INFO8002: Implement a consensus algorithm

Antoine Leroy,[1] **Hubar Julien**,[2] **Lievens François**,[3] and **Antoine Cajot**[4]

[1] *Antoine.Leroy@student.uliege.be (s140618)*
[2] *jhubar@student.ulg.ac.be (s152485)*
[3] *flievens@student.uliege.be (s103816)*
[4] *antoinecajot@student.uliege.be (s125729)*

I. INTRODUCTION

In order to safely shepherd a rocket to a circular orbit of about 100 km, it is imperative to ensure a good consistency and availability of the computing resources used to guide the rocket. One way to achieve this is to rely on a cluster of computers collaborating to guide the rocket, using distributed algorithms. Our implementation will leverage a modification of the Raft consensus algorithm. [1]

In the following sections, we will respectively (i) introduce the original Raft consensus algorithm, (ii) explain what is missing in the original Raft algorithm to operate with the assumptions made in the context of the rocket guiding problem stated, (iii) describe briefly our implementation and (iv) detail some of the experimental results. Finally, (v) we will conclude with what we have learned from all our reflection and experiments.

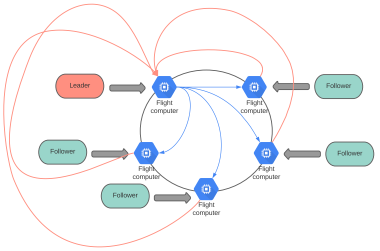

FIG. 1. Illustration of Raft cluster with flight computer

II. RELATED WORK: THE RAFT CONSENSUS ALGORITHM

Let's first examine the basics of the Raft consensus algorithm. Raft distinguishes itself from Paxos by focusing on understandability by separating the different tasks behind consensus its constituent sub-problems, namingly leader election, log replication and safety. This results in an algorithm that is both easier to understand but also easier to implement without sacrificing efficiency. The algorithm works as follows. A *leader* is first elected among all the nodes of the cluster: it has full responsibility for handling operations on the data the cluster replicates. It accepts instructions from the clients (in Raft, data is a replicated log and instructions are new log entries), replicates these instructions to all other nodes (called *followers*) and informs them to apply them to their own copy of the data when it has assessed it was safe. Having the data flow only from the leader to all its followers simplifies greatly the data flow: the leader can manipulate the data without the overhead of consulting the others. *Followers* are therefore fully passive (they can only respond to requests) and only the *leader* (and *candidates* during the election process) can issue requests.

Time in Raft is divided into *terms* which represent the duration for which at most one leader will be in office (one when the election process succeeds, none when a split vote occurs: in that case a new term starts and elections begin anew). A leader authority is maintained by periodic heartbeats sent to the followers. However when a follower fails to receive news from the current leader for a period of time known as the *election timeout*, it increments its current term, flips to candidate state and starts the election process by sending vote requests to every node in the cluster. It will also vote for itself. It will remain in the candidate state until either a node (possibly itself) wins the election or until too much time elapses without a winner. Each node can vote for only one candidate per term (first-come-first-served) and a simple majority is enough to ensure that only one leader can be in office. To prevent split-votes from endlessly happening, Raft randomizes election timeouts per node: there are therefore less chances that followers will switch to candidate state at the same time and in case of split votes there are also less chances several previous candidates will trigger another election at the same time.

Each client request bundles a command that is to be executed on the replicated state machine. The leader starts by appending it to its own log before it sends `AppendEntries` RPCs to all others nodes in the cluster. Only when the entry has been sent to every follower does the leader actually execute the command on its own state machine and return the result of the execution to the client. It may be that `AppendEntries` are lost or that followers fail or run slowly, in that case the leader will continue sending the `AppendEntries` RPC over and over again until every follower has acknowledged it, even after the result has been returned to the client. Then, once the leader assesses a command is safe enough to be executed on followers, it notifies to the followers it is *committed*: they can now apply it to their own local state machine (in order). Committing a command also commits all previous command entries known to the leader to followers. This ensures that committed entries are durable and will be executed by all valid state machines in the cluster.

