FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Decentralized Real-Time IoT Orchestration

Pedro Miguel Sousa da Costa

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Hugo Sereno Ferreira, Assistant Professor

Second Supervisor: André Restivo, Assistant Professor

July 21, 2021

Decentralized Real-Time IoT Orchestration

Pedro Miguel Sousa da Costa

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Gil Gonçalves

External Examiner: Prof. Waldir Júnior

Supervisor: Prof. Hugo Sereno Ferreira

July 21, 2021

Abstract

Internet-of-Things (IoT) represents the seamless connection of anything to the internet. It comprises a growing network of connected edge devices, both virtual and physical, that can communicate among themselves, enabling smarter environments. These devices are of high heterogeneity, with ranging computational and communication capabilities that have been steadily increasing over the years. Several tools have been proposed to build these systems, following distinct architectures and presenting various interfaces, each with its advantages and drawbacks.

One of the most common architectures is centralized ones, where the main component — the orchestrator and processing unit, typically deployed in the Cloud — performs all the computation on data provided by devices. With this architecture (a) the rising computational capabilities of each device are being wasted, (b) a single point of failure is introduced into the system, and (c) the information has to be transferred across boundaries when even it is not strictly necessary. Node-RED is an example of such centralized tool and is amongst the most popular, given its simple flow-based visual programming approach and extendibility. Conversely, a decentralized architecture that aims to increase system dependability, if applied, could possibly tackle the previous issues by distributing the computation across devices.

The recent emergence of Fog and Edge computing has shifted data computing and storage towards the edge of the network and closer to the users, making better use of the capabilities of each device. However, this shift and move towards a decentralized architecture required complex orchestration mechanisms capable of decomposing the system into smaller tasks that could be allocated to edge devices. Our systematic literature review showed that most of the tools used to create IoT systems currently still either follow a centralized architecture or provide only a centralized orchestration solution to the challenges introduced by the Fog and Edge paradigms.

In this work, we propose a strategy towards a more decentralized orchestration for IoT systems capable of operating with minimal dependency on a central entity. We claim that a less capable decentralized orchestration can provide better results than a more capable centralized orchestration when faced with failures from the central orchestrator.

To achieve this, we develop an orchestration strategy based on (1) a pre-computation of the tasks' allocations, (2) a Task Repository containing the code of each task, and (3) a failure detection mechanism, enabling devices to obtain their tasks in a coordinated and decentralized manner. We validate our method through a reference implementation based on an extension of Node-RED, which we test in various scenarios using both virtual and physical devices.

Our evaluation process shows that our decentralized orchestration strategy can correctly allocate the tasks to devices and adapt in run-time to devices' failures while only initially requiring the orchestrator for the pre-computation of the allocations. Additionally, we contribute to the Node-RED community by publishing our work as a package to the Node-RED Library.

Keywords: Decentralized Orchestration, IoT, Node-RED, Allocation, Fog, Edge.

Resumo

A Internet-of-Things (IoT) representa a conexão de praticamente tudo à internet. Abrange uma rede em crescimento de dispositivos conectados, tanto físicos como virtuais, que comunicam entre si, possibilitando ambientes mais inteligentes. Estes dispositivos são de alta heterogeneidade, com capacidades computacionais e comunicativas diversas que têm vindo a aumentar ao longo dos anos. Várias ferramentas foram propostas para construir estes sistemas, apresentando arquiteturas e interfaces distintas, cada uma com as suas vantagens e desvantagens.

As arquiteturas centralizadas são das mais comuns, onde um componente central — o orquestrador e a unidade de processamento, tipicamente localizado na Cloud — efetua toda a computação necessária nos dados fornecidos pelos dispositivos. Com esta arquitetura (a) as capacidades em crescimentos de cada dispositivo são desperdiçadas, (b) é introduzido um ponto central de falha no sistema, e (c) a informação têm de ser transferida entre vários domínios mesmo quando não é estritamente necessário. Node-RED é um exemplo de uma ferramenta centralizada e uma das mais populares, dada a sua abordagem baseada no fluxo da aplicação com programação visual e a sua extensibilidade. Por outro lado, arquiteturas descentralizadas que visam aumentar a *dependability* de um sistema, quando aplicadas, poderiam resolver os problemas previamente mencionados ao distribuir a computação pelos dispositivos.

A recente emergência dos paradigmas de computação Fog e Edge têm deslocado a computação e o armazenamento da informação para a periferia da rede e mais próximo da fonte, utilizando de melhor forma as capacidades de cada dispositivo. Contudo, esta deslocação e aproximação de uma arquitetura descentralizada requer um mecanismo complexo de orquestração capaz de decompor o sistema em várias tarefas e de as alocar aos dispositivos. A nossa revisão da leitura mostrou que a maioria das ferramentas utilizadas para criar sistemas de IoT ainda seguem uma arquitetura centralizada ou apresentam apenas uma orquestração centralizada para colmatar as dificuldades introduzidas pelos novos paradigmas.

Neste trabalho, propomos uma estratégia em direção à descentralização da orquestração para sistemas IoT capaz de funcionar com o mínimo de dependência numa entidade central. Afirmamos que uma orquestração descentralizada com menos capacidades do que uma orquestração centralizada consegue atingir melhor resultados considerando falhas do orquestrador central.

Com isto em mente, desenvolvemos uma orquestração baseada (1) na pré-computação das alocações, (2) num Repositório de Tarefas que contém o código de cada tarefa, e (3) um mecanismo de deteção de falhas, permitindo assim aos dispositivos obter as suas tarefas de uma forma coordenada e descentralizada. Validamos o nosso método através de uma implementação de referência baseada numa extensão do Node-RED, submetendo-a a vários testes usando diferentes cenários e dispositivos virtuais e físicos.

O nosso processo de avaliação mostra que a nossa orquestração descentralizada consegue alocar corretamente as tarefas a cada dispositivo e adaptar-se, em tempo real, às falhas destes apenas requerendo apenas inicialmente o orquestrador para a pré-computação das alocações. Adicionalmente, contribuímos para a comunidade Node-RED ao publicar o nosso trabalho como pacote na biblioteca Node-RED.

Keywords: Decentralized Orchestration, IoT, Node-RED, Allocation, Fog, Edge.

Acknowledgements

First, I would like to thank my supervisors, Professors Hugo Sereno and André Restivo, for guiding me throughout this work, providing me with fundamental insights which allowed me to produce a better work. Additionally, I would like to thank João Pedro Dias for always being present and providing valuable advice and feedback.

Then, I would like to express my deepest and sincere gratitude towards my family, especially my parents and grandmother, for going above and beyond in making sure I had everything I needed. Thank you for everything; I could never be where I am today without you.

To Rita, for always being there for me and reminding me there is so much more to life than work. Thank you for always showing me the better part of life.

Lastly, but not least, I would like to thank all my friends for all the moments of laughter and collective frustration and despair during our journey. Thank you for all the support and jokes. A special thanks to Tiago Fragoso, who developed a work closely related to mine, for pairing with me in tackling our common obstacles and with whom I shared countless rants.

Pedro Costa

"And those who were seen dancing
were thought to be insane by those who could not hear the music."

Friedrich Nietzsche

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Problem . . . 3
- 1.4 General Goals . . . 3
- 1.5 Document Structure . . . 4

2 Background **5**
- 2.1 Internet of Things . . . 5
  - 2.1.1 Fog Computation . . . 6
  - 2.1.2 Edge Computation . . . 7
- 2.2 Real-Time Operating Systems . . . 7
- 2.3 Visual Programming . . . 8
- 2.4 Fault Tolerance . . . 9
- 2.5 Summary . . . 10

3 State of the Art **11**
- 3.1 Systematic Literature Review . . . 11
  - 3.1.1 Methodology . . . 11
  - 3.1.2 Research Questions . . . 11
  - 3.1.3 Databases . . . 12
  - 3.1.4 Search Query . . . 12
  - 3.1.5 Results . . . 13
  - 3.1.6 Expanded Search . . . 20
  - 3.1.7 Results Categorization . . . 23
  - 3.1.8 Evolutionary Analysis . . . 24
  - 3.1.9 Results Analysis . . . 24
  - 3.1.10 Survey Research Questions . . . 25
- 3.2 Extended Review . . . 26
  - 3.2.1 Methodology . . . 26
  - 3.2.2 Results . . . 26
- 3.3 Summary . . . 28

4 Problem Statement **31**
- 4.1 Current Issues . . . 31
- 4.2 Desiderata . . . 32
- 4.3 Scope . . . 33
- 4.4 Main Hypothesis . . . 33
- 4.5 Research Questions . . . 34
- 4.6 Methodology . . . 34
- 4.7 Summary . . . 34

5 Implementation 37
5.1 Overview . . . 37
5.1.1 Initial Allocation Behaviour . . . 38
5.1.2 Subsequent Reallocations . . . 39
5.2 Decentralized Orchestration . . . 40
5.3 Failure Detection . . . 41
5.4 Node Operating Modes . . . 44
5.5 Known Limitations . . . 45
5.6 Summary . . . 46

6 Evaluation and Validation 49
6.1 Experimental Setup . . . 49
6.1.1 System . . . 49
6.1.2 Flows . . . 50
6.1.3 Metrics . . . 52
6.2 Experiments . . . 53
6.2.1 Sanity Checks . . . 53
6.2.2 Improvements . . . 54
6.2.3 Limitations . . . 54
6.3 Discussion . . . 55
6.3.1 Sanity Checks . . . 55
6.3.2 Improvements . . . 61
6.3.3 Limitations . . . 68
6.4 Replication Package . . . 73
6.5 Summary . . . 73

7 Conclusions 75
7.1 Conclusions . . . 75
7.2 Contributions . . . 77
7.3 Challenges . . . 77
7.4 Future Work . . . 78

References 79

List of Figures

2.1 Fog Computing Architecture. 6
2.2 Node-RED Editor. 9
2.3 Fault tolerance techniques. 9

3.1 SLR Methodology Diagram. 13
3.2 DDFlow Architecture. 15
3.3 DDF Coordination States. 18
3.4 CityFlow Architecture. 19
3.5 FogFlow Architecture. 22
3.6 Silva *et al*. High-level Architecture. 23
3.7 Years of the Publications. 24
3.8 Colistra *et al*. Reference Scenario. 27

5.1 Sequence of operations in deploying a flow. 38
5.2 High level overview of the solution modules. 39
5.3 Failure Response Sequence Diagram. 43
5.4 Node-RED fallback mechanism on two temperature nodes. 45

6.1 Flow Setup 1. 50
6.2 Flow Setup 2. 51
6.3 Flow Setup 3. 51
6.4 Flow Setup 4. 52
6.5 SC1-VS1 Measurements. 56
6.6 SC1-PS1 Measurements. 57
6.7 SC2-VS1 Measurements. 58
6.8 SC2-PS1-A Measurements. 60
6.9 SC2-PS1-B Measurements. 62
6.10 I1-VS1 Measurements. 63
6.11 I1-VS1 Measurements. 64
6.12 I2-VS2 Measurements. 66
6.13 I2-PS2 Measurements. 67
6.14 I3-VS2 Measurements. 69
6.15 I3-PS2 Measurements. 70
6.16 L1-VS1 Measurements. 71
6.17 L2-ES1 Measurements. 72

List of Tables

3.1 Inclusion and Exclusion Criteria. 12
3.2 SLR Results Categorization. 24

6.1 System Metrics Gathered During the Experiments. 53

Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
EC	Edge Computing
FC	Fog Computing
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
LWT	Last Will and Testament
MQTT	Message Queuing Telemetry Transport
MWS	Minimum Working System
P2P	Peer to Peer
RAM	Random Access Memory
RTOS	Real Time Operating System
SLR	Systematic Literature Review
SPoF	Single Point of Failure
SRQ	Survey Research Question
VP	Visual Programming
VPL	Visual Programming Language

Chapter 1

Introduction

1.1 **Context** 1
1.2 **Motivation** 2
1.3 **Problem** 3
1.4 **General Goals** 3
1.5 **Document Structure** 4

This chapter introduces the context and motivation of this work, along with goals we aim to achieve. Section 1.1 presents the context of this work and Section 1.2 the motivation behind it. Section 1.3 describes the problem at hand and Section 1.4 briefly presents the goals of this work. Finally, Section 1.5 presents the structure of this work.

1.1 Context

The Internet of Things (IoT) paradigm represents the connectivity of everyday objects to the Internet. This growing network enables both physical and virtual objects to share information, allowing for coordinated decisions in a highly heterogeneous environment [5]. In fact, the high heterogeneity and device distribution are the main characteristics of this network, which comprises devices of ranging computational and communicational capabilities. By exploiting both the capabilities of these devices and their pervasiveness, IoT enables smarter environments with applications in various domains, from home automation to healthcare and smart cities [1].

Interest in IoT has been steadily increasing in the past years, so far as being included in the "Disruptive Civil Technologies" list by the US National Intelligence Council [37]. This rise in popularity derives from the increasing number of existing IoT devices, with projections of reaching 75 billion connected devices and generate up to $11.1 trillion a year in economic value by 2025 [42, 65]. Following this trend, IoT systems started to become more complex, combining a plethora of different devices without adhering to a specific architecture [38], leading to an increasingly challenging environment to develop on [24].

As this kind of systems spread among various application domains, it became increasingly harder to depend on people with technical knowledge to configure and operate these systems [63]. This created a necessity to simplify this process, and one of the solutions was to integrate visual programming solutions in IoT. Visual Programming Languages (VPL) enables the user to describe processes more intuitively and visually, rather than relying only on text. Using elements such as boxes and arrows allows for a clearer view of the program's flow, creating a more pleasant

environment. VPLs are usually designed for a specific goal, such as education, simulation and IoT Systems.

Node-RED[1] is one of the most popular VPLs [27] in the IoT domain that provides a simple flow-based way of wiring together all kinds of devices and online services. It provides a centralized architecture where a single entity performs all the computation on the data provided by the devices and services. However, centralized architectures carry inherent limitations that affect several attributes of the system, such as its availability, resiliency and fault-tolerance by depending on the central entity. This entity presents, in itself, a single point of failure that can bring the entire system to a halt if it fails, which is even more threatening as IoT systems embrace mission critical applications [31, 53].

Conversely, following a decentralized architecture, a system could, in theory, partition itself into smaller tasks that could be allocated to the edge devices, making better use of the computation found at the edge levels. Moreover, the distribution of the application logic could as well remove the need for a central entity, thus possibly eliminating the single point of failures.

1.2 Motivation

The IoT sector has constantly been rising, and its applications are widespread in today's world. However, the lack of standards in developing IoT applications — mainly due to its rapid growth and high heterogeneity — has led to a particularly challenging environment to develop on. Users have to choose from a wide range of languages, protocols and platforms without a unified consensus of what works best [2].

Given this difficulty, VPLs and other low-code approaches have been adapted to this context, providing a friendly and visual way of creating IoT applications, hiding most of the complexity of low-level tiers, languages or protocols from the user [24, 66, 39].

However, most of these tools provide only a centralized orchestration mechanism [45] where a central node performs all the computation, hindering the entire system resilience — if the central component fails, the system cannot operate. Since all the computation is being performed in one place, the remaining devices' rising capabilities are being wasted, and it forces information to cross different boundaries, possibly violating legal constraints even when it is not necessary.

With the emergence of Fog and Edge computing, there has been a shift in computing and storage towards the edge of the network and closer to the users (*cf.* Section 2.1, p. 5), making better use of the capabilities of each device. However, this shift in the computation required a more complex orchestration of the system, *i.e.*, the decomposition of application into smaller computational units and their respective allocation to edge devices [9, 52, 49, 60, 61]. Furthermore, it required a more complex monitor system capable of handling the failure of devices that are no longer simple data sources but a fundamental part in maintaining the correct behaviour of the application logic [56]. Once again, these responsibilities were assigned to a central entity — the

[1] Node-RED, https://nodered.org/

orchestrator — rendering him a single point of failure (SPoF), even when each device is actively contributing.

Thus, in this scenario, it would be ideal to provide an alternative orchestration mechanism capable of decomposing the system and allocating its task according to changes in the availability of the edge devices without entirely depending on a central orchestrator.

1.3 Problem

The majority of IoT systems currently follow a centralized architecture [45] where the main component — the orchestrator and processing unit — performs all the computation on the data provided by the devices. This type of architecture is often imposed by the tool used to create the system, as is Node-RED. Nevertheless, this architecture introduces several limitations that may hinder the entire system dependability. First, the orchestrator presents, in itself, a single point of failure (SPoF), bringing the entire system to a halt if it fails. Secondly, given that all the computation is being performed in one place (*i.e.*, cloud), the remaining devices' rising capabilities are being wasted [64]. Thirdly, information is transferred across different boundaries to the Cloud, possibly violating legal constraints even when it is not necessary [59].

Furthermore, the systems that present a decentralized architecture — with the introduction of the Fog and Edge computing paradigms — still rely on a centralized orchestration, where a central entity decomposes the system into smaller tasks and dictates their allocation to the edge devices.

In this work, we intend to improve an existing orchestration mechanism, that can leverage the rising capabilities of edge devices, by introducing new capabilities that lower its dependency on the central orchestrator. More specifically, an orchestration capable of decomposing and allocating the system tasks to multiple devices that can operate when faced with failures from the orchestrator. The problem under study is further analyzed in Chapter 4 (p. 31).

1.4 General Goals

The main goal of this dissertation is to develop a decentralized orchestration strategy for IoT systems. To achieve our goal, we extend an existing decentralized computation platform created by Silva *et al*. [60]. This platform, created as an extension of Node-RED, provides a centralized orchestration capable of partitioning and allocating the tasks to edge devices and a custom MicroPython firmware that allows the edge devices to run arbitrary MicroPython code.

Our decentralized orchestration should (1) allow for devices to obtain their assigned tasks and (2) provide (re)allocations to handle changes in the system with minimal dependency on the availability of the central orchestrator.

Additionally, as a secondary goal, we intend to provide our platform as an isolated Node-RED package that any user can easily install. For this, our solution must be able to decouple the previous modifications made to Node-RED.

1.5 Document Structure

This document is composed of seven chapters, structured as follow:

- Chapter 1 (p. 1), **Introduction**, presents the problem under study, as well as its motivation, goal and validation process;
- Chapter 2 (p. 5), **Background**, introduces and explains core concepts necessary to fully understand this work;
- Chapter 3 (p. 11), **State of the Art**, describes the current research state on this context, providing a Systematic Literature Review on the decentralized IoT systems and an extended review on decentralized task allocation algorithms and consensus mechanisms.
- Chapter 4 (p. 31), **Problem Statement**, presents the problem under study in more detail and the proposed approach to solve it;
- Chapter 5 (p. 37), **Implementation**, discusses the implementation our of solution and provides the justification for the various technical choices made;
- Chapter 6 (p. 49), **Evaluation and Validations**, presents our validation process and analyzes the obtained experimental data;
- Chapter 7 (p. 75), **Conclusions**, summarizes the developed work, reflecting on the initial goal set forth by our hypothesis and presenting its main contributions and future work.

Chapter 2

Background

2.1 **Internet of Things** 5
2.2 **Real-Time Operating Systems** 7
2.3 **Visual Programming** 8
2.4 **Fault Tolerance** 9
2.5 **Summary** 10

This chapter introduces and describes key concepts necessary to fully understand this work. Section 2.1 presents the IoT paradigm and describes the computing paradigms associated with it. Then, Section 2.2 (p. 7) introduces real time operating systems and their usage in IoT. Section 2.3 (p. 8) introduces Visual Programming Languages and details Node-RED. Then, Section 2.4 (p. 9) provides a definition of fault tolerance. Finally, Section 2.5 (p. 10) summarizes the key concepts introduced in this chapter.

2.1 Internet of Things

Internet of Things is defined by Atzori *et al.* [6] as:

> *"a conceptual framework that leverages on the availability of heterogeneous devices and interconnection solutions, as well as augmented physical objects providing a shared information base on global scale, to support the design of applications involving at the same virtual level both people and representations of objects."*

This global network created by IoT has allowed a seamless connection between humans and things — sensors, actuators and everyday devices — producing enormous quantities of information for analysis [40]. Given the ubiquity of these devices, IoT applications can be deployed to a wide range of domains, such as: transportation and logistics, healthcare and home automation [5].

IoT applications are usually divided into three tiers [69, 25]:

Cloud: characterized by large quantities of easily accessible virtualized resources (data centers and servers), able to dynamically scale, but offering high latency;

Fog: composed of heterogeneous gateways and devices in between the cloud and the edge of the network, providing fewer computational resources but also less latency;

Edge: composed of all the devices at the edge of the network — *e.g.*, sensors, actuators, mobile phones — providing highly constrained computational resources but the minimum latency possible.

The majority of IoT applications depend on Cloud-based computing or storage, requiring large amounts of storage space to be available. However, with the increasing number of IoT devices [42], the data created by such devices is predicted to be two orders of magnitude higher than data stored [18]. A cloud-base centralized approach is showing difficulty in coping with this increase in both the volume and heterogeneity of end devices and the data they produced. For applications with low latency requirements, such as real-time systems, the latency found in the cloud also proves to be unacceptable [30]. This led to the need of distributed computing in IoT and emergence of new computing paradigms in IoT: Fog and Edge Computing.

