Visually-defined Real-Time Orchestration of IoT Systems

Margarida Silva
MIEIC and 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
hugo.sereno@fe.up.pt

ABSTRACT

In this work, we propose a method for extending Node-RED to allow the automatic decomposition and partitioning of the system towards higher decentralization. We provide a custom firmware for constrained devices to expose their resources, as well as new nodes and modifications in the Node-RED engine that allow automatic orchestration of tasks. The firmware is responsible for low-level management of health and capabilities, as well as executing MicroPython scripts on demand. Node-RED then takes advantage of this firmware by (1) providing a device registry allowing devices to announce themselves, (2) generating MicroPython code from dynamic analysis of flow and nodes, and (3) automatically (re-)assigning nodes to devices based on pre-specified properties and priorities. A mechanism to automatically detect abnormal run-time conditions and provide dynamic self-adaptation was also explored. Our solution was tested using synthetic home automation scenarios, where several experiments were conducted with both virtual and physical devices. We then exhaustively measured each scenario to allow further understanding of our proposal and how it impacts the system's resiliency, efficiency, and elasticity.

CCS CONCEPTS

• **Computer systems organization → Embedded and cyber-physical systems**; Embedded systems; **Distributed architectures**; • **Software and its engineering** → Embedded software; **Integrated and visual development environments**; • **Hardware → System-level fault tolerance**.

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 *MobiQuitous 2020 - 17th EAI International Conference on Mobile andUbiquitous Systems: Computing, Networking and Services (MobiQuitous '20),December 7–9, 2020, Darmstadt, Germany.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3448891.3448938

1 INTRODUCTION

The Internet-of-Things (IoT) mostly consists of uniquely identifiable objects (*i.e.*, *things*) and their virtual representations within the Internet infrastructure [12]. Broadly, it refers to the inter-connectivity between ordinary devices alongside their contextual awareness, sensing capability, and autonomy [20]. The interest in IoT has been growing as the number of connected devices rises steadily. Estimations point to around 26 billion physical Internet-connected devices (circa 2020), and predictions are targeting 75 billion in 2025 [1, 2]. 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, security, and autonomy [1, 9, 11].

Although most (I)IoT systems are large-scale, they are typically designed and built around centralized architectures (as most of the existent Web services), where one main component executes most of the computation on data provided by edge devices (*i.e.*, sensors and actuators) [24]. We also observe centralized cloud services in cloud-based IoT architectures, mostly due to the advantages of management and costs (*e.g.*, *the economics of scale when building datacenters, automatic backup of all data, and enforce physical security* [36]). Examples include IoT PaaS such as Amazon Web Services (AWS) IoT, IBM Bluemix, and Microsoft Azure IoT Suite [28]. Other on-premises solutions are more suitable for the *fog* tier (*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) [3]. Their processing is usually 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) [15, 21].

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 [22] — *i.e.*, data and logic should reside locally, independent of third-party services faults and errors — and NoCloud [31] — *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 [25]. Nonetheless, most of the issues remain unaddressed, as the central instance, if it exists, should orchestrate the system so 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 [29].

In this paper, we explore how the computation capabilities of heterogeneous devices capable of running custom code can be leveraged to improve 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 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. We concluded that the system's resilience to failures was improved and once orchestrated, that the system operated in a distributed fashion (even without the orchestrator's presence). We verified that the system scales, at least up to 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 changes in the *channel* (*i.e.*, from a Node-RED *in-process* communication to a decentralized Wi-Fi MQTT-based).

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 [15] 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 [10], being integrated into several IoT platforms as the *flow* designer and event processing runtime.

Blackstock *et al.* [5, 16] present an IoT development approach (DDF), which they claim as suitable for 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 which device a sub-*flow* must be deployed to or the computational resources needed. Subsequent works [17, 18] focus on deploying multiple instances of devices running the same sub-*flow*, and the support for more complex deployment constraints.

