Visually-defined Real-Time Orchestration of IoT Systems

Margarida Silva
DEI
Faculty of Engineering, University of Porto
Porto, Portugal
ana.margarida.silva@fe.up.pt

João Pedro Dias
INESC TEC and DEI
Faculty of Engineering, University of Porto
Porto, Portugal
jpmdias@fe.up.pt

André Restivo
LIACC and DEI
Faculty of Engineering, University of Porto
Porto, Portugal
arestivo@fe.up.pt

Hugo Sereno Ferreira
INESC TEC and DEI
Faculty of Engineering, University of Porto
Porto, Portugal
hugosf@fe.up.pt

ABSTRACT

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Distributed architectures*; Reliability; • **Software and its engineering** → *Embedded software*; Integrated and visual development environments.

KEYWORDS

Internet-of-Things, Orchestration, Distributed Systems, Real-Time Systems, Embedded Computing

ACM Reference Format:
Margarida Silva, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. 2020. Visually-defined Real-Time Orchestration of IoT Systems. In *17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The Internet-of-Things (IoT) mostly consists of uniquely identifiable objects (*i.e.*, *things*) and their virtual representations within the Internet infrastructure [8]. Broadly, it refers to the inter-connectivity between ordinary devices alongside their contextual awareness, sensing capability, and autonomy [16]. With the steady rise in the number of connected devices, the interest in IoT has been growing massively. According to Siemens, there are around 26 billion physical Internet-connected devices (circa 2020), and predictions are pointing at 75 billion in 2025 [1]. Although this presents several opportunities, these devices are highly heterogeneous in both their hardware and capabilities, which causes several issues in terms of development, scalability, maintainability, and security.

Although IoT systems are of large-scale, most existent IoT systems are designed and build around centralized architectures (as most of the existent Web services), where the main component executes most of the computation on data provided by edge devices (*i.e.*, sensors and actuators) [18]. Even for the ones that are cloud-based, they are typically based on centralized cloud services, mostly due to the advantages in terms of management and costs (*e.g.*, *the economics of scale when building datacenters, automatic backup of all data, and enforce physical security* [30]). Examples include several IoT platform-as-a-service such as Amazon Web Services (AWS) IoT, IBM Bluemix, and Microsoft Azure IoT Suite, which are centralized cloud-based solutions where all the processing takes place [22]. There are other, on-premises, and more suitable for the *fog* tier, solutions (*e.g.*, QNAP QIoT Suite, Home Assistant and OpenHAB) which provide features to integrate and build IoT systems with the aid of rules and triggers (usually visually-defined) [2]. However, their processing is also centralized in a single instance, integrating Node-RED (or a similar solution) as the event processing engine for the user-defined rules and triggers (*e.g.*, Fig. 3) [11].

These centralized approaches have several consequences, including: (1) computation capabilities of the edge devices are being ignored, (2) it introduces a single point of failure, and (3) data is being transferred across boundaries (*e.g.*, private, technological and political) either without the need or even in violation of legal constraints. Ideas such as the one of Local-First [17] — *i.e.*, data and logic should reside locally, independent of third-party services faults and errors — and NoCloud [25] — *i.e.*, on-device and local computation should be prioritized over cloud — are mostly ignored. Edge and Fog Computing have been suggested to solve some of these limitations by pushing some processing tasks away from the cloud and into lower-tier devices [19]. Nonetheless, most of the issues remain unaddressed, as the central instance, if it exists, should orchestrate the system in a way that the computational tasks are divided into independent blocks that could then be executed by other devices instead of running everything in a centralized way [23].

In this paper, we delve on how to leverage the computation capabilities of heterogeneous devices that capable of running custom code, as a way of improving the resiliency, efficiency and scalability of IoT systems. For this purpose, a prototype was developed consisting mainly of two parts: (1) extensions and modifications to the Node-RED system — allowing it to orchestrate the computing tasks among the available devices, while taking into account their current capabilities; and (2) a MicroPython-based firmware that

EAI MobiQuitous 2020, 2020,

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

runs on the edge devices that can receive, interpret, and execute the orchestrator-assigned computational tasks.

We evaluated this approach, in terms of functionality, resilience to hardware/software errors, and efficiency (*i.e.*, latency and elasticity); by scaling up the number of available devices and computational tasks. It was concluded that we can improve the resilience of the system to device failures and ensures that the system, once orchestrated, operates in a distributed fashion (even operating without the presence of the orchestrator). We verified that the system scales, at least until a total of 50 devices (affirming its suitableness for most *smart home* setups). We also concluded that our approach increases the delay in communication between *nodes*, mostly due to the changes in the communication channel (*i.e.*, from a Node-RED centralized communication to a Wi-Fi MQTT-based one).

The remaining paper is structured as follows: Section 2 presents an overview on related work and summarizes the open research challenges. Section 3 provides insights on our approach architecture and implementation. Section 4 presents the experiments and results. These results are discussed in Section 5 and some closing remarks and future work directions are given in Section 6.

2 RELATED WORK

Node-RED [11] is a web-based development and runtime environment for developing Internet-of-Things systems. It provides the end-user with a *drag-n-drop* interface to connect devices and APIs, using a flow-based programming approach. Programs are called *flows*, built with *nodes* connected by *wires*. Regarding its architecture, the base class `EventEmitter` maintains a subscriber list of all the *nodes* connected to it and emits events to them. When a *node* finishes processing data, from external sources or another node, it calls the methods `send()` with a JavaScript object. In its turn, this method calls the `EventEmitter emit()` method that sends named events to the subscribed nodes. Being open-source, Node-RED takes advantage of a large community that contributes new *nodes* and improvements to the tool. It is the most popular open-source visual programming tool for IoT, with more than 10100 stars on GitHub [7], being integrated into several IoT platforms as the *flow* designer and event processing runtime.

Blackstock *et al.* [3, 12] present an IoT development approach, named DDF, which they define as more suitable for the development of fog-based applications that are dependent on the context of the edge devices where they operate. The authors extended Node-RED and implemented D-NR (Distributed Node-RED), which contains processes that can run across devices in local networks and servers in the cloud. All devices running D-NR subscribe to an MQTT topic that contains the status of the main *flow*. When the *flow* is deployed, all devices running D-NR are notified and, based on a set of constraints, decide which *nodes* may need to deploy locally and which sub-*flows* must be shared with other devices. Each device has a set of characteristics, from its computational resources, such as bandwidth and available storage, to its location. The developer can insert constraints, by specifying in which device a sub-*flow* must be deployed or the computational resources needed. Subsequent works [13, 14] focus on the issue of deploying multiple instances of devices running the same sub-*flow*, as well as the support for more complex deployment constraints of the application *flow*.

Szydlo *et al.* [27, 29] focus on the transformation and decomposition of programmed *flows* into executable parts (*e.g.*, Lua artifacts). Their contribution includes the concepts of *flow* transformation, and a new runtime environment called *uFlow* that can be executed on a variety of edge devices and seamlessly integrates with Node-RED. *Flows* are transformed based on the developer-defined configurations stating which operations will be run on each edge device. These operations are implemented using the *uFlow* solution, which allows parts of the *flow* to be run edge devices but all the communication between them is cloud-dependant. The results point to a decrease in the number of measurements needed by the sensors. However, there was no automation of the decomposition and partitioning of the initial flow, neither efforts in detecting bottlenecks or addressing their impact. The same authors improved *uFlow* with a set of models and an execution engine which enables the design of applications to be decomposed onto heterogeneous devices according to a defined decomposition schema. Several algorithms for flow decomposition are mentioned [15, 20], but no results were stated.