2.1.1 Fog Computation

Fog Computing (FC) was introduced by Bonomi *et al.* [10] in 2012 as a platform placed in between the Cloud and the end devices to provide intermediary compute, storage and networking services. It can be seen as an extension of Cloud Computing to help it achieve the rising requirements of IoT applications such as low latency, by bringing computation closer to the data source. This computation is powered by typical network devices, *e.g.*, routers and switches, that are placed closer to the end devices. Besides their networking responsibilities, these devices are usually also paired with processing and memory resources to provide support for data processing. However, in some scenarios, *cloudlets* or *Fog servers* are also employed to provide further processing power and storage [41].

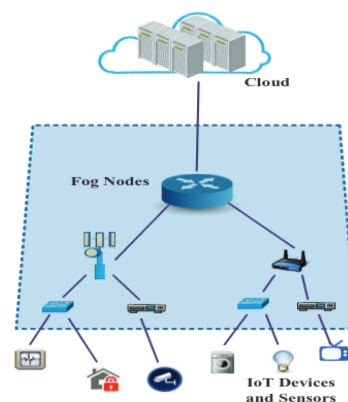

Figure 2.1: Fog Computing Architecture [41].

However, compared to Cloud Computing, FC faces new issues: (1) not all devices found at Fog levels can support general-purpose computation and upgrading them can be challenging; (2) the selection of suitable nodes and their place of deployment becomes vital and complex due to the heterogeneity found at this level. Nonetheless, FC brings a considerable set of advantages over

only Cloud-based solutions, requiring less network traffic and latency, and by offloading some workload from the Cloud — which, as seen before, can start to become bottleneck.

2.1.2 Edge Computation

Edge Computing (EC), based on the idea that computing should be made close to the data sources, pushes the computation towards the edge of the network. Shi *et al*. [59] define edge as *"any computing and network resources along the path between data sources and cloud data center"*.

Besides the advantages mentioned previously in bringing computing closer to data sources, the authors also note that, while in Cloud computing paradigm the edge devices typically act as data consumers (*e.g.*, watching a video), there has been a change from sole consumers to producer and consumer devices. Smartphones and wearables, for example, produce enormous amount of data that would be unfeasible to process it all in the cloud, reaching bandwidth limits in some scenarios. Thus, new techniques and functions are required at the edge level.

This paradigm also prevents unnecessary data communication with the Cloud and, with that, provides more privacy for the user given that private information, such as information produced by physical wearables, does not have to be sent to the Cloud and can be processed locally [59].

However, this approach faces several challenges, of mainly due to the high number and high heterogeneity of devices: (1) the devices present several computing and energy constraints; (2) the runtimes of each device may vary which makes the development of an application harder; (3) there is not a standard way of naming and addressing edge devices that can handle the high mobility and dynamic network topology of edge devices; (4) it becomes very hard to detect that an edge device failed and assess why, reducing the reliability of the system.

Finally, Fog Computing can be confused with Edge Computing. In this work, the differentiation is the same as presented in [59]: *"edge computing focus more toward the things side, while fog computing focus more on the infrastructure side."*

2.2 Real-Time Operating Systems

As seen in Section 2.1 (p. 5), devices found at the edge of the network present highly constrained computational resources (*e.g.*, reduced amount of memory, low computing power, smaller batteries). Nonetheless, each device has to perform several tasks concurrently, such as communication and task computing, and meet real-time requirements imposed by the application, leading to scheduling problems. Leaving this problem to developers is not desirable — the environment is already challenging to develop on — and should be the responsibility of the Operating System. However, typical OS (*e.g.*, Linux and Windows) present excessive resources requirements that cannot by fulfilled by these devices. Furthermore, the schedulers of these OS present difficulties in coping with the real-time requirements [44]. Thus, Real-time Operating Systems (RTOS) become ideal for edge devices.

Generally developed for embedded systems, RTOS offers a scheduler capable of satisfying response-time constraints, while providing a small footprint and overhead (by only including the

necessary functionality). Baskiyar *et al.* [7] present a survey on the existing commercial RTOS, such as LynxOS[1] and VxWorks[2]. They list the basic requirements of a RTOS, which are: (1) Multi-tasking and preemptable; (2) Dynamic deadline identification; (3) Predictable synchronization; (4) Sufficient priority levels; and (5) Predefined latencies.

Milinkovic *et al.* [44] present a short survey on open-source RTOS and their applicability to IoT platforms. The different solutions were analysed the in terms of their kernel structure (*i.e.*, monolithic, layered and microkernel architecture), the different type of schedulers and algorithms provided and the programming model. Following this analysis, the authors selected one of the solutions, RIOT[3], to port into a target test platform, documenting their difficulties. Other solutions analyzed in this paper are FreeRTOS[4] and ChibiOS[5].

2.3 Visual Programming

Visual Programming (VP) is a type of programming that make use of multiple dimensions to build and specify programs in a graphical manner, as opposed to traditional textual languages that only convey semantics in a one-dimensional stream [47, 11]. These additional dimensions are represented graphically, as visual expressions, such as time and spatial relations. Programming languages that make use of these visual expressions are called Visual Programming Languages (VPL).

VP was one of the approaches to make coding generally more accessible to users, given the difficulty in learning and using traditional programming languages. By creating an extra layer of graphical abstraction, the user can better understand how the program works. VP can also improve the correctness and speed with which users develop applications. As presented by [11], VPL often make use of four strategies: (1) *Concreteness*; (2) *Directness*; (3) *Explicitness*; (4) *Immediate Visual Feedback*. Dataflow Visual Programming is one of the most explicit paradigms used in VPL where operations are usually enclosed in boxes and connected with arrows, explicitly highlighting the flow of the application.

Node-RED is an open-source Dataflow VPL for the development of IoT applications, originally developed by IBM[6] and now part of the JS Foundation[7]. It provides a browser-based flow editor to wire together different kind of physical devices, API and online services (*cf.* Figure 2.2, p. 9).

Applications consist of a network of *black-boxes*, called nodes, connected with *wires* that create *flows*. In the browser editor, the user can drag nodes from the *palette* and wired together. The palette can be extended by installing new nodes provided by the active community in JSON files. Each node can then produce, consume and process data with custom scripts written in

[1] LynxOS, https://www.lynx.com/products/lynxos-posix-real-time-operating-system-rtos
[2] VxWorks, https://www.windriver.com/products/vxworks
[3] RIOT, https://www.riot-os.org/
[4] FreeRTOS, https://www.freertos.org/
[5] ChibiOS, https://www.chibios.org/dokuwiki/doku.php
[6] IBM, https://research.ibm.com/labs/uk/
[7] JS Foundation, https://openjsf.org/

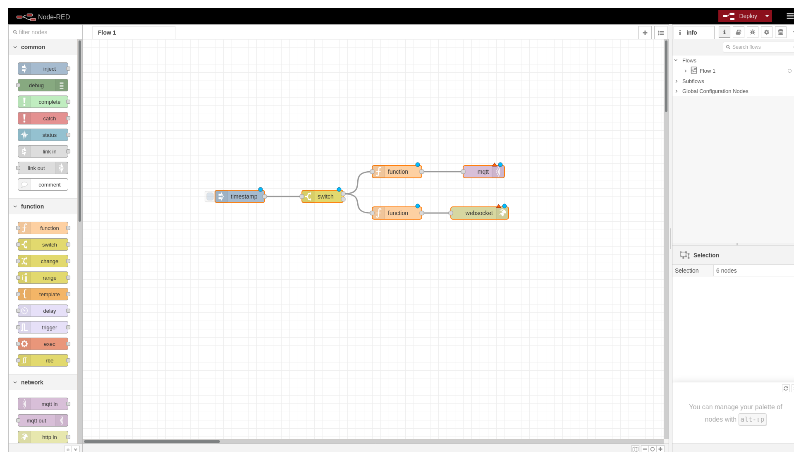

Figure 2.2: Node-RED Editor.

Javascript. The runtime is built on Node.js and, given the event-driven and non-blocking model, it performs well on edge devices.

2.4 Fault Tolerance

Avizienis *et al.* [3] present clear and precise definitions of main concepts related to dependable computing (*i.e.*, a service that can be justifiably trusted), such as errors, service failure, fault prevention and fault tolerance.

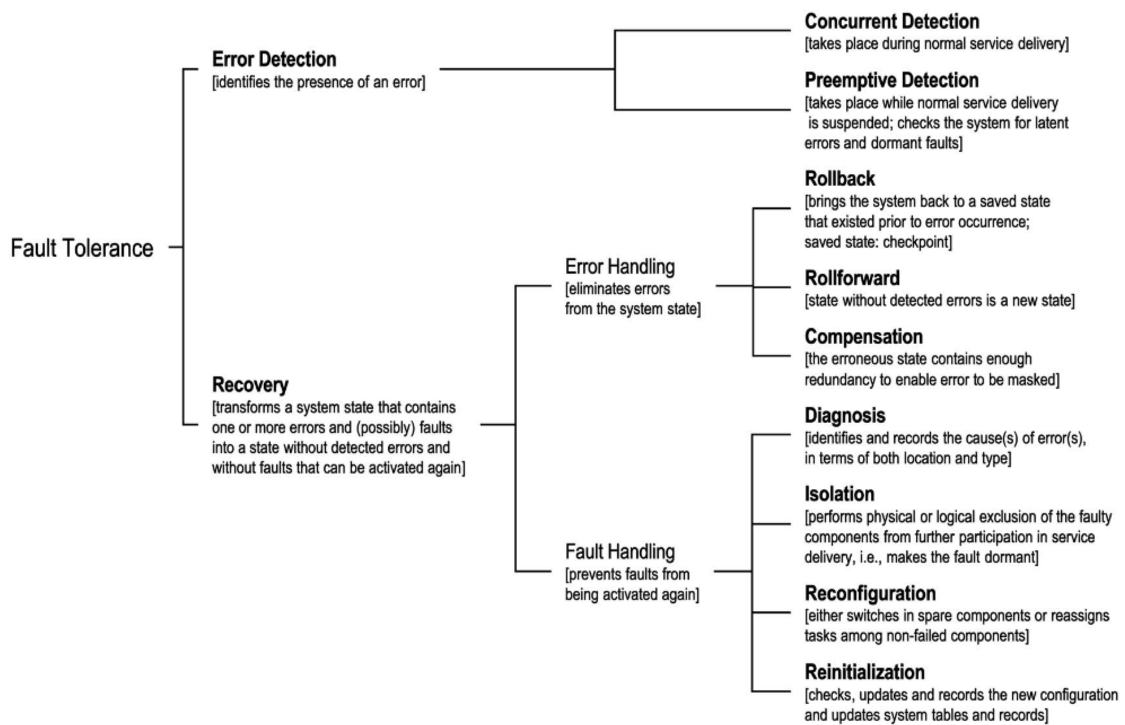

Figure 2.3: Fault tolerance techniques [3].

Fault-tolerance is introduced a mean to to attain system dependability, providing the ability to *"avoid service failure in the presence of faults"*. As illustrated in Figure 2.3 (p. 9), the authors present a strategy for Fault-tolerance based on two phases: (1) error detection, and (2) system recovery. The first phase is dedicated to identify errors in the system, either during normal service delivery or while the system is suspend to check for latent and dormant faults. The second phase aims to transform an erroneous system state into a error-free state that can be activated again by employing both error handling, that eliminates errors from the system, and fault handling, that prevents future faults. The authors present a further method of achieving fault-tolerance based on the redundancy of computational channels, where actions can be made sequentially or concurrently. Additionally, fault tolerance is also a *"recursive concept"*, meaning that the same mechanisms that provide fault-tolerance, should also be protected against faults. Finally, the authors note that fault tolerance and resilience are synonyms. As such, fault tolerance and resilience will be used interchangeably henceforth.

2.5 Summary

This chapter introduced concepts related to the IoT paradigm, along with Real Time Operating Systems, Visual Programming Tools and Fault Tolerance, which are fundamental to understand this work.

Section 2.1 (p. 5) defines IoT and presents the Cloud, Fog and Edge tiers associated with this network. The typical Cloud-based computing paradigm employed by most systems started to show difficulties in coping with the rise in numbers of devices and the volume of data they produced, leading to the appearance of the Fog and Edge paradigms. Fog Computing makes use of devices between the the edge and the cloud for intermediary processing, being considered as an extension of Cloud Computing. Edge Computing moves the computation towards the edge devices, considering that the edge devices are starting to become both producers and consumers of data.

Section 2.2 (p. 7) introduces Real Time Operating Systems, highlighting some of the available solutions. RTOS were created to simplify the development of application with real-time constraints and requirements, by providing a scheduler that can guarantee that a task is accepted and executed in a specific time frame.

Section 2.3 (p. 8) defines Visual Programming Languages, presenting the strategies used to create a more friendly environment to create applications. Node-RED is analysed in more detail, explaining the different parts that compose an application in it, and exposing the expendability of the tool.

Section 2.4 (p. 9) presents a concise definition of fault tolerance — ability to *"avoid service failure in the presence of faults"* — as given by [3]. The different strategies to achieve it are illustrated in Figure 2.3 (p. 9).

Chapter 3

State of the Art

3.1 Systematic Literature Review 11
3.2 Extended Review 26
3.3 Summary 28

This chapter describes the state of the art of decentralization in IoT systems. Section 3.1 presents the Systematic Literature Review conducted to gather information about the state of the art, describing the methodology, survey questions, criteria, and query used during the review. Section 3.1.5 presents the results of this Systematic Literature Review. Section 3.1.6 complements the previous results with other publications that were found in a survey obtained during the review and suggested by experts in the field. Section 3.1.7 organizes the results according to a set of relevant categories and Section 3.1.9 presents an analysis following those categories and answers the previously defined survey questions. Section 3.2 presents an extended review on decentralized task allocation algorithms and consensus mechanisms. Finally, Section 3.3 presents the conclusions of the Systematic Literature Review.

3.1 Systematic Literature Review

A Systematic Literature Review (SLR) was conducted to gather information on the state of the art of decentralizing IoT systems. The SLR process reduces the researcher bias and synthesises results by adhering to a well-defined and repeatable procedure [51].

3.1.1 Methodology

This SLR was done by following a specific methodology to reduce the bias and obtain the best coverage of the work done in this area [51]. First, the research questions to be answered in this work were define as the databases to utilize. Then, the search query was constructed along with the inclusion and exclusion criteria.

3.1.2 Research Questions

The following Survey Research Questions (SRQ) were defined to identify the current state of decentralization in IoT systems that are built using mainly visual programming tools:

SRQ1: *What solutions exist to orchestrate decentralized IoT systems?*

SRQ2: *Which approaches found in SRQ1 move towards a decentralized orchestration?*

SRQ3: *Which approaches found in SRQ1 make use of visual programming?*

SRQ3: *What is the evolution of orchestration solutions to IoT systems over the years?*

By decentralized orchestration we consider a system capable of distributing a set of tasks among its parts without depending on a centralized entity.

3.1.3 Databases

This research includes publications from the following electronic databases: (1) Scopus, (2) IEEE Xplore, and (3) ACM Digital Library.

3.1.4 Search Query

The search process was made with a query that incorporated keywords that were the most relevant and most likely to appear in the target publications:

```
( iot OR "Internet of Things" OR internet-of-things ) AND ( decentralized
    OR distributed ) AND ( "visual programming" OR vpl OR visual-
    programming OR "node red" OR node-red OR data-flow OR dataflow )
```

The results were then filtered according to the inclusion and exclusion criteria defined in Table 3.1.

Table 3.1: Inclusion and Exclusion Criteria.

	ID	Description
Inclusion	IC1	Must be related to decentralized or distributed IoT Systems
	IC2	Explains how the approach works
	IC3	Survey focused on Fog or Edge computing
	IC4	Has a publication date between 2008 and 2020
	IC5	Presents a decentralized method of system resources allocation
Exclusion	EC1	Presents a solution based on blockchain
	EC2	Presents just ideas, tutorials, interviews, and discussion papers
	EC3	Publications older than 2008
	EC4	Not written in English
	EC5	Not available in PDF
	EC6	Duplicated publications
	EC7	Framework or tool cannot handle multiple devices

A diagram of the methodology followed during the SLR is present in Figure 3.1 (p. 13), where we can observe that, in total, 1530 publications were analysed, and 15 were selected, after an evaluation accordingly to the inclusion and exclusion criteria.

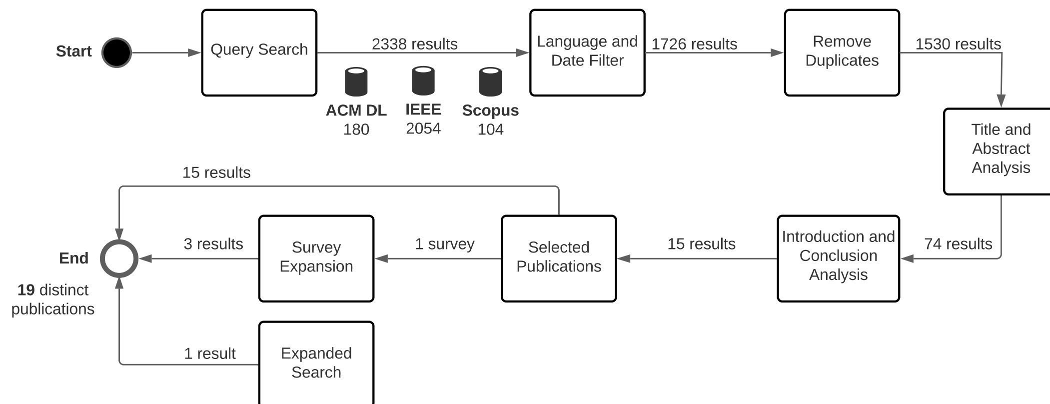

Figure 3.1: SLR Process Diagram.

3.1.5 Results

Of the 15 selected publication, [54] was a survey on Fog Computing (FC) employment in IoT systems. In this survey, Pulifalito *et al.* describe and characterize the concept of FC and the related literature. They provide 6 IoT application domains that may benefit from FC's usage while highlighting the current research directions in this field. Finally, they describe the existing software platforms that provide basic mechanisms to deploy an IoT application over a Fog infrastructure. These platforms are used later on to complement the results of the SLR.

The remaining 14 publication detail tools, approaches and a programming model to build further decentralized IoT system. Three of those publications are relative to the same tool, leaving a total of 12 distinct results. These are:

Reiter *et al.* [57] present an architecture and framework to offload work from mobile devices by leveraging Edge computing, named Hybrid Mobile Edge Computing (HMEC). Whilst the number of transistors per chip has been doubling every 18 months, the performance and battery of mobile batteries usually do the same in ten years. This leads to a clear gap between the growing energy demands by applications and the available battery capacity. This architecture is based on a distributed hash table backed peer-to-peer network comprised of the edge computing units and *super-nodes* that act as contact points for the mobile devices. The mobile devices can then query these nodes for available computing units to offload work; they can then communicate directly with the edge devices to send tasks.

The tasks eligible for offloading are identified via annotations in the client application code that are then processed by a static source code analyzer. This analyzer takes into consideration some parameters such as energy-saving models and latency to decide if that task should be migrated to edge devices or not. With this, they can achieve good improvements in energy consumption and performance. However, the authors fail to mention what happens when edge devices with associated task fail during the process.

Vidyasankar [67] , in the context of Fog Computing, proposes a generic framework to distribute computation tasks in a generic hierarchy of nodes. Using latency as an example, it is stated that if all processing could be done at the edge level, latency could be minimum. However, the typical devices found at this level may not have the needed computational capabilities; at most they could only process some part and delegate the remaining to nodes in upper levels. Given this, the problem lies in decomposing any task into smaller sub-tasks and assign them to appropriate nodes. Characteristics such as processing power, network band with and available storage are taken into consideration. Several algorithms are presented to deal with different scenarios (single, multiple, homogeneous and heterogeneous source inputs). However, the partitioning of tasks into sub-tasks is assumed to have already been done, and no attention is given to that problem.

Noor *et al.* [49] , motivated by the current difficulty in leveraging the innate heterogeneity of IoT devices and coordinating a complex set of these devices in an ad hoc network, present DDFlow, a macro programming abstraction to build high-quality distributed IoT applications. DDFlow allows developers to specify high-level requirements instead of dealing with low-level implementation. The authors developed a distributed system based Node-RED and its graphical interface to validate the abstraction, supporting the DDFlow applications and tools for dynamically scaling and reconfiguration.

As illustrated in Figure 3.2 (p. 15), the system is comprised of two main components: *Coordinator* and *Device*. The Coordinator is a web server that handles DDFlow applications composed of three main parts: (1) a web interface where developers can create the applications; (2) a *Deployment Manager* responsible for interacting with each device; (3) a *Placement Solver* responsible for allocation tasks to available devices, formulated as a linear programming problem to minimize end-to-end latency based on the details of each device. The *Coordinator* is also responsible for inter-device communication and the dynamic adaptation and recovery of the system, periodically checking every device to see if a remap of the application is needed. It can be replicated onto multiple devices, thus increasing the resilience and fault tolerance of the system. Each device offers its own set of *Services* (concrete implementations of actions Nodes in the interface, *e.g.*, play sound) and has a *Device Manager*, a lightweight web server responsible for managing the *Services*, intra-device communication, and exposing the device details for the Coordinator.

The authors devised a video surveillance scenario to validate the tool and demonstrated that DDFlow could reallocate computation to other devices when a critical node fails or becomes disconnected from the system and can switch between network mechanisms to cope with a varying network.

