COMP2310 ASSIGNMENT1 REPORT

Author : Jiaxu Liu

Student ID : U6920122

Introduction

To mention above: The submit file consists of 5 .adb/.ads files, where ***staged_manager.adb&staged_manager.ads*** are just for ***stage d method 1,*** which is not a fully distributed method. Method 2 is fully distributed. I provides both methods' implementation just to compare the accuracy and except for stage d(method 1), my code is not using any functions in the above package.

- As the vehicles are keep consuming energy when they are in idle condition, they need to be charged using the globes, thus we need to control them and tell them the correct position of the globes. The problem is, we cannot get the correct position directly, while we can achieve this by letting the entities which detect the globes to send message to those who need to be charged, and guide them to that position. Another problem is that the message that vehicle get would not be the latest, so we also need avoid the expiration of message.
- For the vehicles are equipped with message passing system which broadcast messages which will be received by all vehicles, the goal of the assignment is to design a proper structure of the message and set the proper time to receive&broadcast in order to ensure the version of message is up to date. Nevertheless, the efficiency and some safety issues of the program should also be taken into consideration, and we also care about the expiration of messages.
- I started the assignment from attempting stage B and its almost successful while still with some little issues and concerns, with only one globe, it can hold approximately 160 vehicles and for stage c, with multiple globes, it

can hold up to approximately 300 or more, we will discuss these details in evaluation part.

- In the following of the report, we will mainly discuss the following topics, and basically they're starting from stage C:
  1. The structure of message and the meaning&effect of its attributes.
  2. The relationship and effect of "*local message*" and "*broadcasting message*".
  3. Basic running strategies and Implementation of tasks on Stage C.
  4. Attempt & Implementation on Stage D.
  5. Evaluation of the entire model in all stages.

Structure Of Message (Inter_Vehicle_Messages)

- The message basically contains 4 attributes, which are:
  1. ***The closest surrounding globes***: This is an Energy_Globe type variable which record the position of the closest globe around the vehicle. This message is broadcasted to tell other vehicles for compare to get the correct position of the closest globe.
  2. ***The current condition of the vehicle***:

     For vehicles need to request for new information of globes when in low battery, and they also need to respond to others' requests, so I defined three basic condition of the message to show the behaviour, which is consists of:

     1. Default condition: All messages are initialized as Default condition.
     2. Requesting condition: If the vehicle did not catch new message for long time or the updated message is not new enough, we switch the condition to requesting_condition.
     3. Responding condition: Corresponding to requesting, the message will be switched into responding condition if it is trying to respond to a request, only broadcasting message could respond to local one, while the local one can replace the broadcasting one and respond to other task requests.

3. ***System time when message is updated***: Whenever a message is constructed if the vehicle found the globe, the time need to be update using *Ada.Real_Time.Clock*.
4. ***Vehicle number array:*** This is an array that helps implement stage D, the detail will be discussed in the 4th chapter, attempt 2.

"Local Message" and "Broadcasting Message"

These two variables in tasks are the key attributes of message passing system, the attributes are defined in **vehicle_message_type.ads**:

- Relationship & Effect:

  These two vars are both *Inter_Vehicle_Messages* type.

  "Broadcasting Message" is a variable which dynamically updating itself by using the *Receive()* function and send itself out by *Send()* function. This is more like a temp message that always trying to get the newest message.

  "Local Message" is storing the updated message locally, we need this to keep comparing its update time with *"Broadcasting Message"* to get the newest local data.

- Some further explanation:

  Usually the Broadcasting message is newer than the local one, and we will replace the local one with the broadcasting one. While there is still condition that the local one is newer, for instance, vehicle ***A*** just found a globe and it update the local message at ***time_1***, while then it received a message from ***B*** at ***time_2***, if ***time_2*** is less than ***time_1***, it means the broadcasting message is expired and we need to replace the update_message with the local one, and broadcast the local message. This is just an example to show that whether local or broadcasting message is the latest is not deterministic. We will discuss in detail in next chapter.