III. THE BYZANTINE ENVIRONMENT

The Raft algorithm allows us to take into account possible node crashes, to regenerate it if possible, or to avoid possible slow flight computers thanks to heartbeat timeouts. But in a byzantine environment, some of the computers of a cluster can do worse than fail: they can misbehave and return erroneous computations (intentionally or not). The original Raft implementation could not deal with that kind of issue because, since data and instructions were only flowing from the leader the the followers, a byzantine leader could have imposed erroneous instructions/data to all its followers without them being able to give their "opinion" about it. Therefore, it was necessary for us to extend Raft so that followers would be able to respond to the leader and have all of the nodes agree on the correct instructions to be executed instead of them being imposed by the leader. This is further described in the next section.

IV. IMPLEMENTATION

A. Choice of leader

Just like in the *Raft* algorithm, we have implemented an election system. These elections happen every time a follower triggers a time out, who means that the actual leader maybe fail. During these, a candidate will send a message containing his ID, term and `time index` to every other nodes of the system. When another node receive a vote request from a candidate, it performs several security checks to detect whether the node is correct or not (i.e. term and state are up to date). If the candidate satisfies all constraints, the other node give him a vote. A node can vote for only one candidate per term, and if this candidate obtain strict majority of vote, he becomes to be the leader. Figure 2 shows how an election is conducted.

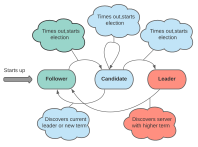
Times out,starts election
Times out,starts election
Times out,starts election
Starts up
Follower
Candidate
Leader
Discovers current leader or new term
Discovers server with higher term

FIG. 2. Illustration of an election process

B. Heartbeat

Once a leader is elected, he will prevent followers from triggering an election by sending them heartbeats (If a follower has not received heartbeat message over some time, then it triggers an election). So, this heartbeats mechanism implement a failure detector who allow the system to detect a crash or a slowdown of the leader, who can't so making a proper job.

Given the unpredictable behaviour and workload of the different hardware on which we can run our flight computers, it is common for a good leader to cause a time out in a follower. However, this will have no consequences, as this node, once downgraded to follower, will continue to function perfectly.

Figure 3 shows the sending of messages by the leader to different followers.

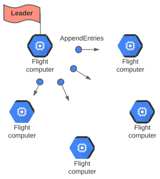

FIG. 3. Sends messages from the leader to the different followers

C. Broadcasting of the state

In our algorithm, the first role of the leader will be to receive messages who contain the state of the rocket, to broadcast this messages to each others node, and to make sure that the majority of nodes accept and replicate this state.

So, once the leader is elected, he will be able to broadcast the state he has just received (see *Fig* 4).To make this, we are using http POST method, using the *request* package of *Flask* who let us to reliably communicate between nodes and to handle link failures. Using this component, we can so assume a perfect link implementation between our nodes.

When a leader broadcasts a state to a follower, this request is received by an end point of that follower who will organise the state replication procedure like this: First, the *acceptable_state* method of *FlightComputer* is used to verify that this state is accepted by the node. If this is the case, the Deliver state method is used to update the flight computer's knowledge about the rocket state. If this is not the case, a disagreement notification is send back to the leader.

When the leader receive a majority of acknowledgment from the followers, and that it can also update his own state value and to send back a validation to the rocket.

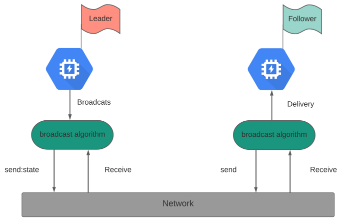

FIG. 4. Diffusion of the state by the leader to the different followers via the network

D. Consensus and broadcasting of the action

Once the state is sent, received and approved by all followers and the leader, the rocket need to receive a good action to execute. So, the rocket (modelled by our *test_consensus.py* file), after receiving a notification confirming that the state has been successfully replicated, will send an action request to the leader. This request will triggers the *action_consensus* method (*raft.py*) of the leader, who will manage the consensus.

As explained above, a node choosing actions arbitrarily cannot be detected as failing by the election system. In fact, in this project we are going to consider these random flight computers as good flight computers, in the sense that despite their random personal decisions, they are quite capable of organising the democratic consensus that we have implemented.

So, in addition to the consensus achieved on the choice of the leader, we decided to implement an additional consensus on the choice of the action to be carried out at each request of the rocket, and who will be based on the assumption that the majority of our flight computer are taking the decision.

So, the *action_consensus* method will first send a request to every nodes of the system to find out what action each one recommends, given the previously breadcasted state. Each follower will reply the dictionary returned by the *sample_next_action* method of his flight computer.