Pallewatta *et al.* [50] propose a microservice oriented approach to modelling IoT applications. Each application comprises a set of containerized microservices deployed to a multi-level, hierarchical Fog architecture following a decentralized placement algorithm. One of these

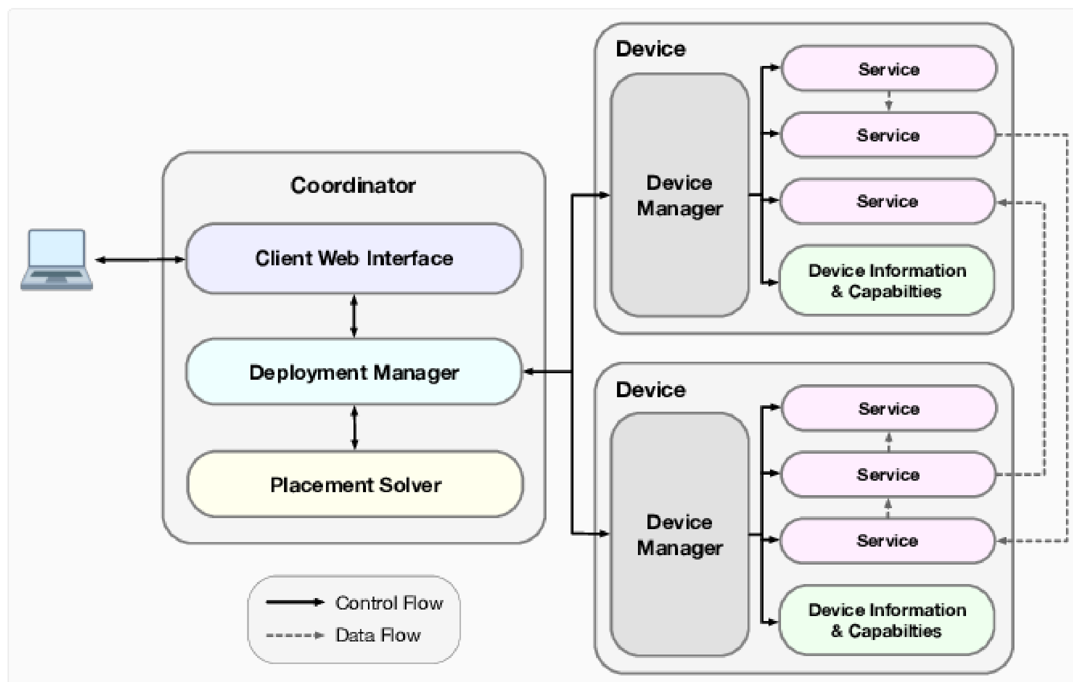

Figure 3.2: DDFlow Architecture, reprinted from [49].

microservices represents the *Client module* deployed onto edge devices (*e.g.*, tablets, smartphones) and allows the user to make requests. Given that there may be data dependencies among the microservices that compose the application, each application is depicted as a Directed Acyclic Graph (DAG). Nodes at the same level on the Fog tree form *clusters* that can communicate following the Constrained Application Protocol (CoAP), a simple and effective protocol for IoT networks [19]. The placement algorithm also handles service discovery and load balancing, based on the number of resources allocated for each node following a Weighted Round Robin method. The algorithm follows a heuristic of minimizing latency and network usage by placing each application's microservices horizontally in the same cluster. When this is not possible, the request is sent to the cluster's parent node (higher level in the hierarchy, closer to the Cloud). The placement of a microservice in a node might not be possible due to constraints in available resources — each microservice has its own set of requirements defined in terms of CPU, bandwidth, RAM and storage usage and each node keeps track of its own available resources.

To implement their solution, the authors utilized the iFogSim Simulator [35]. Since this simulator offers only a centralized module placement and communications can only be made vertically, *i.e.*, *intra-cluster* communication is not supported, the authors extended the simulator to accommodate their solution. They achieve significantly better results in both latency and network usage while providing a smaller delay in microservices placement than a centralized approach. However, Fog nodes' failure is still not addressed, being pointed as future work, and the container images for each microservice are obtained through a centralized container image registry that presents a SPoF.

Seeger *et al.* [58] present a framework for dynamic and automated IoT systems management. Motivated by the growing computational power of sensors and actuators that can render central orchestrators superfluous, they move towards a distributed choreography of devices, achieved by semantically modelling the service composition as *recipes*. Each recipe provides a template for the composition of ingredients (*i.e.*, placeholders for devices and services), their interactions. Each ingredient describes how to access it, following a JSON-like structure where properties such as the inputs and outputs of each device or service are specified. To make each recipe executable a central component must instantiate it, *i.e.*, replace the placeholder ingredients with an actual service or device instance selected from the system. The selection of such instance is based on each candidate's set of properties and, optionally, the user can define a set of rules at the recipe level that dictates non-functional requirements that the selected device must fit (*e.g.*, location or energy consumption levels).

After instantiating the recipes, the central component no longer represents a SPoF. Even if it fails, all the recipes will continue to work, given that upon the connection of a new component, a description of its communication behaviour in that choreography is spread to other participants, meaning that every device knows how to communicate with any device. However, the addition and removal of new components are impacted; there is no other way of disseminating this event without it. The authors also state that the application's current state does not handle nodes' failure, being placed under future work.

Blackstock *et al.* [9] and Giang *et al.* [32, 33] , in subsequent works, present a Distributed Dataflow (DDF) programming model for the development of IoT applications in a Fog environment. In DDF, each program's several flows are deployed onto the multiple physical devices, instead of only in the central orchestrator (as is the case with vanilla Node-RED). Each device may be responsible for multiple flows and, since there might be dependencies between flows, DDF allows for inter-device communication through a central message broker. The authors implemented the DDF framework based on Node-RED to validate this model, given the dataflow similarities. This originated Distributed Node-RED (D-NR), an extension to Node-RED that leverages edge devices' computational power for increased efficiency and timeliness. At the time of writing, the model has gone through 3 iterations.

In the first iteration, a *device id* property was added to a set of Node-RED nodes. This property allows the user to specify, through the interface, the device where that node should execute. Nodes of the same flow running on different devices can communicate using a shared MQTT message broker.

In the second iteration, the authors improved on two major points: (1) broader support of deployment constraints; (2) management of the replicated deployment of nodes. To start, the authors introduced the *constraint* primitive that specifies how each node should be deployed to a device (instead of only specifying a device id). This primitive can specify the type of device on which the node should run (*e.g.*, mobile, server) and the processing capability,

memory size, and location of the device. However, this meant that the same task could be assigned to multiple nodes (*e.g.*, if the constraint is only location, there can be several devices in the same location). This raised the need for the notion of *wire cardinality*, a relationship between nodes that dictate how their devices communicate. In this version, the deployment of a flow to all devices was done through the broker. Whenever a developer deploys a new or updated flow through the interface, all the devices receive a notification and can pull and parse the latest version. Following the set of constraints specified on each node, each device can then decide which nodes should be deployed locally. However, this deployment method proved to not be enough in highly dynamic and large-scale environments; devices can join and leave the network at any time, making the decision to run a node in a particular device not permanent.

This led to the appearance of a stateful coordination layer in the third and final iteration. This layer centrally coordinates the flow of data between nodes. First, between two nodes, a new node, *Coordination Node*, is introduced to intercept the communication. This node communicates with a *Flow Coordinator*, running locally on each device, and, in its turn, communicates with a central *Global Coordinator*. This Global Coordinator, through each Flow Coordinator, updates Coordination Nodes to one of 4 states. As illustrated in Figure 3.3 (p. 18), the 4 states are: (1) NORMAL, the device can run both nodes, and they pass data normally; (2) DROP, the nodes can drop all data they receive; (3) FETCH_FORWARD, the Coordination Node acts as a *wire in* node to retrieve data from an external Node A, typically happening when the device can run Node B but not Node A; (4) RECEIVE_REDIRECT, the Coordination Node acts as a *wire out* node, sending information from Node A to an external Node B, typically happening when the device can run Node A but not Node B. These 3 iterations led to a tool that is well equipped to handle dynamic and heterogeneous Fog environments. However, DDF assumes that all can run Node-RED, which limits the range of available devices. Moreover, the system does not provide any secondary communication mechanism, being totally dependent on the Broker's availability (introducing a SPoF). Finally, the fault-tolerance of the tool was not evaluated.

Simpkin *et al.* [62] make use of Vector Symbolic Architecture (VSA) to semantically represent workflows of services that can be processed without a central point of control. They focus on Node-RED and in parsing and migrating Node-RED workflows, through VSA, to a decentralized execution environment, specifically in the CORE/EMANE[1] network emulator; providing a new mode Node-RED processing while keeping the intuitive graphical interface. VSA use large vectors to represent objects, their feature and their semantic relation with each other in a hyper-dimensional space; similar objects are placed closer in such space. They describe how a Node-RED workflow can be encoded using VSA and provide a Python2 module for such.

[1] Core/EMANE, http://coreemu.github.io/core/emane.html

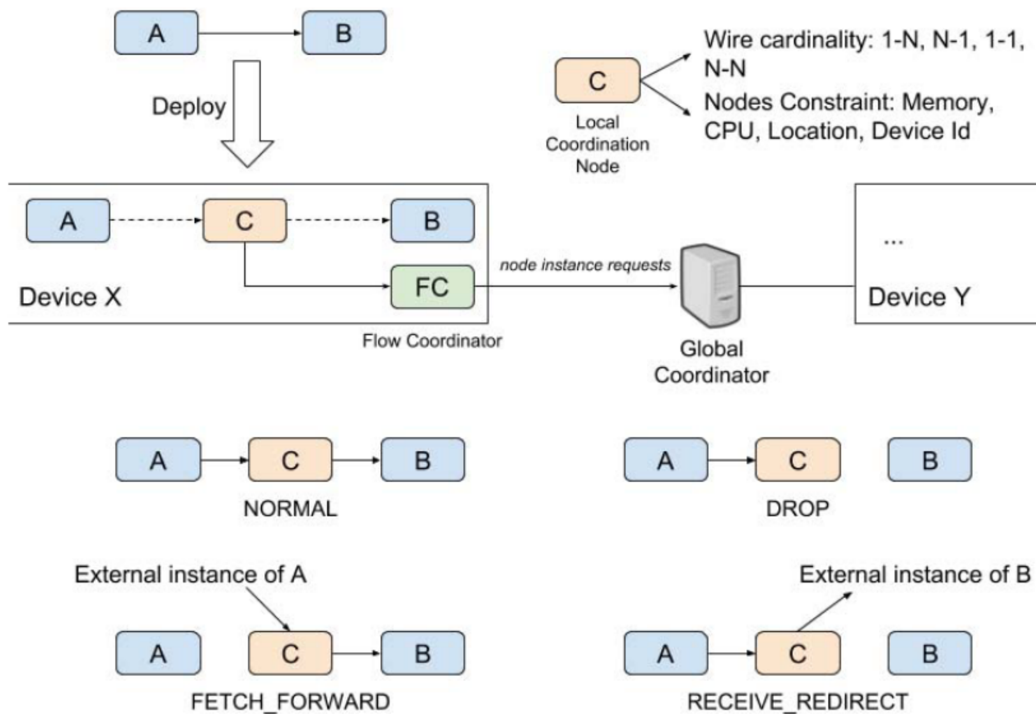

Figure 3.3: DDF Coordination States, reprinted from [33].

Hong *et al.* [36] , building on the growing ubiquity of mobile devices, present Mobile Fog, a programming model for large-scale and dynamic IoT applications. Mobile Fog provides a high-level programming model that aims to simplify the development of highly heterogeneous and distributed network while providing a runtime to scale the application dynamically. It assumes the existence of a Fog computing infrastructure that provides mechanisms to manage on-demand computing instances, similar to those found in typical cloud providers, for the creating and termination of new computing instance in a certain level in the hierarchy. This is needed to accommodate dynamic scaling based on a user-provided policy with CPU utilization and bandwidth metrics. Mobile Fog applications must implement a set of event handlers to integrate the application's lifecycle with the underlying infrastructure. Furthermore, the authors provide an API that allows applications to obtain detailed information about available system resources. The code for each application runs on various devices in the network, meaning that developers must not write different versions of the same program for all devices' varied connectivity aspects. The API also provides communication abstractions (both vertically and horizontally in the hierarchy) between devices. To evaluate the solution, the authors created a realistic traffic simulation with `OMNeT++`[2], obtaining latency values of 20 milliseconds. However, the Mobile Fog runtime's implementation details, that handles dynamic scaling are not discussed, being unclear if it runs in a centralized manner.

[2] `OMNeT++`, https://omnetpp.org/

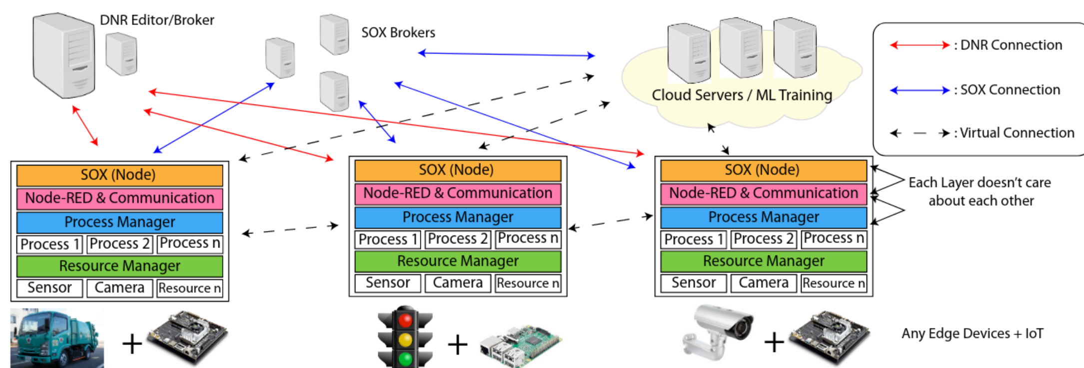

Figure 3.4: CityFlow Architecture, reprinted from [34].

Chopra *et al.* [17] explore the problem of optimally allocating computation and communication in IoT. The problem is formulated as an Integer Linear Programming problem having three optimization metrics: (a) energy-efficiency; (b) available energy maximization; and (c) delay-deadline optimization. Their findings show that if the objective is only to maximize energy efficiency, the clear strategy is to process all data on one node (either the first node on a homogeneous network or the node with the lowest cost in a heterogeneous network). However, if the battery level of devices is introduced into the optimization objective, the workload distribution presents a better alternative; although it increases the overall energy consumption, it allows for a longer network lifetime (given the high disparity in the battery levels across the network). The introduction of deadlines also supports the distribution of the workload for heterogeneous networks across nodes, providing an algorithm to select them. However, this allocation presents the following property (proved by contradiction): in the face of delay and deadline requirements, a two-node allocation will always consume less energy than a three-node allocation.

Giang *et al.* [34] present CityFlow, a platform for developing and deploying large scale smart city applications. Developing Smart cities applications present the same common challenges as seen before (*e.g.*, different computational and communication capabilities, high geographic distribution), but on a much larger scale. CityFlow aims to ease the development by providing an abstraction layer over the complex management of devices in the network, allowing developers to focus on the application logic, specifically integrating Machine Learning (ML) algorithms. To achieve so, it combines D-NR [32], a distributed extension of Node-RED as seen before, and SOXfire [70], a universal sensor data exchange system for Smart Cities, as seen in Figure 3.4. SOXfire utilizes the XMPP[3] protocol, providing the needed communication transparency for distributed processing. It is important to note that this project is still in its early stages and under research.

[3]XMPP, https://xmpp.org/

VIPLE [12] , Visual IoT/Robotics Programming Language Environment, is an open-source VPL for programming IoT devices and robots mainly used in schools and universities for teaching. Similarly to Node-RED, it provides a flow-based interface with a substantial set of general and robots services (including simulations). VIPLE also supports parallel computing, with a service runtime that shares computation load among any number of CPU cores, but it is best suited for event-driven programming. Events can be attached to devices to perform data processing unique to each device [21].

Nguyen *et al.* [48] propose a market-based framework for the allocations of resources in a Fog architecture. One of Fog computing's main challenges is efficiently distributing the workload across the nodes given their limited and highly heterogeneous capabilities. In the proposed framework, each service (described as a market participant) has a certain budget for resources procurement and each resource has a certain price — low demand resources have low prices, and high demand resources have high prices. The goal is to achieve an allocation that maximizes each market participant's interest while maintaining a high resource utilization. They present both a centralized and distributed algorithm for obtaining said allocation while preserving the privacy of services in the latter. To evaluate their framework, they simulated an environment consisting of a maximum of 100 Fog nodes and 40 services, achieving allocations in a reasonable number of steps with the distributed approach (*e.g.*, 130 steps in a system of 100 nodes and 20 services).

3.1.6 Expanded Search

The results from the SLR were presented and analysed in the Section 3.1.5. However, the survey found in those results contains approaches and tools that were not discovered during the process. Two factors explain this divergence: (1) some tools do not have an academic publication associated; (2) the publications contain some variations of the keywords specified in the search query (*cf.* Section 3.1.4, p. 12) that were not considered.

To complete this search process, the solutions from Puliafito *et al.* [54], were filtered accordingly to the inclusion and exclusion criteria defined in 3.1.4 and are presented next:

AWS Greengrass [4] and Microsoft Azure IoT Edge [43] are two very similar products that extend the AWS Cloud and the Azure Cloud, respectively, towards the Edge, allowing for the deployment and management of code directly into IoT edge devices. Each device can locally process data and connect with other devices over local networks. The devices can still operate under a non-reliable connection to the Cloud, keeping a local state and later synchronise.

Cheng *et al.* [14, 13] present FogFlow, an IoT edge computing framework for IoT, to incorporate standards in such the IoT environment. The framework is based on the dataflow standard where tasks have to be defined as dockerized applications, named *operators*, and adhere to the Next Generation Service Interface (NGSI) [8] standard. NGSI defines both the data

model and the communication interface of applications, allowing for a contextual exchange of information over a lightweight message broker. As illustrated in Figure 3.5 (p. 22), the framework is composed of three main components: (1) *Service Management*; (2) *Data processing*; (3) *Context Management*. The *Service Management* is responsible for providing the *Task Designer*, which presents the web-based interface to design and monitor the services, the *Docker Image Repository*, that manages the dockerized operators' images, and the *Topology Manager* (TM), which is responsible for the service orchestration. The *Data Processing* component is responsible for assigning a set of workers to an available cloud or edge node to perform data processing on the tasks allocated by the TM. The communication between TM and each worker is done through RabbitMQ [55]. Finally, the *Context Management* is responsible for: (1) a set of IoT Brokers, providing workers with context updates and subscriptions; (2) a centralized IoT Discovery that handles the registration of new entities and their discovery; (3) a Federal Broker responsible for sharing information with other Federal Broker in other domains.

In Fogflow, an IoT service is represented by a *service topology*, comprised of multiple operators. This topology is defined as a DAG in a JSON format to identify the operators' data dependencies. Additionally, for each operator, a set of *hints* (*e.g.*, number of tasks assigned to a specific input stream) must also be specified to guide the service orchestration. Upon submitting the *service topology*, the TM calculates the assignment plan and sends each task to its work, monitoring each's status. However, due to bandwidth limitation, some edge devices might not support NGSI and, to communicate with the IoT Broker, it is necessary to add a virtual proxy between them to convert the messages into the right format. Currently, FogFlow appears to lack fault-tolerance mechanisms, being pointed a future work by the authors.

Following the analysis of the solutions in the discovered survey [54], another work as suggested by experts in the studied field and is analysed next.

Silva *et al.* [60] present a decentralized computation environment for Node-RED. The authors provide an extension to Node-RED that supports automatic decomposition and partitioning of tasks and custom firmware for the devices responsible for exposing each device's capabilities and running custom MicroPython scripts on demand.

The main changes to Node-RED were: (1) replacing the built event-driven communication with an MQTT-based one to allow external communication; (2) introduction of the *Registry node* that maintains a list of available devices and their capabilities; (3) introduction of the *Orchestrator node* responsible for partitioning and assignment of tasks; and, finally, (4) support to translate Node-RED nodes to MicroPython code (by adding specif code generation methods to each node).

The custom firmware, running of each device, is composed of two main modules: (1) *Announcer* responsible for announcing the capabilities of the device to the Registry node;

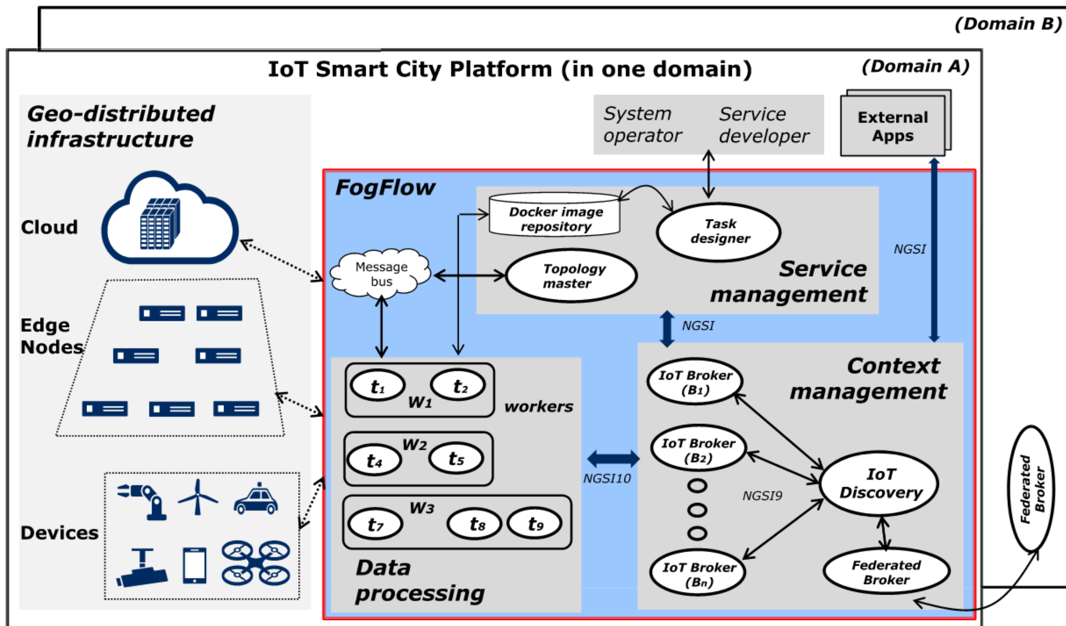

Figure 3.5: FogFlow Architecture, reprinted from [14].

