An Energy-efficient FaaS Edge Computing platform over IoT Nodes: Focus on Consensus Algorithm

David Fernandez Blanco, Frédéric Le Mouël, Julien Ponge, Trista Lin

An Energy-efficient FaaS Edge Computing platform over IoT Nodes: Focus on Consensus Algorithm

David Fernández Blanco*†, Frédéric Le Mouël*, Julien Ponge‡ & Trista Lin†

ABSTRACT

With the increasing computing needs of the new systems and applications, cloud offloading has become a popular choice for constructors to keep the prices of their devices affordable. However, this solution only shifts the scaling problem from the end devices to the cloud, increasingly enhancing the capacities of cloud infrastructures. As a way to reinforce the cloud capabilities on the edge without needing to add extra computing resources, we propose PyCloudIoT, a collaborative energy-efficient Function-as-a-Service (FaaS) computing platform (pltf.) with low-to-medium availability targeting the execution of punctual stateless functions over the already deployed IoTs and gateways. As these resources are extremely dynamic, with intermittent availability, heterogeneity and faultiness, the addition of strong control mechanisms is key to efficient operation. In this paper, we discuss the PyCloudIoT Consensus Model (PCM), which enables the coordination and orchestration of resources dynamically and compensates for the faults of the IoT computing farm. Compared to SOTA, PCM shows promising results with a performance and energy consumption improvement of 20% and 66% and 37% and 65% respectively compared to the best configurations of Raft and Pirogue (4+1 quorum), achieving at the same time a slightly stronger fault tolerance level.

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*; *Dependable and fault-tolerant systems and networks*; • **Hardware** → *Impact on the environment*;

KEYWORDS

Consensus Algorithms; FaaS Edge Computing; Internet of Things; Distributed Computing; Resource Orchestration.

*Univ Lyon, INSA LYON, Inria, CITI, EA3720, 69621,Villeurbanne, France.
†STELLANTIS, 78140, Velizy-Villacoublay, France.
‡Red Hat, Lyon, France.

1 INTRODUCTION

Society today is more connected than ever before. With new forms of connected objects, smartphones, and computers, applications have progressively increased their demand for connectivity and computing power. This rise in the application's resource requirements has been paired with a steady increase in the device's embedded characteristics, energy consumption, and price. With the objective of keeping the devices affordable for the public and continuing calculations in a more resource-optimal environment, cloud computing has extended its services by adding nodes at the edge of the network; this is commonly known as Fog [1, 3] or Edge [11, 19] computing. In these new models, the cloud nodes bear fewer computational resources compared to traditional cloud nodes. However, their lower latency and network load are sufficient to meet the demands of user applications [8, 9, 17, 18]. Even though adding resources to edge support infrastructures solves immediate resource demand problems, these models still rely on infinitely scaling computing resources; as we saw with the micro-controller crisis last year, this is a fragile development model, economically and environmentally, that can be significantly impacted by future shortages and resource crises. It shall not be preserved in the long term.

Thus, if we try to improve cloud edge layer capabilities without deploying massive amounts of additional computing resources, the answer seems clear: we need to profit from the resources that are already being deployed elsewhere. IoT devices seem to be a major source of underused computing resources and are exponentially growing in number. Additionally, they mostly follow long sleep-wake periods. However, since these nodes are typically low-end nodes running on batteries (§2), they are often faulty and have limited network access; they need strong control mechanisms to be implemented over them to ensure platform safety and availability.

We propose a collaborative energy-efficient FaaS computing pltf. with low-to-medium availability targeting the execution of punctual stateless functions over the already deployed IoTs and gateways. In this paper, and as the central contribution of our distributed computing platform, we focus on its consensus mechanism PCM, which is similar to existing consensus algorithms (mostly Raft[2] and Pirogue[15]) but revisits some of their main characteristics:

(i) Node roles & classification: PCM divides nodes into two layers according to their energetic and computational capabilities: the gateway layer, which has few nodes responsible for the infrastructure management, and the leaf-node layer, which has many nodes responsible for the function execution. These second nodes are dynamically split into sub-computing groups, according to their availability constraints, to optimise task scheduling. Thus, PCM operates over a heterogeneous ensemble of layers (§4.1).

(ii) Leadership & Leader election: PCM also splits the leadership into two levels, with a dynamically chosen strong resource management leader and several imposed calculus coordination sub-leaders

(§4.2 & 4.3). PCM bases its leader election algorithm on global resource optimisation. It then selects a strong leader that minimises the number of messages exchanged and configuration changes, favouring the previous leader in an even vote. However, during the calculus coordination sub-leader election, which is fully managed by the aforementioned strong leader, PCM aims to globally optimise the performance and availability of each sub-cluster.

(iii) Membership changes & Fault detection: PCM's dynamicity and clustering mechanisms allow a response to the highly dynamic mobile IoT environment that causes each of the sub-networks, in case of partitioning or configuration, to change. Considering the aforementioned intermittent availability of IoT nodes, PCM implements an asynchronous connection maintenance and fault detection mechanism that allows distinction of a faulty node from an expectedly unreachable node (§4.3).

The remainder of this paper is organised as follows: §2 depicts our case study and context, §3 gives a background of classical consensus models and limits for the context, §4 discusses our proposal and §5 evaluates the PCM's performance in terms of computing performance, fault-tolerance and energy consumption. Finally, §6 has our conclusions and future research directions.

2 CASE STUDY: TOPOLOGY DESCRIPTION

In this section, we discuss the details of our case study topology, which consists of a wide set of IoT devices and their interconnected gateways forming a complex mesh network that is shown in Fig. 1. In this figure, this mesh network has a fully connected backbone between the gateways and a set of sub-star networks in which each gateway can obtain a group of IoT devices.

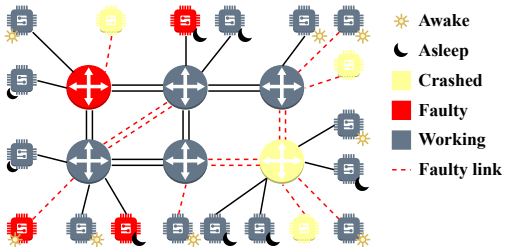

Figure 1: IoT / Gateway mesh topology example.

IoT devices can be characterised by their lack of embedded resources (computing, energy and storage), their intermittent availability and their faultiness. These nodes usually experience network interruptions and unexpected crashes that leave them unjoinable for long periods of time. These nodes may also experience byzantine faults due to low end characteristics or malicious attacks. The elevated number of IoT nodes already deployed represents an enormous mine of underused resources despite their characteristics. However, they need strong control mechanisms to run above them to compensate for their faults. Alternatively, we have the gateways that, as they manage a big ensemble of IoT nodes, are considerably less numerous but more resourceful and available than the IoT nodes; their availability is almost full. However, even though this may happen considerably less often than for the IoT nodes, they might experience network cuts, unexpected crashes or malicious attacks. These attacks have a higher impact than before because they usually deny the service of both the gateway and the IoT devices in its sub-network. Thus, being able to handle the errors and continue operating normally is critical for a cloud computing platform. Adding a consensus mechanism targeting this heterogeneity to safely profit from the IoT nodes' capacities is a strong requirement. This consensus model shall (i) ensure the system safety under both byzantine and non-byzantine faults that include network partitioning, delays, node unjoinability and packet loss, (ii) maintain the service availability while the system has at least one gateway and three followers, (iii) not depend on timing for synchronising the topology and executing the tasks, implementing an asynchronous communication middleware in-between and (iv) be completed as soon as a majority is achieved, being able to compensate the performance loss caused by slower nodes. Note that a detailed description of the hardware characteristics of the IoT-like and Gateway-like nodes used in our experiments, as well as the precise topology, is presented in §5.