Cheng et al. [6] provide an implementation of a standards-based programming model for Fog Computing and scalable context management. They extend the data-flow programming model with hints to facilitate the development of fog applications. The scalable context management introduces a distributed approach, which allows overcoming the limits of a centralized approach, achieving much better performance in terms of throughput, response time and scalability. Follow-up approaches [5] provide infrastructure managers with an environment that allows to build decentralized IoT systems with increased stability and scalability. Dynamic data representing the IoT system and logical flows is orchestrated between sensors and actuators. The application is designed using the *FogFlow* Task Designer, a hybrid text and visual programming environment, which results in an abstraction called Service Template, which contains specifics about the resources needed for each part of the system. Once the Service Template is submitted, the framework will determine how to instantiate it using the context data available. Each task is associated with an operator and its assignment is based on (1) how many resources are available on each edge node, (2) the location of data sources, and (3) the prediction of workload.

DDFlow [21] presents another distributed approach by extending Node-RED with a system runtime that supports dynamic scaling and adaption of application deployments. The coordinator of the distributed system maintains the state and assigns tasks to available devices, minimizing end-to-end latency. Notions of *node* and *wire* are expanded, with a *node* in DDFlow representing an instantiation of a task that is deployed in a device, receiving inputs and generating outputs. *Nodes* can be constrained in their assignment by optional parameters, such as *Device* and *Region*. A *wire* connects two or more *nodes* and can have three types: *Stream* (one-to-one), *Broadcast* (one-to-many) and *Unite* (many-to-one). Each device has a set of capabilities and services that correspond to a *Node*. The devices communicate this information through their Device Manager or a proxy. The coordinator is responsible for managing the DDFlow applications and is composed of three main components: (1) a visual programming environment, (2) a Deployment Manager that communicates with the Device Managers of the devices and (3) a Placement Solver, responsible for decomposing and assigning tasks to the available devices. When an application is deployed, a

Table 1: Small circles (•) mean *yes* and empty means *no*. Node-RED is used as a comparison baseline.

Tool	Leverage devices	Comm. capabilities	Open-source	Computation decomposition	Run-time adaptation
Node-RED [11]		•	•		
DDF [3, 12–14]	Limited[1]	•	•	Limited[2]	•
uFlow [27, 29]	•	Limited[3]		•	Limited[3]
FogFlow [5, 6]	•	N/A	•	Limited[2]	•
DDFlow [21]	Limited[4]	•		Limited[2]	•

[1] Assumes that all devices run Node-RED, which limits the type of devices.
[2] Do not specify the algorithm used.
[3] Communication between devices is made through the cloud.
[4] Assumes that all devices have a list of specific services they can provide.

network topology graph and a task graph are constructed based on the real-time information retrieved from the devices. The coordinator proceeds with mapping tasks to devices by minimizing the task graph's end-to-end latency of the longest path. If changes in the network are detected, such as device failure or disconnection, task assignment adjustments are made. The coordinator can be deployed in multi-devices to improve the system's reliability.

The mentioned tools were characterized based on their mentions or support for the following features and characteristics:

Leverage devices A decentralized architecture takes advantage of the computational power of the devices in the network. However, some tools have limitations on the type of devices, being specific for certain devices or tiers.

Communication capabilities The devices need to communicate to the orchestrator their capabilities so that it can make an informed decision regarding the decomposition and assignment of tasks.

Open-source Open-source license allows access to the code, making it possible for its analysis, improvement, and reuse, playing a key role in terms of research.

Computation decomposition A decentralized architecture must decompose the computation of the system into independent and logical tasks that can be assigned to devices. The algorithms used for this can be specified or mentioned.

Run-time adaptation A system needs to adapt to runtime changes, such as non-availability of devices or even network failure. The system notices these events and can take action to circumvent the problems and keep functioning.

From the analysis (*cf.* Table 1), we can conclude that the current research in decentralized architectures in visual programming tools applied to IoT is incomplete. All the tools leverage the devices in the network but different ways. DFF [12] assumes that all devices run Node-RED, which limits the type of devices that can be leveraged since it needs to have minimum resources to run it. *uFlow* [27, 29] is the only tool that specifies how it truly leverages constrained devices, with the transformation of sub-flows into Lua code, with DDFlow [21] assuming that all devices have a list of specific services they can provide, that should match the *node* assigned to them.

Regarding the method used to decompose and assign computations to the available devices, DDFlow describes the process with the use of the longest path algorithm focused on reducing end-to-end latency between devices. *uFlow* [27, 29] mention several algorithms that could be used, but do not specify which one was implemented. Both DDF [12] and *FogFlow* [5, 6] do not specify the algorithm used besides some constraints but are the only tools with their source code accessible and with an open-source license. All the tools claim to have support for runtime adaptation to changes in the system, such as device failures.

Overall, these solutions solve specific problems or make assumptions regarding the scale of the system and devices constraints. Thus, we can identify the following research challenges: (1) how to leverage the computational capability devices in the network, (2) how to communicate computational capabilities of devices, (3) how to detect device non-availability, (4) how to generate code for sub-flows, and (5) how to make the system self-adaptable.

We tackle these challenges by creating and evaluating a prototype of an IoT system having decentralized orchestration. The motivational use case is a home automation system, where Node-RED is used as a programming and runtime environment. We assume that all the devices have firmware capable of running MicroPython code, accept custom code and can announce their capabilities.

3 ARCHITECTURE AND IMPLEMENTATION

We use Node-RED for both (1) defining programs (as flows), and (2) send tasks to other devices in the network, acting as a system orchestrator; depicted in Fig. 1. The devices in the network make themselves known by announcing their address and capabilities to a registry *node* running in Node-RED. Consequently, Node-RED assigns *nodes* to devices taking into account their capabilities and communicates each node's assignment via HTTP. Due to the devices' limitations, they cannot run an instance of Node-RED, so Node-RED needs to translate the *nodes* code in JavaScript to artifacts that can be interpreted by these devices.

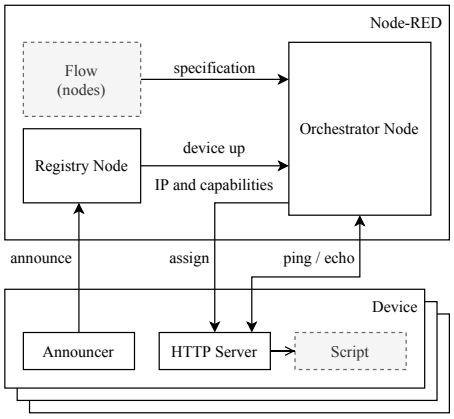

Figure 1: Implemented proof-of-concept overview.

To meet the distributed computation communication demands the built-in communication was replaced by an MQTT-based one. Two main components, were introduced to the Node-RED Palette: (1) the *Registry node* which maintains a list of available devices and their capabilities and, (2) the *Orchestrator node* which partitions and assigns computation tasks to the available devices. Support was added to Node-RED to generate MicroPython-compatible code from custom *nodes*. A MicroPython-based firmware was developed

that can receive and run arbitrary Python code scripts generated by Node-RED, and communicate with other devices or Node-RED itself by MQTT.