Szydlo *et al.* [32, 35] proposed a transformation and decomposition of *flows* into executable Lua artifacts. Their contribution includes *flow* transformations, and a portable runtime environment called *uFlow* that seamlessly edge devices with Node-RED. *Flows* are transformed based on developer-defined configurations stating which operations will run on which device. These operations are implemented using the *uFlow* solution, which allows parts of the *flow* to run on edge devices but keeping the communication *cloud-dependent*. Results point to a decrease in the number of measurements needed by sensors. However, there is no automation of the initial flow's decomposition and partitioning, nor efforts in detecting bottlenecks or addressing their impact. They further improved *uFlow* with an execution engine that enables the design of applications to be decomposed onto heterogeneous devices according to a defined decomposition schema. Several algorithms for flow decomposition are mentioned [19, 26], but no results are presented.

Cheng *et al.* [8] 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 a centralized approach's limits, achieving much better performance in throughput, response time, and scalability. Follow-up approaches [7] provide infrastructure managers with an environment that allows building 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.

Noor *et al.* [27] present another distributed approach (DDFlow) by extending Node-RED with a system runtime that supports dynamic scaling and adaption of application deployments. The distributed system coordinator 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 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 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.

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

Leverage devices. Decentralized architectures take advantage of the available computational power in the network. However, some tools have limitations on the devices they can operate on, as they are tailored for specific devices or tiers.

Communication capabilities. The orchestrator has to be aware of the devices capabilities so it can make an informed decision for the decomposition and assignment of tasks.

Open-source. An open-source license allows access to the code, making it possible for its analysis, improvement, and reuse, playing a pivotal role for future 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 are mentioned.

Run-time adaptation. Systems need 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 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 in different ways. DDF [16] 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* [32, 35] is the only tool that specifies how it truly leverages constrained devices, with the transformation of sub-flows into Lua code, with DDFlow [27] assuming that all devices have a list of specific services they can provide, that should match the *node* assigned to them.

Table 1: Check marks (✓) 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 [15]		✓	✓		
DDF [5, 16–18]	Limited[1]	✓	✓	Limited[2]	✓
uFlow [32, 35]	✓	Limited[3]		✓	Limited[3]
FogFlow [7, 8]	✓	N/A	✓	Limited[2]	✓
DDFlow [27]	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.

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* [32, 35] mentions several algorithms that could be used, but does not specify which one was implemented. Both DDF [16] and *FogFlow* [7, 8] 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. Our 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 to (1) define programs (as flows) and (2) send tasks to other devices in the network, acting as a system's orchestrator (*cf.* Fig. 1). The network devices make themselves known by announcing their address and capabilities to a particular registry *node.* Consequently, Node-RED assigns *nodes* to devices (taking into account their capabilities) and communicates each node's assignment via HTTP. Constrained devices cannot directly run Node-RED flows, so the orchestrator translates the nodes' JavaScript code to artifacts that can be interpreted by these devices.

Two main components were introduced to the *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. The capability of generating MicroPython code for supported nodes was added, and a MicroPython-based firmware was developed that receives and runs Python code generated by the orchestrator. The centralized Node-RED built-in node's communication was also replaced by an MQTT-based distributed approach, leveraged by our firmware.

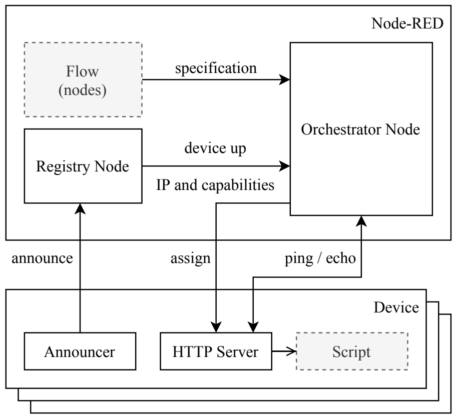

Figure 1: Implemented proof-of-concept overview.

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) [13, 14]. 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 [4]. 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 receive arbitrary Python scripts (sent by Node-RED) and run them, a HTTP server was used to receive the Python payloads and save them in the device SPI Flash to be executed later. The same HTTP server is used to implement an endpoint that returns the state of the device, and an announcing mechanism (*cf.* Section 3.3). These features were 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 device's lifespan (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 errors occurring due to the device's memory constraints.