3 CONSENSUS BACKGROUND

Consensus algorithms were born from the necessity of bringing non-trivial solutions to reach an agreement between multiple machines in the most effective and reliant way and reinforce the system's fault-tolerance. These algorithms, through the use of multiple processes and mechanisms, ensure a single output value between a set of possible values proposed by different machines (referred to as a *node*) on a machine ensemble (referred to as a *cluster*). Hence, to establish a background and comparison criteria for our model, we highlight in this section the most iconic consensus models (i.e., Paxos and Raft), analysing the majority of these algorithms and some of their variants.

Paxos [4, 6] works by allowing a cluster of nodes to reach an agreement over a value, synchronising these values and making them learnable for all the nodes in the cluster. To do so, Paxos classifies the nodes into three roles (proposer, acceptor and learner); it is possible for a node to act as more than one at the same time. To simplify the vote counting and to prevent the desynchronisation between the nodes when replying to different simultaneous consensus requests, this protocol implements a two-phase process. The process is composed of (1) a promise phase and, once the majority of nodes reach an agreement over a value, (2) a commit phase, during which the proposer determines the final answer to the request and propagates the result to all the cluster nodes to ensure all the infrastructure gets synchronised. This model has been widely criticised because of its complex applicability to real distributed systems due to the system's heterogeneity and cluster node constraints. Multiple propositions (such as Fast [5], Flexible [13], Fast-Flexible [14] or multi[12] Paxos) have appeared to try to enhance this adaptability and to lower the cost of the consensus algorithm. However, they were still too complex for IoT powered systems. Having the objectives of reaching performance levels similar to multi-Paxos and easing the understanding and implementation of the model, along with the adaptability to the real distributed systems, Raft[2] appeared as a Paxos alternative when managing replicated logs through all the nodes in the cluster. Even though Raft focuses on the data replication storage instead of the task execution, the way to reach agreements over the synchronisation and integrity of the data stored relies on the same principles as Paxos and our solution.

Thus, to simplify the task distribution (i.e., system coordination or topology maintenance), Raft proposes three different exclusive roles: (a) leader, which will be in charge of coordinating all these system management actions and requests and guaranteeing coherence and organisation and (b) follower, passively listening and following the leader's orders. A third temporary role, (c) the candidate, can happen during the leader election process. In addition to this power centralisation on the leader, and as a way to resolve inconsistencies in the synchronisation of the log tables, Raft bases itself on the logical clocks [7] by adding a counting number (known in Raft as a *term*) that allows the detection of obsolete information and the restoration of the nodes' log table. However, despite the simplifications and optimisations brought by Raft, the complexity and the cost of the voting process on which this model relies harden its applicability to the IoT highly mobile and faulty networks.

As a way to deal with this complexity, Pirogue [15] presents himself as a Raft variant, which is the closest to our consensus model, focusing on reducing its high energy footprint with a similar performance. In this model, the authors propose a change in the voting process from the classic static voting process to a dynamic- linear voting algorithm [20]; this substitutes the classic static quorum majority into a dynamic quorum, needing less nodes to achieve a majority if there are server disconnections. Pirogue proposes a new role: the witness, a role able to participate in the consensus quorum in case it is needed. The role is implemented by low-power devices not having the capacity to store the whole system's data, reducing the energy footprint of the cluster. All these optimisations mentioned above allow Pirogue to achieve performance, availability and fault-tolerance close to the one in a five-node Raft cluster with lower energy footprint configurations such as four-node or three-node-one-witness clusters.

However, despite the efforts to achieve more lightweight consensus models, to the best of our knowledge, no solution allows for performing fault-tolerant consensus asynchronously in complex heterogeneous mesh topologies over highly restrained IoT devices. We also have not found a consensus algorithm over highly faulty nodes nor intermittent available nodes.

4 THE PCM CONSENSUS ALGORITHM

PCM is an algo. for managing a replicated infra. topology and orchestrating fault-tolerant execution of stateless tasks over often faulty nodes. Fig. 2 summarises the props., rules and states guaranteed by the each of the roles at all times; the elements of these detailed tables are discussed piecewise in the rest of the section.

Given the differences between the gateways and IoT nodes, both in terms of computing capabilities and availability, PCM implements consensus in two levels. First, at a gateway level, by dynamically electing a strong leader referred to as the Resource Manager (RM). This leader is given complete responsibility for managing the replicated infrastructure topology and sub-clustering of the IoT nodes into Calculus Units (CUs). The RM also accepts new task execution requests, assigns them to a CU and re-transmits the final calculus answer back to the client. Having this centralised strong leader simplifies the management of the highly dynamic routing table. For example, the RM can determine the IoT sub-clustering and new device addition without consulting with other servers, having to only propagate this information once the decision is taken. Even though gateways' failures are considerably less frequent than those of the IoT nodes, an RM can fail or disconnect from the network, in which case a new one is elected. PCM also implements a consensus layer at the IoT level, more precisely inside each CU. In this case, a set of weak dynamic leaders is elected by the RM; they are referred to as Calculus Orchestrators (COs). However, the additional responsibilities of these leaders are limited to collecting the cluster answers to an assigned request, then operating a consensus over the final result through a voting process. Thus, PCM decomposes the consensus problem into four relatively independent sub-problems, which are discussed in the following subsections:

(1) RM Election. A new RM must be chosen when an existing one fails or there are none in the system (§4.2).

(2) Topology management & segmentation. The RM must be able to manage the IoT device dynamicity, replicate the current status across the rest of gateways and optimally update the dynamic CU distribution (§4.3). The RM shall also be able to optimally choose the optimal set of COs.

(3) Task Execution. The RM must handle the client task requests and pick the best available CU to offload it. After that, the IoT nodes in the cluster must operate independently and reach an agreement over a final answer (§4.4).

(4) Safety. The key safety property for PCM is the Topology and Task Tracking Safety Property, which enables the availability of the infrastructure without data loss. Thus, if any server has applied a particular topology entry to its state machine, then no other server may apply a different command for the same index and era. §4.5 describes how PSM ensures the safety of each PCM mechanism.