3.1 Devices Setup

We consider constrained devices that are capable of running custom code. Amongst the available hardware solutions, taking into consideration both costs and features, we picked two IoT development devices based on the Espressif Systems ESP32 and ESP8266 systems on chip (SoC) [9, 10]. The first challenge is to find a way to take advantage of the constrained devices by making them run arbitrary scripts of code and communicate with other devices. Since both selected devices can run MicroPython firmware, Python language was used. MicroPython already packs a small-footprint HTTP server and packages are available to implement asynchronous operations (`uasyncio`) and MQTT publisher-subscriber (*i.e.*, pub-sub) communication (`MicroPython-mqtt`).

As the devices must be able to receive arbitrary Python scripts (sent by Node-RED) and run them, the HTTP server was used to receive the Python payloads, which are then saved in the device SPI Flash and can be executed later. The same HTTP server is used to implement an endpoint that returns the state of the device, as well as an announcing mechanism (*cf.* Section 3.3). These features built as an integral part of the firmware that runs on the devices.

The firmware also includes a FAIL-SAFE mechanism, safeguarding against several errors (including *Out-of-Memory*) that may happen during the lifespan of the device (SRAM usage). This mechanism resets all running tasks and recovers the HTTP server and communication channels, being essential due to the high probability of these error's occurrence due to the devices memory constraints.

3.2 Decentralized Node-RED Computation

Node-RED is a centralized tool by design, which takes advantage of events to allow communication between *nodes* in a *flow*. Implementing a decentralized architecture required some changes to the Node-RED runtime. These changes consisted mainly in (1) implementing a different communication channel for *node*-to-*node* communication and (2) add code generation features (*i.e.*, JavaScript to MicroPython). These are described in the following paragraphs.

3.2.1 Node-RED Node-to-Node Communication. Node-RED *nodes* communicate using events — node.js `EventEmitter`. The communication is forward-only, with the *nodes* only sending data to the next *nodes* it is connected to. These output wires are used to access the *nodes* the message must be sent to, and their `receive()` method is called. This method triggers the event `emit()` which will the caught by a specific method of each *node*, implementing its own logic. This implementation is local and JavaScript specific, making it hard to be used in a decentralized architecture where *nodes* will be executed outside of Node-RED. It was necessary to implement a way of communicating between *nodes* external to Node-RED that could be supported by constrained devices.

Node-RED `Node` class was modified to use MQTT pub-sub communication [28] instead of the in-place communication. Each *node* publish messages to an unique and addressable topic generated at the flow start, to which the next *node* in the *flow* subscribes to. This happens for every *node* with the exception of *producer nodes* that only act as publishers and *consumers* that only act as subscribers. Since the modifications were made at the base class level (from which every *node* derives from) all the existent *nodes* and sub-*flows* become mostly compatible with this modification without further changes. However, if we want a *node* to be orchestrabable, the code of the *nodes* themselves needs to be changed (*cf.* 3.2.2).

3.2.2 Code Generation. To orchestrate Node-RED *nodes* amongst devices, there is the need to generate MicroPython-compatible code from the existent JavaScript (*i.e.*, code generation). It is also necessary to support multiple *nodes* in one script; thus, a generalized strategy was defined that could fit any type of *node*. This was accomplished by adding specific code generation methods to each orchestrabable *node*, which provide (1) their functionality, and (2) input/output capabilities. Since every *flow* communication is now MQTT-based, the only input and output a *node* can have through its topics. An exception to this are the *nodes* that are producers, meaning that they generate input and do not receive it.

Code generation happens after the *node*-device assignment. This generation creates device-specific code that carry out the tasks assigned to the device (which can correspond to several *nodes*), adding some wrapping code that is responsible for subscribing to all input topics of all nodes, stopping the script's processes, and forwarding the messages to the respective *nodes*.

3.2.3 Custom Nodes. As previously mentioned, all the existent *nodes* are compatible with the modified Node-RED. Nonetheless, for a *node* to be orchestrabable, it must be modified to comply with the code generation needs. Each of these *nodes* has two available properties: Predicates and Priorities. Similar to the Kubernetes logic of assigning containers to machines [4], the predicates dictate constraints that cannot be violated, and priorities are requests that are advisable and recommended but can be ignored if needed.

3.3 Device Registry

IoT systems are typically built on top of heterogeneous parts, with different capabilities and resources, and their network can be highly-dynamic (devices can go off/on due to battery levels, hardware/-software failures and communication issues). To maintain a *list* of network available devices along with their capabilities, there is a need for a Device Registry [24] inside Node-RED.

When a device becomes available, information about itself is sent to an MQTT topic. This information contains the device's IP address, their capabilities and their status (*i.e.*, if the device has failed before). Node-RED contains a *Registry node* that listens to the announcements MQTT topics and saves the devices information. If this *node* is connected to an *Orchestrator node*, each time a new device appears a message is sent to the orchestrator, so that it can consider the new resources in a following orchestration.

When a device has an OUT-OF-MEMORY error, it triggers a FAIL-SAFE, where it reboots the HTTP server, stops running any script and restarts all communications. After this action, the device announces itself again but with a flag that indicates that it has failed. This way, the *Orchestrator node* knows that a device is active but not running any code, being possible that it has failed because of having to many *nodes* allocated. In that case, it can dynamically

adapt and assign fewer *nodes* to the device, reducing the chances of causing another Out-of-Memory error.

3.4 Computation Orchestration

The requirements to achieve this are two-fold: (1) a *node* should act as coordinator, which when provided with an available devices list, along with their respective capabilities (*cf.* Section 3.3), should decide which device should execute specific computation *nodes* — *Orchestrator node* — and, (2) the orchestrabable *nodes* should provide both Predicates and Priorities that must be meet to assure their correct execution (*cf.* Section 3.2.3).

Algorithm 1: Greedy algorithm for *node* assignment.

Input : deviceList, node, $\alpha = 0.5$, $\beta = 0.4$, $\gamma = 0.1$
Output : bestDevice

onInput
  electible $\leftarrow \{d \in \text{deviceList} \mid \text{hasMem} \wedge \text{isReady} \wedge \text{isCapable}\}$
  where
    hasMem $\leftarrow \#d.\text{nodes} < \#d.\text{lastError.nodes}$
    isReady $\leftarrow d.\text{status} = \text{OK}$
    isCapable $\leftarrow \text{node.predicates} \subseteq d.\text{capabilities}$
  return
    $\underset{d \in \text{electible}}{\arg\max}\ \text{fitness}(d) = \alpha \cdot \text{overlap} + \beta \cdot \text{vacancy} + \gamma \cdot \text{specificity}$
  where
    overlap $\leftarrow \frac{\#(d.\text{capabilities} \cap \text{node.priorities})}{\#\text{node.priorities}}$
    vacancy $\leftarrow (\#d.\text{nodes} + 1)^{-1}$
    specificity $\leftarrow \frac{\#(d.\text{capabilities} \cap \text{node.predicates})}{\#d.\text{capabilities}}$

The assigning algorithm uses the devices capabilities and each node's Predicates and Priorities to assign *nodes* to devices. With a greedy approach, the algorithm filters the devices that comply with each node's predicates and assigns the one having the best fitness (*cf.* Algorithm 1). The fitness takes into account the number of priorities the device can provide, which is the most valued factor (0.5), the number of already assigned *nodes* the device has (0.4), and the specialization of a device (0.1); meaning that a device with priorities not requested by the node would be better if left for a future node that might request them. The goal is to assign each *node* to the best possible device, spreading the tasks through all the available devices. An example of a possible assignment can be seen in Fig. 2, where the assignment matches the nodes' priorities with the devices' tags while spreading the *nodes* over the devices.

