LSINF2345 - PROJECT REPORT

GREGORY CREPEULANDT[1]
ELIAS OUMOUADENE[2]
GROUP C

December 18, 2019

CONTENTS

1 Introduction 3
2 Part I : Building a Distributed IoT Application 4
2.1 Application purpose 4
2.2 Software benefits and limitations 6
2.3 Discuss how your implementation leverages distribution of IoT sensor data using Erlang/OTP. What do you consider as the key benefits of your solution compared to their non-distributed equivalents, and why? 7
2.4 Describe the limitations that the users should be aware of when considering your software. What possible solutions could solve the identified problems in the future? 8
2.5 Solution 8

[1] **NOMA :** 59191300
[2] **NOMA :** 81211400)

LIST OF ALGORITHMS

1 Fault Resistant pseudo-code. 4
2 Leader election pseudo-code. 5
3 Daemon pseudo-code. 5
4 One push per period pseudo-code. 6
5 Get Global Mean pseudo-code. 6

1 INTRODUCTION

For "LSINF2345 - Languages and Algorithms for Distributed Applications" class, we were asked to develop a small distributed application. The idea is to learn a new (distributed) approach for application design while using a provided framework. The general framework is achlys, the distributed variables are CRDT variables in lasp and the general (ideal) goal is to run this in a distributed way on Grisp boards. During this cold winter, our ideas quickly guided us to a heating-related application as you will discover on the next page.

At the start we thought our idea was relatively trivial because of its apparent simplicity. But when implementing it and facing few little problems, we ended up implementing an application that, we believe, is relatively good performing and implements some interesting features such as fault-tolerance and leader election.

To implement our solution we forked the achlys repository and changed only the "achlys_app.erl" file.

Our project is publicly available at https://github.com/eoumouadene/achlys .

2 PART I : BUILDING A DISTRIBUTED IOT APPLICATION

2.1 Application purpose

Our application measures the temperature on several Grisp boards and compute global average temperature on a regular time basis. The temperature measurements and mean computations are refreshed on every board every 5 seconds (this very short period was designed mainly for testing purpose). Every node computes the mean temperature based on the most recent temperatures available from other nodes but only the leader node can push its computed mean on a dedicated distributed variable (accessible from any node).

2.1.1 *State a motivating example for your software and describe what potential users can expect as an added value.*

In many houses, the heater is triggered by a thermostat. For example, if the temperature measured at the thermostat is lower than 19 Celcius degrees, the heating system will be triggered. This is a very simple solution but has the big drawback of only considering the temperature at one specific point. For example, my mum used to put decorative candles on the table just in front of the heater thermostat. This made the thermostat believe it was hot in the house when we were actually very cold (true story). Our idea is to put one little GRISP board in several rooms of the house to collect temperatures in multiple points and compute a temperature average which would be much more relevant for the heater trigger mechanism.

2.1.2 *Provide the list of features that your project offers and document their usage with code samples.*

- Join when you want : You can launch the application on GRISP boards or on an emulated version without having to rush to launch all of them at the same time at all. The order you launch them has no importance.
- Join as many as you want : In the released version, you can join up to 10 nodes without any problem. If you want to use our application on a bigger scale, you can simply modify the variable "NumberOfNodes" that was initially set to 10 (We tested with up to 30 without any issue).
- Leave and come back as you want : You can decide to kill one of the node without causing any trouble to the global system. The dead node will simply not be taken in consideration. You can also launch it again and it will be detected by the others and taken into consideration again.
- Fault resistant : If any node stops updating temperatures for a while, is not able to communicate with others for a while or simply crashs, it will not hurt the system at all. Actually any suspect node will simply be ignored until it acts correctly again.

```
case RESPONSE of
  undefined ->
       nothing
  Value ->
       Time_diff = timestamp() - erlang:element(2,Tuple), % We take the elasped time since
    last update from this node
       if Time_diff < (Sleep*3) -> % The queried node recently pushed a temperature
            do
       true ->
            nothing
       end
    end
```

Algorithm 1: Fault Resistant pseudo-code.