Strategies And Implementation

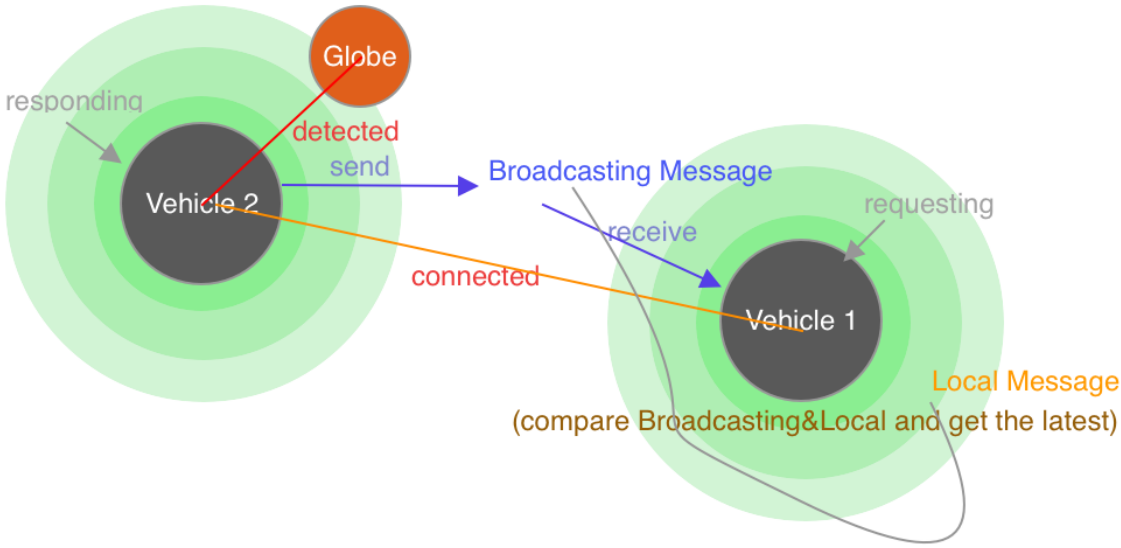

fig 3-0

The above figure demonstrates the basic working process of vehicle. In order to describe my program, I would like to divide the entire into 5 stages:

- ***Initialization of broadcasting message:***

  From line 59 to approximately line 178

  1. Detect globes: If one has detected globes around, then calculate the closest globe and store it in the *broadcasting_message*, update the local message with new message and send out for other vehicles.

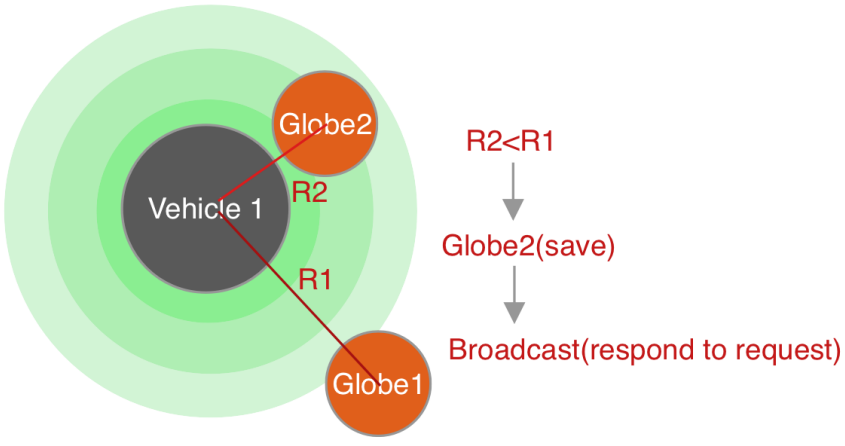

fig 3-1-1

  2. No globe around: If a vehicle does not detect any globes for long, then its only mission is to wait for receiving message from someone else.

We can use *Receive()* function to achieve this. The strategy is: try to receive a new message at the beginning of main task loop.