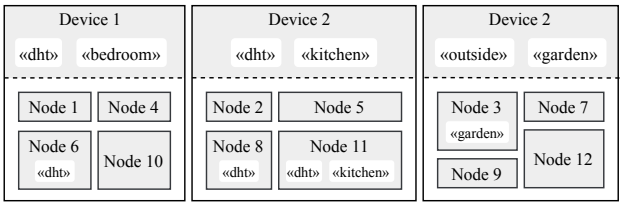

Figure 2: Node assignment example.

After assigning all *nodes* to a specific device, a code script is generated for each one (*cf.* Section 3.2.2). Due to the constrained memory of the devices, the number of *nodes* assigned to a device may exceed their resources. In that case, it will fail-safe and return an error to the assignment request. The orchestrator will receive this information and repeat the process, assigning fewer *nodes* to the ones that returned an Out-of-Memory error. If a device does not return any response, the orchestrator will assume that the device is unavailable and not assign any *node* to it.

The *Orchestrator node* can be triggered — proceeding to a system (re)orchestration — by the following events: (1) start of the system, when there is already a defined flow in the configuration, the assignment start after a period of 3s, to give time for the devices to be registered by the registry node, (2) deployment of the entire flow using the Node-RED editor or API, (3) appearance of a new device detected by the *Registry node*, and (4) failure or recovery of a device, which, working as a complement to the *Registry node*, is detected using Ping/Echo pattern [26] which periodically *pings* the devices in the system to assert their operational status.

4 EXPERIMENTAL OVERVIEW

We evaluate our approach in scenarios using both virtual and physical setups. Physical setups used ESP8266[1] and ESP32[2] devices connected to the same Wi-Fi network. Virtual setups used Docker containers with constrained resources. The experiments were performed in a i5-6600K@3.5GHz w/16Gb RAM, Linux Manjaro 5.6.16, Node-RED 1.0.6, Mosquitto 1.6.10, and MicroPython 1.12.

4.1 Experimental Scenarios

We defined two experimental scenarios. In **ES1**, a room has three sensors that provide temperature and humidity readings every minute. There is a virtual sensor that compares these readings and triggers depending on certain thresholds. An AC reads it and decides (a) if it *switches on/off*, and (b) its operating mode: *cool*, *heat*, or *dehumidify*. The Minimal Working System (MWS) consists in (a) one temperature sensor, (b) one humidity sensor, (c) one *node* capable of making the decision, and (d) a working communication channel amongst them. For **ES2**, the system has 20 devices that are responsible for propagating an injected message in a long chain of nodes, until it reaches a specific sink MQTT topic. The goal of **ES1** is to isolate the features of our work with a moderately simple, although realistic, Node-RED *flow* (*cf.* Fig. 3). **ES2** aims to measure possible overheads of our solution.

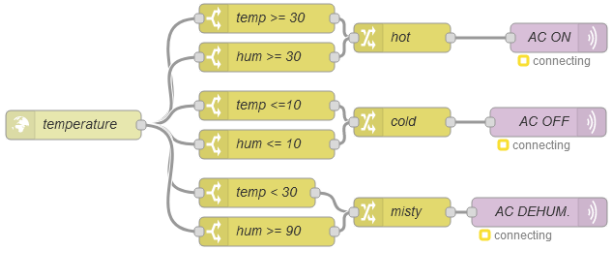

Figure 3: Partial *flow*, which is repeated three times to enable *consensus* and *fault-tolerance* strategies in ES1.

[1] ESP8266 has a single-core at 80Mhz, 160Kb SRAM and 4Mb flash.
[2] ESP32 has a dual-core at 160Mhz, 512Kb SRAM and 4MB flash.

4.2 Experimental Tasks

For each one of the experimental scenarios (**ES1** and **ES2**) we defined a set of experimental tasks, detailed in the next paragraphs.

4.2.1 Experimental Scenario 1 (ES1). Two sanity checks were performed, namely (**ES1-SC1**) with virtual devices, and (**ES1-SC2**) with physical devices. A set of readings and message forwarding tasks were performed with no compensation or any other fault-tolerance strategies. Each sensor only provided environmental readings to the system. Orchestration is centralized. We expect all roundtrips to take less than the smallest part that can be resolved (measurement capability estimated to be < 1s). We then defined a set of (re-)orchestration experiments where the system must allocate computation tasks among the available resources:

- **(A)** MWS is achieved via multiple possible configurations by provoked device failure (fail-stop) using only virtual devices;
- **(B)** MWS is achieved via multiple possible configurations by provoked device failure (fail-stop) using physical devices;
- **(C)** Inconsistent device behaviour, *e.g.*, appear and disappear in shorter intervals lower than the time needed for orchestrating convergence (OCT), possibly impacting the MWS;
- **(D)** With 4 devices, each with different processing capabilities. During orchestration, some devices will throw an*Out-of-Memory* error because they cannot handle all the processing tasks assigned to them (*i.e.*, the size of the provided script). The orchestrator should decide to send fewer tasks to these devices, and converge to a working solution;
- **(E)** With 4 devices, some of them exhibit a memory leak from an unknown cause. These problematic devices stop working with an *Out-of-Memory* error at a random time. The orchestrator should assume these devices cannot handle the number of processing tasks assigned to them, so assign them fewer tasks. Since the devices will keep breaking, the orchestrator should eventually ignore them;
- **(F)** With 4 devices, there is a device that is sensitive to a particular *node*. Whenever the orchestrator assigns this *node* to that specific device, it throws *Out-of-Memory*. The orchestrator should eventually converge to a solution where the specific *node* is not assigned to that device.
- **(G)** With 50 devices, there is a given probability every second of a particular device failing. The downtime can go from 0s to 10s at random. The orchestrator must deal with the devices' failure and re-orchestrate. This experiment is considered a *stress* test, since it forces constant re-orchestrations.

During this experiments we should verify (a) **restrictions (predicates) are enforced**, by checking that possible configurations lead to solutions that enforce defined predicates, and (b) that **priorities are honored**, by checking that all specified priorities were taken into account, and only violated if necessary. In these scenarios, (a) priority is always given to edge devices and (b) priority is given to the maximum level of decentralization by attempting to maximize the number of available devices are being used.