(2) *HTTP Server* responsible for the endpoint that allow the monitoring of the device status and the reception of the assigned MicroPython script. An overview of this architecture is present on Figure 3.6 (p. 23). Additionally, a new property was added to each node that allows it to specify *Predicates* (constraints that cannot be violated) and *Priorities*(requests that can be violated when necessary). The Orchestrator then follows a greedy algorithm for the assignment of nodes to devices, based on the devices' capabilities and the priorities and predicates of each node. This heuristic first prioritizes the number of priorities that can be respected, followed by the equal distribution of the number of nodes associated with each device. After the orchestrator assings nodes to devices, it collects the code generated by each node into a script to be sent to each the device, meaning that a single script can contain the code of several nodes.

The Orchestrator is capable of handling both the addition of device and the failure of existing ones. When a device becomes available, it announces itself through a specific MQTT topic, being received by the Registry node. In its turn, the Registry node communicates this new information with the Orchestrator, triggering a new orchestration. Upon the failure of a device due to memory problems, a *FAIL-SAFE* mechanism is triggered, that restarts all the components, leading to a new announcement, as described before. However, this time, a flag is added that indicates that the device has failed, informing the Orchestrator that it should assign fewer tasks to that device.

Finally, the authors present the known limitations of the solution: (1) script generation forces all communications to be through MQTT, even in sequential nodes assigned to the same device; (2) the system is halted between orchestrations; (3) the detection of device failure is

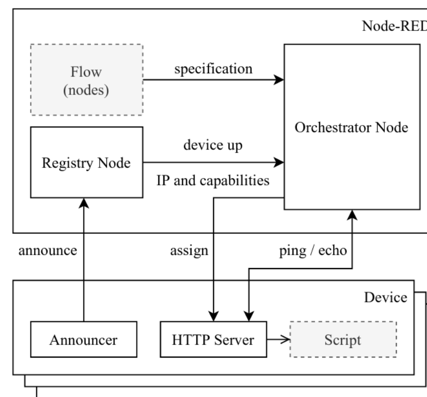

Figure 3.6: Silva *et al*. high-level architecture, reprinted from [60].

too sensible, possibly interpreting temporary unavailability with total unavailability, leading to a costly re-orchestration; (4) the current node assignment algorithm can lead to impossible assignments.

3.1.7 Results Categorization

After the initial analysis, the results were divided into the following categories according to their characteristics and features:

1. **Scope:** Some solutions were created for a specif context, while others present a more general approach. Thus, it is important to understand the current scope of the solutions. Example values: *General*, *Smart Cities*, *Task Allocation*.

2. **Approach:** The solutions present different methodologies for building the application. While some provide a graphical interface, others present a framework or just the overall architecture. Possible values are VPL, SDK, Framework or Methodology.

3. **Communication:** Communication can either be P2P or centralized. Centralized communications are typically made through a message broker.

4. **Task Allocation:** Task allocation can either be made by a central component or through a decentralized algorithm.

5. **Dynamic Adaption:** Can the system handle device or network failure and reorganize itself after the initial task allocation is done? This can be done in a centralized or decentralized manner.

The characterization of the results is present in Table 3.2 (p. 24).

Table 3.2: SLR Results Categorization.

Solution	Scope	Approach	Communication	Task Allocation	Dynamic Re-orchestration
Reiter *et al.* [57]	Task offloading	Framework	P2P	Decentralized	
Vidyasankar [67]	Task Allocation	Framework	N/A	Centralized	N/A
DDFlow [49]	General	VPL	Centralized	Centralized[1]	Centralized[1]
Pallewatta *et al.* [50]	General	Framework	P2P	Decentralized[2]	-
Seeger *et al.* [58]	General	Framework	P2P	Centralized	-
D-NR [9, 32, 33]	General	VPL	Centralized	Centralized	-
Simpkin *et al.* [62]	General	VPL	P2P	Decentralized	
Mobile Fog [36]	General	Programming Model	P2P	Centralized	
Chopra *et al.* [17]	Task Allocation	ILP	N/A	N/A	N/A
City Flow [34]	Smart Cities	VPL	Centralized	Centralized	-
VIPLE [12]	Education	VPL	Centralized	Centralized	-
Nguyen *et al.* [48]	Task allocation	Framework	N/A	Both	N/A
AWS Greengrass [4]	General	Framework	P2P	Centralized	-
Microsoft Azure IoT Edge [43]	General	Framework	P2P	Centralized	-
FogFlow [13, 14]	General	VPL	Centralized	Centralized	
Da Silva *et al.* [60]	General	VPL	Centralized	Centralized	Centralized

Hyphens (-) mean *no information available*, and empty means *no*.
[1] This responsibility is centralized on the Coordinator, which can be replicated introducing redundancy (but not decentralization).
[2] The tasks must be dockerized and are available through a centralized registry.

3.1.8 Evolutionary Analysis

The publication's years of the results from both the SLR and the survery [54] were grouped and analysed. Figure 3.7 depicts the evolution, where we can clearly observe that the vast majority of work done in decentralized IoT orchestration is still very recent, from 2017 onwards.

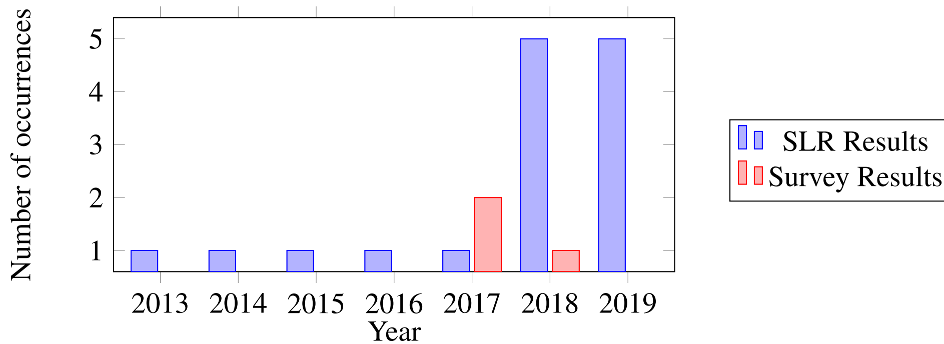

Figure 3.7: Years of the Publications.

3.1.9 Results Analysis

According to the categories defined in Section 3.1.7 (p. 23) and the results in Table 3.2, the following conclusions were taken:

1. **Scope**: Of the 16 solutions, 10 have a general scope, capable of being applied to multiple scenarios. From the remaining, 3 were related to studying the task allocation problem, 1 to offloading computation tasks from mobile devices, 1 education and the remaining one to Smart Cities.

2. **Approach**: The most common approaches are Frameworks and VPL, with 7 solutions each. The remaining are a programming model and an ILP problem. It is important to note that, from the 7 VPL solutions, 4 of them are based on Node-RED.

3. **Communication**: There are 6 solutions which depend on a central communication mechanism, 7 that employ P2P communication and 3 that do not take that into consideration.

4. **Task Allocation**: The majority of solutions depend on a central orchestrator to perform the task allocation. Only 3 tools provide a decentralized mechanism, however with some limitations. Those are: Reiter *et al*. [57] only deal with task offloading, requiring developers to specify applications parts that can be offloaded through code annotations; Pallewatta *et al*. [50] depends on a centralized image registry to obtain each microservice to deploy.

5. **Dynamic Re-orchestration**: The vast majority of solutions do not provide any type of Adaptive Orchestration, with 7 solutions that don't handle device failure, 4 solutions that do not mention it. Two other solutions [49, 60] present a centralized orchestration mechanism that can handle device failures and [49] handle network failure. The remaining 3 solutions don't apply to this category. It is important to note that the work present [49] is currently closed-source.

3.1.10 Survey Research Questions

The SRQ presented in Section 3.1.2 (p. 11) were defined to guide and limit the research process to what was relevant. In this section, those same questions are answered.

SRQ1: *What solutions exist to orchestrate decentralized IoT systems?*

Of the 16 solutions found, 12 offer a solution to orchestrate IoT systems. Most follow a VPL approach, and the remaining offer a framework to build upon.

SRQ2: *Which approaches found in SRQ1 move towards a decentralized orchestration?*

While 3 solutions offer a decentralized task allocation mechanism, none of them offers a truly decentralized orchestration. The orchestration responsibility is typically assigned to a central component. It is also important to note that all but 2 solutions either don't offer any mechanism to handle failures or don't provide sufficient analysis of such mechanisms after the initial task allocation.

SRQ3: *Which approaches found in SRQ1 make use of visual programming?*

Of the 16 solutions, 7 made use of visual programming to expose the system. Four of them, built upon, providing extensions or frameworks than operate on top of it, portraying the popularity of Node-RED in the IoT domain.

SRQ4: *What is the evolution of orchestration solutions to IoT systems over the years?*

As illustrated by Figure 3.7 (p. 24), the number of publications in this context increased significantly in 2018 and 2019, with the oldest publication being made in 2013. It becomes clear that this topic has gained general interest only recently.

3.2 Extended Review

During the Systematic Literature Review presented in Section 3.1 (p. 11), the different methods of task allocation of each solution were analyzed and categorized. It became clear that the majority presented a centralized task allocation algorithm, done by a central node that communicated the tasks to each device. In this section, the current state of decentralized task allocation is further explored, focusing on consensus-based approaches.

3.2.1 Methodology

To collect relevant work, an iterative literature review process was followed. Starting from a set of relevant keywords, the same databases used in Section 3.1.3 (p. 12) were queried and the results were filtered accordingly to their relevance to the subject, by an analysis of the title and abstract, and added to a list. This list was further expanded by relevant references found in the original results (*i.e.*, *snowballing*). The search query was the following:

```
("task allocation" OR consensus) AND (decentralized OR distributed) AND ( iot  OR
    "Internet of Things")
```

3.2.2 Results

From the obtained results, the following were considered to be the most relevant.

Colistra *et al.* [20] present a consensus-based distributed resource allocation protocol for IoT networks. The reference scenario for this allocation is a network comprised of *Task Groups* and *Virtual Objects*. *Task Groups* are groups of devices that perform similar and replaceable tasks (*e.g.*, measuring temperature in a room). These groups are assigned specific application tasks by the application deployment server (*e.g.*, the central node in Node-RED). The deployment server can either specify the exact device for the task or leave this decision for the task group to autonomously decide how to distribute the workload amongst themselves, removing the need for a central registry. The authors build on the latter, introducing *Virtual Objects* (VO) present on each *Task Group*. Each VO is implemented by a node in each group and is responsible for receiving a task request from the central server and forwarding to its

peers. After this moment, the responsibility of allocating resources for the task lies on the group, offloading work from the central server. This scenario is illustrated on Figure 3.8.

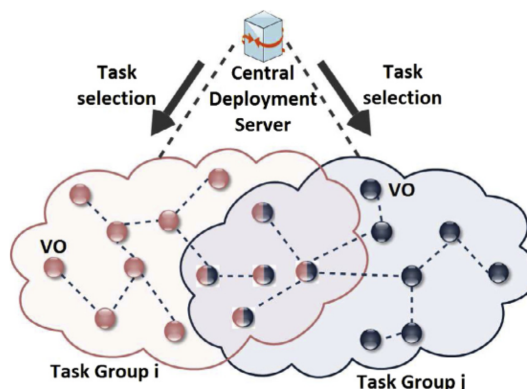

Figure 3.8: Colistral *et al*. Reference Scenario, reprinted from [20].

The communication between the different devices in each group is made through middleware with two layers: (1) the *Semantic Layer* responsible for providing the different descriptions needed for the subdivision of application; (2) the *Resource Allocation and Management Layer* that, based on the information provided by the previous layer, identifies tasks groups and correctly allocates resources. The central deployment server creates the tasks groups and assigns the devices. This middleware is implemented in all the network devices, which, upon joining a new network, broadcast a SPARQL query to discover the VO responsible for the tasks it can perform in order to join a task group — or create, in the case of no VO, notifying the central server.

Inside each group, the task allocation is an application of the consensus protocol to achieve agreement on a common execution frequency for a specific task. In a highly heterogeneous network where devices join and leave the network frequently, the authors' idea is that balancing the task frequency across the different devices of each group, the accuracy of the results will not be compromised and the network resources will be better distributed, improving the lifetime of the network.

The authors tested two types of communication in the network, broadcast and gossiping, obtaining a better convergence on the former, obtaining an error lower than 5% relative to a centralized solution after the transmission of 20 packets per device. Finally, in overlapping tasks groups (*i.e.*, groups that share a common subset of devices), there is the need for inter-task coordination to maintaining homogeneous allocation.

Mudassar *et al.* [46] present a decentralized algorithm for the grouping of edge nodes to execute a certain task in a distributed manner and a mathematical model to achieve the needed resource allocation. Some of the devices present in the edge network are previously defined as *organizer nodes*, responsible for receiving a task and form a group for processing. Each task is defined in terms of the associated workload, data size, and the time deadline. Upon

receiving the task, the *organizer* will announce the task in the edge network and build a local group according to the responses. The grouping phase ends when the organizer has secured a fair resource allocation for the task (as defined by the mathematical model). After, this task is subdivided into smaller independent tasks which will be executed in the different devices. Additionally, the authors provide an fault-tolerant mechanism that allows each node to maintain a robust subset of its neighbors that could potentially handle tasks in the case of its failure. This subset is built considering the available resources of each neighbor (*i.e.*, CPU, RAM and Bandwidth). If a neighbour does not answer a periodic request, it is removed from the set.

The authors tested their model in the iFogSim Simulator [35], achieving a reduced network usage, deadline missing ratio and latency.

Choi *et al.* [15] present an auction approach for decentralized task selection coupled with a consensus procedure for decentralized conflict resolution in the context of the autonomous vehicles. They present two versions of the algorithm: (1) Consensus-based Auction Algorithm (CBAA), a single-assignment version; (2) Consensus-based Bundle Algorithm (CBBA).

In CBAA, the algorithm is divided into two phases. The first phase is the auction process where each agent (*i.e.*, node in the network) places a bid on a certain task asynchronously with the remaining agents. Each agent keeps two vectors of length equal to the number of tasks, the first representing which tasks have been allocated to that agent and the second representing the current winning bids for each task in the network. The second phase details the consensus process, where agents share their winning bids vector with their neighbours, updating it to the highest value found. Ties that may occur are assumed to be handled systematically (*e.g.*, lexicographical tie-braking).

In CBBA, the algorithm is also divided into two phases — bundle construction and conflict resolution. Here each agent keeps a list of task potentially assigned to itself and the auction process is still done at the task level. The agents now share three vectors: (1) the winning bids; (2) the winning agents; (3) the timestamps of the last update of each other agents. The timestamps are used to determine the action an agent should take upon receiving messages from other agents.

The authors further extend their work to incorporate cooperation constraints for heterogeneous tasks [16] and coupled constraints in subsequent publications [68]. Di Paola *et al.* [22] also presents an extension to CBBA, named Heterogeneous Robots Consensus-based Allocation, capable of better handling heterogeneous robots by extending the second phase of the base algorithm to allow pruning of tasks from overloaded robots.

3.3 Summary

Section 3.1 (p. 11) details the Systematic Literature Review conducted to gather information about the state of the art of decentralized IoT systems. In total, 1530 publications obtained from three

distinct databases were analysed, and 15 were selected. One of them was a survey that provided an additional 3 publications. Furthermore, another publication was suggested by experts in this work's field, resulting in 19 distinct publications.

These publications show that there are already several solutions for IoT orchestration that adopt different approaches. Additionally, the popularity of Node-RED is also noticeable given that, from the 7 solutions based in VPL, 4 of them use Node-RED to implement their solution. However, currently, all follow a centralized orchestration mechanism. Moreover, most fail to provide dynamic adaption mechanism to device and network failures. The two solutions that provide it [49, 60], do so in a centralized manner. These two points, *i.e.*, decentralized orchestration and decentralized dynamic re-orchestration, represent the research challenges still not currently addressed by the research community.

Section 3.2 (p. 26) presents an analysis of decentralized task allocation. The solutions made use of two different techniques: (1) formation of self-managed clusters in the edge of the network according to their capabilities; (2) usage of market auctions and consensus algorithms to achieve an agreement.

Chapter 4

Problem Statement

4.1 **Current Issues** . . . 31
4.2 **Desiderata** . . . 32
4.3 **Scope** . . . 33
4.4 **Main Hypothesis** . . . 33
4.5 **Research Questions** . . . 34
4.6 **Methodology** . . . 34
4.7 **Summary** . . . 34

This chapter describes the problem under study, as well as its hypothesis, the proposed solution to solve it and the validation process. Section 4.1 presents and contextualizes the problem under study and Section 4.2 defines the *desiderata* we wish to fulfil. Then, Section 4.3 presents the scope and focus of this work, and Section 4.4 defines its hypothesis. Then, Section 4.5 present the research questions that will guide our work. Section 4.6 present the experimental methodology to validate and evaluate our work. Finally, Section 4.7 summarizes this chapter.

4.1 Current Issues

The Internet of Things (IoT) paradigm represents the connectivity of everyday objects to the Internet. This growing network enables physical and virtual objects to share information, allowing for coordinated decisions in a highly heterogeneous environment [5]. In fact, the high heterogeneity and device distribution are the main characteristics of this network, which comprises devices of ranging computational and communicational capabilities.

The majority of IoT systems currently follow a centralized architecture [45] where the main component — the orchestrator — performs all the computation on the data provided by the devices. This type of architecture is often imposed by the tool used to create the system, as is Node-RED. Nevertheless, this architecture introduces several limitations that may hinder the entire system. First, the orchestrator presents, in itself, a single point of failure (SPoF), bringing the entire system to a halt if it fails. Secondly, given that all the computation is being performed in one place, the remaining devices' rising capabilities are being wasted. Thirdly, information is transferred across different boundaries to the Cloud, possibly violating legal constraints even when it is not necessary [59].

Recently, with the emergence of Fog and Edge Computing paradigms (*cf.* Section 2.1, p. 5), there has been a migration of data computing and storage towards the network edge and closer

to users. In these paradigms, the computational workload can be better distributed across the network, offloading some work from the orchestrator.

However, even in Fog or Edge Computing, new problems arise. With the distribution of the workload across the network, a need arises for the decomposition of IoT applications into smaller tasks and their allocation to available devices (*i.e.*, the orchestration of the application). It becomes critical to provide fault-tolerant mechanisms, given that each device can be running a fundamental part of the application. However, failure detection and recovery also become more complex in a distributed scenario than in a centralized one — managing distributed application logic is more challenging than handling input or output sources.

The systematic literature review made in Chapter 3 (p. 11) shows that there has been a rising interest in the research community in incorporating Fog and Edge computing into IoT solutions, either by extending previous tools or by creating new frameworks. Additionally, several publications focused entirely on studying and analyzing task allocation algorithms, a central problem in orchestrating devices in such a heterogeneous environment [67, 17, 48]. Finally, the review revealed the evident centralization in the orchestration of the different solutions (*cf.* Section 3.1.9, p. 24), further highlighting the scarcity of either centralized or decentralized resilient mechanisms. Once again, the responsibility of handling the inherent problems of Fog and Edge computing has been assigned to the orchestrator, rendering him a SPoF, even in a scenario where each device is actively contributing. In summary, none of the solutions succeeds at presenting a system capable of allocating a set of tasks among its parts without fully depending on a central entity, *i.e.*, a decentralized orchestration.

We can then conclude that there are clear gaps in providing a resilient, decentralized orchestration in IoT applications, namely (1) a failure detection and recovery mechanism, and (2) an allocation strategy — that involves both computing the allocations and sending the tasks to each device — that does not depend on a central entity.

4.2 Desiderata

A system capable of rectifying the issues mentioned above would be one that necessarily presented the following characteristics:

D1: Automatically decompose the system into isolated computational units, so that they can be assigned to each device independently.

D2: Remove the single point of failure of the central orchestrator, so that the system can adapt, in run-time, to changes in the availability of devices even when the orchestrator has failed.

D3: Remove the single point of failure of the communication channel, so that the devices can still communicate even when faced with failures from the communication medium.

The main focus of this work will be on providing a system capable of fulfilling *desiderata D1* and *D2*. Any advancement made towards *D3* is secondary.

