INFO8002: Large-Scale Data System
Developing a Distributed System Consensus

Olivier Beauve,[1] **Robin Wallon**,[2] and **Alexandre Baré**[3]

[1]*olivier.beauve@student.uliege.be(s171195)*
[2]*robin.wallon@student.uliege.be(s171708)*
[3]*alexandre.bare@student.uliege.be (s172388)*

I. PROBLEM STATEMENT

In order to effectively control a rocket with several flight computers, we have to solve a consensus between them. Each time the rocket proposes a state, it is decided and then replicated identically among the flight computers. An action depending on the current state is also decided right after the state and returned to the rocket to effectively adjust its trajectory.

In our case, each flight computer and the rocket are running in different processes. We implement a REST API via *Flask* to deal with the communication between the processes.

Our code can be run with the same command as advised in the instructions with the desired parameters (`--flight-computers > 0` and `--correct-fraction > 0.5` to have an execution where the flight computers are guaranteed to reach a consensus)

II. ARCHITECTURE

- `without-ksp.py` represents the client/rocket and is responsible for delivering the state at the current time to the instance service of class Service.
- `service.py` is the link between the client and the current leader/proposer of the distributed system of flight computers. It communicates directly with the servers.
- `server.py` represents each *Flask* server-side application associated to a flight computer to enable HTTP communication between the servers and between the service and the servers.
- `computers.py` represents each flight computer and handles the internal computation associated to those.

At initialisation, in `without-ksp.py`, we create an instance of the class Service. This object will instantiate each sub-process for each server so that they can all run in parallel. Each server will then instantiate its own flight computer from `computers.py`. See Diagram in Appendix. To operate correctly, it is imperative that the execution waits some time (`TIMEOUT` can be adjusted in `service.py` depending on the number of computers) so that each computer is correctly initialized before launching the rocket.

A. Server-Side

As said before, our communication between processes rely on the HTTP protocol. Each process is thus assigned an address of the form *127.1.0.0:4000* where the port is unique to distinguish it from the other processes. As security is not a concern in our project, we decided that the communication between processes would be done via GET requests.

In practice, we follow the following structure to have access via GET requests to the different functions associated to their endpoints: *http://<address>:<port>/<endpoint>*

Here are the main endpoints we put in place:

- **/ask_leader_address**: asks a flight computer the address of who it has saved as current leader.
- **/decide_on_state_and_action**: decides on a state and an action that is coherent with the state machine of the majority of computer nodes and returns the action.
- **/find_leader**: finds the current leader of the most recent term and ask it to bring a consensus on a given state and its proposed action with the majority of computer nodes and returns the action.
- **/acceptable_state**: checks that the given state is consistent with the flight computer state machine.
- **/acceptable_action**: checks that the proposed action is consistent with the action that the flight computer would have computed by himself.
- **/deliver_state**: delivers the next decided state to the flight computer.
- **/deliver_action**: delivers the next decided action to the flight computer.
- **/request_vote**: requests a vote for the leader election from the target flight computer
- **/heartbeat_message**: sends an heartbeat message (see RAFT algorithm) to a flight computer.

B. Client-Side

The client side is responsible for delivering the current state of the rocket and waits for a consensus to get back the action decided.

In order for the rocket to ask the distributed system of flight computers for an action in accordance to a given state, we must first establish the link between the rocket and the current leader of the distributed system. To do so, we save the last address of the leader. We ask to this computer (thus not randomly chosen) who is the current leader. Either it is still the current leader and it returns its own address, or it is not the leader anymore. In this last case, it will ask its own saved leader address, or rather the computer associated to it whether it is still the leader. If it is, it will return its address, else it will ask to its own saved leader address, or rather the computer associated to it whether it is still the leader. And so on...

Moreover, if this process takes too long, timers will cut the process short and the client will ask another computer (this time randomly chosen) to find the leader that will propose the next action given the rocket state.

It is worth mentioning that this approach will lead us to a very satisfactory convergence time (see graph in appendix) though we had not the time to optimise the different timeouts in our implementation. We can see that the greater the correct fraction of flight computers, the shorter the convergence time is. And that with a fraction lower or equal to 0.5, the convergence time increases drastically and in fact the convergence is not even guaranteed anymore. Most often the leader does not encounter problems if it is a correct flight computer. Thus, most reelections will occur for leaders being malicious flight computers because of failed heartbeat message broadcasts (see later) or because of failed immediate consensus on the leader proposed action. It is thus not surprising that a correct leader will often have a longer term successfully deciding many states and actions. Our link between the client and the servers, i.e.: in `service.py`, is thus well designed: we always consider the last leader in first place for communicating the current state and asking for the corresponding action. And only if this first attempt fails, we randomly choose another computer.