4.2.2 Experimental Scenario 2 (ES2). Regarding **ES2**, a total of 20 devices were connected in a line topology. A message is sent to the starting device, which will propagate it to its output. All the devices implement this propagation logic, which should result in the initial message reaching the end of the line. The propagation time is measured, starting when the message is sent and ending when the message reaches the last node. This scenario was implemented with different experimental configurations, namely:

- **(A)** Non-modified version of Node-RED, using the default *node*-to-*node* communication channel (`EventEmitter`), with all the *nodes* sharing the same runtime;
- **(B)** Modified version of Node-RED that uses MQTT as the *node*-to-*node* communication channel, with all the *nodes* sharing the same runtime;
- **(C)** MQTT-based modified Node-RED, where each *node* of the *flow* is assigned to a different virtual device (*i.e.*, a MicroPython-running Docker instance). The Docker instances and MQTT broker run in the same host machine;
- **(D)** MQTT-based modified Node-RED, where each *node* of the *flow* is assigned to a different virtual device. The Docker instances are in one host, but the MQTT broker is in another one. All parts are connected to the same Wi-Fi network;
- **(E)** Each physical device runs a simple script that performs the desired behaviour, on top of a non-modified MicroPython firmware image, communicating over MQTT. Node-RED is not used, and there is no orchestration being performed;
- **(F)** MQTT-based modified Node-RED, along with the modified MicroPython firmware running on physical devices. Each *node* is assigned to a different device. The devices communicate by MQTT overt the same Wi-Fi network.

5 DISCUSSION

The results of the experimental tasks on both scenarios are discussed in the following paragraphs.

5.1 ES1: Sanity Checks

5.1.1 ES1-SC1. This experiment was used to observe the overall approach functionality in a controlled way (the use of virtual devices reduce the proneness to hardware-provoked failures). The usage of RAM was significant, varying from 60Kb to 200Kb. The flash size only decreases to 150Kb when the device receives a script for executing, *i.e.*, matching the size of the payload received by the devices. As the orchestrator defines the *nodes* assignment, the corresponding scripts are built and sent to the devices. A confirmation of this delivery is necessary for the system to conclude the assignment phase and start monitoring the state of the system. The time it takes to deliver the script averages to 0.303 ± 0.165s. The usage of virtual devices running in the same host as the Node-RED instance allows for shorter times, which are measured in milliseconds. Once the devices status executing its assigned script, each allocated *node* will communicate with each other. All the messages of all communicating topics were capture to check if the system worked as expected. This allowed us to verify that all *nodes* are receiving and producing the expected output messages. The observed behaviour matched the expected by (1) spreading the computation amongst available resources and (2) maintaining the system within expected behaviour.

5.1.2 ES1-SC2. The previous experiment was repeated using physical devices, more specifically four ESP32. The assignment of *nodes* to devices spread the number of *nodes* equally, with each device

running 9 *nodes*. The usage of RAM in physical devices is smaller to the used by virtual devices, which can be explained with the possible optimization differences in the Docker-compatible and ESP-compatible MicroPython firmwares (*e.g.*, MicroPython Garbage Collector runs) and libraries. The free flash space of the physical devices is smaller than the virtual ones, as expected. The script delivery time for physical devices is longer than their virtual counterpart, with an average of 6.776 ± 0.476s. Since computation is distributed among devices, the Wi-Fi connection and hardware specs have a non-negligible impact the communication speed.

5.2 ES1: Experimental Tasks

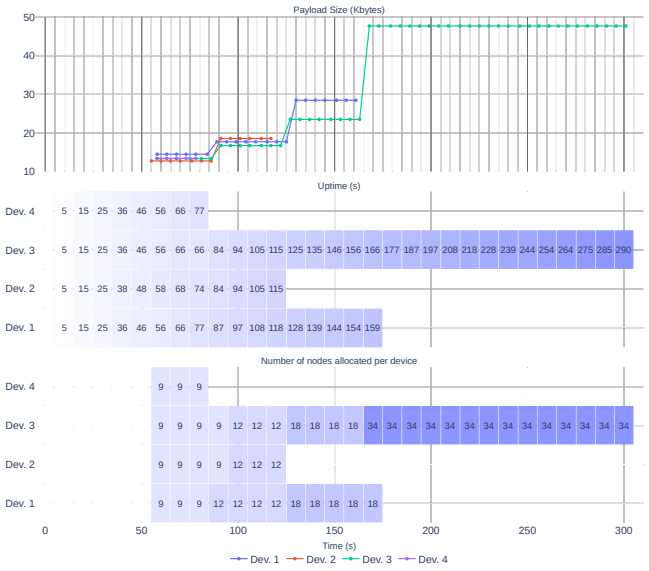

Figure 4: ES1-A measurements.

5.2.1 ES1-A. This experiment evaluates if the system is able to re-orchestrate when a device either fails or (re-)appears (*i.e.*, new or recovered). During this experiment, devices were turned off one by one until only one was left running. It is expected that the system detects when a device has become unavailable and re-orchestrates, assigning *nodes* to the available devices. In the end, only one device should be running, with all the *nodes* assigned to it. Fig. 4 shows that the uptime of the devices stops increasing one by one, identifying the moment the device fails. Once a failure happens, the system re-orchestrates, assigning the *nodes* of the device to the other available devices, increasing their number (*cf.* Fig. 4). The increase in the number of *nodes* assigned to the available devices can also be observed in the payload size. When all devices fail except one, the remaining is the only that receives the payload, which is bigger than any other previously received. The information regarding the number of *nodes* is not updated to zero once the device fails, since it is no longer active to send the updated metric. We can then conclude that the system identifies the failure of devices and takes actions to rectify it by repeating the assignment process.

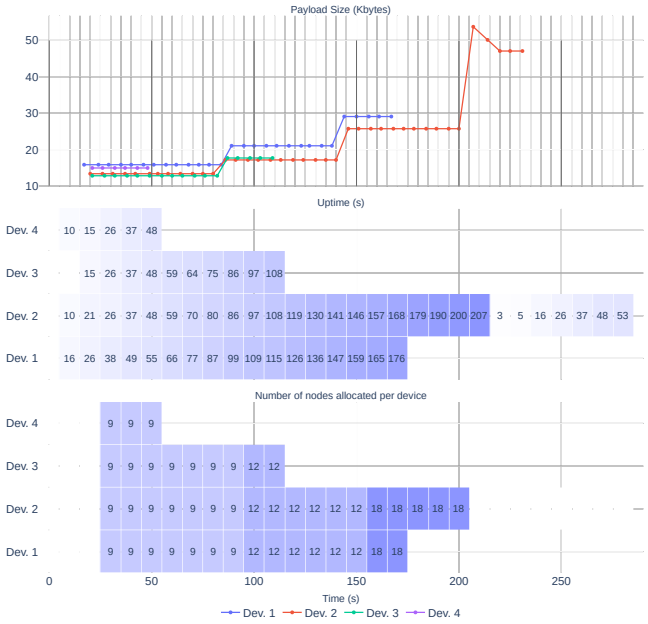

Figure 5: ES1-B measurements

5.2.2 ES1-B. Based on **ES1-A**, this experiment replaces the virtual devices by physical ones. The payloads and number of *nodes* assigned through the experiment are very similar the experiment **ES1-A** (*cf.* Fig.5). However, it is noticeable that *Device 2*, the last remaining active device, fails when receiving the final-step payload — which contains the code for all the *nodes* of the system, since no other device is available. The device constrained memory cannot handle the payload size, so it FAIL-SAFES, informing the system that there was an *Out-of-Memory* error, which results in the *Orchestrator node* assigning fewer *nodes* to the device.

