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.*