Here for example (algorithm 1), let's imagine we query a distributed variable to get the temperature from a specific node. If it returns an undefined value or if the timestamp related to that value is not recent enough, we simply ignore it.

- Leader election :

```
mean_compute(Sleep, Id, GlobalMean, Buffer, NumberOfNodes, Mean, Counter, CounterValid, Chef
    ) when Counter =< NumberOfNodes ->
    case RESPONSE of
  undefined ->
      nothing
  Value ->
      Time_diff = timestamp() - erlang:element(2,Tuple), % We take the elapsed time since
    last update from this node
      if Time_diff < (Sleep*3) -> % The queried node recently pushed a temperature
          if Chef == 0 -> % If no chef yet and this node was recently active, it becomes
    the chef (leader)
          become_leader
      true ->
          nothing
      end
    end
```

Algorithm 2: Leader election pseudo-code.

Here (algorithm 2), we are in a loop that check every node temperature ordered by the node's Id. If no leader is elected yet, the first encountered node that is correctly running (has recently pushed a temperature) will be elected as the leader (logically, should be node with Id 1 if it is not crashed). This leader election is run at every new loop of the deamon (every 5 sec) on every node, which means that if a node crashs it should be quickly detected and will not be elected as the leader. After multiple execution tests, we were able to confirm that all the nodes were always electing the same leader, according to the specification of our leader election protocol.

- Daemon :

```

start(_StartType, _StartArgs) ->
    start_grisp(),
    init_variable(),
    daemon(5000, Id, Buffer, NumberOfNodes, GlobalMean),
    end.
daemon(Sleep, Id, Buffer, NumberOfNodes, GlobalMean) ->
    %New round
    set_temperature(),
    sleep(Sleep/3),
    compute_mean(),
    sleep(2*sleep/3),
    daemon(Sleep, Id, Buffer, NumberOfNodes, GlobalMean).
```

Algorithm 3: Daemon pseudo-code.

Here (algorithm 3) you can have an idea of our daemon. Actually it will be called at the end of the start and will loop every 5 seconds. Each loop, it will compute the average temperature based on all the valid nodes and will confirm the leader election or elect a new leader if required.

- One push per period : Only one node (the leader) will push its computed temperature average on the GlobalMean distributed variable. Since a node may use one temperature for an other node that was updated up to 5 seconds ago, there can be some very little variation on the nodes computed means. Since for testing reasons, we use some random values for the temperatures (no access to the real sensors during the tests), this little variations may be not negligible. To avoid having multiple nodes pushing their nearly equal computed means, only the leader is allowed to push it. Note: the fact

some nodes may read few-seconds-old values to compute the temperature mean is not a problem at all in particular for the main use-case we designed the application for; private house heater.

```
mean_compute(Sleep, Id, GlobalMean, Buffer, NumberOfNodes, Mean, Counter, CounterValid, Chef
    ) when ALL NODE COVERED ->
    [***]
    if (Chef == My_Id ) or (Chef == 0) ->
    % If I am the chef or no chef exist (all the nodes were considered not valid, I am alone
    , ...), I push my mean as the current GlobalMean
        push_to_global_mean
    true ->
       else
    end;
```

Algorithm 4: One push per period pseudo-code.

- Get Global Mean from any node : Even if only the leader will push the mean temperature on the distributed variable, you can ask to get it from any node.

```
get_GlobalMean() ->
    GMType = {state_pair, [state_lwwregister, state_lwwregister]},
    GMName = "GlobalMean",
    GMSet = {GMName, GMType},
    {ok, Tuple} = lasp:query(GMSet),
    erlang:element(1,Tuple).
```

Algorithm 5: Get Global Mean pseudo-code.