III. DISTRIBUTED SYSTEM

A. Consensus

A consensus is reached once a leader proposing a state and an action obtains the majority of votes among itself and its peers. A flight computer will only vote for a state and an action that are consistent with its own 'state machine' (see later). A consensus should always be reached. This can only be guaranteed if the fraction of correct flight computers is bigger than 0.5 (for the quorum to work). In case of a failed consensus with the current leader, a reelection is enforced so that eventually, a charismatic leader will be able to bring about a consensus. A charismatic leader is a leader that proposes a state and an action that will be voted for at the majority.

B. Failures Handling

If a slow correct flight computer is a leader, it will probably not have the time to propose an action. Timeouts caused by the election timer will enforce a reelection among its peers.

If a slow correct flight computer is a follower, it will probably not have the time to participate in the consensus votes to decide an action. As soon as a majority of positive answers have been received, the leader can reply with the action asked by the client. It also does not need to wait for the latecomers thanks to the established timeouts.

If there is a majority of slow correct flight computers in the distributed system, the time needed for a consensus will be lower bounded by the necessary slow flight computers response time thanks to the eventually perfect failure detector abstraction.

If a corrupted flight computer is a leader because it has not yet been detected as a suspected process, it will probably not propose the right action given the current state. In such a scenario, a consensus will not occur. We thus enforce a reelection in case of failed consensus so that other computers have their chance to propose an action.

If a corrupted flight computer is a follower, it will probably not accept the action proposed by the leader. Its vote will be negative but as long as a majority of nodes are correct, the consensus will eventually be reached among these.

If a computer crashes while being a leader or crashes and then becomes a leader, it will not be able to bring about a consensus. In such a scenario, a reelection is enforced to retry the consensus on another term.

C. State Machine Consistency

State machine consistency is checked during the consensus. Each time a leader broadcasts a proposition of state and action, its computer peers can verify that it is indeed consistent with their internal state machine. The state is always consistent but the action proposed should be equivalent to the action that the computer can compute by itself knowing the current state.

In addition, to ensure that an old state or action can not be accepted, the computer verify that the action or the state is proposed by a computer with a bigger or equal term and bigger step than him. The term refers to the current epoch consensus id and the step is used to know the order of the heartbeat messages.

D. Leader Election

The election process is done according to the RAFT algorithm. At first, flight computers are set to the status of FOLLOWER.

Each flight computer is assigned a random timer in the initial range [1, 2]sec, the election timer. This timer is reset to a random value in the range every time it fires as long as the flight computer is not a LEADER.

Once this timer is up, the associated flight computer becomes a CANDIDATE of the incremented term. It votes for itself and then requests votes among its computer peers. A peer will vote if the current term is larger than its own and thus if it has not voted yet in this term. It updates its current term to the candidate's one. This ensures that the computer can not vote anymore in this term as it has the same term as CANDIDATE.

In the first case, the candidate becomes a LEADER if it receives a majority of votes. It then sets a second timer to a fixed value, reset each time it fires, the heartbeat timer.

In the other case, it waits until the next timeout of its election timer to become again a CANDIDATE or it votes for another CANDIDATE if the occasion comes. In the last option, the flight computer becomes again a FOLLOWER.

A LEADER broadcasts heartbeat messages to its peers each time its heartbeat timer times out. If a FOLLOWER or CANDIDATE receives a heartbeat message of a term greater or equal to its own before the election timer times out, it resets its random election timer and waits in the FOLLOWER state.

Additionally, if the current LEADER after having proposed a state and an action does not reach a consensus, we enforce a reelection by stopping the LEADER from sending heartbeat messages. We then wait for the election timer of some FOLLOWERs to fire and new CANDIDATEs will request votes to become the new LEADER. This way, other computers could potentially become a LEADER and in turn propose their state and action for consensus.

In summary,

- the FOLLOWER state is the initialisation state but also the state in which the computer has voted for a CANDIDATE in the higher known term.
- a CANDIDATE increments the current known term and waits for votes from its peers.
- a LEADER is an elected candidate, i.e.: it has received the majority of votes in the current known term. It is responsible for proposing a state (sent by the rocket) and action (really proposed by this computer) for consensus and reply to the rocket with this action.