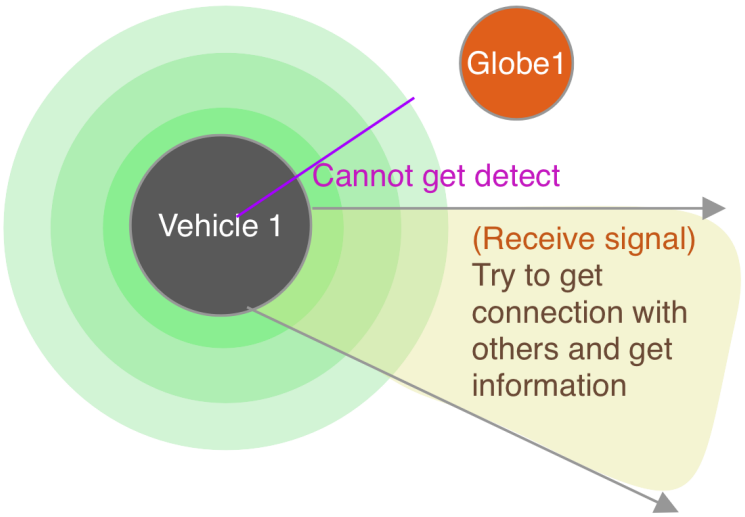

fig 3-1-2

- ***Received a new message from other vehicle:***

  From line 190 to 219

  1. If the time that the received message is updated is later than the local one, which means that the received message is newer, then we need to update the local one with the received one.
  2. While if the received message is updated earlier than local one, this means the currently broadcasting message is a old message and we need to replace the received message with the local message and broadcast to all. This is a process of fixing the responding broadcasting message.

- ***Haven't received a message for long time (Local message expired):***

  From line 221 to 231

  If one did not catch any message for long time, this could be the situation that there are too less vehicles in an area or the vehicle is drifting out of the orbit, thus we need it to go directly to the position of globe which is stored locally even if the message is out of date , then we set the mode of vehicle into 'requesting'. This process could successfully save the vehicle

from using up the battery for the position of globe on record is the most possible place to have other vehicle that send responding message.

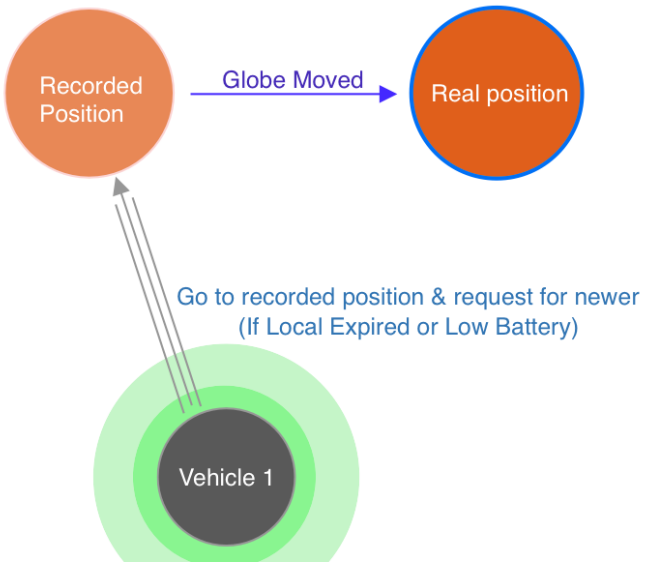

fig 3-3

- ***Supervision if the vehicle have enough battery charge:***

  From line 234 to 264

  In this situation, we do not need to focus on the target destination of the vehicle and just set it idle, however, setting the vehicle free is not convenience for vehicles in good condition to help others, thus leading them to idle on a spherical surface around the globe is an optimal selection. In order to avoid collision, I assigned different radius of spherical surfaces around each globes for vehicles to spin around if they are not busy.