3.2 Decentralized Node-RED Computation

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

3.2.1 Node-RED Node-to-Node Communication. Node-RED *nodes* communicate using events — node.js `EventEmitter`. The communication is forward-only, with *nodes* only sending data to the following *nodes* in the flow. Output wires are used to access which *nodes* a message must be sent to by calling its `receive()` method. This method triggers the *emit()* event, which will be caught by a specific method, implementing its own logic, in each *node*. 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 [34] instead of in-place communication. Each *node* publishes messages to a unique and addressable topic generated at the start of the *flow* and subscribed by the next *node*. This happens for every *node* except for *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* became compatible with this modification without further changes. However, if we want a *node* to be orchestrable, the code of the *nodes* themselves needs to be changed (*cf.* Section 3.2.2).

3.2.2 Code Generation. To orchestrate Node-RED *nodes* amongst devices, we 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, we defined a generalized strategy appropriate for any *node* type. This was accomplished by adding specific code generation methods to each orchestrable *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 are its topics.

Code generation happens after the *node*-device assignment. This generation creates device-specific code that carries 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 existing *nodes* are compatible with the modified Node-RED. Nonetheless, for a *node* to be orchestrable, it must be modified to comply with code generation needs. Each of these *nodes* has two available properties: Predicates and Priorities. Similar to the Kubernetes logic of assigning containers to machines [6], 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 and their capabilities, we need a Device Registry [30] 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, its capabilities, and status (*e.g.*, 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 to consider the new resources in the 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 and that it has possibly failed due to having too many allocated *nodes*. 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 *Orchestrator 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* and, (2) the orchestrable *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
1  onInput
2      electible ⟵ $\{d \in \text{deviceList} \mid \text{hasMem} \land \text{isReady} \land \text{isCapable}\}$
3      where
4          hasMem ⟵ $\#d.\text{nodes} < \#d.\text{lastError.nodes}$
5          isReady ⟵ $d.\text{status} = \text{OK}$
6          isCapable ⟵ $\text{node.predicates} \subseteq d.\text{capabilities}$
7      return
           $\arg\max_{d \in \text{electible}} \text{fitness}(d) = \alpha \cdot \text{overlap} + \beta \cdot \text{vacancy} + \gamma \cdot \text{specificity}$
8      where