E. RAFT

It is worth noting that we did not actually implement a full RAFT algorithm because the problem at hand can be handled with simplifications. With our simplified RAFT, we make the assumption that the only operations handled are assignations. By this, we mean that the state is only assigned to the flight computers after it has been decided. No append operations should be implemented. We therefore do not need to implement log replications as past assignations are irrelevant to keep. A leader only overwrites the previous entry if its entry is effectively more recent, i.e.: the term being higher and timestep being higher or equal to the receiving process's ones

Also, though we reach a consensus over the action, it is not replicated to all flight computers. Only the leader needs to compute and return it to the client after it has been decided. In case of failure of either the consensus over the state or the action, a reelection is enforced to retry a consensus. The leader does not need to seek for the most recent state machine among its flight computer peers to update its own state machine. Even an old state machine would not compromise the consensus as only assignations are taken into account. The state proposed by the leader is the state just given by the client (and past states are not used). Thus, if only the consensus over the action failed, the decided state will have to be decided again with the new leader in our implementation.

IV. DATA SYSTEM ABSTRACTIONS

A. Perfect Link, Best-Effort Broadcast & Eventually Perfect Failure Detector

The perfect link is assured by the TCP protocol as we are using HTTP which is based on TCP.

- Messages are not duplicated and are received in the order they were sent.
- Messages are not lost / Messages are eventually delivered.

The messages used by RAFT are in our implementation sent via HTTP using best-effort broadcast. Indeed, we have the following properties:

- Validity: if a correct process broadcasts a message m, then every correct process eventually delivers m. This is ensured by continuously resending messages after each heartbeat timeout.
- No duplication: no message is delivered more than once. This is guaranteed by the perfect link abstraction.
- No creation: if a process delivers a message m with sender s, then m was previously broadcasted by process s. This is guaranteed by the perfect link.

This type of broadcast is using perfect links. Thus, best-effort broadcast guarantees reliability only if the sender is correct. In the other cases, this type of broadcast will just fail silently.

To suspect incorrect nodes, we decided to implement an eventually perfect failure detector satisfying the following properties:

- Strong Completeness: Eventually, every process that crashes is permanently suspected by every correct process.
- Eventual Strong Accuracy: Eventually, no correct process is suspected by any other correct process.

We are thus in fail noisy environment encapsulating the timing abstraction of a partially synchronous system with a crash-stop process, a perfect link and an eventually perfect failure detector. In such an environment, we can evade the FLP impossibility and ensure a consensus.

We can now summarise the whole discussion of our modified RAFT algorithm with the following implemented abstractions that ensures a consensus.

B. Eventual Leader Detector

Thanks to our eventually perfect failure detector, we can implement an eventual leader detector satisfying:

- Eventual Accuracy: There is a time after which every correct process trusts some correct process.
- Eventual Agreement: There is a time after which no 2 correct processes trust different correct processes.

C. Epoch Change

An epoch change abstraction satisfies the following properties where the term is the current epoch id and l is the leader of the current epoch:

- Monotonicity: If a correct process starts an epoch (term, l) and later starts an epoch (term', l'), then term' > term.
- Consistency: If a correct process starts an epoch (term, l) with term = term', then l = l'.
- Eventual Leadership: There is a time after which every correct process has started some epoch and starts no further epoch, such that the last epoch started at every correct process is epoch(term, l) and process l is correct.

D. Epoch Consensus

An epoch consensus/term abstraction satisfying:

- Validity: If a process ep-decides v, then v was ep-proposed by the leader l' of the current epoch consensus.
- Uniform Agreement: No 2 process ep-decides differently.
- Integrity: Every correct process ep-decides at most once.
- (There is no need for a Lock-In property in our implementation as we restart completely each failed epoch consensus from `service.py`, ignoring the previous ones)
- Termination: If the leader l is correct, has ep-proposed a value and no correct process aborts this epoch consensus, then every correct process eventually ep-decides some value.
- Abort Behaviour: When a correct process aborts an epoch consensus, it eventually will have completed the abort; moreover, a correct process completes an abort only if the epoch consensus has been aborted by some correct process. Indeed, only a leader can abort an epoch in our implementation.

E. Consensus

Our algorithm thus ensures a consensus:

- Termination: Every correct process decides eventually some value. If the leader is an incorrect process, the majority quorum to decide a state and action will not be met and a reelection will occur. If the leader is a correct process, it will eventually obtain a majority quorum among correct peers and decide then the action and state.
- Validity: If a process decides v, then v was proposed by some process. The action and state are always proposed by the current leader to its peers and are only decided upon a majority quorum.
- Integrity: No process decides twice. Indeed, the leader only asks once each of its peers to decide on a state and action. Each computer checks the term and the timestep to see if it has already voted for the current action and state.
- Agreement: No two correct processes decide differently. This is ensured as the implementation of `acceptable_state` and `acceptable_action` is identical for all correct flight computers. They will always accept a state and they will all come to the same decision for a proposed action given the state decided just before.