5.2.3 ES1-C. Similar to **ES1-A** and **ES1-B**, this experiment focuses on testing the system's ability to recover when devices fail and re-appear. Fig. 6, *Device 3* and *Device 4* fail early, and the system recovers, leads the orchestrator to assign those *nodes* to other devices. *Device 4* recovers around the 100s, fails again and then recovers. The system did not catch this change since it was swift, and the system only re-orchestrates the second time *Device 4* recovers. During this experiment, *Device 3* and *Device 4* continue to fail and recover, and the system always re-configures itself. This re-orchestration ability when a device recovers can be taxing to the functionality of the system. If a device is continuously failing and recovering, the system will always try to adapt itself, halting its functionality to orchestrate.

5.2.4 ES1-D. The memory constraints of IoT devices can negatively impact the functioning of the system, by raising memory errors when writing the received script into the device SPI flash. This experiment verifies how the system recovers and adapt to the device's memory constraints.

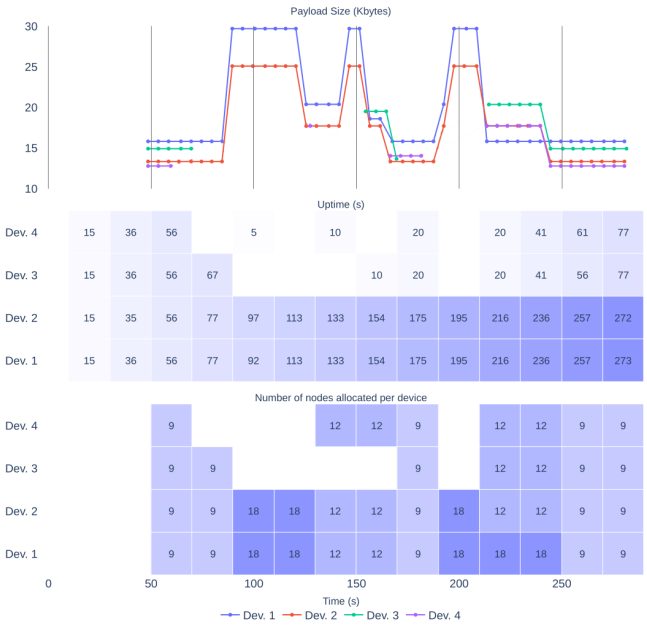

Figure 6: ES1-C measurements.

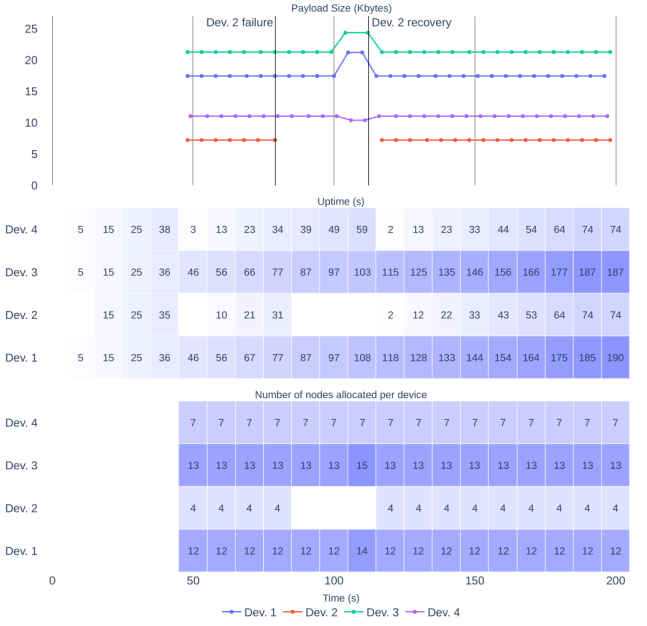

Figure 7: ES1-D measurements.

Fig. 7 shows the constrained memory of the *Device 2* and *Device 4*. When the first assignment is made, at 50s, both these devices FAIL-SAFE due to *Out-of-Memory* errors. The number of *nodes* present on these devices are the ones assigned after they communicate to the orchestrator their limitations. To assess if the system saves information about the limitations of the devices, one of them was turned off and later turned on. As it can be observed, *Device 2* uptime stops increasing around the time of the event and its *nodes* are distributed by the other devices, except for *Device 4*, which is memory constrained. After the recovery of *Device 2*, the system re-orchestrates and the same number of *nodes* are assigned to the devices. However, *Device 4* failed when *Device 2* recovered, which implies that the system repeated the assignment process, ignoring the previously known information about memory constraints.

5.2.5 ES1-E. In addition to the handling of memory limitations, it is expected that the system can handle a damaged device which has a memory leak issue. *Device 2* was modified to always generate an *Out-of-Memory* error after a random period. The system should be able to exclude this device during the assignment process.

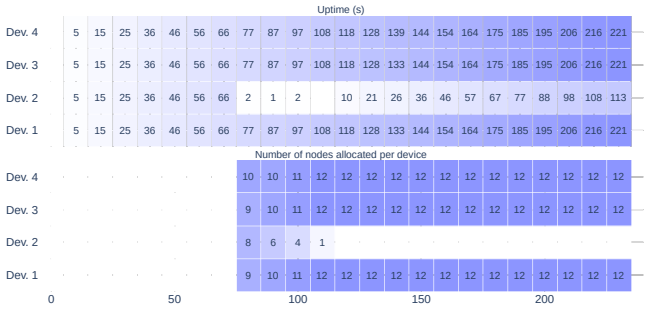

Figure 8: ES1-E measurements.

Fig. 8 shows that *Device 2* is consistently failing after the first assignment of *nodes*, at 75s. The number of *nodes* assigned decreases, until no *node* is assigned and the device is excluded from consideration. This is an iterative process, in which the system will decrease the number of *nodes* it assigns to a device if the device communicates an *Out-of-Memory* to the orchestrator. As the system runs, if the device is not able to handle any *node*, the minimum number of *nodes* the device can handle is zero, thus excluding it from the assignment process.

5.2.6 ES1-F. To further assess the resilience of the system to *Out-of-Memory* errors, a *node* was deliberately injected that causes such error in specific devices. It is expected that the system re-orchestrate and converge to a solution where the specific *nodes* are assigned to devices not affected by them. In turn, the devices affected by these *nodes* should have fewer *nodes* assigned. The system and devices do not know that a specific *node* is creating the *Out-of-Memory* errors and interpret the error as a device problem. Since the first assignment could be correct by sheer chance, meaning that these faulty *nodes* would be assigned to devices not affected by them, we force the system to re-orchestrate. The devices were all turned off and on in different order, repeated by three times. Fig. 9 shows these on/off events at approx. 125s, 200s and 275s timestamps. It is important to note that the devices affected by the faulty *nodes* are *Device 2* and *Device 4*.

The event we aim to test occurs at 300s. As it can be seen (*cf.* Fig. 9), *Device 4* is assigned 10 *nodes*. The uptime of *Device 4* resets in this small time period (the next uptime is less than 20s.) meaning that an *Out-of-Memory* occurred and the device performed a

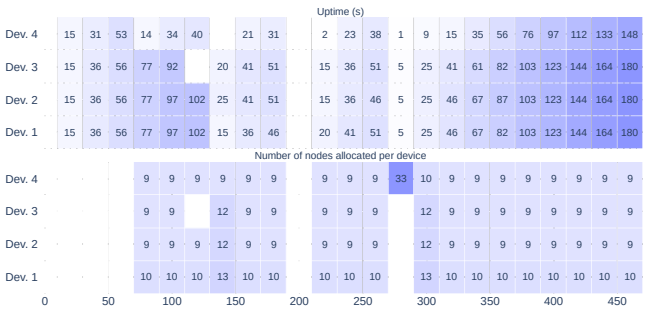

Figure 9: ES1-F measurements.