4.1 PCM basics

A PCM infrastructure contains many gateways and IoT devices. They are sub-clustered into CUs. At any given time, each IoT device will be in one of two states: *Coordinator orchestrator* (CO) or *Follower*. The dispatchment of the followers into CUs and the election of the CO are done centrally by the RM. The IoT nodes are only informed if the status changes. While followers are passive, they issue no requests on their own and simply respond to the execution tasks assigned. The CO is in charge of collecting the request answers and operating a consensus over the final answer. In the gateway layer, the nodes can be in three different states: *Resource manager* (RM), *Dispatcher* or *Candidate*. Being passive, their only goal is to maintain updates of their Global Topology Table (GTT), Subnetwork Follower Topography Table (SFTT) and Task Tracking Table (TTT). The RM will handle all the client requests (if a client contacts another gateway, it will be redirected to the RM) and coordinate the synchronisation and collection of both topology and task execution state machines across the rest of nodes in the infrastructure. The last state, Candidate, is used to elect a new RM as described in §4.2. The possible node states and their transitions are deeply discussed in further sections.

Similar to the Raft-based models, PCM divides time into periods of variable length referred to as *eras*. Eras will be numbered with a consecutive integer, and they will end with an election. In the election, one or more dispatchers will calculate the optimal emplacement for the next RM. Since the RM election is based on the infrastructure resource optimisation, this election process can be triggered in different scenarios that we will detail in §4.2. Once a

State

Follower state :

Static *(Updated on stable storage before joining the network)*

duty-cycle	The periodicity definition in miliseconds with the tuple <Active, Busy, Asleep>. Where the addition of the three shall be smaller than 120s.
lib-cache-size	Maximum size dedicated to the library caches.
entry-points [ ]	List of gateways to which it can connect a follower.
Era & GTT index	Numbers allowing to order requests.

Persistent *(Updated on stable storage before answering a request)*

calculus unit id	Id of the calculus to which it belongs at a given time.
lib-cache [ ]	Contains a set of libraries already used to fasten up the task execution in the future. The RM is also aware of this list for each follower.
am_i_CO	Boolean indicating if the node is the cluster's calculus orchestrator.

Volatile *(Reinitialized after giving the final request answer)*

last_function [ ]	In this list it is stored the execution result for the last function offloaded. This list is also used if the node is the CO to store the follower votes before calculating the final answer.

Dispatcher state :

Static *(Updated on stable storage before joining the network)*

broker_port	The port to access the MQTT broker.
RM_timeout	RM Timeout definition.

Persistent *(Updated on stable storage before answering a request)*

gateways [ ]	Initial set of known gateways. Every time a new gateway is discovered is appended to this list.
Sub-network Follower Topography [ ] (SFTT)	Table used to keep track of the followers and their characteristics.
Global Topology Table [ ] (GTT)	Table synchronised by the RM exclusively formed by the merge of the SFTTs of all dispatchers.
gtt_table_index	Table clock-like index inside an Era. This is used to detect a most synchronised GTT.
Task Tracking Table [ ] (TTT)	Table containing the tasks to be executed, their execution status and, for a limited sized, a cache of the task executed. It is updated and sync by the RM. It can be used to reply to client requests already in cache by non-RM dispatchers.
ttt_table_index	Table clock-like index inside an Era. This is used to detect a most synchronised GTT.

Volatile *(Reinitialized after giving each leader election)*

RM_id	Name of the RM id. The node will behave as the Resource Manager (RM) if this id is his.
Era	Similar to raft terms.

Rules for Nodes

All servers :

- When receiving a new message, either containing a GTT, a TTT or a configuration change for the followers, if the *era* > *local_era* or *index* > *local_index*, the receover will overwrite the concerned state and refresh the concerned indexes.

Followers :

- Execute the task offloading requests assigned by the Resource Manager (RM) and send the result to the Calculus Orchestrator (CO) asigned to its Calculus Unit (CU).
- Consume as many messages from their MQTT pile as possible when active. However, before any of that, these nodes always start by sending a keepalive message to their gateway to refresh their availability.

Calculus Orchestrators :

- Collects and merges the answers of the rest of followers in the Calculus Orchestrator, and itself. Whenever it has enough votes to take a decision, it will end up the voting process and return the final answer, droping all the rest of the follower answers.
- The Calculus Orchestrator for each of the clusters is chosen dynamically by the RM and is, without exception, the most available node in the cluster.

Dispatchers :

- If there is a RM in the architecture, they respond to its requests with the objective of keeping the GTT and TTT on sync and serving the client requests without service cuts.
- However, if an RM election timeout elapses without receiving any request emitted by the RM, the dispatcher will increment by one its current era and execute the RM election algorithm described in Figure 4, informed the most optimal candidate of this process.

Candidates :

- On conversion to candidate:
  - Increment the current Era and reset the GTT and TTT index to 0, which will contain its locally stored data.
  - Trigger the election algorithm in the rest of dispatcher
  - Reset the election timer
- If he receives back votes from majority of dispatchers: becomes RM
- If receives a GTT message from a node claming to be a leader for the same or higher era: go back to dispatcher state.
- If election timer elapses without a majority: the candidates merge their GTTs, increase the era and recalculate the optimal RM.

Resource managers :

- Upon election: It will have the duty of maintaining the system in sync, both for the TTTs and the GTTs, orchestrate the followers in calculus units, dynamically select the COs and establish its authority and prevent new unnecessary elections.
- If a SFTT is received by a dispatcher it compares it to the GTT and, if there is something different, merges the changes to the GTT and increment the gatt_table_index. These changes will then be periodically propagated and update the rest of dispatchers in the system.
- If a task is sumbitted by a client to the RM, it appends it to the TTT and assigns it to a cluster. If it does not receive an answer before the offloading timer elapses, the task gets asigned to a different cluster again. Once the result received, it sends it back to the client and adds it to a FIFO-based cache to serve future similar client requests.

Properties guaranteed

- **Election safety :** at most one RM can be elected in a given era. At most one CO per calculus unit can be chosen in a given gtt_table_index.
- **Topology & Task tracking safety :** if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index, being the index formed by the era and the gtt_table_index / ttt_table_index in the dispatchers and by task_id, its hash and the ttt_table_index for the followers.
- **Cluster exclusiveness :** a follower will always be part of exactly one calculus unit. Also, a task will always be ofloaded to exclusively one CU simultaneously.
- **CO Append-Only :** A CO never overwrites or deletes entries in its results log before giving a final answer; it only appends new entries to the list. Whenever the voting ends, it completely deletes its log and overwrites the last function result with this last.
- **Table Matching :** If two tables, either the GTT or the TTT, contain an entry with the same index and era, then the tables are identical in all entries.
- **RM Completeness :** If a new node or task is commited in a given era, then that entry will be present in the logs of the dispatchers for all higher-numbered eras and table indexes, unless the node exceeds the disconnection threshold or the task gets of the TTT on a FIFO basis.

Figure 2: Basic properties, rules and states for the different roles.

new dispatcher is found optimal, it will become a candidate. The rest of the nodes will verify if it is indeed the most optimal dispatcher. When this verification is done, it will become the new RM. Since the RM election process is calculated following a deterministic algorithm, split votes are unusual. However, if this happens, the split nodes will merge their GTTs and re-run a second tour election, increasing the era once again by one. The different dispatchers may observe the transitions between eras at different times, and they may not always be aware that one or multiple elections happened while it was off the network. Thus, eras act as logical clocks, allowing servers to identify which information is most recent. This same sequencing mechanism is used inside an era to identify the changes on the GTTs or TTTs, which are called gtt_index & ttt_index. Each dispatcher will then store the RM era and the table indexes which increases with the different messages exchanged, forcing then every server to update itself to the highest era and index when one its found. At the same time, if an RM or a candidate is "out-of-date", it assumes that it has temporally lost the network connection. It will sync to the RM with the highest era and index and trigger an election, returning to the dispatcher state. This same mechanism works in the opposite way when receiving an elder era number from another dispatcher.