9          overlap ⟵ $\frac{\#(d.\text{capabilities} \cap \text{node.priorities})}{\#\text{node.priorities}}$
10         vacancy ⟵ $(\#d.\text{nodes} + 1)^{-1}$
11         specificity ⟵ $\frac{\#(d.\text{capabilities} \cap \text{node.predicates})}{\#d.\text{capabilities}}$
```

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 ($\alpha = 0.5$), the number of already assigned *nodes* the device has ($\beta = 0.4$), and the specialization of a device ($\gamma = 0.1$); meaning that a device with priorities not requested by the node would be better if left for a future node that might request them. We decided to opt for these particular values of these hyper-parameters, as they performed well in preliminary tests; their optimization is out-of-scope of this paper. 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.

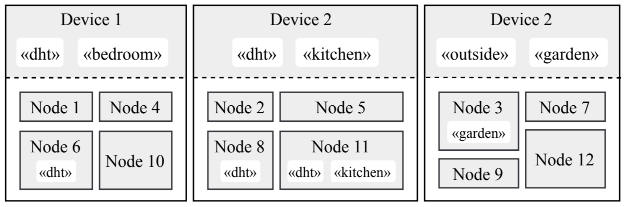

Figure 2: Node assignment example.

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 [33] 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.

[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.

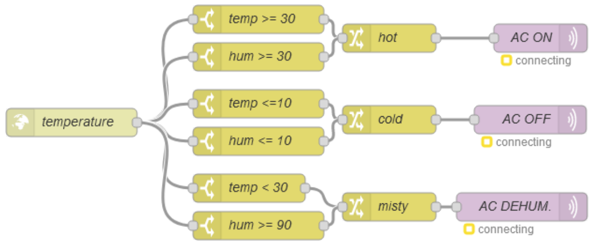

Figure 3: Partial *flow*, which is repeated three times to enable *consensus* and *fault-tolerance* strategies in ES1.

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.*, appearing and disappearing in intervals shorter 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, and 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 a *Out-of-Memory* error. 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 of a particular device failing in each second. 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 these experiments we should verify that (a) **any restrictions (predicates) are enforced**, by checking every obtained configuration, and (b) that **priorities are honored**, by checking that all specified priorities were taken into account, and only violated if necessary. If specified priorities must be violated, (a) edge devices should be used first, and (b) the level of decentralization should be maximized by using the most available devices.

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 over the same Wi-Fi network.

5 DISCUSSION

We now discuss the results from our experimental tasks.

5.1 ES1: Sanity Checks

5.1.1 ES1-SC1. This experiment was used to observe the overall approach in a controlled fashion. By using virtual devices we reduced the chance of hardware failures. The free flash size decreases by ≈150Kb when the device receives a script for executing, *i.e.*, matching the size of the payload. As the orchestrator assigns the *nodes*, the corresponding scripts are built and sent to them. The time it takes to deliver the script averaged 0.303 ± 0.165s. All exchanged messages were captured, which allowed us to check that the system behavior was the expected by (1) spreading the computation amongst available resources, and (2) resulting in a system with the expected functional behaviour.