When the leader receive a reply who was the same for a majority of the nodes (including himself), a second broadcasting process share the decided action to every nodes who can use this action in her *deliver_action* method. When the leader has received a mojority of action delivery notifications from the followers, he can in turn deliver the action to himself and respond to the rocket with the decided action.

This procedure is shown in the figure 5: An action request is retrieved by the leader who will then send it to the followers. This is action 3 in Figure 5. Once the action is submitted to the different followers. They answer the action they want to take according to the previously received state. This corresponds to action 5 in Figure 5. Then, the algorithm waits for a majority of the same answers. This corresponds to action 7 in Figure 5. Once this one is obtained, the action is delivered to all the servers (leader and followers).

Figure 5 shows each step of the algorithm to make a decision on the action to be taken according to a given state.

Given its implementation, it is possible ,in some cases, for a slow flight computer to be elected as leader. An additional security added to speed up the reject of a slow leader is the following: while the leader is executing its own *sample_next_action* function, and since it is this function that can take too much time, the sending of the heartbeats is suspended. If this were not the case, the slow flight computer will continue to emit heartbeats, and thus remaining in the lead and slowing down the whole system.

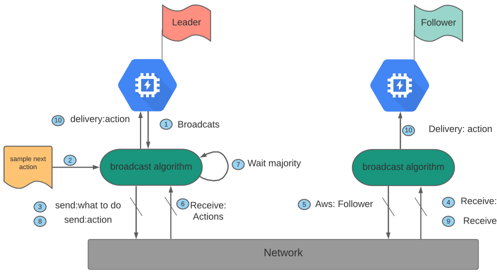

FIG. 5. Illustration of a consensus of an action and broadcasting of this one

E. Network

The network has been simulated thanks to the file *run_server.py* Using flask and requests module, we were able to set up our own decorator. This decorator allows us to create links illustrated in Figures 4 and 5 between the leader and the different followers. But also between our test file and the leader.

F. Rocket simulation and cluster testing

All interactions with the rocket are simulated by our *test_consensus.py* file. The take of procedure, is the following:

1. First of all, we have to start each rocket's flight computer by executing *run_server.py* in a new console for each computer. This python file have to be executed in the *fbw* environment and with the argument *–port 800x*, where $x \in 0, ..., n$ and n is the number of flight computer in the rocket. The *–flight-computer-type* argument, can be set to 1 if the operator want a random flight computer allocation. We also add a new argument *–type* who let create a faulty flight computer of the wanted type: 0 for a full throttle flight computer, 1 for a random throttle, 2 for a slow and 3 for a crashing flight computer.

2. Once the determined number of flight computers are up and running, we can run the *test_consensus* file with no arguments. To make good job, the *peering.json* file have to be update according to the number of working flights computers. By default, our code test 7 flight computers.

3. The main loop of the *test_consnsus* file will, for each time step, send the state of the rocket, the action demand, and the receive the consensus action to and from the leader.

   By default, The leader is randomly selected. But if the node who receive the request is a follower, an bad leader notification is return with the id of the leader. In this case, the time step restart with the given leader id. If the node who receive the request is a candidate, a no leader notification is return. So, the test script wait the next time step without taking an action before, letting the time to the flight computers to elect a new leader.

4. At each start of the main loop, a small time sleep of 0.1 second simulate the the passing of time between state sampling and let the time of elections process when the system is changing of leader.

5. for each time step, the returned action is compare with the corresponding action in the actions pickle. So, the test script print on the screen the proportion of good taking actions returned by the concensus process.

V. DISCUSSION ON THE CONVERGENCE RATE OF YOUR CONSENSUS ALGORITHM

We firstly investigate the effect of the size of the cluster on the needed time to get a consensus decision each time that the cluster have to take a decision. Our figure 6 represent the time done during our tests to perform 100 time step consensus with clusters respectively of 4, 5, 6 and 7 flight computers. This tests are using only no faulty flight computers and each cluster size are tested 10 times.

During the execution of the system, a part of the time is strictly fixed: at each time step, the *test_consensus.py* algorithm (who simulate interactions with the rocket) wait a small time sleep of 0.1 sec to simulate the pass of the time between states. This time sleep permit us to simulate a fixed time passing, who is not accelerate when the rocket have no consensus answer during an election process.