Considering the intermittent availability and dynamicity of the IoT nodes, PCM nodes communicate by using asynchronous remote procedure calls (RPCs) through a mesh of asynchronous message brokers deployed over the gateway nodes. The mesh is used as a mail- box for the devices in its sub-network. Messages will then be delivered a maximum of one time to the recipients in order to enhance the performance. The gateways might clean the message piles of the IoT nodes to erase the past unconsumed messages that came while they were unavailable. It is then useless to retry RPCs unless they have an error when joining the asynchronous broker. The message will be preserved until consumed or erased. To increase the performance and protect the system against loopholes, we have implemented some timeouts for the different messages that we will discuss in the following sections.

4.2 Resource Manager Election

PCM uses a modified heartbeat mechanism to trigger the RM election. When a gateway starts up, it begins as a dispatcher spreading its SFTTs for node advertising and remains in this state as long as it receives a valid RM or candidate advertising message. RMs send the GTT table (Fig.3) to all followers in order to maintain their authority and keep all the nodes in sync. However, if a dispatcher receives no communication over a period of time (which is known as RM election timeout), it assumes there is no RM nor election process happening and begins a new election to choose a new RM. To begin an election, the dispatcher will increment (by one) its current era and execute the RM election algorithm described in Fig.4. The input used is the most updated version of the GTT, taking off the disconnected RM. With the goal of optimally choosing the emplacement of the RM (and by minimising the quantity of messages to be exchanged), PCM uses a deterministic leader election algorithm. Afterwards, every dispatcher that exceeded the RM election timeout will run this algorithm and inform the output dispatcher that, to the best of its knowledge, it is the most optimal RM for this new era. If a node receives one of these messages, it will try to reach the

Async Advertisement

Invoked by any node to inform about its internal properties, follower / CO, or replicate log entries for both GTT and TTT, dispatchers / RM; also used as heartbeat.

Arguments:

kind	It can be (1) follower to dispatcher to inform about internal properties, (2) GTT replication, (3) SFTT propagation & (4) TTT replication.
< sender, dest >	Sender node id and destinatary node id.
era	The era of the informmation.
index	Index of the information. This parameter takes the gtt_index value if (1-3) & the ttt_index if (4)
payload	This field changes completely depending on the message *kind* attribute. If (1) the payload contains, the follower duty_cycle, if (2) it contains the GTT, if (3) the SFTT and if (4) the TTT.

Receiver implementation :

1. If term < local.term or (term = local.term & index < local.index) information is not up to date, refuse the package and send your GTT or TTT to the node.
2. If more recent, synchronise your internal information concerned with the received package information. Append any new entry not already in the log and erease all entry no longer inside.
3. If a message of any kind arrives, refresh the node joinability.
4. If an existing entry conflicts with a new one, delete the existing entry and all that follow it.

Figure 3: Asynchronous advertising message options.

Leader Election Algorithm

Invoked by any dispatcher exceeding the RM timeout or having received a preliminary RM election vote request.

Initial: GTT containing a list of dispatchers, each having a list of nodes connected and its capabilities, but without the disconnected RM.

External functions: **disp1.calcMessages(disp2)** calculates the number of messages expected to be exchanged between the dispatcher 1 and the dispatcher 2 if the dispatcher 1 was the RM. It does not consider the re-transmissions needed, which need to be added to the total calculus.

Implementation :

```
optimalDispatcher ← null;
optimalNumberOfMsgs ← null;
for dispLeadTmp ∈ GTT do
totalMessages ← 0;
  for disp ∈ GTT do
    expectMsgs ← dispLeadTmp.calcMessages(disp);
    totalMsgs ← totalMsgs + expectMsgs ∗
                distance(dispLeadTmp, disp);
  end for
  if totalMsgs < optimalNumberOfMsgs (or null) then
    optimalDispatcher ← dispLeadTmp;
    optimalNumberOfMsgs ← totalMsgs;
  end if
end for
return optimalDispatcher
```

Results:

new era & index	current era and index, for candidate to update itself .
optimal RM	id of the most optimal dispatcher to become RM.

Figure 4: Resource Manager election algorithm.

current RM. If the node is unable to do so, it will trigger an election in the rest of dispatcher nodes that did not already make a vote for it. Afterwards, it will switch to the candidate state until one of three things happens: (a) it wins the election, (b) another node wins the election or was already the RM or (c) a period passes with no winner. These outcomes are discussed separately below.

(a) A candidate wins an election only if the majority of available dispatchers in the infrastructure agree that it is the most optimal RM. Each server will vote for exactly one candidate each election, which, given the deterministic behaviour of the optimal RM calculation algorithm, should always be the same if they are in sync throughout the network; this never leads to a split vote. However, to ensure the correctness of the system, the majority rule ensures that only one candidate can win the election for a particular era (Fig. 2). Once a candidate wins an election, it will become the RM until another election is triggered. It will then have the duties of keeping the system in sync (both for the TTTs and the GTTs), orchestrating the followers in CUs and dynamically selecting the COs.
(b) While waiting for votes, a candidate may receive a message from another candidate claiming to be the RM. If this RM's era is at least as large as the candidate's current era, the candidate will recognise it as a legitimate RM and return to the follower state. However, if it is smaller, the message will be rejected and the candidate will remain in the candidate state.
(c) At the end of the election time threshold, neither of the candidates have enough votes to win or lose the election; this could happen if there were multiple network fractures not allowing the dispatchers to have a global up-to-date sync table. When this happens, each candidate will send its GTT to the rest of candidates. This will merge the GTTs and re-execute the RM election algorithm, increasing the era. This time, it is confirmed that all the nodes participating share the same GTT, and thus the same output. Afterwards, once a candidate wins the election, it will propagate the decision to the rest of the followers.

Until now, we have believed that an RM elect. always happened because of a node disconnection. However, it can be triggered with the objective of readjusting the RM position. In that case, the RM node will calculate the optimal position. If it is no longer the best candidate, it will increase the era and determine its successor, propagating its choice to both the new RM and the rest of dispatchers in the system. This *succession* mechanism allows us to fasten the election process and add less system overhead while keeping the RM placement optimal.

If a dispatcher receives a message from another node, RM or dispatcher at any point with a higher era or table index (or with the same era or table index coming from a different RM) while already having an active RM itself, this would most likely be because of either malicious behaviour or a network segmentation. In that case, the dispatcher will alert its RM, which will merge its GTT with the other RM and re-trigger an internal RM election, propagating the election results and bringing all the nodes to a consensus. Note that if one of the candidates is unjoinable during an election, the era will increase by one, and the election starts all over again with the new updated table. Even though this could last longer than the RM election process, given the characteristics of the gateway nodes, this situation does not happen often.