5.1.2 ES1-SC2. The previous experiment was repeated using physical devices (four ESP32 devices). The orchestrator attributed 9 *nodes* to each device. The RAM usage in physical devices was smaller than in virtual devices[3]. The free flash space was also smaller, as expected. The script delivery time was longer than in the first experiment, averaging 6.776 ± 0.476s. This is mainly attributed to *nodes* being in different devices, with the Wi-Fi communication and hardware specs having a non-negligible impact.

5.2 ES1: Experimental Tasks

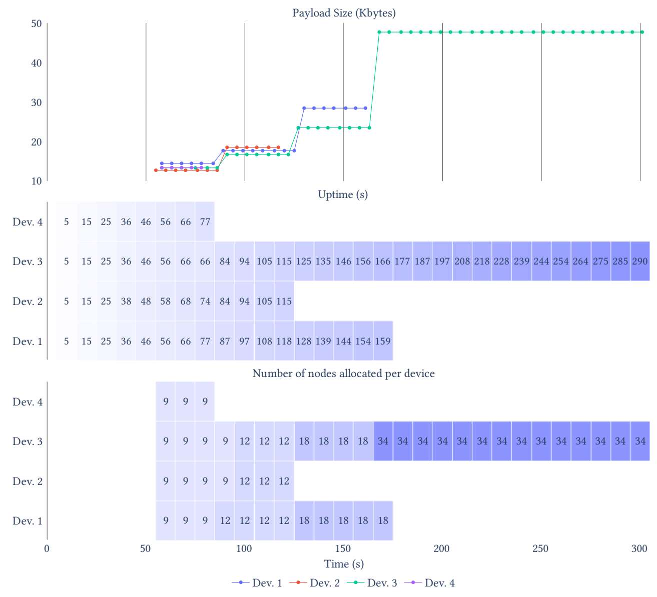

Figure 4: ES1-A measurements.

5.2.1 *ES1-A.* This experiment evaluates if the system is able to re-orchestrate when a device fails. A set of virtual devices were turned off one by one, until only one was left running. It was expected for the system to detect when a device became unavailable and to re-orchestrate by assigning *nodes* to the remaining devices. In the end, we expected only one device to be running, with all the *nodes* assigned to it. Fig. 4 shows the uptime of the devices, allowing us to identify the moment each one fails. We can also observe an increase in the payload size and the number of allocated *nodes* in the remaining devices each time a device was turned off.

5.2.2 *ES1-B.* This experiment repeats **ES1-A** with physical devices. The payloads and number of *nodes* assigned through the experiment are very similar (*cf.* Fig.5). However, we observe that *Device 2* (the last remaining active), first fails when receiving the payload containing the code for all the *nodes* of the system. This was *expected*, as its constrained memory cannot handle the full payload. It then enters a FAIL-SAFE state, reporting an *Out-of-Memory* error, and forcing the *Orchestrator* to assign it fewer *nodes*.

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 then recover. In Fig. 6, we can observe *Device 3* and *Device 4* failing very early. The system recovers by assign the corresponding *nodes* to other devices. *Device 4* then (1) recovers around 100s, (2) fails again, and (3) recovers. The system disregards this swift failure, and only re-orchestrates the second time *Device 4* recovers. During this experiment, *Device 3* and *Device 4* continue to fail and recover in a predictable pattern, and the system keeps re-configuring itself.

[3]Which may be due to optimization differences between the Docker-compatible and ESP-compatible MicroPython firmwares, garbage collector runs, and libraries.

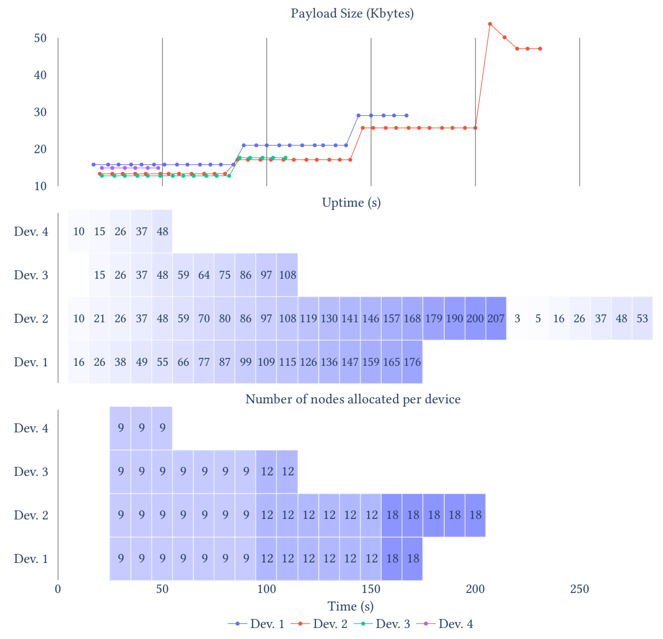

Figure 5: ES1-B measurements

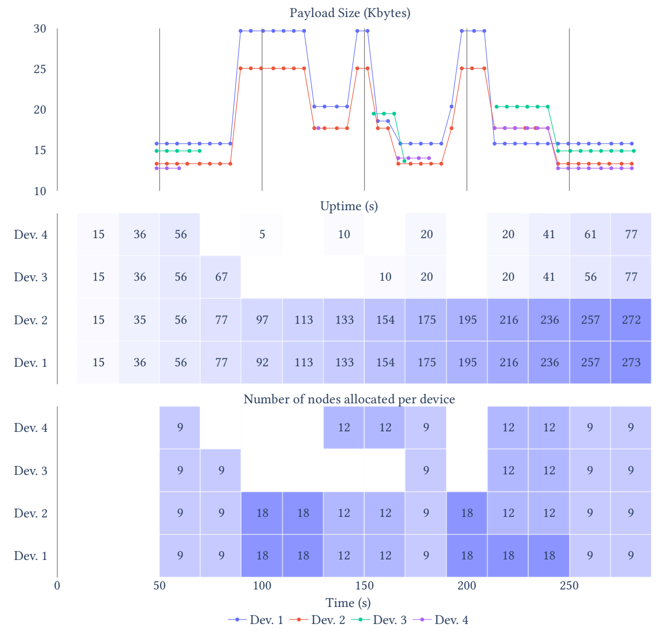

Figure 6: ES1-C measurements.

The precise decision heuristic might need further investigation, as a device that enters a fail/recover loop might introduce continuous re-orchestrations (essentially an unintended DoS).