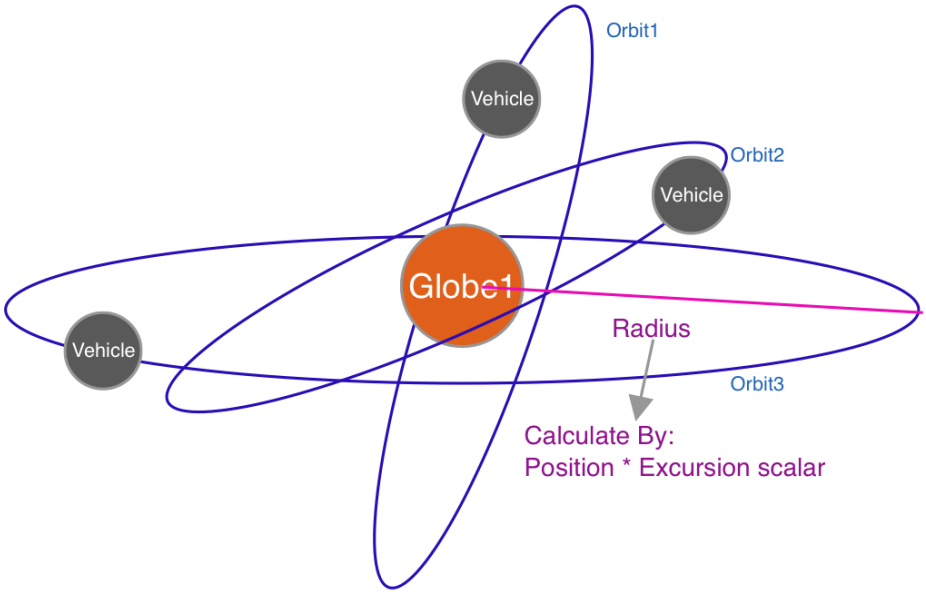

fig 3-4

- ***If the vehicle need to be charge:***

  From line 267 to 273

  Broadcast the request for the position of globe and go directly to the stored position of closest globe with max speed. The picture demonstration is same as [fig 3-3]

Attempt & Implementation on stage D

Attention!: There are two versions of implementation on stage D. The code won't execute stage D by default, and if you want to test stage D, please switch ***"Active_Stage_D" (line 79)*** in the declaration part of task to ***"1" or "2"***. This value is by default ***"0"***

```
78      -- If want to active stage d, set this to 1(method 1),2(method 2)
79      Active_Stage_D : constant Integer := 0;
80
```

For The First Attempt: This is not a fully distributed method but it provides relatively high performance, if you want to test this method, please switch ***"Active_Stage_D"*** to ***"1"***.

The implementation on this stage would still need a messager(agent) for vehicles to communicate with and a section of shared memory, the idea is really intuitive:

- Each vehicle tries to add its vehicle number into a shared queue, the max size of the queue is a constant [Target_No_of_Elements]. When the queue is full, all the vehicle stop attempting to access the queue. To ensure the deterministic of this process, we'll need a protected entry which access the queue, the entry(**Add_Itself()**) can be found in **staged_manager.adb**.
- When the set is full, each vehicle will ask if they are in the set, if they are, then it just do nothing, otherwise, the vehicle will exit its main loop and run out of its energy.

The problem is that although this method works really well, it still need a coordinator and is not fully distributed. I attempted to implement a fully distributed one which send message from vehicle to vehicle, while the accuracy is not great and the work efficiency is not exceptional to shared memory, and that was my second attempt.

For The Second Attempt: I got inspired by discussing with my classmate Ruikai Cui & Jiahao Zhang, we came up with an idea which could solve this problem distributively. Again, if you want to test this method, please switch ***"Active_Stage_D"*** to ***"2"***.

The code is from line 37 to 73, its concept is:

- We change the structure of message by adding an attribute which is an array that saves the vehicle number. And we add an another attribute which stand for the update time of the array. We also create a local array which store the vehicle number, and the max size of both arrays are [Target_No_of_Elements].
- Each time an vehicle received a message, it try to add the vehicle No of itself into the array of the message, the strategy is:
    1. At the beginning of receiving each broadcast message, we compare the length of its array with the local one:
        1. If their length are the same, we see the time stamp of the upgrade of each array. We always want the local array to be a older one to ensure the increment of the array.