4.3 Topology management & segmentation

4.3.1 Topology management replication. With the objective of reducing the network congestion and the global number of messages, topology management for each sub-network is also delegated to its gateway. Thus, each gateway, despite its role in the infrastructure, will track the IoT node's heartbeat (Fig.3-1), containing all its

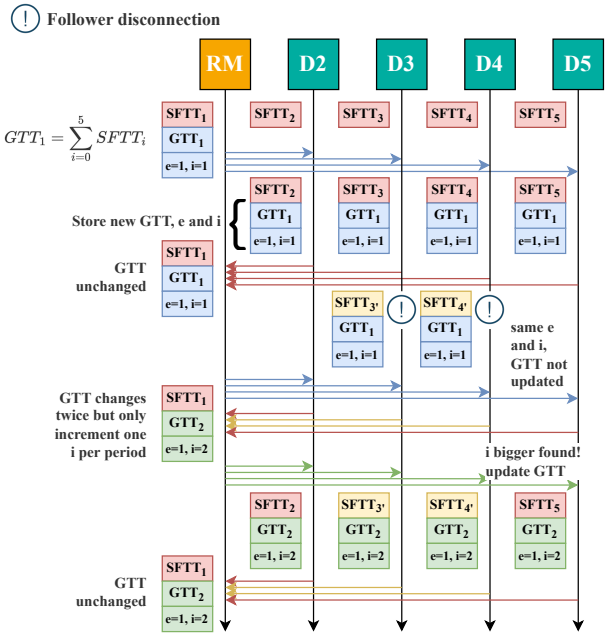

Figure 5: GTT and SFTT synchronization example.

availability properties, storing them into its SFTT. Each gateway will then track the faults of the IoT nodes in a flexible manner since each one has a different availability and periodicity. Thus, if a gateway does not receive a heartbeat or task-execution message from an IoT node in its sub-network over a certain period, it assumes that the node got disconnected and purges it from its SFTT. This disconnection detection period, dynamic and independent for each IoT node, is then equivalent to the 1.5 connection periodicity, which is at least one second. On the other hand, as a way to coordinate and replicate each of the sub-network tables throughout the rest of the dispatchers in the infrastructure, PCM implements a replicated state machine layer. As shown in Fig.5, this process starts with the periodic GTT spread message (Fig.3-2) by the RM. Once this message is received by another dispatcher, it will compare its era and table index with those previously stored and save the most updated version. Once this is done, the dispatcher will reply to the RM with its SFTT (Fig.3-3), which it will compare to its locally previously known information about the same sub-network. If the RM finds a change, it will merge the changes into its GTT and increase its index. The index will only increase by one each period no matter how many changes are detected. Finally, since this mechanism relies on the comparison of the SFTTs and GTTs, it is critical to keep the comparison as lightweight as possible. However, the sizes of these two tables are different. The SFTT has a list of a dozen nodes, and the GTT has a list of dozens of SFTTs. Thus, the GTT tables will get verified by comparing the tuple <era, table index> to the one locally stored, which allows us to identify which table is more recent. However, for the comparison of the SFTTs, given the high dynamicity of the nodes and the reduced size of the list, a more extensive verification is required.

4.3.2 Follower CUs clustering. Having a distributed full sync GTT between the gateways, which allows us to use any gateway as a backup RM in case of disconnection and detect the disconnections in both layers, we can explore the second use of the GTTs: the

clustering of the IoT nodes into CUs. As shown in Fig.1, the IoT nodes are characterised by their intermittent availability. The perf. of each CU is highly dependent on receiving at least half of the CU follower votes as quickly as possible so that the CO can execute the voting algorithm (§4.4). As a result, PCM proposes a clustering algorithm that tries to minimise the gap between the available periods of the nodes. In this algorithm, the RM will first classify the nodes into performant and not performant. More performant means the nodes have higher availability. However, given the constrained capacities of the IoT devices, and in order to lose the lowest amount of calculus periods possible, this classification algorithm also looks at the current clustering distribution to order the nodes. It attempts to make as few insignificant changes as possible. It will then calculate the number of clusters needed (N) according to the desired cluster size (M), which is optimally set to five nodes per CU. Afterwards, it will distribute the (N) fastest nodes. One is assigned to each cluster, which will be designated as the CO. Finally, it will end the fulfilling of the cluster by taking two performant nodes and two low performant nodes, appending the CU clustering to the GTT and informing the followers of the changes. This way, we can enhance the number of calculus periods as well as homogenise the infrastructure capacities. However, the algorithm will look each time to the precedent position of the node and, if still in the same group of nodes, this position will not be modified.

This node clustering process is highly resource consuming and adds a considerable overhead to the system. Given the high dynamicity of the IoT networks, we cannot execute this algorithm every time a node disconnection happens without drowning the system's performance. In addition, this clustering process will only take place in three scenarios: (a) any CU has less than one CO and a performant node, (b) the number of nodes in the infrastructure is smaller than $N \cdot 3$ and (c) periodically to re-optimise the node distribution. The RM will track the IoT devices to ensure that these rules are always followed. To keep the infrastructure as performant as possible, palliating the node losses, the RM will always prioritise the task offloading to the CUs with all their performant quorum active (§4.4). Finally, in order to optimise the system performance and safety, two mechanisms are implemented to manage the intra-CU disconnections. To begin, if we focus on the follower's disconnections, the RM will inform the CO whenever a change occurs to its CU that is related to the total cluster size for its CU; this is done to adapt its local DLV. On the other hand, whenever a CO disconnection happens, the RM needs to choose another CO from within the nodes in the cluster to be operative until a new re-clustering happens. This way, each CU can tolerate one node disconnection out of the performant quorum and up to three disconnections if two of them are non-performant nodes.

4.4 Task execution

Once an RM has been elected, the CU clustering has been done and the basic topology maintenance tools have been ensured, it begins serving client requests. Each client request contains a stateless function to be executed by the IoT nodes, which is referred to as a Function Execution Req. (FER). The RM checks for the presence of the function in its local cache. If it has already been requested and matches one of the FERs stored in the cache buffer, it will directly return the past result and refresh it. However, if a pre-existing entry

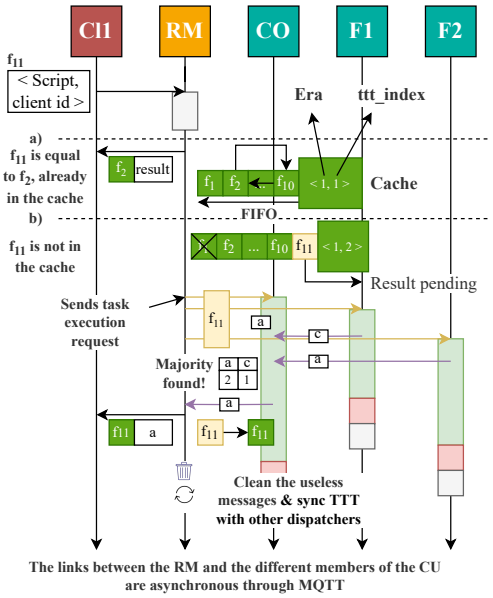

Figure 6: Task execution example.

cannot be found, it appends the new FER to the TTT as a new entry, increments the TTT table index and issues an asynchronous FER to the best available CU; this will prioritise the CUs with less faulty nodes at a given time. When the CO of the assigned CU answers to the FER after handling the voting process (as described below), the RM applies the entry to its local TTT, returns the result to the client and synchronises the TTT with the rest of dispatchers. The process of CU task execution is depicted in Fig.6.