5.2.4 ES1-D. The memory constraints of IoT devices can negatively impact the functioning of the system, by raising out-of-memory errors when writing the received script into the device's SPI flash. This experiment assesses how the system recovers and adapts in these conditions. Fig. 7 depicts the system behavior due to the constrained memory of *Devices 2* and *4*. When the first assignment is made, at ≈50s, both these devices enter a FAIL-SAFE state 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. We then turn *Device 2* off, and later on. As it can be observed, once *Device 2* stops, *nodes* are distributed to 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. The fact that *Device 4* fails after *Device 2* recovered implies that the system repeated the original assignment decision, ignoring previously known information about memory constraints. This is a known limitation to be addressed in the future.

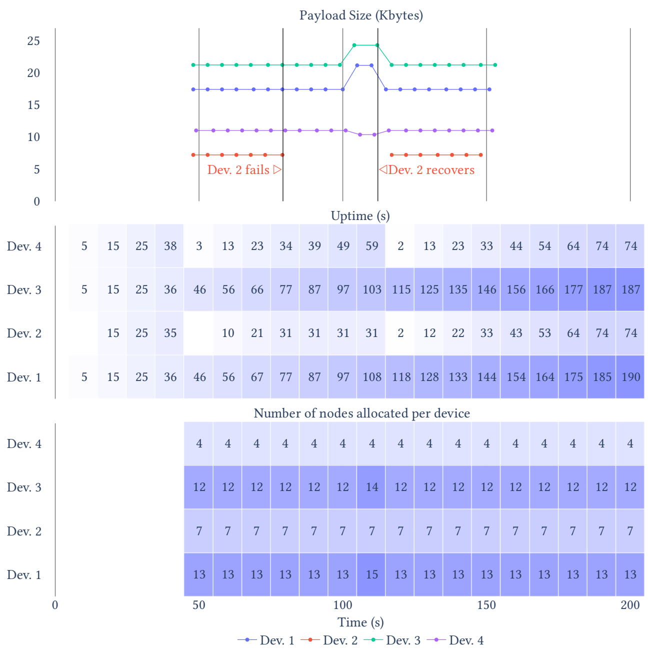

Figure 7: ES1-D measurements.

5.2.5 ES1-E. Besides memory limitations, we also expect the system to be capable of handling unhealthy devices with memory leaks. *Device 2* was modified to always generate an *Out-of-Memory* error after a random period. We expected the system to, eventually, exclude this device during the assignment process. Fig. 8 shows that *Device 2* consistently fails after the first assignment of *nodes* at ≈75s. The number of *nodes* assigned keeps decreasing, until the device is excluded from consideration. This is currently a simple process, in which the system will decrease the number of *nodes* it assigns to a device every time it reports an *Out-of-Memory* to the orchestrator. Once the minimum number of *nodes* reaches zero, the device is excluded from the assignment process.

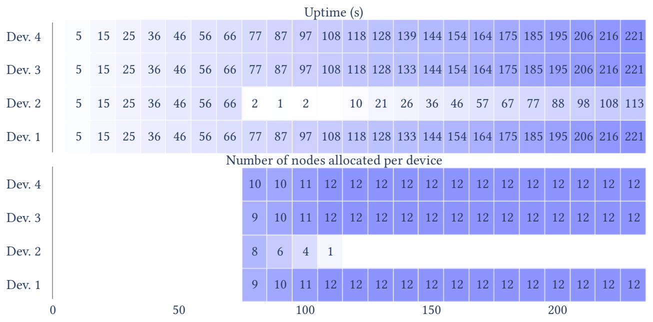

Figure 8: ES1-E measurements.

5.2.6 ES1-F. To further assess the resilience of the system, *nodes* causing errors in specific devices were deliberately added. We expected for the system to re-orchestrate and converge to a solution where those specific *nodes* were assigned to devices not affected by the problem. Neither the system nor the devices know which specific *node*/device combinations are causing the faulty behaviour, so this scenario is overall interpreted by the orchestrator as a device problem. As the first assignment could be correct by sheer chance, we forced a system re-orchestration, by turning off and on all devices in a random order, and repeating the process three times. Fig. 9 shows these on/off events at the ≈125s, ≈200s and ≈275s timestamps. In this case, the devices affected by the faulty *nodes* were *Device 2* and *Device 4*. The event we aim to test occurs at ≈300s. As can be seen in Fig. 9, 10 *nodes* are assigned to *Device 4*. 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 entered a FAIL-SAFE state. 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 the missing *node* is assigned to *Device 1*. Since *Device 4* does not enter a FAIL-SAFE state, the *node* assigned to *Device 1* must have been the faulty one.

5.2.7 ES1-G. To further investigate possible limitations in our current approach, we proceeded to inject constant failures in the available devices. Every second, each device has a $p = 5\%$ of becoming unavailable for 0–10s. During this period, the device becomes unresponsive, announcing itself only when it recovers. Fig. 10 shows that the system is kept continuously re-orchestrating. But once the majority of devices fail, 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 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. Despite that at ≈100s there is a period where all devices are available, the *orchestration* does not converge during