2.2 Software benefits and limitations

The main benefit compared to a classical thermostat is that it is able to compute an average based on multiple temperatures from different places (rooms). But beside this point, compared to other IoT applications, our implementation has some strengths and drawbacks.

Strengths:

- Easy to launch: As explained above, the order and the time between each node launch has no influence, making the distributed application easy to configure and launch.
- Resistant: As explained above, crashed nodes or nodes that has not updated their temperature for too long will not be taken in consideration. Basically, as long as one node is still correctly running, the app will be fine.
- Simplicity : Our code is very simple to understand and does not require tons of heavy computations nor communications.
- Nearly no settings required: When you create a shell with a command like "make shell n = 1 PEER_PORT = 27001" our application will directly start running on the created node (simulated node). You don't need to specify which task to start.
- Clear outputs : We tried to have information outputs inside the created shells (simulated nodes) as clear as possible. They show the current execution state in an easy to understand way.
- Easy to test : The outputs showed during execution will tell what is the number of valid nodes, what are the collected temperatures, who is the leader, ... This makes it easy to test different scenarios (stopping an other node during execution, starting it again, joining a new node during execution, etc...). You can simply create new shells and joining them to watch in detail what is happening during the entire life of the application.

Weaknesses:

- One hard-coded variable: The "NumberOfNodes" variable is initially hard-coded at 10. It represents the maximum number of nodes you can join using our application. If you want to use more nodes you have to manually modify its value.
- All the code at once : Since we noticed that, on the clean unmodified code from the initial achlys github repo, the executed code when creating a new shell was the `achlys_app.erl` code, we decided to code our application directly inside this file. Everything is running fine but we believe it would have been a better approach to respect the architecture with multiple files representing separated functions such as workers.
- Only one task possible : Since we designed our implementation inside `achlys_app.erl`, our application is the only one you can launch. Actually it has the advantage of being directly launched but the drawback of not allowing any other task.
- Many distributed variables : In our implementation, every node has its own dedicated distributed variable that is queried from others to compute their means. This means for 10 nodes, 10 distributed variables will be created (with each node pushing to its dedicated variable every 5 seconds). We believe an implementation with less distributed variable would be possible but more complexe to implement.
- Manual joins : As explained above, the launch of our application is relatively easy since you don't need to specify tasks to run on nodes. But you still have to run few manual commands, especially the joins to cluster all the nodes together.
- No real tests with sensors : Since we had no access to the real GRISP boards with sensors during our testing phase, we represented the sensors data by some randomly generated values that would mimic possible real data.

2.3 Discuss how your implementation leverages distribution of IoT sensor data using Erlang/OTP. What do you consider as the key benefits of your solution compared to their non-distributed equivalents, and why?

To compute the temperature average, we decided, on each node, to take the most recent temperature pushed by every other node, to count them and to sum them then divide it by their number. The idea is that since every node will push their current temperature on a very regular and small time basis (5 seconds), even if some nodes may compute their mean based on older values (few seconds older values), it should not impact the system and should not be a problem at all in the real scenario of private house heater. Beside that, to be able to have one global reference for the global temperature mean, we decided to have a leader election that would push its computed mean on a distributed variable. This is useful in the case we want, for example, the heater to be triggered based on this single value (GlobalMean).

Finally, as explained above, we believe the main benefit of our application in the case of private house heat trigger is the fact it will consider multiple rooms temperatures instead of a single point temperature. This, for purely practical and comfort reasons is a real benefit compared to traditional non-distributed versions. We can discuss the fact that the computations are not really distributed themselves and, indeed, we only exploit the collect of distributed data aspect but not the capacity to distribute the computations. The fact is, for our particular application, the main focus is about the collect of information from multiple boards, the computations themselves are very small and easy to compute even if we decide that every node has to do them.

2.4 Describe the limitations that the users should be aware of when considering your software. What possible solutions could solve the identified problems in the future?