The FER gets stored in the RM in a limited sized cache following a first-in-first-out policy, as shown in Fig.6, that is synchronised among the different disp. in the same way as the GTT (§4.3). Each FER is generated directly in the client side with the use of the PCM parser and is composed by the function script to be executed and the needed libraries to execute it. As in the GTT spreading mechanism, whenever a new FER reaches the RM, it will be added to the cache and increase the table index by one. In addition to that, whenever a new RM is found, the era will be increased by one. Both the era and the table index allow the different disp. to detect inconsistencies between logs and stay updated at a low computing cost, ensuring the system safety and guaranteeing the props. from Fig. 2.

If we focus now on the CU execution and voting process, the CO decides when it is safe to give a final answer to the request. Once any node in a CU receives a FER message, they will run the given task and send their result back to its CO. The CO, that is aware of the CU status, will verify each time if it is able to achieve a majority vote. To do so, instead of using the classic static quorum voting algorithms, we implemented a variant of DLV [20]. DLV will then dynamically re-set the majority threshold depending on the number of connected nodes, which allows us to enhance the system flexibility and performance. In case of a tie, the CO will always prefer its answer or, if its answer is not among those tied, the first answer that reaches the tie quantity. PCM-DLV requires at least n + 2 servers per cluster to be able to tolerate n failures instead of 2n + 1 required by Raft. This property allows the CUs to

operate with less members with an acceptable fault tolerance level, which is critical in a faulty and highly dynamic environment.

4.5 Correctness argument

Given the complete PCM algorithm, we can now discuss now its safety and liveness. The correctness of PCM is verified by proofing the correctness of the different mechanisms forming it, ie., (4.5.1) RM election, (4.5.2) GTT & TTT synchronization, (4.5.3) CU clustering and CO election and (4.5.4) task execution.

4.5.1 RM Election argument. If we start by analysing the RM election correctness, as presented in §4.2, this mechanism can be triggered in three different scenarios and can bring two different outputs. In normal conditions, the safety of the mechanism can be ensured by Property 1 in Fig. 2: *"election safety"*. However, in the case in which the nodes are not able to achieve a majority because of missynchronisation of their GTTs, the correctness of the algorithm depends on the correctness of the GTT merge (§4.5.2) and the limited quorum voting process, which can be as well ensured by the same aforementioned property. On the other hand, the liveness of the protocol is verified by the deterministic characteristics of the RM election algorithm and the fact that any leader will be elected with less than $\frac{N}{2} + 1$ votes. Thus, the order in which the voting takes place is irrelevant and only impacts the perf. of the algorithm.

4.5.2 GTT & TTT Synchronization argument. We will focus on the correctness of the second mechanism, starting with the GTT (which is a single variable state-machine) and followed by the TTT, which is a limited sized state-machine. The correctness of both is verified by Properties 2 and 5 from Fig.2: *"Topology & Task tracking safety"* and *"Table Matching"*. This is why we are sure that, for a given era and index, tables are the same and are spread among all the nodes in the net work under a specific RM. The GTT & TTT merging mechanism is like Raft's log compaction mechanism, which has already been proven correct, in which we will take the previous entries of the different tables we want to merge and (in case of the GTT) append them. For the TTT, instead of appending the tasks, we will preserve the pending tasks. The liveness of this mechanism can be ensured by the heartbeat synchronisation mechanism, in which the dispatchers send their SFTTs and the RM acknowledges them with the missing chunks of its GTT / TTT table. Thus, even if a new follower device registers after the message is sent by a given dispatcher, its inclusion in the GTT will only be delayed until its next sync heartbeat.

4.5.3 CU Clustering and CO Election argument. This mechanism correctness and liveness depend on the GTT synchronisation correctness (§4.5.2) and the correctness of the clustering deterministic algorithm, which is computed strictly on the RM and spread to the rest of the nodes. The correctness of this is easy to prove since it is a simple list order depending on a set of criteria.

4.5.4 Task Execution argument. The correctness and liveness of this mechanism depends on the DLV mechanism (proven in the literature), the clustering and CO election (§4.5.3) and the TTT synchronisation mechanism (§4.5.2), which were discussed previously.

5 EVALUATION

In this section, we have implemented the PyCloudIoT FaaS Edge Computing platform. We evaluate the impact of PCM and its different possible configurations by putting the focus on the trade-off between performance, energy consumption and fault tolerance. We have used two different experimental set-ups. The first one is composed of 3 to 15 ESP32-MINI-1U nodes as the IoT nodes and a Raspberry Pi 3B as the gateway, with the objective of finding the optimal parameters and discussing their impact on the system performance to compare with the state of the art. Note that these nodes were orchestrated similar to Fig. 1. The second one mixes real and emulated nodes to discuss the overhead evolution with the platform scalability. Note that the ESP32-MINI-1U has a ESP32-U4WDH Single Core processor with 160 MHz core clock maximum frequency and 4MB Flash packaged in chipset. The Raspberry Pi 3B bears a Quad-Core ARM Cortex-A53 processor with 1.3GHz core clock maximum frequency and 1GB RAM.

For the evaluation benchmark, we have used an intensive calculus benchmark based on NumPy[1] that we adapted to the micropython operating system and tagged with some internal parsing flags to coordinate the execution of the task and its previous requirements. Note that you can find the code of the project in my github repository [2]s. All the results presented follow normal distributions with a standard deviation smaller than 2%, which allows us to use only the mean of the data to enhance the readability. All the fault injection models in this paper follow a pseudo-random distribution (cf. Mersenne Twister [16]) simulating byzantine faults. We used energy models described for the ESP32 in [21] having $V_{node} = 3.3V$, $I_{active} = 0.260A$ and $I_{sleep} = 2.5 * 10^{-6}A$, as stated on the ESP32 technical documentation, and for the Raspberry PI 3B nodes in [10] that we have considered constant in order to simplify the calculus while setting the value to the mean energy consumed by these nodes when doing the tests, which is of around 2.5923 W (considering a mean CPU load of almost 50%). Finally, in this section, the duty-cycle represents the time that the node cannot be used by PyCloudIoT and comprises both the time it is asleep and the time it is doing its main tasks.