4.3 Scope

The scope of this work is to develop a platform that offers a decentralized orchestration strategy with minimal dependency on a central orchestrator in the context of IoT applications. This platform will be based on an existing solution that provides a centralized orchestration and decomposition of distributed IoT applications. Thus, the focus of our work will be on modifying and introducing mechanisms that will allow the existing orchestration to operate with a lower dependency on the availability of the central orchestrator. Our target audience is IoT developers who wish to leverage edge devices' computational capabilities while still using a visual programming environment.

Although providing different communication mediums between the edge devices would result in a more resilient system, any development towards this goal is secondary. Additionally, notwithstanding the importance of security in decentralized environments, it is also not the focus of this work.

4.4 Main Hypothesis

Assuming a distributed IoT system, composed of various tasks and multiple heterogeneous devices, that is fully dependant on a central entity for the orchestration, the fundamental research question this work intends to answer is the following:

> *"What type of alternative orchestration mechanism can we provide that enables the system to continue operating after the central entity has failed?"*

Following the previous question and the requirements set by the *desiderata*, we claim the following hypothesis:

> *"Given an IoT system with multiple connected heterogeneous devices, a weak decentralized orchestration mechanism offers better results than a strong centralized orchestration when faced with failures from the orchestrator."*

For all purposes, we define henceforth a strong orchestration as a mechanism capable of: (1) handling the failure of devices; (2) handling the appearance of new devices; (3) managing the addition or removal of tasks; and (4) computing new allocations to respond to the previous events, and a weak orchestration as any orchestration that does not comply with at least one of the capabilities of a strong orchestration.

We evaluate the results in terms of the availability and proper functioning of the system resulting from the orchestrations. Thus, an orchestration presents *better results* when it either increases the availability when another hinders it or restores the proper functioning of the system when another fails to provide it.

4.5 Research Questions

To validate our hypothesis and guide our work towards fulfilling our *desiderata*, we have defined the following four research questions:

RQ1: Can we provide a way for the devices to obtain their assigned tasks without a central orchestrator?
To maintain the correct behaviour of the system after the orchestrator has failed, each device should be able to obtain its tasks independently of the status of the orchestrator.

RQ2: How can we orchestrate the system without a central orchestrator and with a weak orchestration that cannot compute new allocations?
If we could pre-compute all possible allocations and share them with each device, the system could re-orchestrate itself without a central orchestrator. Also, to maintain the correct behaviour of the system, the pre-computed allocations should contemplate an appropriate response for any change in the availability of the devices or tasks (*i.e.*, addition and removal).

RQ3: Can we provide a failure detection mechanism without depending on a central orchestrator?
To maintain a consistent state across all devices according to the pre-calculated allocations, the devices must detect the failure of their peers without requiring a central entity.

4.6 Methodology

The main hypothesis of our work is centred around providing a decentralized orchestration mechanism that can provide better results than than than the existing centralized ones when faced with failure from the orchestrator. We will validate our work through a reference implementation based on the work of Silva *et al.* [60]. We will add new capabilities to the existing orchestration mechanism, allowing it to operate without as much on the central orchestrator.

To assess if our system fulfils the *desiderata* presented in Section 4.2 (p. 32) and answer to the Research Questions raised in Section 4.5, we will test it in multiple experiments using both physical and virtual devices and different scenarios where we inject failures in the orchestrator [23, 26]. Each scenario was purposely created to highlight the different features of our solution, providing the basis for the validation of the *desiderata*.

4.7 Summary

The current issues of centralized orchestration in IoT systems are presented in Section 4.1 (p. 31) along with the identification of a knowledge gap in providing decentralized solutions found in the Systematic Literature Review made in Chapter 3 (p. 11). Based on this gap, in Section 4.2 (p. 32) we present a set of requirements that our solution intends to fulfil in order to provide a decentralized orchestration mechanism.

In Section 4.3 (p. 33) we defined the scope and focus of our work, which consists of the development of a decentralized orchestration mechanism on top of an existing distributed IoT computing platform. The hypothesis of this work is presented in Section 4.4 (p. 33), claiming that a weak decentralized orchestration can offer better results when compared to a centralized orchestration when the orchestrator fails. Then, in Section 4.5 (p. 34) we define the Research Questions formulated to guide the development of our work.

In Section 4.6 (p. 34) we present our experimental methodology to validate our hypothesis and assess to what extent the *desiderata* has been fulfilled and the Research Questions answered. Our solution will be tested in multiple scenarios, using physical and virtual devices, to gather relevant information to validate our work and compare its resiliency with the underlying platform.

Chapter 5

Implementation

5.1 Overview 37
5.2 Decentralized Orchestration 40
5.3 Failure Detection 41
5.4 Node Operating Modes 44
5.5 Known Limitations 45
5.6 Summary 46

This chapter introduces and discusses the implementation of our solution along with the justification and reasoning for each choice made. Section 5.1 presents an overview of the whole solution. Section 5.2 further explains our mechanism for a decentralized orchestration. Then, Section 5.3 presents, in detail, the failure detection mechanism developed. Section 5.4 introduces the different operating modes of Node-RED nodes on our solution. Section 5.5 presents the known limitations of our solution. Finally, Section 5.6 summarizes this chapter.

5.1 Overview

In this work, we extend the platform developed by Silva *et al.* [60] for a decentralized computation environment in Node-RED, providing a more decentralized orchestration strategy by reducing the dependency on the central orchestrator — the Node-RED instance itself.

The existing platform comprises two main modules (1) the Node-RED instance and (2) the Micropython based firmware developed for the edge devices, capable of running Python scripts generated by Node-RED. In our version, both of these modules were updated, with significant changes made in the communication layer, and a third module — the `Task Repository` — was added. However, excluding the orchestrator node, the behaviour of the remaining nodes created for the platform was mostly kept (*e.g.*, the registry node suffered no alteration) as well as the announcing mechanism and the heuristics for the assigning algorithm, *i.e.*, predicates and priorities (*cf.* Section 3.1.6, p. 20).

We present a weak decentralized orchestration that violates the second, third and fourth capabilities of a strong orchestration, as defined in Section 4.4 (p. 33). More specifically, a decentralized orchestration that cannot compute new allocations and is dependant on the pre-computation of the allocations by the central orchestrator, capable of adapting to device failures but unable to react to new appearances of both devices and tasks. We will revisit this topic in the following sections to better understand our implementation choices and their impact on the violated capabilities.

5.1.1 Initial Allocation Behaviour

Upon assigning the flow nodes to the devices, the orchestrator node is responsible for sending to each node the identifier of its assigned device (*e.g.*, device IP) and sending each device the allocation list (explained in detail in Section 5.2 (p. 40). In their turn, each of the flow nodes generates its own Micropython code and sends it to the `Task Repository` for storage. After receiving the allocation list, each device is responsible for obtaining its tasks from the Repository and installing them. Finally, after every device has installed all of its assigned tasks, the flow is operating normally. The sequence of operations can be seen in Figure 5.1.

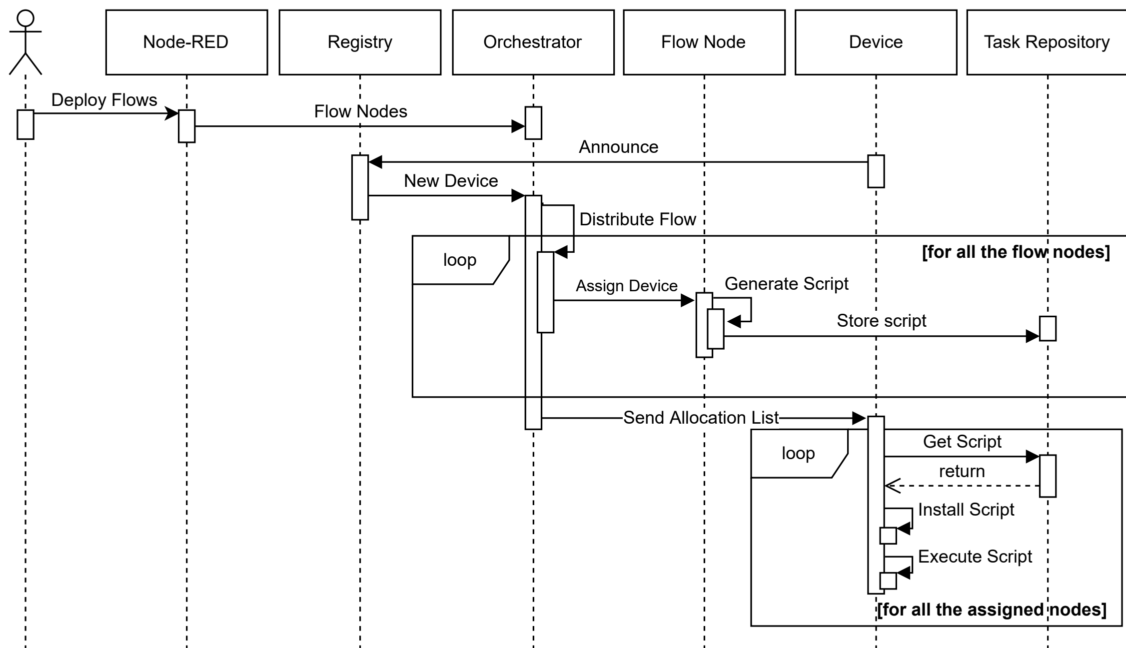

Figure 5.1: Sequence of operations in deploying a flow.

In the existing platform, the way how nodes communicated was altered at the library core level, replacing the Node-RED event-based mechanism with an MQTT based one. This modification meant that the newly created nodes capable of running on edge devices would only work on this specific version, forcing users to install it should they want to run this decentralized environment. Not only does this severely hinders the adoption of this platform amongst Node-RED users, as it goes against the common practice in the Node-RED community of creating and sharing new nodes that can be easily installed as an external package (*cf.* Section 2.3, p. 8). Our solution isolates and extracts those modifications to a separate class that the nodes can utilize to run in the decentralized environment, providing an isolated package that can be installed into the Node-RED *palette* without interfering with the original communication mechanism. This meant that a node could be interacting with other nodes running either locally, on Node-RED through events, or externally through MQTT, in a device, having to select between these distinct communication mediums. Thus it becomes essential to define exactly how each node should dispatch each input

it receives in the flow. To tackle this issue, we introduce different *operating modes* that are assigned to each node by the orchestration upon each allocation. This topic is further analyzed in Section 5.4.

5.1.2 Subsequent Reallocations

The allocation list contains entries for each node in the flow and, for each node, a sorted list of the devices that can run that node. It is ordered by the affinity of each device to that node, making use of the assigning algorithm devised by Silva *et al*. [60], where the first device is the most adequate and the last the least.

All devices and Node-RED monitor each other status — either *failed* or *alive* — to detect possible device failures, leading to two different responses depending on the status of the Node-RED instance. If it is still *alive*, the orchestrator will receive a message with the identifier of the device that has *failed*, marking it as such and triggering a re-orchestration of the system, repeating the process as mentioned above. If Node-RED has *failed*, the failure detection mechanism will notify each device of that failure, triggering an update to the device internal state. Each device enters an *independent* state where it will self re-orchestrate accordingly to the allocation list it possesses. In this state, upon the failure of another device, each device iterates over the allocation list and installs every node of the failed device for which it is the next best device. A high-level view of the different modules and their interaction can be seen in Figure 5.2.

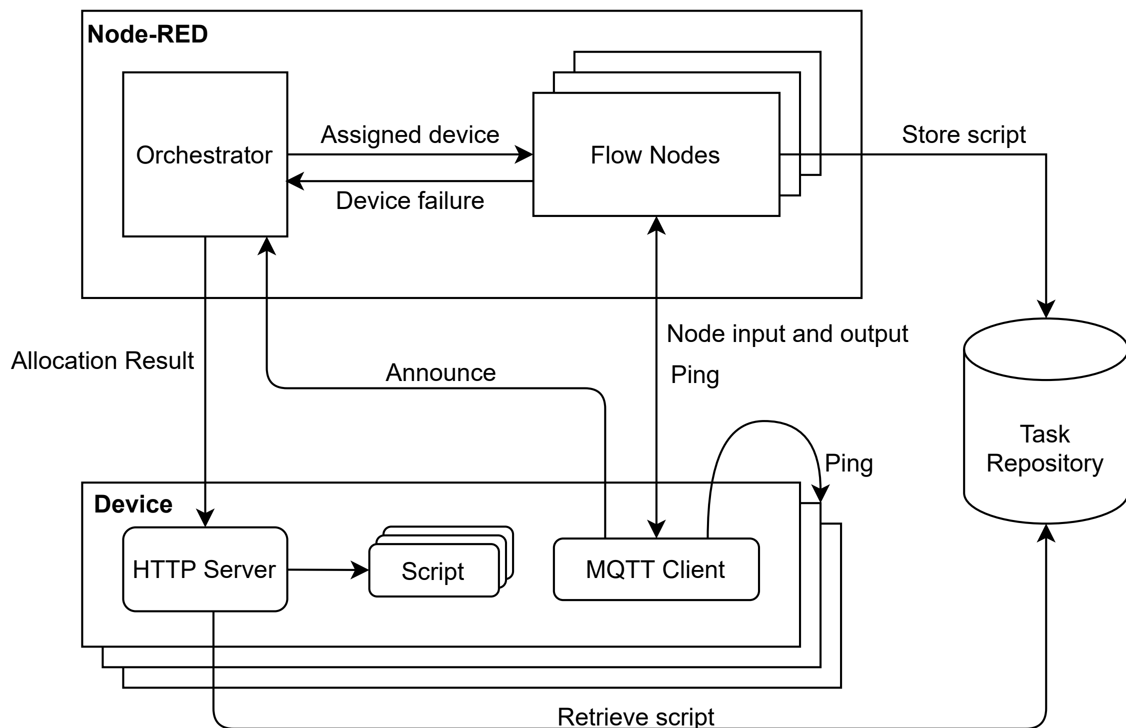

Figure 5.2: High level overview of the solution modules.

5.2 Decentralized Orchestration

In order to achieve a decentralized orchestration, the devices in the system must be able to define and agree on a re-distribution of the tasks allocated to a device that has failed without depending on a central entity. For this, the system must provide a way of detecting such failures (*cf.* Section 5.3, p. 41) and a decentralized algorithm to reallocate such tasks correctly.

As seen in Section 3.2, several decentralized task allocation algorithms have been presented, making use of auctions coupled with consensus mechanisms [15, 68, 22] or creating local groups at the edge that collaborate on tasks following a consensus procedure [46, 20]. However, these algorithms require coordinating asynchronous processes, divided in multiple phases, in order to agree upon a certain decision. In a scenario such as IoT systems that are failure-prone and composed of many heterogeneous devices, the implementation of these algorithms becomes increasingly difficult. Thus, to simplify the development, we devised an alternative method, making use of the centralized allocation algorithm made by Silva *et al*. [60].

Originally, for each node, the algorithm finds the best possible device to receive a node by calculating a score for each device based on either it complies with the node predicates and priorities and the number of nodes already associated with that device. Instead of returning only the best device, in our solution, the algorithm produces a score ordered list with all the devices capable of running that node (*i.e.*, complies with the predicates). If the score is the same for any two devices (*e.g.*, they share the same capabilities and the current number of allocated nodes), they are sorted alphabetically according to their identifier. Then, the complete allocation list is built, containing entries for each node with the respective device list, and sent, by the orchestrator, to every device. The devices can then adhere to this allocation list during device failures to decide if they should take over a task (*i.e.*, if they are the next in the list for any node associated with the failed device). An example of an allocation list can be seen in Listing 1 (p. 41).

Therefore, with the introduction of the allocation list, which represents a pre-computation of the allocations, we have opted to violate the fourth capability of a strong orchestration, further defining our weak decentralized orchestration (*cf.* Section 4.4, p. 33).

This approach has clear limitations regarding the efficiency of subsequent allocations due to its static nature. A device with many capabilities that can satisfy most of the predicates and priorities for a set of nodes will be placed high in the list for that set. Upon the failure of other devices responsible for some of those nodes, this device will start to accumulate a significant number of assigned nodes, possibly leading to its failure. A dynamic algorithm, ran by each device, could potentially prevent this situation, given that it would have access to the current number of assigned nodes of each device at any time.

Given that storing all the tasks in every device would be unfeasible, especially considering the constrained aspects of these devices, we created a `Task Repository`. Upon a flow allocation made by the orchestrator, each node is responsible for generating its own Micropython script and sending it to the `Task Repository`. After this, each device can query the Repository for a task and install it. While the Repository introduces a new SPoF, we consider it to be relatively easy

```
{
    "ae3d1231.865b58": {
        "devices": [
            {
                "id": "192.168.1.117"
            },
            {
                "id": "192.168.1.116"
            }
        ]
    },
    "acd4d85c.0c2e18": {
        "devices": [
            {
                "id": "192.168.1.116"
            },
            {
                "id": "192.168.1.111"
            }
        ]
    }
}
```

Listing 1: Example Allocation list containing two nodes and three devices. The key of each initial entry is the node ID, given by Node-RED, and each IP is the identifier of each device.

to replicate this Repository across several machines, introducing redundancy while maintaining a consistent state.

The generation of a single script per node in our solution contrasts with the single scripts per device present in the previous platform. By using a single script per device, each script could contain the code of several nodes and thus reduce the number of transferred files. However, this approach would not be efficient nor feasible in a decentralized environment. If the system had to reallocate all the tasks of a failed device as a single unit to a single device, this would result in a severely unbalanced system and lead to impossible allocations.

5.3 Failure Detection

As previously mentioned, to attain a decentralized orchestration, all of the devices must detect the failure of their peers, preferably promptly, to avoid system downtime. We explored two mechanisms for the detection in our solution: (1) a standard heartbeat mechanism based on pings sent through MQTT; and (2) a failure notification system based on the Last Will and Testament (LWT) feature of MQTT brokers.

The heartbeat mechanism relies on sending periodic ping messages on a shared topic, `ping/+` where the last topic level is either the device identifier or the string *"node-red"* for the Node-RED

instance, subscribed by all devices and nodes. Then, for each distinct identifier received in this topic, each device periodically checks if the timestamp of the last received message is below a certain threshold, triggering a failure response if such happens.

The Last Will and Testament is a feature of MQTT brokers that provides a way of informing clients of any *ungraceful* disconnect from another client by sending a predefined *Will* message. As specified in the MQTT Version 5.0 Documentation[1], the Will message is published in the following scenarios:

- An I/O error or network failure detected by the Server;
- The Client fails to communicate within the Keep-Alive time;
- The Client closes the Network Connection without first sending a DISCONNECT packet with a Reason Code 0x00 (Normal disconnection);
- The Server closes the Network Connection without first receiving a DISCONNECT packet with a Reason Code 0x00 (Normal disconnection).

With this mechanism, each device, similarly to the heartbeat mechanism, is subscribed to the predefined topic of the LWT messages, in our case `alive/+`, and triggers the failure response accordingly.

Comparing both mechanisms, LWT provides a more reactive system. Each device receives a message whenever a device fails, discarding the need for any polling on the status of devices. Furthermore, it dramatically reduces both the number of published messages at any moment since there is no need to send ping messages (that typically are published frequently) and the run-time load of devices by eliminating subscription callbacks and time interval calculations. Thus, for our solution, we chose the LWT mechanism. However, given that the heartbeat mechanism is not platform-dependent, as is LWT, and can be applied over different communication mediums, increasing the system's resiliency, we chose to keep it in our solution for future improvements.

The failure response of the devices can vary according to the status of the instance of Node-RED. While the instance is still operating, the devices do not perform any re-orchestration following the pre-calculated allocations. This is the main responsibility of the orchestrator node, and since it is operating, it will trigger a new re-orchestration. The devices only rely on the allocation list sent to them when the Node-RED instance has failed. To better isolate the responsibilities of the orchestrator, it does not subscribe to the LWT messages of devices. This is done by the nodes that are assigned to devices, which, upon receiving the LWT message, forwarding it to the orchestrator. The sequence diagram for the failure response can be seen in Figure 5.3 (p. 43).