As discussed before, there are few drawbacks that the user should be aware of;

Firstly, the fact if he wants to launch more than 10 nodes, he should modify the "NumberOfNodes" value inside achlys_app.erl. This little drawback could be solved in the future by modifying the structure of the app to encapsulate it inside a task that could be launched from a command line using some arguments to specify the number of nodes without hard-coding it.

Secondly, the commands the user has to enter to launch every nodes are relatively easy to do but they are many especially if he wants to launch a large number of nodes. A very clean solution would be to write a script with the number of nodes as argument that would launch all the nodes directly without other manipulations from the user. We believe it is possible to write such a script but we were not confident enough in our mastery of bash scripts and preferred to focus on the erl application instead.

2.5 Solution

How to launch our application :

To launch our application, with simulated nodes, open a terminal inside the achlys folder and enter the command "make shell $n = 1$ PEER_PORT $= 27001$".
This will create a first emulated node and launch our application on it. The created shell will start by showing something like "application launched on node achlysX@130.104.164.101". You should note this IP address, we will need it later (let's call it MyIp). If you want, you can already check the outputs inside this terminal. You can notice that the node is detecting that it is alone and already considers itself as the leader.

Then you can launch a second node by opening a new terminal inside the achlys folder and enter the command "make shell $n = 2$ PEER_PORT $= 27002$". For now, this node is also alone and will behave as the previous one, considering itself as a leader. Then inside this shell, enter the command "lasp_peer_service : join('achlys1@130.104.164.101')" where you should replace 130.104.164.101 by the real value you got for MyIp (see above).
You have now two nodes clustered together, they will detect each other (output shows " 2 valid nodes") and elect Node1 as the leader.
Here you are, to create a bigger cluster of nodes, you can repeat the same procedure inside new terminals with the same commands as before "make shell n=X PEER_PORT $= 2700$X" where X is the number of the board (1,2,3,4,5... up to 10) then the join command inside each created shell.

We made the outputs inside these shells as clear as possible to make it easy to understand the actual state of the application.

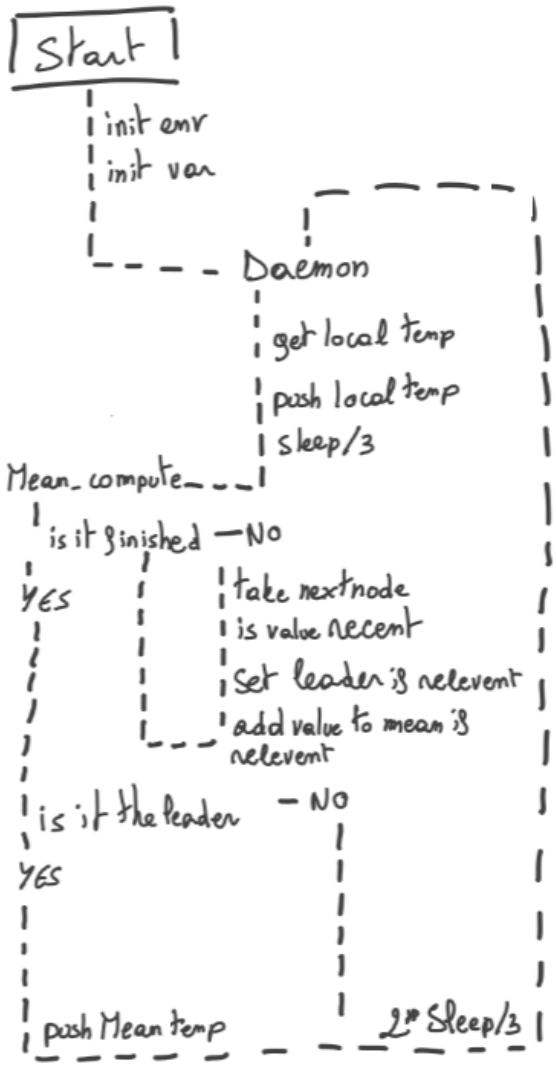

Figure 1: A general diagram of the global structure running on our grisp board.