FAIL-SAFE. The system updates, allocating the 10 *nodes* previously assigned to *Device 4* through all the available devices. Since Fig. 9 shows the data in intervals of 20s, the assignment in *Device 4* happens before the assignment present in the other devices. When the system receives information that *Device 4* is available again, it already knows that it has a limitation, so it only assigns 9 *nodes* to it. It can be seen that missing *node* is assigned to *Device 1*. Since *Device 4* does not FAIL-SAFE, the *node* assigned to *Device 1* must have been the faulty one.

5.2.7 ES1-G. To assess our approach limits, we proceed to inject constant failures in the available devices. Every second, each device has a 5% probability of becoming unavailable from 0 to 10s. During this period, the device is unresponsive to the orchestrator requests and, when recovers, announces itself.

Fig. 10 shows that the system is kept continuously re-orchestrating, and once the majority of devices failed, the system becomes unstable. It is important to note that, similar to previous experiments, once a device fails, the number of *nodes* does not update to zero. We then conclude that devices with the same number of *nodes* during the total execution of the system failed early on and continued to fail, stopping the orchestrator assignment. At 100s, there was a period where all devices were available. However, the *node* assignment in Fig. 10 does not converge during that time period. This is due to the system re-orchestration whenever a device becomes available (since each device announces itself individually, each announcement triggers a new orchestration). This process takes time and results in several failed orchestrations due to outdated data on the device's operating status; being also taxing for the devices, causing an overload of received assignments that will never make the system function as a whole.

5.3 ES2: Experimental Tasks

To benchmark our approach against partial implementations and the original Node-RED (to check the communication delay between modifications), we proceed to experiment with a *flow* that passes a message through several devices, recording the elapsed time for the message to pass through all of them. The `NOP` *nodes* execution consists of only redirecting their input to their output. A message containing only the current timestamp is inserted into the system by triggering the *Inject node*, and the same message is expected to appear in the Node-RED *Debug* console.

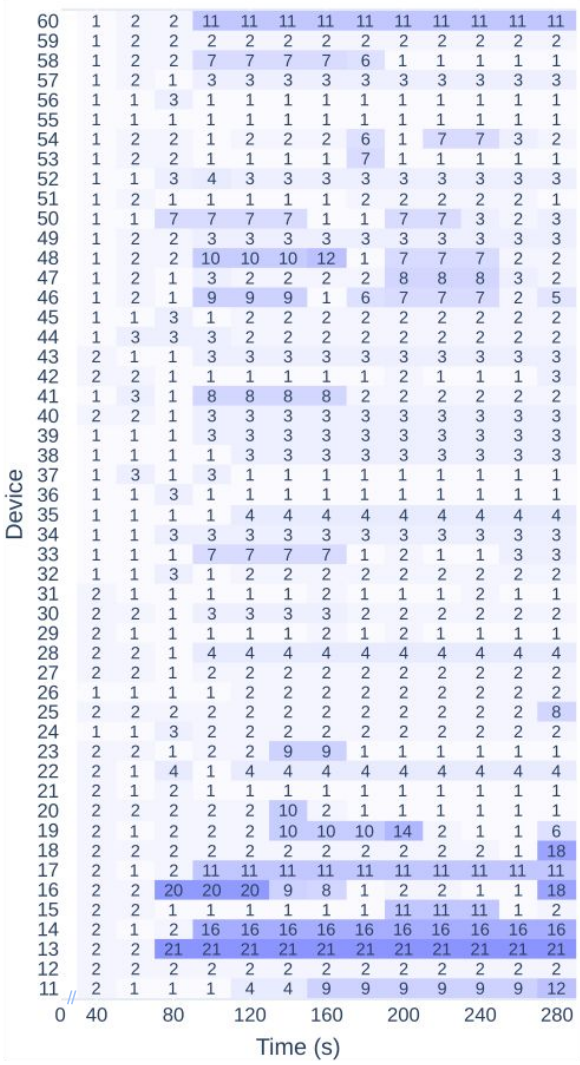

Figure 10: Nodes assignment distribution over time.

Table 2: ES2 elapsed time measurements.

Label	Min	Q1	Q2	Avg	Q3	Max
ES2-A	3	8	10	10	13	15
ES2-B	134	353	431	489	711	883
ES2-C	1217	1260	1318	1400	1574	1665
ES2-D	1445	2332	2536	2392	2708	3059
ES2-E	3616	4031	4142	4133	4372	4452
ES2-F	4168	4357	4569	4751	5088	5940

This experiment was run with different configurations (**ES2-A** to **ES2-F**) to assert the impact of each modification/module, as described in Section 4.2.2. Each experiment was replicated ten times, and the resulting time measurements are shown in Table 2.

When the decentralization is applied inside Node-RED (*cf.* **ES2-B**) it is possible to see that the introduction of the MQTT communication (Mosquitto broker) running in the same host causes some latency. The introduction of Dockers running the firmware in the same host as the Node-RED instance and MQTT causes additional latency (*cf.* **ES2-C**), making it possible to conclude that the MicroPython-based developed firmware also delays the communication. By repeating the same experiment but with the broker running in another machine (same network) (*cf.* **ES2-D**), it is noticeable that the times are more spread out and the overall latency of the system increases. As the Node-RED and the broker run in different machines connected over Wi-Fi, we conclude that this is the leading cause for the additional delay.

The experiment was repeated in physical devices: (1) by running a simple code in the MicroPython flashed devices and injection of messages directly in the broker (*cf.* **ES2-E**), and (2) by using our approach as a whole, *i.e.*, modified Node-RED and designed firmware (*cf.* **ES2-F**). The results shows that the use of physical devices produces higher times (as expected), but that the developed firmware has little impact, visible by the comparison of their results. We conclude that our approach, including the *node*-to-*node* communication change, is slower than the original Node-RED, but it mostly results from the Wi-Fi communications and the base MicroPython firmware. Also, this modification makes Node-RED more *modular*, allowing the use of any other communication mechanism.

6 CONCLUSIONS

Given the results of the experiments with our approach proof-of-concept, we conclude that the challenges that we focus on this work were attained, more specifically the decentralization of computation, with the handling of the device's memory constraints and failures, and dynamic adaptation of the system (self-reconfiguration).

In terms of resilience, we consider that our approach is moderately robust, handling device failures and memory constraints dynamically at runtime. However, there are some limitations to this robustness. The system reaches a maximum point of adaptability when several devices fail and recover continually.

Efficiency-wise, our approach is slower than the original Node-RED system. However, this is due to the change in the communication channel, that introduces some extra latency. Despite this, the developed modules, such as the orchestrator and the device's firmware introduce little latency to the latency of the whole system.

Regarding systems' elasticity, our approach handles a different number of devices, as it was demonstrated. It also handles the number of devices changes throughout the lifespan of the system, adapting the orchestration to the number of devices available.