As seen in Section 6.3.1.4 (p. 59), there is a significant difference in how the MQTT connection is dealt with when triggering a failure in virtual and physical devices. To the best of our knowledge, the process of killing the container of a virtual device most likely also explicitly terminates the MQTT connection without sending the DISCONNECT packet, leading to an immediate detection

[1] MQTT v5.0 Documentation, https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

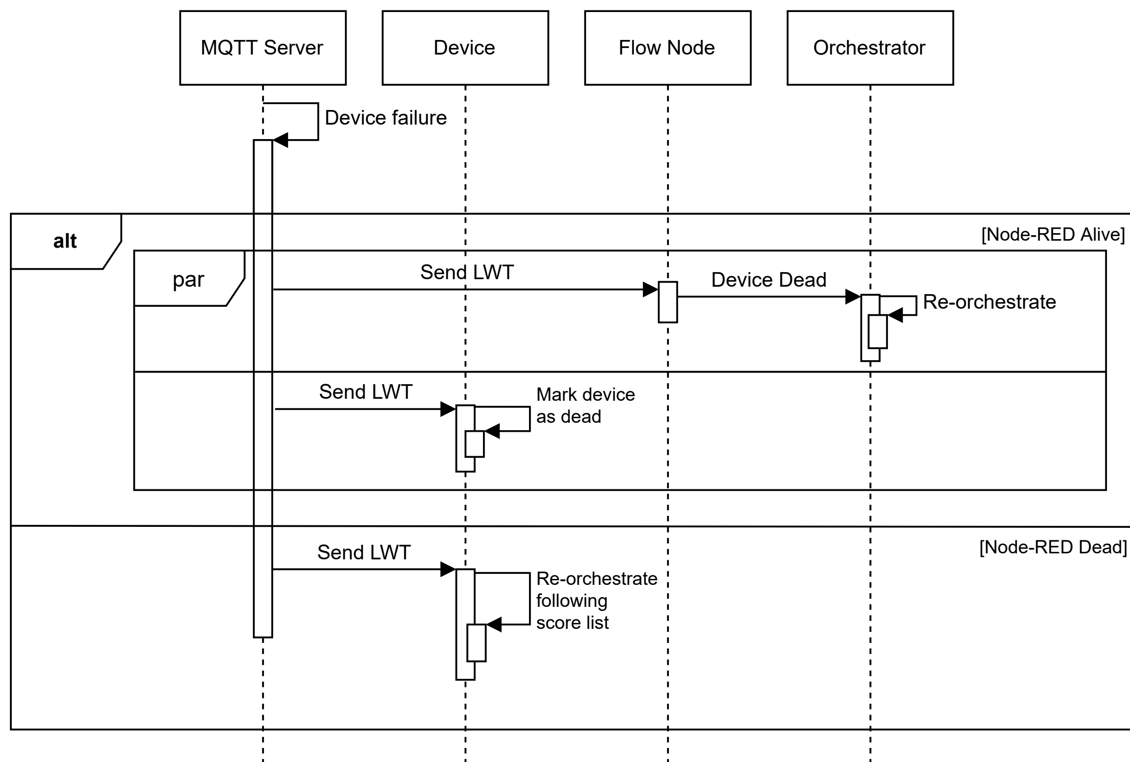

Figure 5.3: Failure Response Sequence Diagram.

by the MQTT server of this failure and, therefore, the publication of the LWT message. However, with physical devices, if we unplug the power source, no connection is explicitly closed, and the MQTT server only takes note of the failure after waiting for one and a half times the Keep-Alive time (as defined in the documentation[2]). Additionally, we also discovered that if instead of unplugging the power source, we connect to the REPL of the device, provided by the Micropython firmware, and send a SIGINT signal, the behaviour is identical to the virtual devices, triggering an immediate failure response. Nevertheless, it is our opinion that the method of unplugging the power source better mimics the majority of failures observed in real scenarios, and we have decided to maintain it during our experiments with physical devices. For the sake of having shorter experiments and a more reactive system, we reduced to Keep Alive time to 5s, from the default 60s, resulting in a 7.5s delay between the failure of a device and the publishing of its LWT by the Server.

Moreover, the platform created by Silva *et al.* also provides a `failsafe` mechanism that was added to the Micropython firmware. This mechanism can deal with memory errors on the device, stopping all the current tasks allocated to that device and clearing the allocated memory. Then, it announces itself once again to the orchestrator, informing it of this failure. In its turn, the orchestrator performs a new orchestration but allocates fewer nodes to that device to prevent future memory errors.

[2]MQTT v5.0 Documentation, https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

5.4 Node Operating Modes

In the existing platform, the modifications made to the communication layer created a distinct version of Node-RED. The event-based mechanism of Node-RED, using the `EventEmitter` class of node.js, was replaced by MQTT based communication. Instead of triggering `emit` events, a node outputs its data by publishing to a unique and addressable topic — generated at the flow creation for each node that represents the node output topic — that is subscribed by the following nodes, independently of being executed in Node-RED or in a device. Since these modifications were made at the base class level, all existing nodes became compatible with this environment. This severely hinders the platform's adoption, given that it forces users to install a different version of Node-RED. To tackle this issue, we isolated and extracted these modifications to an external class, the `Device Handler`, creating a package that can be installed separately into the Node-RED palette.

Extracting these changes to an external class meant that different nodes in the same flow could be communicating through two mediums. Non-modified nodes from the original Node-RED palette would still be sending and receiving messages through events, and the modified nodes would be publishing and subscribing to MQTT topics. Thus, to maintain compatibility with vanilla nodes, modified nodes should adapt their input and output depending on the types of nodes connected to them. For this, we introduce three different *node operating modes* — *local*,*proxy*, and *remote* — that dictate how a node should communicate its output. First, the *local mode* indicates that no device was allocated to the node, and it must operate locally on Node-RED, communicating through events. Secondly, the *proxy mode* means that the node is to be deployed to a device, but, given that one of its outputs is running locally on Node-RED, the node must send its output through an event. For this, an intermediate topic, the *proxy topic*, was created to forward the output from the device to the node. Thirdly, the *remote mode* indicates that the node and all of its outputs and running on devices, not requiring any data transfer to Node-RED. In this case, only the devices' *input topics* are being used, and no information is transmitted to Node-RED. In these last two modes, *proxy* and *remote*, if the node has any input that is being produced by a *local* node, it must forward it to the device by publishing it to the device *input topic*. The orchestrator sets the nodes' modes at the end of each orchestration.

With the introduction of the operating modes, the orchestrator can assign the *local mode* to a node that can not be executed locally. Let's consider a temperature node whose responsibility is to measure and send the temperature of a specific place. Suppose every device with a temperature actuator fails or becomes unreachable. In that case, the orchestrator has no choice but to assign it a *local* mode, even when Node-RED can't completely assume this task. However, this actuator could be replaced by, for example, a weather API call. Although not as accurate and precise as the actuator, the API information — that the Node-RED instance could obtain — could be sufficient to keep the flow alive. This led us to introduce a node *fallback mechanism*.

The fallback mechanism is triggered by a second output that was added to every modified node. This output is meant to be connected to another node — the *fallback node* — that can replace, or at

least compensate for, the initial node in case of an impossible allocation. Upon receiving the *local mode*, a node can send a message through its second output — through the `EventEmitter` — informing the fallback node to start its process. An example of this setup can be seen in Figure 5.4.

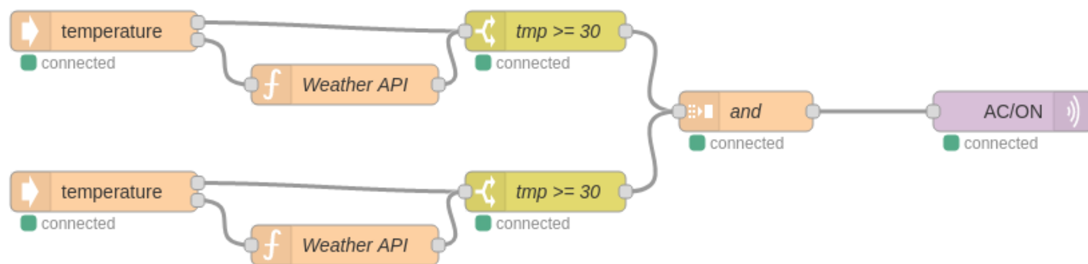

Figure 5.4: Node-RED fallback mechanism on two temperature nodes.

5.5 Known Limitations

While our solution successfully reduces the necessity of the central orchestrator there are still some limitations, the majority being a direct consequence of the weak type of our decentralized orchestration.

First, the decentralized orchestration mechanism can only deal with device failures, ignoring new appearances of new or previously failed devices. If a new device appears after the orchestrator has failed, the devices have no mechanism to calculate and assign a score for the existing nodes to that device, meaning that it will never participate in the flow. This can lead to scenarios where multiple devices are alive, but only a few share the tasks (*cf.* Section 6.3.3.2, p. 72). Furthermore, the decentralized orchestration fails to accommodate the addition or removal of tasks since this would require the calculation of a new allocation list. Considering this limitation, we can conclude that our weak decentralized orchestration further violates the second and fourth capability of a strong orchestration.

Secondly, our orchestration scores calculated by Node-RED at the beginning do not change throughout the lifetime of devices. In certain scenarios, the order of failure of devices might lead to an overload of nodes assigned to a device, and there is no current mechanism to prevent that from happening (*cf.* Section 6.3.2.2, p. 63).

Thirdly, the pre-calculated scores do not distribute the nodes evenly across devices upon failures in some scenarios. In scenarios where all the devices share the same capabilities, the only parameter that affects the score calculated for each node is the current number of allocated nodes to each device. For example, let's assume three different devices with the identifiers A, B, and C and with the same capabilities. In the first iteration of the allocation algorithm, the score for the first node is equal for every device. Given that collisions are handled by sorting alphabetically according to the device identifier, the resulting list for that node is $[A, B, C]$. In the second iteration, device A already has a node allocated and thus a lower score than the others, resulting in the list

$[B,C,A]$. Similarly, the following iteration will yield $[C,B,A]$. Finally, in the fourth iteration, all the devices have the same number of allocated nodes, one node each, meaning that the process will essentially loop, yielding the results list for each node in the same order as before. The consequence of this repeating process is that for every node where device A has been placed in the first place, device B will always follow. Therefore, when device A fails, all of its nodes will be assigned to device B, instead of evenly distributing them between device B and C, leading to an unbalanced system in terms of assigned nodes (*cf.* Section 6.10, p. 63).

All of the previous limitations could be tackled by a dynamic mechanism, performed by each device during run-time, being able to take into consideration the current number of assigned nodes.

5.6 Summary

In this chapter, we present and discuss the design and implementation choices made during the development of our solution.

To employ a decentralized orchestration, we utilize the allocation algorithm present in the previous platform to pre-calculate the allocations and create an allocation list that is sent to the devices. We modify the script generation mechanism to generate a single script per node which is then stored in the `Task Repository`. Additionally, we introduce a failure detection mechanism based on LWT messages sent by the Server upon an *ungraceful* disconnection by any device. Combining all of these aspects, we obtain a decentralized orchestration capable of reallocating the tasks across devices and adapt to device failures that only depends on the central orchestrator for the initial calculation of the allocation list. After this list is sent to every device, the central orchestrator is no longer necessary to re-orchestrate the system when faced with device failures.

Additionally, we extract and isolate the previous modifications made to the communication layer of Node-RED, and present a decoupled solution that can be separately installed and used. However, by decoupling both parts, new problems appeared regarding the communication between sequential flow nodes running locally on Node-RED and on devices. To tackle this issue, we introduce the concept of *node operating modes* that further allowed us to develop a *node fallback mechanism*.

Our solution still faces limitations, mainly regarding the allocation list. Since all the allocations are pre-calculated, our decentralized orchestration cannot account for the evolution of the number of assigned nodes to certain devices without a central orchestrator. In some scenarios, this can lead to highly unbalanced allocations, as we've demonstrated in Section 5.5 (p. 45). Furthermore, our orchestration is currently incapable of managing the appearance of either new devices or tasks.

With the introduction of the `Task Repository`, the allocation list and the LWT mechanism, we hypothesise that our solution answers affirmatively to **RQ1** and **RQ3** and partially to **RQ2**, but we only attempt to find supporting evidence in Chapter 6 (p. 49).

Finally, considering the details of the developed orchestration mechanism, we can confirm that our weak decentralized orchestration violates the second, third and fourth capability of a strong

orchestration — as mentioned in the beginning of this chapter — and is, as defined by us, a valid weak orchestration.

Chapter 6

Evaluation and Validation

6.1 Experimental Setup 49
6.2 Experiments 53
6.3 Discussion 55
6.4 Replication Package 73
6.5 Summary 73

This chapter presents and discusses our experimental process devised for the validation and evaluation of our solution. Section 6.1 presents the experimental setup used throughout the experiments. Then, Section 6.2 enumerates the experiments to be conducted, detailing the flows and devices to be used. Section 6.3 presents and discusses the obtained results and Section 6.4 presents the replication package. Finally, Section 6.5 assesses to which degree our solution complies with the defined requirements, summarizing this chapter.

6.1 Experimental Setup

To validate our hypothesis and measure to what extent our goals have been reached, we devised several scenarios and experiments on which we tested our solution. During these experiments, we gathered relevant metrics to assess the system behaviour.

6.1.1 System

The experiments were performed on a computer with an Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz and 16GB of RAM, running Arch Linux x86_64. The software stack was containerized using Docker 20.10.6, and it was comprised of: (1) Node-RED 1.2.9; (2) Eclipse Mosquitto Broker 2.0.11; and (3) InfluxDB 1.8. Regarding the IoT devices, we used both virtual and physical devices, where all the virtual devices are running a Unix port of Micropython and running on Docker, and the physical devices are Espressif Systems ESP32 devices. This resulted in the following setups:

VS1 4 virtual devices with the same capabilities, able to execute general Micropython code and the temperature sensor.

VS2 2 virtual devices with different capabilities, where one device can only execute general Micropython code, and the other can also run the temperature node.

VS3 5 virtual devices with different capabilities, where each device's capabilities are represented by "devX", in which X is the device's index number.

PS1 4 physical devices with the same capabilities, able to execute general Micropython code and the temperature sensor.

PS2 2 physical devices with different capabilities, where one device can only execute general Micropython code, and the other can also run the temperature node.

6.1.2 Flows

We devised four Node-RED flows to highlight and evaluate different aspects of our solution:

FLW1 Created to represent a home or small office scenario, the flow comprises three different temperature sensors, placed in different places, that report the measured temperature every 5 seconds. Depending on the combined value of these readings, a message is sent by the flow to the AC to either turn it on, off, or active the humidifier. The flow, in total, has 38 nodes, 36 of which can be allocated to devices and can be seen in Figure 6.1.

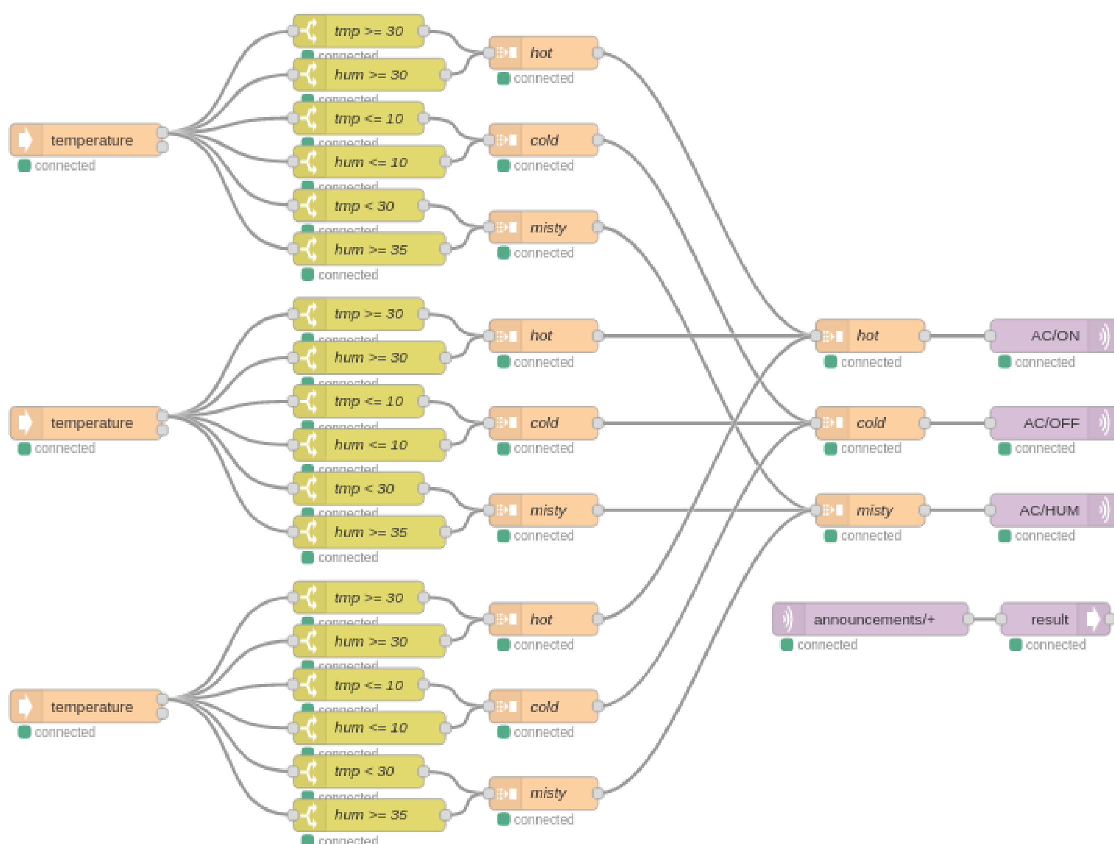

Figure 6.1: Flow 1 containing three temperature nodes that control an AC unit depending on the measured temperature

FLW2 This flow is a smaller version of FLW1, where only two temperature nodes are used, and the only action is to turn on the AC. Two additional nodes were added, connected to the second output of the temperature nodes. These nodes can obtain the temperature from a Weather API and replace, to some degree, the temperature nodes should the physical temperature sensors fail. The flow, in total, has ten nodes, 6 of which can be allocated to devices and can be seen in Figure 6.2.

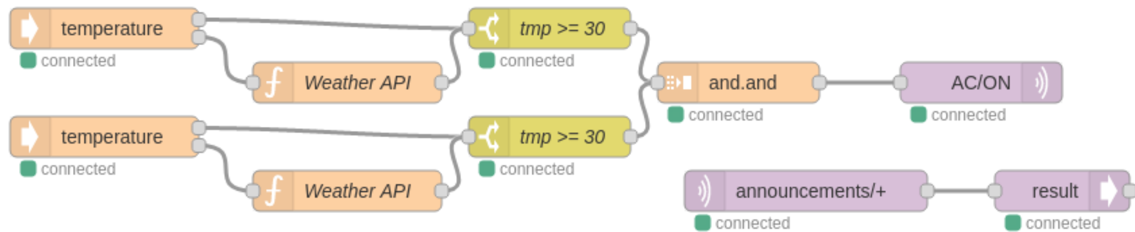

Figure 6.2: Flow 2 containing two temperature nodes that can turn on the AC unit depending on the measured temperature. Each temperature node is connected to a Weather API node that can replace it if the physical device fails.

FLW3 This flow is another smaller version of FLW1, without one temperature sensor and the humidifier check for the AC. It can be seen as the substitute for FLW1 for the ESP devices, given that FLW1 had too many nodes causing failures on the physical devices. The flow, in total, has 20 nodes, 18 of which can be allocated to devices and can be seen in Figure 6.3

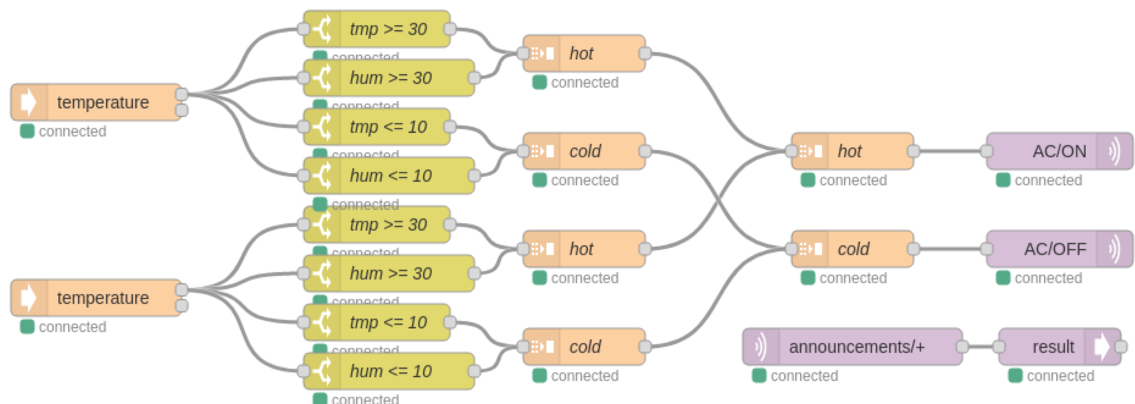

Figure 6.3: Flow 3 containing two temperature nodes that can turn off or on the AC unit depending on the measured temperature.

FLW4 This flow does not represent any real-life scenario, and its only purpose is to test the allocation result of our orchestration. The flow, besides the orchestrator and registry nodes, comprises 32 nodes, all of which share the same predicate — the device must be able to run general Micropython code — and have two priorities equal to “dev0 devX” where X is a number between 0 and 4, excluding 1. These priorities have no concrete meaning

outside this experiment, only to be used in combination with setup VS3 to impact the score calculation during orchestrations. (*cf.* Figure 6.4).