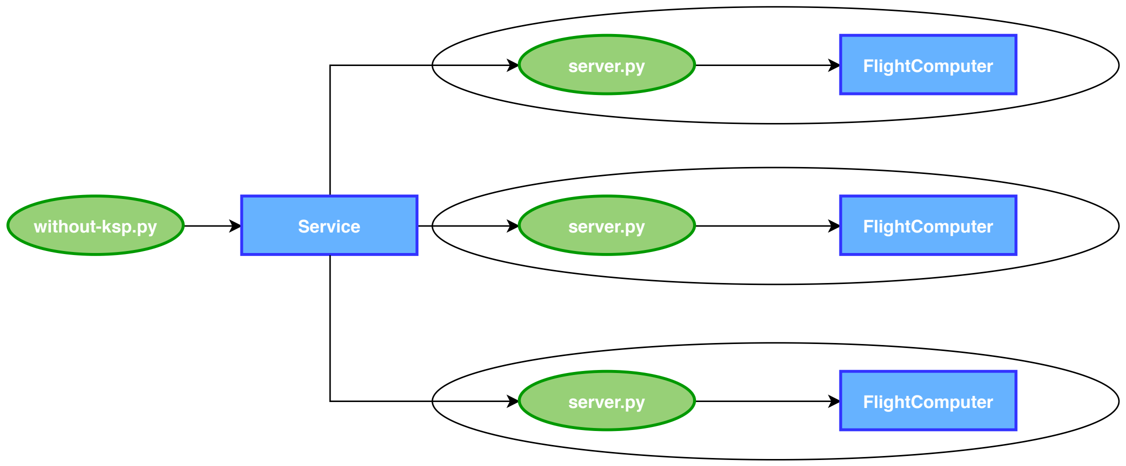

FIG. 1. Architecture Diagram

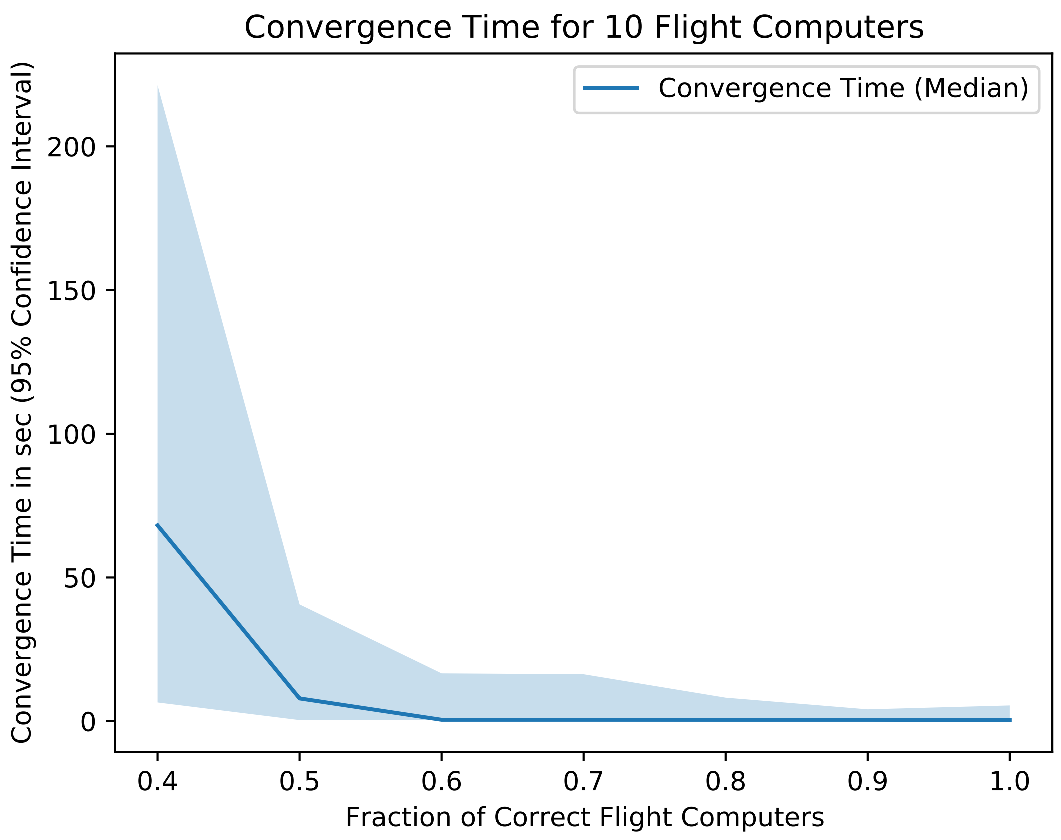

FIG. 2. Convergence Time