In summary, the developed solution is more scalable than a centralized one, dealing with a dynamic number of devices while taking advantage of their computational capabilities. It is also robust to handle the failures and constraints of these devices. Lastly, the developed solution is not more efficient, but its latency is due to factors external to the developed firmware, but it was mandatory for having a decentralized solution. There are, however, some limitations in our approach that we are aware of and consider future work, namely: (1) the greedy algorithm used to orchestrate the *nodes* among the available resources can fail if there is no fit for the *nodes* requirements, thus there may be alternatives more suitable, (2) several optimization considerations were disregard, such as if more than one *node* is allocated to the same device there is no optimization of communications (*i.e.*, *nodes* in the same device have to communicate through the MQTT broker), (3) due to limitations on assessing if a message was delivered correctly to a node, there is a chance of duplicated MQTT messages being not handled correctly, (4) only MicroPython is supported but additional firmware supports can be added, and, (5) the (re)orchestration is, so far, only possible by deploying the entire system, and not specific *flows* or *nodes*.

REFERENCES

[1] Tanweer Alam. 2018. A Reliable Communication Framework and Its Use in Internet of Things (IoT). 3 (05 2018).

[2] Nayeon Bak, Byeong Mo Chang, and Kwanghoon Choi. 2018. Smart Block: A Visual Programming Environment for SmartThings. In *Proceedings - International Computer Software and Applications Conference*, Vol. 2. 32–37.

[3] Michael Blackstock and Rodger Lea. 2014. Toward a distributed data flow platform for the Web of Things (Distributed Node-RED). In *ACM International Conference Proceeding Series*, Vol. 08-October. 34–39.

[4] Brendan Burns and Craig Tracey. 2018. *Managing Kubernetes: operating Kubernetes clusters in the real world.* O'Reilly Media.

[5] B. Cheng, E. Kovacs, A. Kitazawa, K. Terasawa, T. Hada, and M. Takeuchi. 2018. FogFlow: Orchestrating IoT services over cloud and edges. *NEC Technical Journal* 13 (11 2018), 48–53.

[6] Bin Cheng, Gurkan Solmaz, Flavio Cirillo, Ernö Kovacs, Kazuyuki Terasawa, and Atsushi Kitazawa. 2017. FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities. *IEEE Internet of Things Journal* PP (08 2017), 1–1.

[7] João Pedro Dias, Bruno Lima, João Pascoal Faria, André Restivo, and Hugo Sereno Ferreira. 2020. Visual Self-healing Modelling for Reliable Internet-of-Things Systems. In *Computational Science – ICCS 2020*. Springer International Publishing, Cham, 357–370.

[8] Manuel Díaz, Cristian Martín, and B Rubio. 2016. State-of-the-art, challenges, and open issues in the integration of Internet of things and cloud computing. *Journal of Network and Computer Applications* 67 (2016), 99–117.

[9] Espressif Systems. 2019. *ESP8266 Technical Reference Manual.* Technical Report. Espressif Systems, Shanghai, China. https://www.espressif.com/sites/default/files/documentation/esp8266-technical_reference_en.pdf

[10] Espressif Systems. 2020. *ESP32 Technical Reference Manual.* Technical Report. Espressif Systems, Shanghai, China. https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf

[11] OpenJS Foundation. 2020. Node-RED. Available: https://nodered.org/. Last access 2020. [Online].

[12] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C.M. Leung. 2015. Developing IoT applications in the Fog: A Distributed Dataflow approach. In *Proceedings - 2015 5th International Conference on the Internet of Things, IoT 2015*. 155–162.

[13] N. K. Giang, R. Lea, M. Blackstock, and V. C. M. Leung. 2018. Fog at the Edge: Experiences Building an Edge Computing Platform. In *2018 IEEE International Conference on Edge Computing (EDGE)*. 9–16.

[14] N. K. Giang, R. Lea, and V. C. M. Leung. 2018. Exogenous Coordination for Building Fog-Based Cyber Physical Social Computing and Networking Systems. *IEEE Access* 6 (2018), 31740–31749.

[15] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. 2017. iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. *Software: Practice and Experience* 47, 9 (2017), 1275–1296.

[16] Md. Mahmud Hossain, Maziar Fotouhi, and Ragib Hasan. 2015. Towards an Analysis of Security Issues, Challenges, and Open Problems in the Internet of Things. *2015 IEEE World Congress on Services* (2015), 21–28.

[17] Martin Kleppmann, Adam Wiggins, Peter Hardenberg, and Mark McGranaghan. 2019. Local-first software: you own your data, in spite of the cloud. 154–178.

[18] Julien Mineraud, Oleksiy Mazhelis, Xiang Su, and Sasu Tarkoma. 2016. A gap analysis of Internet-of-Things platforms. *Computer Communications* 89-90 (2016), 5–16. arXiv:1502.01181

[19] N. Mohan and J. Kangasharju. 2016. Edge-Fog cloud: A distributed cloud for Internet of Things computations. In *2016 Cloudification of the Internet of Things (CIoT)*. 1–6.

[20] Mohammed Islam NAAS, Laurent Lemarchand, Jalil Boukhobza, and Philippe Raipin. 2018. A Graph Partitioning-Based Heuristic for Runtime IoT Data Placement Strategies in a Fog Infrastructure. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (Pau, France) *(SAC '18)*. Association for Computing Machinery, New York, NY, USA, 767–774.

[21] Joseph Noor, Hsiao Yun Tseng, Luis Garcia, and Mani Srivastava. 2019. DDFlow: Visualized declarative programming for heterogeneous IoT networks. In *IoTDI 2019 - Proceedings of the 2019 Internet of Things Design and Implementation (IoTDI '19)*. ACM, New York, NY, USA, 172–177.

[22] P. Patel, M. Intizar Ali, and A. Sheth. 2017. On Using the Intelligent Edge for IoT Analytics. *IEEE Intelligent Systems* 32, 5 (2017), 64–69.

[23] D. Pinto, J. P. Dias, and H. Sereno Ferreira. 2018. Dynamic Allocation of Serverless Functions in IoT Environments. In *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*. 1–8.

[24] Antonio Ramadas, Gil Domingues, Joao Pedro Dias, Ademar Aguiar, and Hugo Sereno Ferreira. 2017. Patterns for Things That Fail. In *Proceedings of the 24th Conference on Pattern Languages of Programs* (Vancouver, British Columbia, Canada) *(PLoP '17)*. The Hillside Group, USA, Article 7, 10 pages.

[25] Reza Rawassizadeh, Timothy Pierson, Ronald Peterson, and David Kotz. 2018. NoCloud: Exploring Network Disconnection through On-Device Data Analysis. *IEEE Pervasive Computing* 17 (03 2018).

[26] James Scott and Rick Kazman. 2009. *Realizing and Refining Architectural Tactics : Availability*. Technical Report August. Software Engineering Institute.

[27] Joanna Sendorek, Tomasz Szydlo, Mateusz Windak, and Robert Brzoza-Woch. 2019. *FogFlow - Computation Organization for Heterogeneous Fog Computing Environments*. 634–647.

[28] Dipa Soni and Ashwin Makwana. 2017. A survey on mqtt: a protocol of internet of things (iot). In *International Conference On Telecommunication, Power Analysis And Computing Techniques (ICTPACT-2017)*.

[29] T. Szydlo, R. Brzoza-Woch, J. Sendorek, M. Windak, and C. Gniady. 2017. Flow-Based Programming for IoT Leveraging Fog Computing. In *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. 74–79.

[30] R. Want, B. N. Schilit, and S. Jenson. 2015. Enabling the Internet of Things. *Computer* 48, 1 (2015), 28–35.