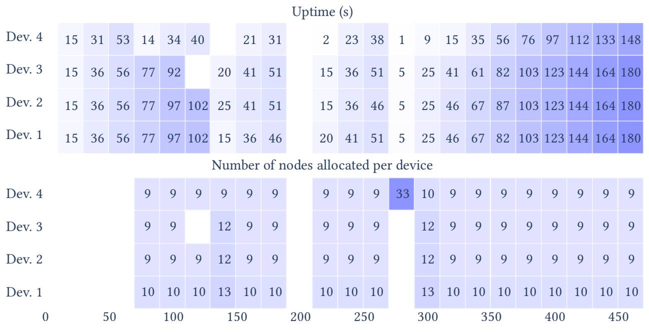

Figure 9: ES1-F measurements.

that time. This is due to the system re-orchestrating 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 the impact of our approach we proceeded to incrementally instrument NodeRED with partial implementations and measure each of them. Our setup consists in a *flow* that passes a message through several devices, recording the total roundtrip. 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.

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 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

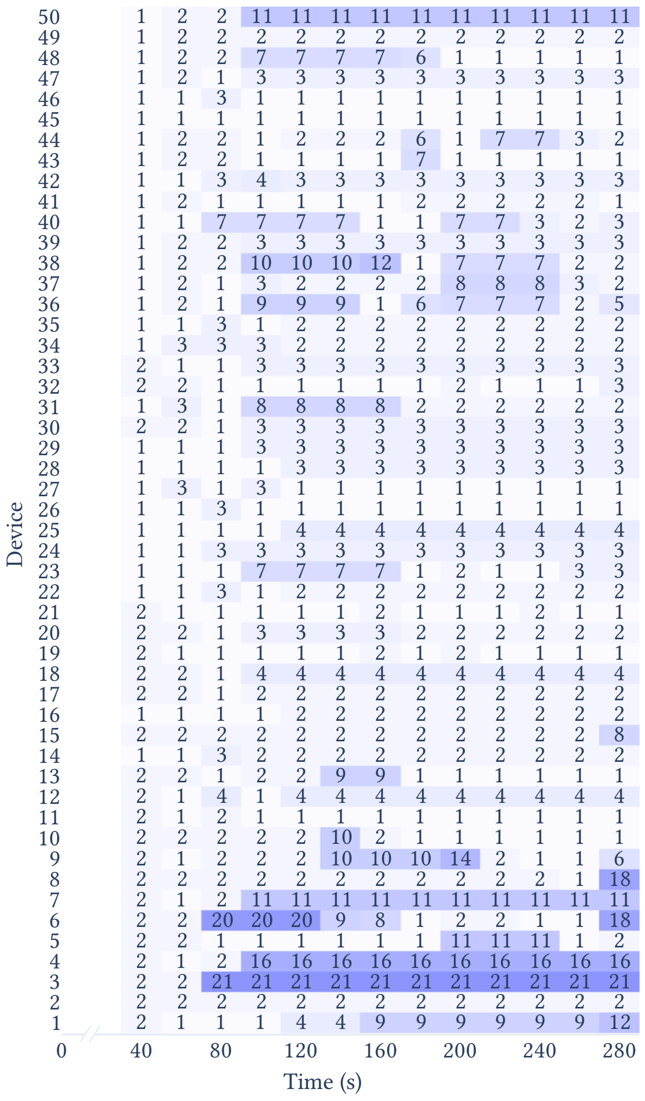

Figure 10: Nodes assignment distribution over time.

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 other communication mechanisms.

6 CONCLUSIONS

In this paper we presented both a method and an extension to NodeRED that provides automatic decentralized orchestration of constrained devices in an IoT network. We proceeded to evaluate it through 2 experimental setups, divided into 13 scenarios. We have shown that our approach is able to provide a decentralized substrate for computation and dynamic adaptation of the system via self-reconfiguration. We mainly addressed three quality attributes: (1) resilience, to which we provide evidence that it is moderately robust, handling device failures and memory constraints dynamically; (2) elasticity, by showing a moderate-sized IoT system functioning in a decentralized fashion with devices being added and removed in runtime; and (3) efficiency, where we investigate the overheads introduced by our approach, and conclude that most of them come from the extra latency introduced by the communication channels, and very few from our proposed firmware. We identify some limitations and possible improvements to our approach, namely: (1) the algorithm used to orchestrate the *nodes* among the available resources can fail to find a suitable configuration, (2) there's room for optimizations, *e.g.*, bypassing the communication substrate between nodes assigned to the same device (although this might hinder observability) and trying to increase the likelihood of such sets of nodes being assigned to the same device (via static and/or dynamic analysis of the flow [23]), (3) other firmware approaches could be explored beyond MicroPython, and (4) the (re)orchestration currently redeploys the entire system and does not attempt to take into consideration a set of minimal changes.

ACKNOWLEDGMENTS

This work was partially funded by the Integrated Masters in Informatics and Computing Engineering (MIEIC) of the Faculty of Engineering, University of Porto (FEUP) and the Portuguese Foundation for Science and Technology (FCT), *ref.* SFRH/BD/144612/2019.

REFERENCES

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. 2015. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys Tutorials* 17, 4 (2015), 2347–2376.