a) Study of the CU-size impact. In this first subsection, we study the impact of varying the CU size on the aforementioned trade-off, cf. Fig. 7. For the middle and left portion of Fig. 7, we want to study the evolution of the performance and the complexity of the calculus when executed on different sized CUs. In this case, the maximum complexity is limited by the hardware constraints of the follower nodes. From this experience, we can highlight two facts. Firstly, the efficiency when executing low-complexity short tasks is questionable in terms of energy consumption and performance (regardless of the CU size) compared to the local execution due to the high overhead of the system maintenance and the transmission process compared to the length of the task. However, this gets more efficient with the task length, being less than 30% slower in medium-high complexity tasks. There are considerable energy savings (more than 50%) if you compare a 3-node CU and the local execution of the same function. Secondly, the performance and energy consumption do not evolve linearly. The performance gain when passing from 3 to 5 nodes in the CU (15%) is considerably more significant than when passing from 5 to 7 nodes (5%); there is an energy consumption decrease with an energetic overhead of 17% between having 3 or 5 nodes in the CU and of 20% between

[1] https://github.com/serge-sans-paille/numpy-benchmarks
[2] https://anonymous.4open.science/r/PyCloudIoT-BD64/README.md

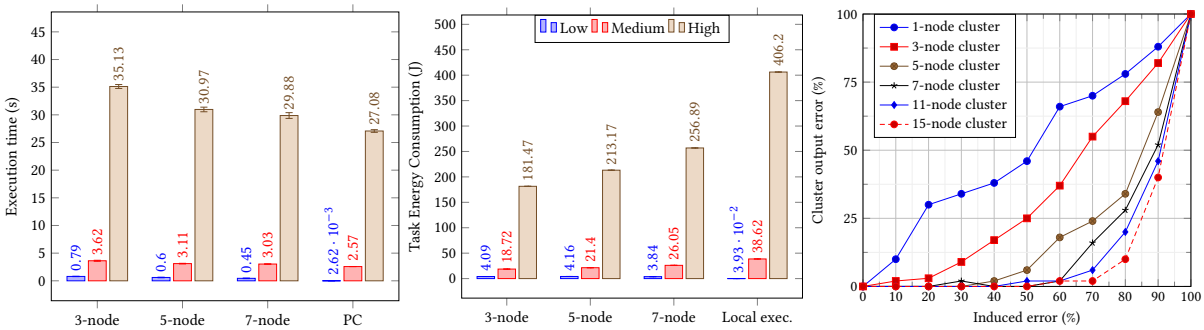

Figure 7: Study of the cluster size.

5 and 7. Finally, if we look again at Fig. 7, we can appreciate that the fault-tolerance gain with the cluster size is considerable, the evolution of this gain confirms that the 5-nodes configuration is once again the most efficient. Thus, we consider 5 to be the optimal CU size from now on.

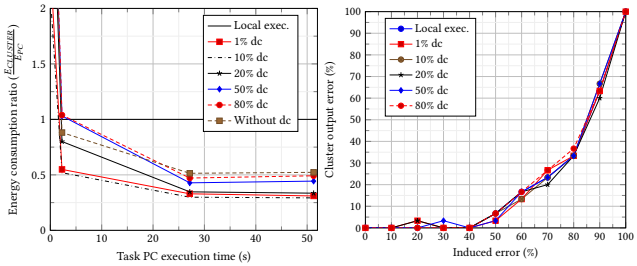

Figure 9: Energy and fault tolerance with duty cycle.

b) Study of the duty-cycle impact. In this second subsection, we focus on the availability of the follower nodes and their influence on the system. To simplify the comparison for this test, we suppose that all the followers have the same duty-cycle, even though they are not necessarily aligned between them. From Fig. 8, in which we compare the performance ratio between the different functions of the benchmark compared to a local execution for high-complexity parameters (for stability), we can conclude that the system performance (even though it mostly converges on a threshold for each of the duty-cycles when the tasks are long enough), the impact of the duty-cycles on the performance and the energy (cf Fig.9) are not linear; this indicates a significantly higher performance gain between 1% and 10% duty-cycle configurations (~ 15% performance) than between 10% and 50% (≤ 10%) or even between 50% and no duty-cycle (≤ 10%), which makes the transition from 1 to 10% the only configuration in which we increase the performance gain and reduce the energy consumption at the same time. Finally, if we look at the right portion of Fig. 9, we can see that the fault tolerance is not impacted by the availability of the IoT nodes in the CU.

c) Study of the RM election mechanism. In this section, we will deepen the understanding of the performance of the RM election mechanism detailed in 4.2. Fig. 10 shows the system overhead induced by this mechanism depending on the number of IoT nodes and gateways present in the system. This way, we can state the limits of our infrastructure concerning the number of IoT nodes on the same infrastructure. We can extract two conclusions. Firstly, due to the fact that each dispatcher handles its subnetwork, the number of IoT nodes barely impacts the RM election algorithm overhead; the verification of the routes and different possibilities are responsible for most of the impact ($o(N^2)$ complexity). Secondly, we shall keep the number of dispatcher nodes in the infrastructure as small as possible, using followers as the gateways if there are too many. Thus, the time it takes the system to detect, orchestrate an RM election and get back to the operative state depends on the number of dispatchers in the architecture. For our test scenario this process took a mean of $1.06 * (RM\ disconnection\ threshold(RMDT))$ over more than 1000 thousand tests, process taking the fastest execution $0.92 * (RMDT)$ and the slowest $2.27 * (RMDT)$.

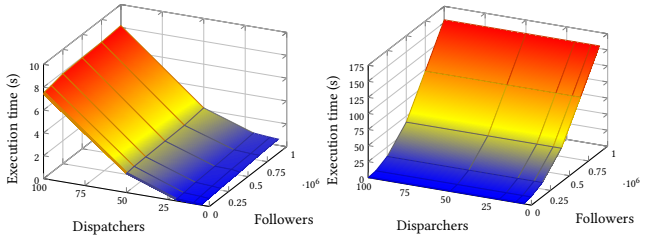

Figure 10: RM election algorithm (left) and CU clustering & CO election (right) execution time.

d) Study of the CU clustering and CO election mechanism. For this second mechanism (§4.3), as you can see in Fig. 10, the number of dispatchers has no impact on its performance. On the other hand, the number of followers considerably raises its complexity.

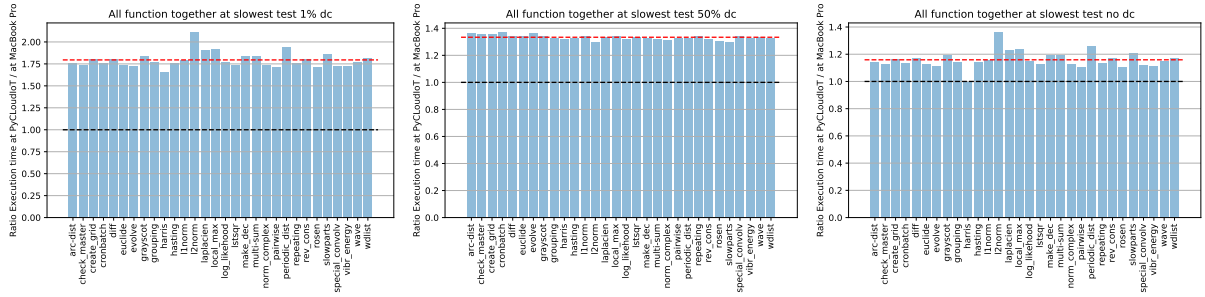

Figure 8: Study of the duty-cycle performance homogeneity.