Despite that, we can notice a relatively linear increase in the time needed for the system to achieve its consensus with the increase in the number of flight computers. This result looks normal to us: during each consensus process, the master have to to wait a majority of acknowledgment or a majority response during multiple part of the process. Moreover, when the master wait a majority action from followers, the lock process who handle the encoding of each node answer take some time.

In a second experiment, we have decide to evaluate performances of our system by the same way of time computing, but this time on a fixed size cluster of 10 flight computers. On this cluster, 10 test realisations are perform after the shut down of one more flight computers for each test, in order to simulate the crashing a part of the system. The figure 7 represent theses result for 10/10 to 6/10 correct flights computers in the cluster.

The same report as in the first experiment can be interpret out this figure: our system is faster with less active flight computers. It can look surprising, but we have to remember that we are in a situation where crashed flight computer don't emit request, and so, don't saturate the leader while good flight computers are replying.

Finally, to investigate the effect of the proportion of random flight computers in the cluster, we have perform exactly the same test. But this time instead of closing a part of the cluster, we replace this part by random throttle flight computers. Result of this test are show in the figure 8.

As we could expect, the computing time increase with the number of random flight computer. It could be explain by the fact that in addition of having a classical weight in the network, the random flight computers will generally return actions who are not the same that a good computer. So, the leader, who have to wait a majority of same answers, have to globally wait answers from a bigger part of the cluster to obtain this majority. The small superiority of the computing time that we observe for the cluster who only contain good computers seems to be not significant to us, and look like to come from a small change in the computing environment like a load change in the testing computer.

VI. CONCLUSION

Our experimental part permit to highlight the fact that our consensus algorithm work well if the number of node in the system is not too height. It is so appropriate to the situation of flights computers, in the sens that a rocket will only embark a limited number of computers for evident weight reasons. In most general case, this type of system where a leader have to deal with all nodes could quickly saturate due to the bottleneck represented by the leader.

Beyond this aspect, our cluster guaranteed an exact consensus as long as the strict majority of the computers are correct. To go further, it seems possible to reduce this necessity by making stronger assumptions about the detection of faulty computers in order to exclude them definitively of the system. In real-world situations, performances of the algorithm can also be greatly improved by using much more hardware-optimised time out values. Indeed, we use wide and safe values that allow us to stabilise the system on a wide range of different hardware.

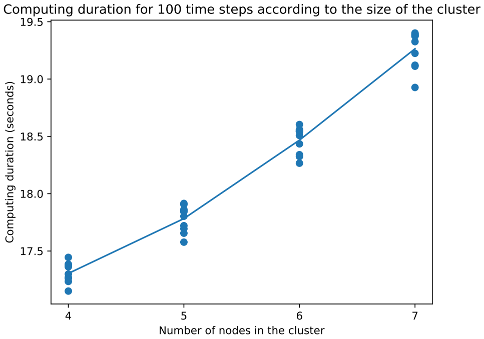

FIG. 6. Ten tests are perform for each cluster size.

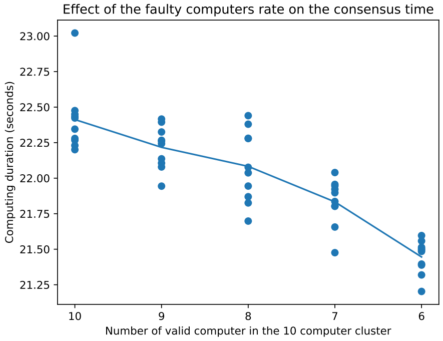

FIG. 7. Computing time according to the proportion of crashed computer in the cluster

[1] D. Ongaro and J. Ousterhout, Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014 , 305 (2019).

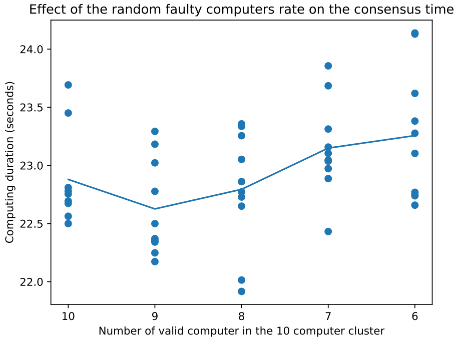

FIG. 8. Computing time according to the proportion of random computer in the cluster