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

[3] 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.

[4] Charles Bell. 2017. *MicroPython for the Internet of Things.* Springer.

[5] 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.

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

[7] 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.

[8] 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.

[9] J. P. Dias, J. P. Faria, and H. S. Ferreira. 2018. A Reactive and Model-Based Approach for Developing Internet-of-Things Systems. In *11th International Conference on the Quality of Information and Communications Technology (QUATIC).*

[10] 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, 357–370.

[11] João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. 2020. A Pattern-Language for Self-Healing Internet-of-Things Systems. In *Proceedings of the 25th European Conference on Pattern Languages of Programs (EuroPLoP 2020)* (Online). ACM.

[12] 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.

[13] 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

[14] 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

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

[16] 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.* 155–162.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Felicien Ihirwe, Davide Di Ruscio, Silvia Mazzini, Pierluigi Pierini, and Alfonso Pierantonio. 2020. Low-Code Engineering for Internet of Things: A State of Research. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (Virtual Event, Canada) *(MODELS '20).* Association for Computing Machinery, New York, NY, USA, Article 74, 8 pages.

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

[23] Tiago Matias, Filipe F Correia, Jonas Fritzsch, Justus Bogner, Hugo S Ferreira, and André Restivo. 2020. Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis. In *European Conference on Software Architecture.* Springer, 315–332.

[24] Julien Mineraud, Oleksiy Mazhelis, Xiang Su, and Sasu Tarkoma. 2016. A gap analysis of Internet-of-Things platforms. *Computer Communications* 89-90 (2016).

[25] 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.

[26] 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.

[27] Joseph Noor, Hsiao Yun Tseng, Luis Garcia, and Mani Srivastava. 2019. DDFlow: Visualized declarative programming for heterogeneous IoT networks. In *Proceedings of the 2019 Internet of Things Design and Implementation.* ACM, 172–177.

[28] 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.

[29] 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.

[30] 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.

[31] 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).

[32] Joanna S., T. Szydlo, M. Windak, and R. Brzoza-Woch. 2019. *FogFlow - Computation Organization for Heterogeneous Fog Computing Environments.*

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

[34] 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).*

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

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