2. If their length are not the same, then we replace the local one with the longer one. Again, ensure the increment.

2. After process 1, If the vehicle number is not in the message array, the vehicle task start to try to add itself into it, until the array is full(to [Target_No_of_Elements]). After the whole update, we broadcast the local message it.
3. We set a time for some tasks to kill itself, if a task reached this time, and its vehicle number is not in the local array which stores the vehicle number, it will exit the main loop and run out of its energy.

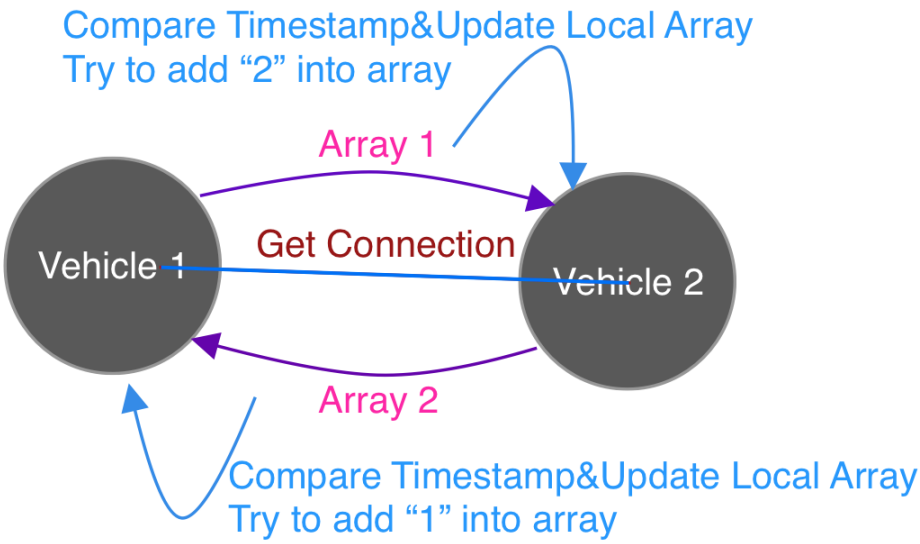

fig 4-2

Even this method sounds reasonable and reliable, there are still some problems. For instance, we will need to create a constant ***"kill_time"*** which stands for a time to kill itself, while we cannot make sure how long should this time last, as well as how long will the final message be fully spreaded hold same to each vehicle, therefore, the only measure to increase the accuracy is to make this time as long as possible. Other problems are like low efficiency, high memory cost, etc.

The Ultimate Solution: Referencing [https://raft.github.io/], Raft algorithm can solve this problem in fully distributive way with high efficiency:

Raft algorithm is a consensus algorithm which can increase error-tolerant rate of distributive system. This algorithm is explained in paper [raft-atc14 Diego

Ongaro & John Ousterhout June 19–20, 2014 • Philadelphia, PA].

In raft, each tasks(vehicles) will be in one of the three conditions:

- Leader: Managing all requests between tasks(vehicles), no more than one.
- Follower: Electorate, completely passive.
- Candidate: Proposer, could be elected as new leader

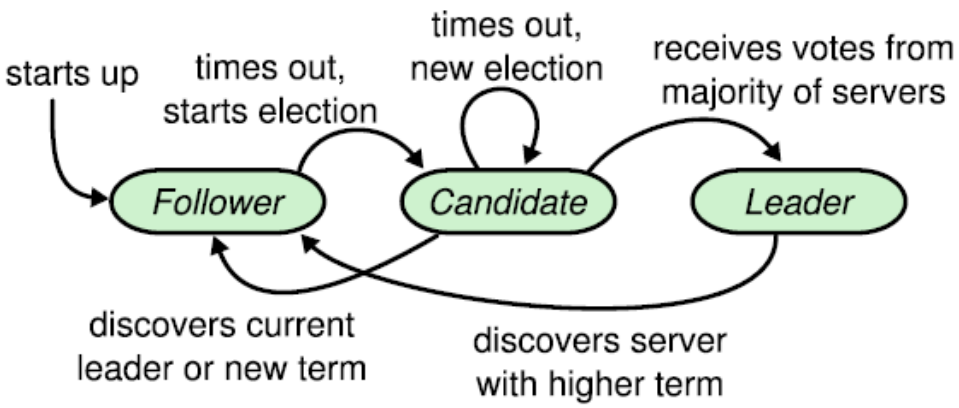

Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

raft-atc14 page. 309 figure 4

The figure above gives a good instruction on the three roles. As each start of an election, we call it "term", each term has a integer connect with it which is called term-number.