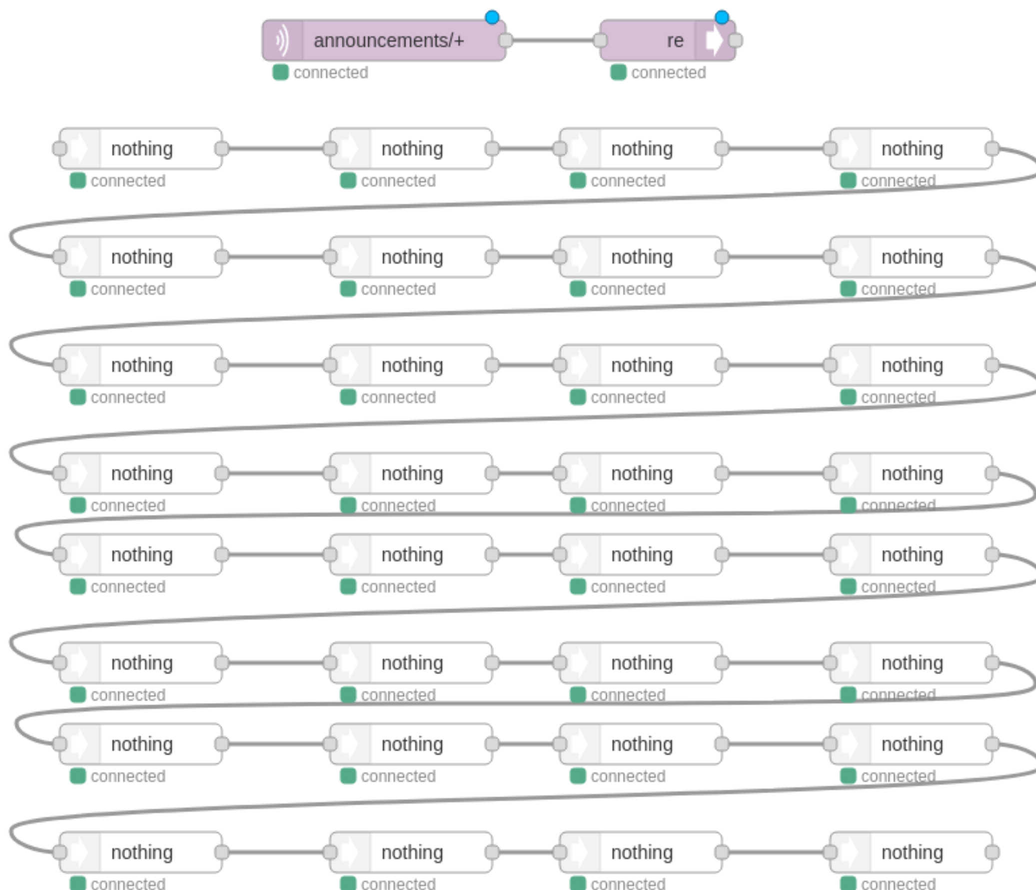

Figure 6.4: Flow 4 containing 32 *nothing* nodes that simply forward the message it receives. Each node has two priorities equal to “dev0 devX“ where X is a number between 0 and 4, excluding 1.

6.1.3 Metrics

We used InfluxDB, a time series database, to collect multiple metrics from both the devices and Node-RED during the experiments. These metrics were published by both ends through MQTT and forwarded to InfluxDB and are detailed in Table 6.1 (p. 53).

In flows FLW1, FLW2, and FLW4, we expect to receive a message in each of the AC’s input topics at a specific interval. Thus, we collect the output of those topics to assess any system downtime during re-orchestrations. In all of the following experiments, the timer of each temperature node was set to 5s, meaning that, in theory, the AC should receive a message every 5s.

We defined a 10s time interval as the minimum accepted interval for the system to re-orchestrate, which presents, in our opinion, a reasonable interval to change the operating mode of an AC unit in a home scenario. Therefore, if the final nodes fail to report a message to the AC unit in that time interval, we consider it to be system downtime. The causes for this downtime in our scenario can

Table 6.1: System Metrics Gathered During the Experiments.

Metric	Unit	Frequency	Description
Allocated RAM	Bytes	5s	Measure the run-time load of each device.
Free RAM	Bytes	5s	Measure the available memory for new tasks.
Payload Size	Bytes	At each script received	Confirm if each script is being received correctly.
Uptime	Seconds	5s	Pinpoint the time at which a device failed.
Assigned nodes	Number	5s	Confirm to allocation result and if every node is being distributed and assess the overall system balance.
Messages exchanged per device	Number	5s	Assess the overall message traffic.

be, but are not limited to: (1) delay in failure detection; (2) time spent in re-orchestrations; and (3) setup time spent by each device in installing its scripts. To prevent the loss of metric data due to synchronization issues, we maintained the reporting frequency of 5s and later aggregated each metric data in 10s intervals.

6.2 Experiments

We divided our experiments into three main groups: (1) *Sanity Checks* to verify if our solution can comply with the most basic scenarios; (2) *Improvements* to assess the new features and capabilities of our solution; and (3) *Limitations* to show the current shortcomings of our work. For each group, a set of experiments, using combinations of the different flows and setups presented in Section 6.1 was created and is presented in this section.

6.2.1 Sanity Checks

We devised two Sanity Checks to verify that our solution could manage the most basic scenarios. The first evaluates if our platform can decompose and allocate the nodes to the available devices properly. Then, the second check verifies if it can reallocate those same tasks when faced with device failures. In both sanity checks, the central orchestrator in Node-RED was kept alive. The setup in terms of devices and flow used can be seen in the following list:

SC1-VS1 Using FLW1, with Node-RED and all devices operations at all times, mimicking the most basic scenario without requiring any re-orchestrations or failure detection.

SC1-PS1 Follows the same procedure as SC1-VS1 but utilizes setup PS1.

SC2-VS1 Using FLW1 and setup VS1, with Node-RED operational, but the devices are turned off one at a time until the Minimum Working System (MWS) is reached, requiring the orchestrator to perform multiple re-orchestration upon detecting device failures.

SC2-PS1-A Follows the same procedure as SC2-VS1 but utilizes setup PS1.

SC2-PS1-B Follows the same procedure as SC2-VS1 but utilizes setup PS1 and FLW3.

6.2.2 Improvements

We defined three experiments to validate our system's behaviour when the orchestrator fails and the introduced features. The first experiment evaluates if our platform can provide a correct decentralized orchestration mechanism. The second and third evaluate the correct behaviour of the nodes' operating modes and the fallback mechanism (*cf.* Section 5.4, p. 44). The setup in terms of devices and flow used can be seen in the following list:

I1-VS1 Using FLW1, keep Node-RED operational only until the first allocation is made and sent to the devices. After, turn the devices off one at a time until the MWS is reached, forcing the devices to re-orchestrate without the central orchestrator.

I1-PS1 Follows the same procedure as I1-VS1 but utilizes setup PS1.

I2-VS2 Using FLW2, keep Node-RED operational and first turn off the device with the most capabilities off, followed by the remaining device, forcing Node-RED to activate the fallback nodes for the non-allocated nodes. After, turn them on in the reverse order, forcing Node-RED to deactivate the fallback nodes.

I2-PS2 Follows the same procedure as I2-VS2 but utilizes setup PS2.

I3-VS2 Using FLW2, keep Node-RED operational and introduce a random memory error in the virtual devices, which, in its turn, will trigger the failsafe mechanism of devices. This will lead to the orchestrator assigning fewer and fewer nodes to the devices, as they keep failing and forcing it to manage nodes running in all three operating modes.

I3-PS2 Follows the same procedure as I3-VS2 but utilizes setup PS2.

6.2.3 Limitations

We created two experiments to present and assess the current limitations of our decentralized orchestration. First, we compare the results of the centralized and decentralized allocation. Then, we show the inability of the decentralized orchestration to respond to device appearances.

L1-VS3 Using FLW4, keep Node-RED operational only until the first allocation is made and sent to the devices. After, keep the device with the most capabilities operational and turn off, in any order, the remaining devices, forcing them to re-orchestrate without the central

orchestrator. Then, repeat the same process but keep Node-RED operational throughout the experiment.

L2-VS1 Using FLW1, keep Node-RED operational only until the first allocation is made and sent to the devices. After, keep one device alive and turn off, in any order, the remaining devices, forcing them to re-orchestrate without the central orchestrator. Afterwards, turn on the dead devices until every device is operational.

6.3 Discussion

In this section, we present and discuss the obtained experimental results for each experiment defined in Section 6.2. First, in Section 6.3.1 we evaluate if the system was able to handle the most basic scenarios during the sanity checks with the central orchestrator. Then, in Section 6.3.2 we deactivate the central orchestrator and verify if our decentralized orchestration can produce working allocations when faced with device failures. Additionally, we verify if our fallback mechanism is working as intended Section 5.4 (p. 44). Finally, in Section 6.3.3, we proceed to assess the limitations of our decentralized orchestration by comparing the allocations made by both orchestrations.

6.3.1 Sanity Checks

The sanity checks made to test if our solution could cope with the most basic scenario are presented and discussed next. Each sanity check was done with both virtual and physical devices and, due to the constrained nature of the latter, an additional variation was added.

6.3.1.1 SC1-VS1

The objective of this experiment is to ensure that the allocation of nodes to devices and the communication between them is working correctly, without any device failures. The flow is composed of 36 nodes and, given that all devices share the same capabilities, the algorithm should evenly allocate nine nodes per device.

As can be seen in Figure 6.5 (p. 56), this does indeed happen when, at approximately 25s, each device receives nine nodes, which stay constant throughout. As expected, this also coincides with a decrease in the available RAM space and, consequently, the increase of the allocated RAM. The uptime of each node also never decreases, indicating no failure.

Devices 2, 3, and 4 present a higher count of exchanged MQTT messaged, concerning Device 1, due to no temperature device being assigned to Device 1. It is also important to notice that, despite Node-RED having no nodes allocated to it, it still has a considerable number of sent MQTT messages due to the ping mechanism.

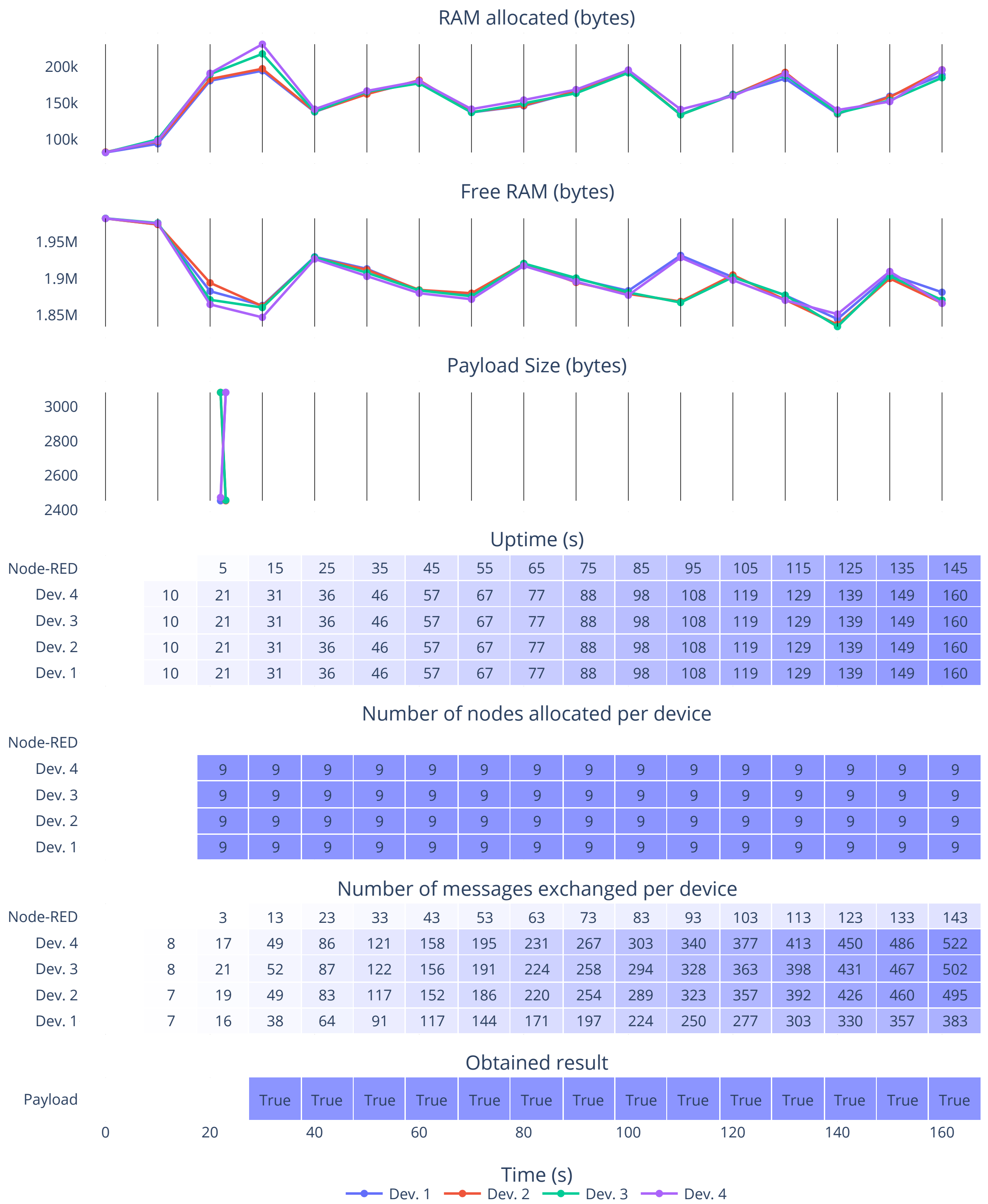

Figure 6.5: SC1-VS1 measurements showing the correct and balanced allocation of the flow nodes through the available virtual devices.

6.3.1.2 SC1-PS1

This experiment repeats the same process as the previous SC1-VS1 but makes uses of four ESP32 physical devices.

As seen in Figure 6.6, the results are very similar, as expected, with the only difference being the usage of RAM. The virtual devices in SC1-VS1 use a maximum of 200k bytes, while the physical devices only require 60k bytes. This can be explained by possible optimizations or an increased number of garbage calls between the ESP and Docker firmware ports.

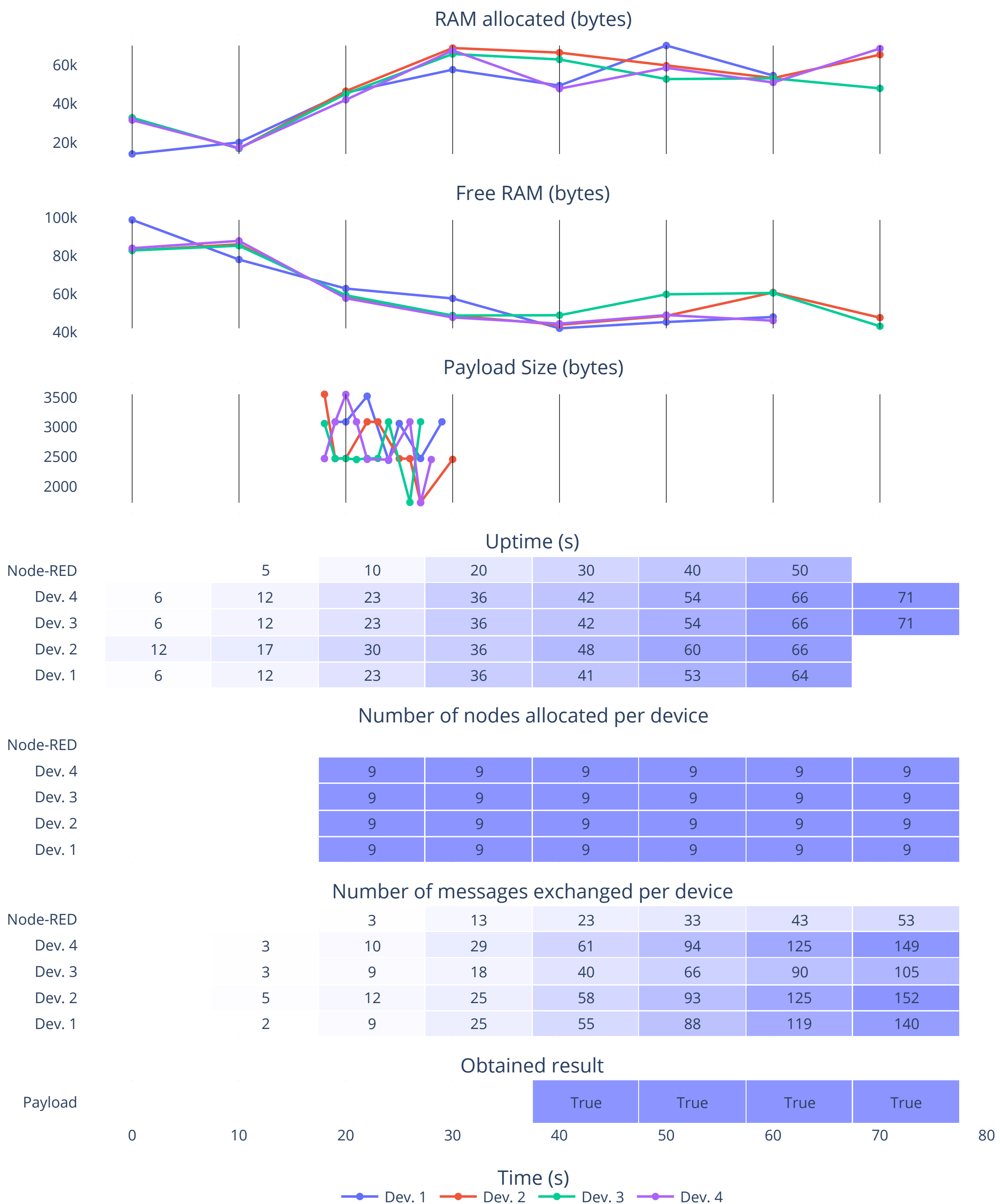

Figure 6.6: SC1-PS1 measurements showing the correct and balanced allocation of the flow nodes through the available physical devices.

6.3.1.3 SC2-VS1

This experiment has the same objective as SC1-VS1, however, the devices are turned off one by one until the MWS is achieved. Since this experiment utilizes virtual devices with no actual memory limits or constrained aspects, the MWS is one device.

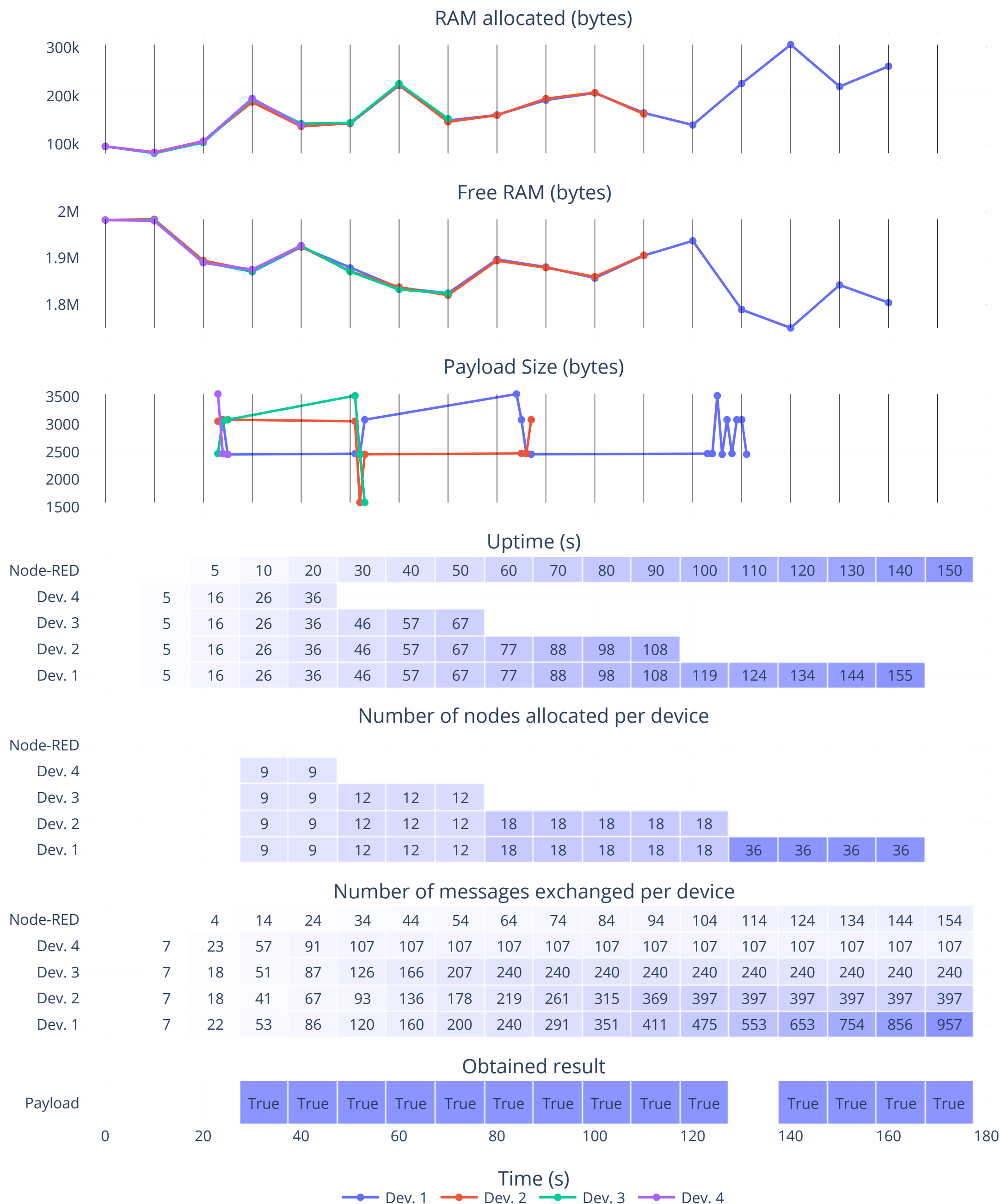

Figure 6.7: SC2-VS1 measurements showing the failure of the devices and the reallocation of the tasks among the still available virtual devices, confirming the correct behaviour of the failure detection mechanism with LWT messages.