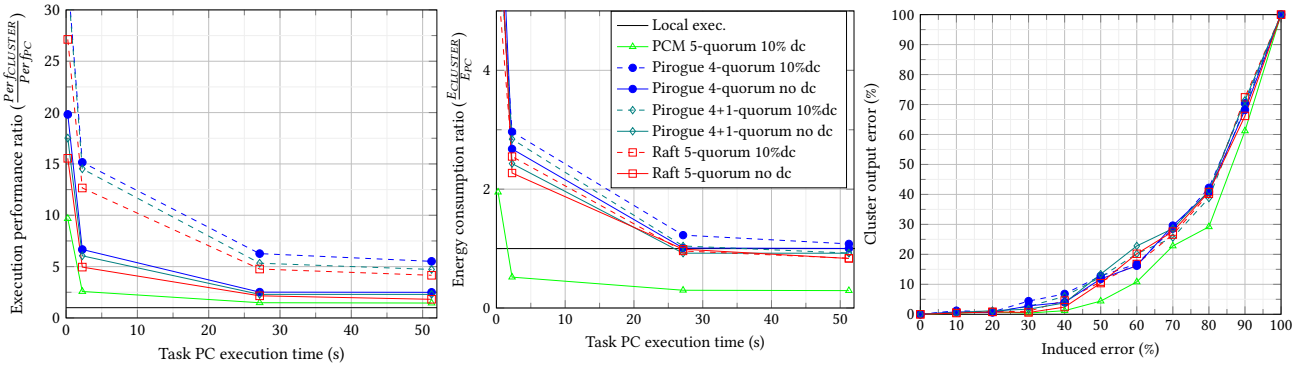

Figure 11: Performance comparison to the state of the art.

However, as this calculus takes place only in the RM node, this calculus can be scheduled to be done periodically depending on the system charge and the previously detailed rules; it is then able to palliate the impact of this calculus on the system overhead, which allows the system to go up to 250k followers without high overhead. Finally, concerning the follower and CO disconnection detection by the RM in our test infrastructure, it oscillates between 1.5 to 2 times the follower's disconnection threshold if there is no need for re-clustering and between 2.2 to 2.7 times if re-clustering is needed.

e) State-of-the-art comparison. We compare PCM, Pirogue and Raft on the first experimental set-up. In order to make possible the comparison with duty-cycles, we have implemented an asynchronous communication broker in Raft and Pirogue. However, we didn't implement any topology maintenance tools; in both Raft and Pirogue, the set of notes participating is supposed to be static. As you can see in Fig. 11, the PCM algorithm achieves a higher performance and fault tolerance level with lower energy consumption than all the other topologies tested. This difference is due to the optimal RM and CO selection and the decrease in the number of messages. In small topologies, the overhead added by the control mechanisms is almost negligible. Another reason for this performance gain is the higher weight of the CO to solve tie votes. It is able to solve ties faster, decreasing the waiting time. Note that this performance gain is minimised in this example, however, the adaptability of PCM to new incoming nodes would make PCM even more useful in highly dynamic topographies.

6 CONCLUSION

PyCloudIoT is a collaborative energy-efficient FaaS computing platform with low-to-medium availability targeting the execution of punctual stateless functions over the already deployed IoTs and gateways putting the focus on its consensus mechanism, PCM. PCM maximises the trade-off energy (performance) fault tolerance for a 10% duty-cycle 5-node CU configuration, which shows increases in performance and energy consumption of 20% and 66%, 37% and 65%, respectively compared to the best configurations of Raft and Pirogue (4+1 quorum) while achieving a slightly stronger fault tolerance level. This gain is measured for static topologies and would be even higher in highly dynamic topologies. However, despite these encouraging results, there are still many open issues in PCM that need to be addressed in the future. These issues include formal correctness, liveness model verification and studying the cache sizes and positions. Researchers should also consider the task orchestration algorithm, the node miss-functions and malicious attacks. Finally, in order to enhance the usability of this platform, a smart automatic function parser would help to analyse, dynamically control and determine calculus offload to IoT nodes.

7 ACKNOWLEDGMENT

This research was mainly supported by the *SPIE ICS-INSA Lyon IoT Chair*. The writing end-phase was supported by STELLANTIS under the collaborative framework OpenLab VAT@Lyon involving Stellantis and CITI Lab (ANRT contract n°2020/1415).

REFERENCES

[1] A.Yousefpour. 2019. All one needs to know about fog computing and related edge computing paradigms: A complete survey. In *J. of Syst. Architecture*. Elsevier.
[2] D.Ongaro. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference*.
[3] H.Atlam. 2018. Fog computing and the internet of things: A review. In *Big Data and Cognitive Computing Journal*. Multidisciplinary Digital Publishing Institute.
[4] L.Lamport. 2001. Paxos made simple. In *ACM Special Interest Group on Algorithms & Computation Theory News (Distributed Computing Column)*.
[5] L.Lamport. 2006. Fast Paxos. *Distributed Computing* (2006).
[6] L.Lamport. 2019. The part-time parliament. In *Concurrency: The Works of Leslie Lamport*.
[7] L.Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*.
[8] A.Munir *et al.* 2017. IFCIoT: Integrated Fog Cloud IoT: A novel architectural paradigm for the future Internet of Things.. In *IEEE Consumer Electronics Magazine*.
[9] B.Cheng *et al.* 2019. Fog function: Serverless fog computing for data intensive iot services. In *IEEE International Conference on Services Computing*.
[10] F.Kaup *et al.* 2014. PowerPi: Measuring and modeling the power consumption of the Raspberry Pi. In *IEEE Conference on Local Computer Networks*.
[11] G.Premsankar *et al.* 2018. Edge computing for the Internet of Things: A case study. In *IEEE Internet of Things Journal*.
[12] H.Du *et al.* 2009. Multi-Paxos: An implementation and evaluation. In *Department of Comput. Sci. and Engineering, Dept University, Tech. Rep. UW-CSE-09-09-02*.
[13] H.Howard *et al.* 2016. Flexible paxos: Quorum intersection revisited. In *arXiv preprint arXiv:1608.06696*.
[14] H. Howard *et al.* 2021. Fast flexible paxos: Relaxing quorum intersection for fast paxos. In *International Conference on Distributed Computing and Networking*.
[15] J.Pâris *et al.* 2015. Pirogue, a lighter dynamic version of the Raft distributed consensus algorithm. In *IEEE IPCCC*.
[16] M.Matsumoto *et al.* 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM TOMACS* (1998).
[17] M.Król *et al.* 2017. NFaaS: named function as a service. In *ACM Conference on Information-Centric Networking*.
[18] M.Shahrad *et al.* 2019. Architectural implications of function-as-a-service computing. In *IEEE/ACM International Symposium on Microarchitecture*.
[19] N.Hassan *et al.* 2018. The role of edge computing in internet of things. In *IEEE Communications Magazine*.
[20] S. Jajodia *et al.* 1990. Dynamic voting algorithms for maintaining the consistency of a replicated database. In *ACM Transactions on Database Systems*.
[21] T.Zhu *et al.* 2009. Leakage-aware energy synchronization for wireless sensor networks. In *ACM MobiSys*.