When the system start up, every tasks become follower initially. As follower begin to work, it open up a timer which record the overtime of election, when time is up, it switch itself to candidate to start a new election, it keeps doing this until the system got a new leader. A candidate will switch back to follower if it received a message from leader or a message that contains higher term-number than local. A leader can be switched follower if it received a message with higher term-number.

According to the rules defined above, we could apply this model to the assignment, while sadly I did not complete the implementation in ada. Still, this algorithm inspired me a lot in dealing with consensus problems.

Evaluation of the model

The max amount of vehicle that [**Single_Globe_In_Orbit**(stage b), **Random_Globes_In_Orbits**(stage c)] can hold. Followings are test results:

Single Globe Test

Table1	Amount Of Vehicles	Number Of Globes	Remain vehicles After 10 min
1	64(default)	Single	64
2	100	Single	100
3	150	Single	150
4	200	Single	144

Multiple Globes Test

Table2	Amount Of Vehicles	Number Of Globes	Remain vehicles After 10 min
1	64(default)	Double (fast)	64
2	100	Double (fast)	100
3	200	Double (fast)	193
4	250	Double (fast)	247
5	300	Double (fast)	285
6	64(default)	Random	64
7	100	Random	100
8	200	Random	200
9	250	Random	250
10	300	Random	292

As you can see from the above tables, the implementation does work properly, one globe can at most hold approximately 150 vehicles and multiple globes can hold up to about 300 globes.

For most cases, vehicles died because of crashing into others or others are always accessing globes before itself which causes its starvation. To solve this problem, its possible to dynamically set a priority on each vehicles and let the lowest charged vehicle to have the most priority to access each globes.

Another situation that causes their death is that some vehicles are initially "stray", as they reached the dangerous level and need to charge, for they haven't receive any messages from others and still idle, they don't know the position of globe, which cause its death. The reason can be explained as the radius of orbit is too small and cannot "infect" all vehicle initially. This problem can be fixed by dynamically adjust the radius of each spinning orbit.

For stage D, the method I implemented can work properly in good accuracy, however, it might costs great time consumption per loop, I think the ultimate solution is *Raft*, while its hard to implement and I could only state my idea in *part4*. For I have already provided two implementations in one piece of code, hope you could play with my code with pleasure. Here I provides some of the test results:

Test Result On Method 1 (Not Fully Distributed)

Table3	Globes	Initial Amount	Targe	Final Amount	Accuracy
1	Single	64	42	42	100%
2	Single	150	42	42	100%
3	Single	200	42	42	100%
4	Random	64	42	42	100%
5	Random	150	42	41	97.61%
6	Random	200	42	42	100%

(For test on method 2, I set the kill time to 10s, which means that the system will react to kill itself after 10s, just to ensure its accuracy)

Test Result On Method 2 (Fully Distributed)

Table4	Globes	Initial Amount	Targe	Final Amount	Accuracy
1	Single	64	42	42	100%
2	Single	150	42	42	100%
3	Single	200	42	42	100%
4	Random	64	42	42	100%
5	Random	150	42	42	100%
6	Random	200	42	42	100%

References

In Search of an Understandable Consensus Algorithm[Diego Ongaro and John Ousterhout June 19–20, 2014 • Philadelphia, PA] [https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14]

The Raft Consensus Algorithm [https://raft.github.io/]

Discussion Partner

- u6919043(Ruikai Cui)
- u6921098(Jiahao Zhang)