As we can see in Figure 6.7 (p. 58), the devices initially get nine nodes each. After Device 4 fails at around 50s, its nine nodes are evenly redistributed by the other three devices, totalling 12 nodes per device. This happens two more times when Devices 3 and 4 fail, leaving Device 1 with the complete set of 36 nodes. As expected, the free RAM space keeps diminishing for this device, and the allocated RAM and the number of MQTT messages increases.

We can then conclude that the failure detection mechanism — with LWT messages — is working correctly and in a timely manner given that almost immediately after a device fails, the other nodes receive their scripts, as we can see looking at the payload graph (*i.e.*, new payload points appear closely after a device stops reporting uptime data). Furthermore, we can also conclude that the allocation algorithm is dealing correctly with device failures.

However, the system experienced downtime between the 120 and 140s. This was due to the re-orchestration made after Device 2 failed — the last device to do so — meaning that Device 1 had to install the full 36 nodes, failing to produce the required temperature messages or pipe them along the way to the final node while doing such.

6.3.1.4 SC2-PS1-A

Following SC2-VS1, this experiment repeats the same process but replaces the virtual devices for 4 ESP-32 and highlights the differences between the virtual and physical scenarios. As we can see in Figure 6.8 (p. 60), for this flow of 36 nodes, the MWS was revealed to be two devices instead of one as previously.

At around 200s into the experiment, only Device 2 and 4 were alive, and, after the failure of Device 2, Device 4 did not report any more data and failed shortly after, leaving Node-RED to assume all the nodes. Since this flow does not have any fallback nodes for the temperature nodes, there are no additional messages sent to AC after the failure, leading to constant system downtime. It is also in the Obtained Result graph that we can observe the most significant difference between virtual and physical devices.

While in SC2-VS1, we received at least one positive message to turn on the AC in every 10-second block but one (after the failure of the third device), in this experiment, we can see multiple and continuous failures in the reception of that message. These failures coincided with the two re-orchestrations made during the experiment, after the failure of Device 3 (at around 60s) and Device 1 (110s), and lasted for 30 and 70 seconds, respectively.

The reason for this difference is twofold. The first reason is explained in Section 5.3 (p. 41), and it regards the interval the MQTT server has to wait before detecting the failure of the physical device. The second reason regards the time an ESP-32 device takes to install one script when compared to the virtual device. While this task is almost instantaneous in the latter, it takes between 1 and 2 seconds to install in the former, causing a significant delay. Visually, this can also be seen by the amount and space between data points in the Payload chart between this and the previous experiment; in SC2-VS1, fewer points appear since the device sends them almost simultaneously. Combining the two reasons, we obtain a system that takes longer to react and adapt to changes, leading to more extensive intervals without obtained its desired output.

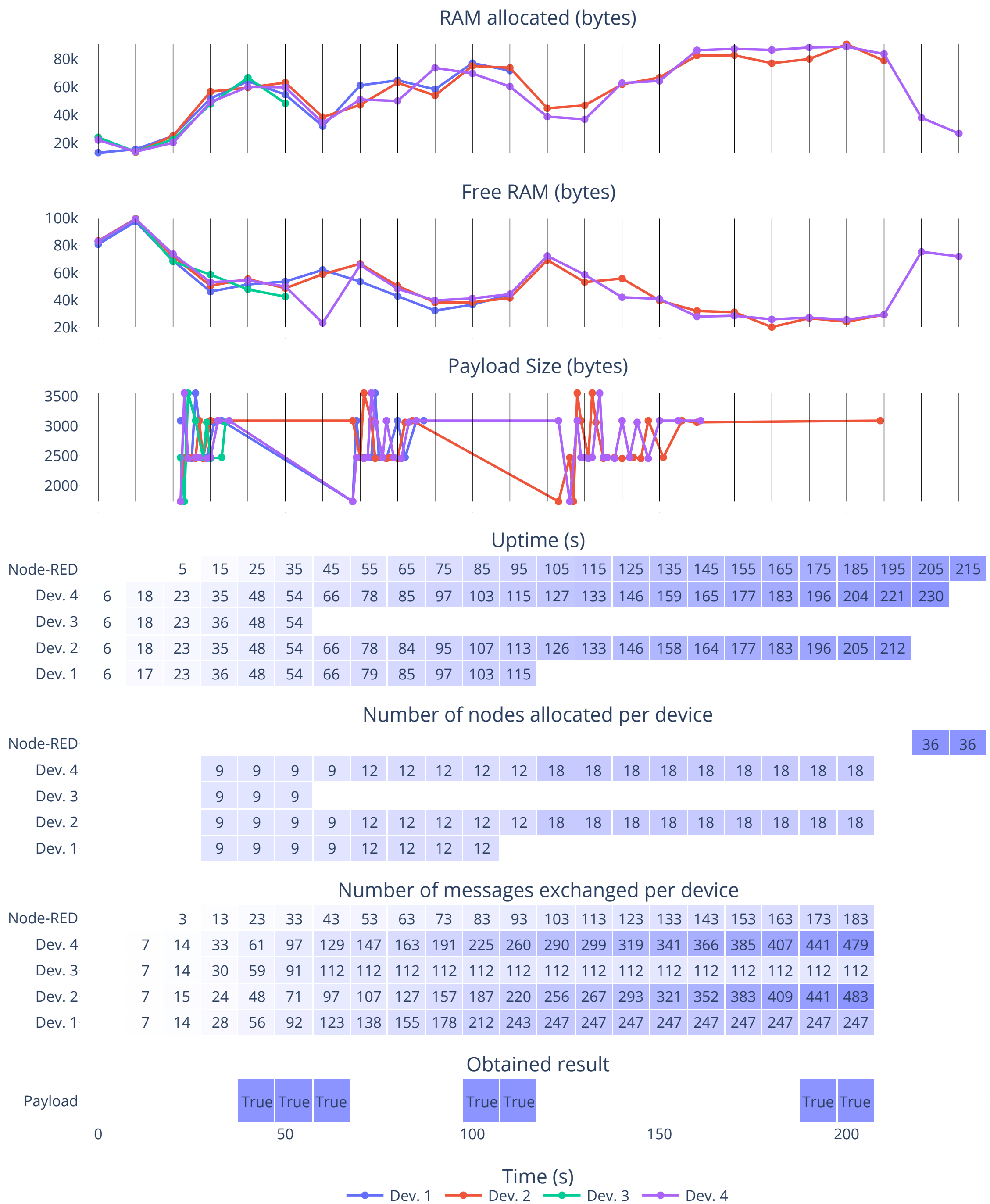

Figure 6.8: SC2-PS1-A measurements showing the failure of the devices and the reallocation of the tasks among the still available physical devices until the MWS of 2 physical devices is achieved (highlighting the constrained aspects of the physical devices).

Moreover, in this experiment, we also reached the memory limits of an ESP-32 device, further highlighting the constrained aspects of the IoT edge devices. After the failure of Device 2, at around 210s, all of the nodes are allocated to Device 4, the last device alive. However, Device 4

was never able to install and report the number of allocated nodes. Instead of reporting the correct number of allocated nodes and higher RAM usage, the device reported a significant decrease in the allocated RAM (and consequently an increase in the free RAM) and zero nodes assigned. Since both the allocated and free RAM levels did not reach their initial values, we suppose that the device failed whilst deleting and installing the new nodes, which led to the failure of this sub-routine in the firmware, and thus the reporting of zero allocated nodes.

6.3.1.5 SC2-PS1-B

Since in the previous experiment we observed prolonged system downtime and the overload of a device when assigned the entire flow, we devised a similar flow with fewer nodes, 18 in total, to assess the impact of it on the system downtime and obtain an MWS of 1 device.

As seen in Figure 6.9 (p. 62), the reduction of the number of nodes led to a shorter system downtime between re-orchestrations.

After the failures of Devices 3, 4, and 1, the downtime was, respectively, 10, 30, and 30 seconds. Additionally, upon the failure of Device 1, Device 2 was able to install and execute all the flow nodes after taking around 30s to install.

6.3.2 Improvements

The following experiments intend to validate the new capabilities of our solution. First, we assess the behaviour of the decentralized orchestration mechanism by removing the Node-RED instance. Then, we test the fallback mechanism and node operating modes and their interaction with the failsafe mechanism.

6.3.2.1 I1-VS1

The objective of this experiment is to validate the developed decentralized orchestration mechanism by forcing the devices to re-orchestrate without the orchestrator. As we can observe in Figure 6.10 (p. 63), Node-RED is only up at the 15s mark for 5 seconds, the time required for it to pre-compute the allocations and send them to the devices. Then, the Node-RED instance is terminated, and the devices are manually turned off, one by one, until the MWS is achieved.

After the first Device 2 fails, we can see the first moment of the decentralized orchestration. All the nodes assigned to Device 2 are now Device 3 responsibility, following the allocation list computed by the orchestrator. Following Device 4 failure, the same applies to Device 1, now also responsible for 18 nodes. After Device 3 fails, Device 1 is left with all 36 nodes. During these reallocations, no system downtime was recorded.

However, when comparing the results of this orchestration to the one seen in Figure 6.7 (p. 58) with Node-RED operating, we can see one of the limitations of our decentralized orchestration mechanism: the nodes are not evenly distributed across the still-functioning devices. Instead of splitting the nine nodes of the first failed device across the other three devices, as seen in the

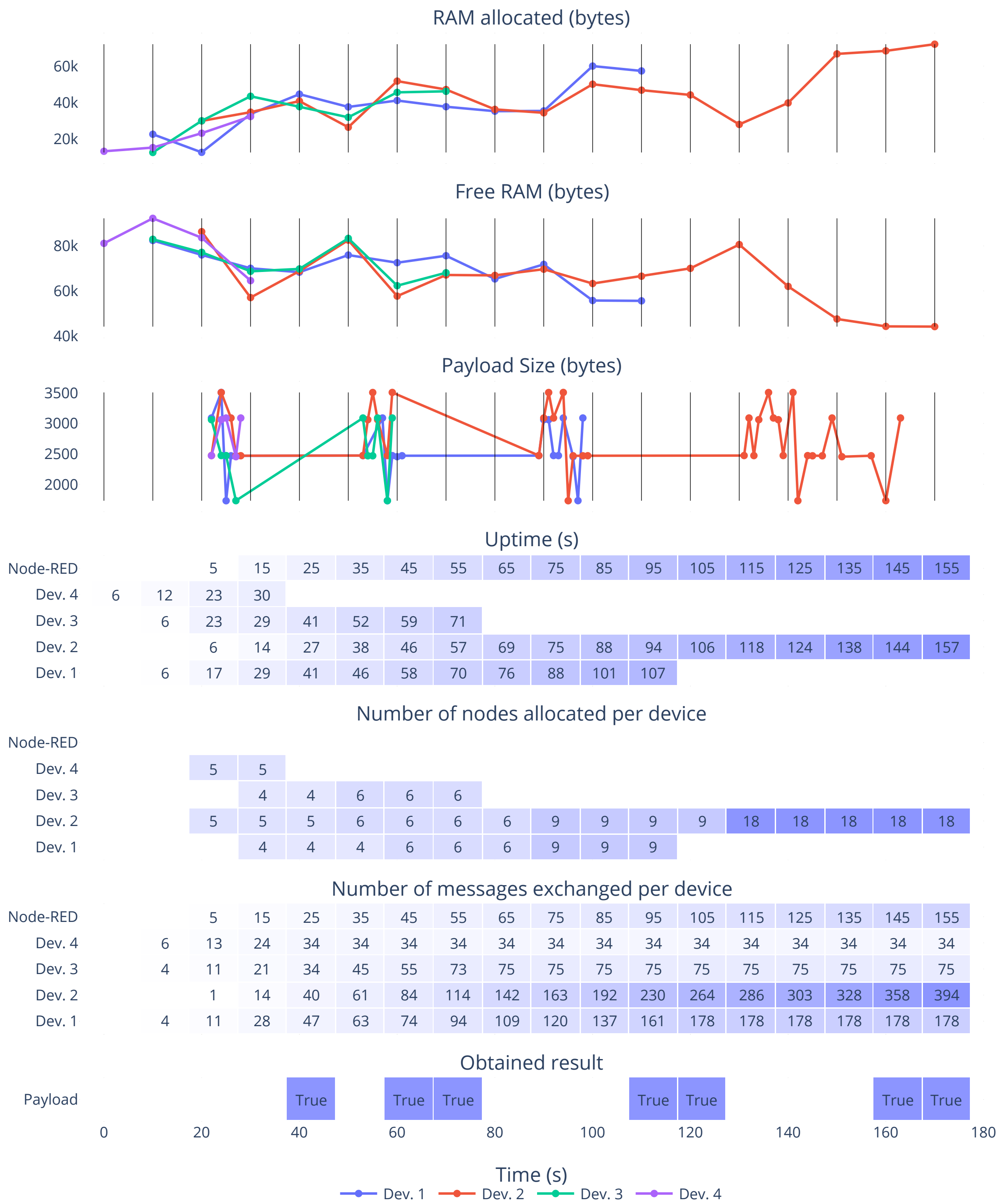

Figure 6.9: SC2-PS1-B measurements showing the failure of the devices and the reallocation of the tasks among the still available physical devices until only one device is alive.

second sanity check (*cf.* Section 6.3.1.3, p. 58), all of those were assigned to Device 3. The reason for this behaviour was explained in Section 5.5.

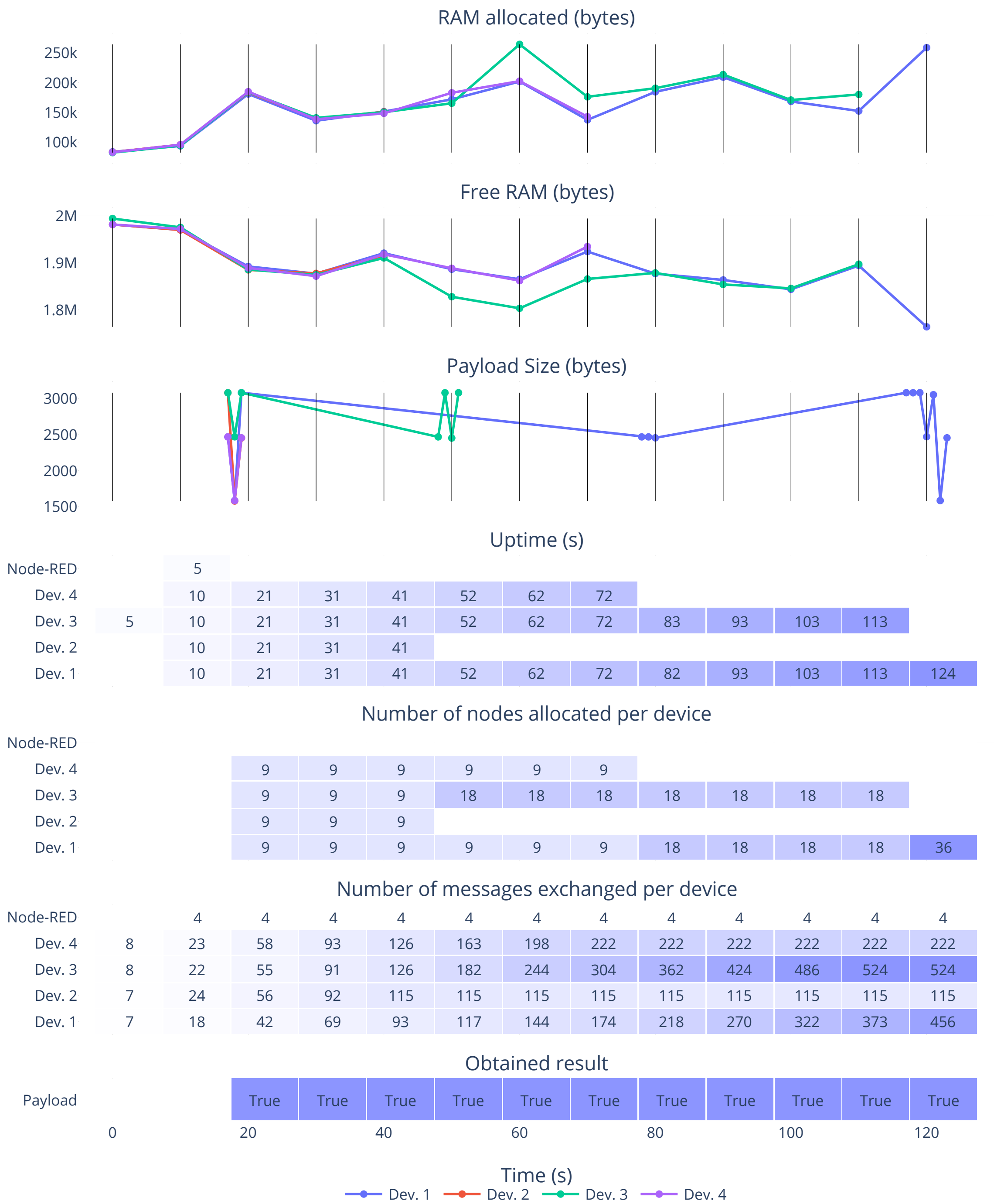

Figure 6.10: I1-VS1 measurements showing the reallocations made by the decentralized orchestration with virtual devices after Node-RED has failed.

6.3.2.2 I1-PS1

This experiment repeats the same process as the previous but makes use of physical devices and a smaller flow, FLW3, for the reasons presented in Section 6.3.2.1.

As we can see in Figure 6.11 (p. 64), and similarly to the previous experiment, the devices

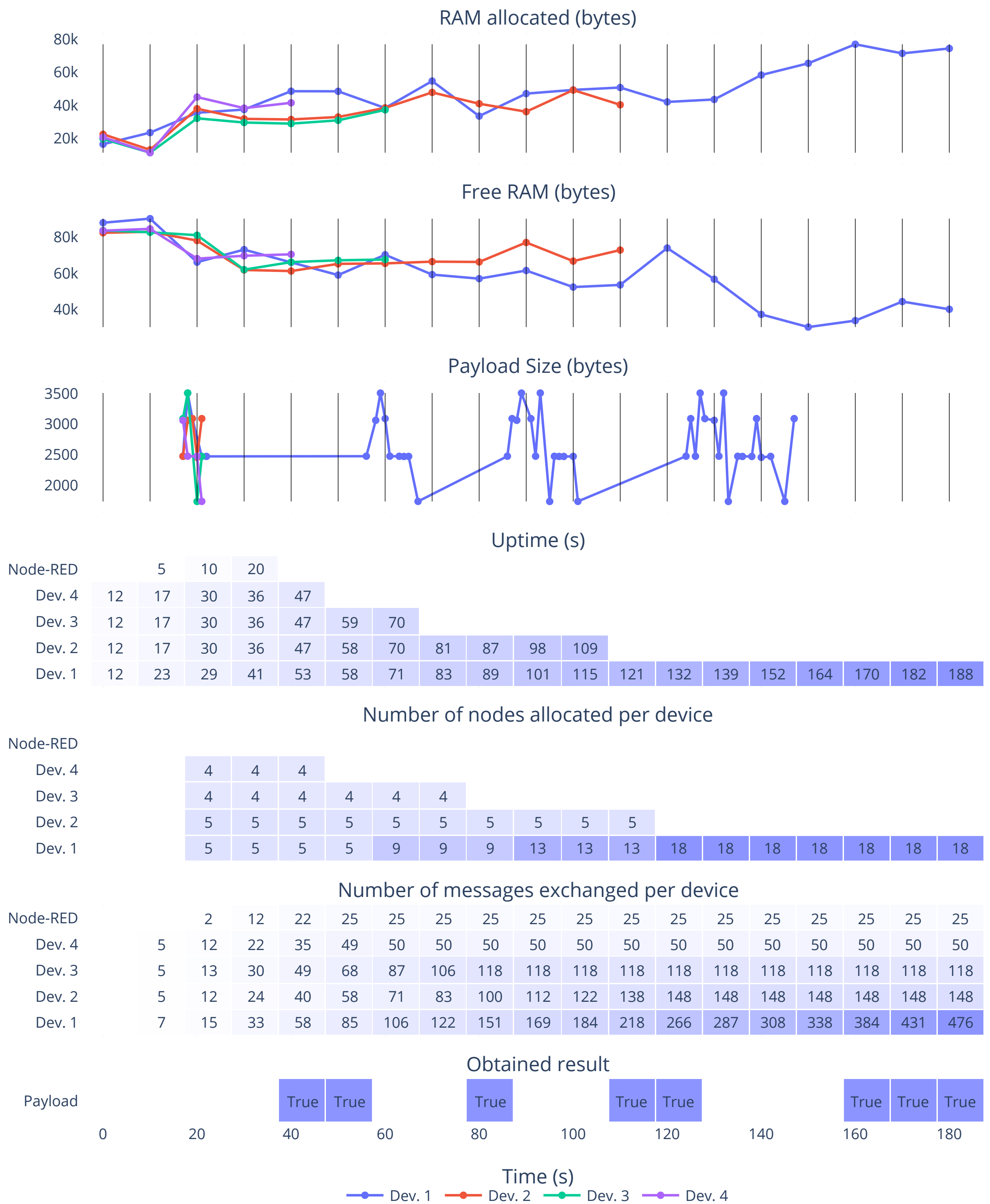

Figure 6.11: I1-PS1 measurements showing the reallocations made by the decentralized orchestration with physical devices after Node-RED has failed.

can distribute the nodes among themselves upon failures without a central orchestrator. Similar to the sanity checks with physical devices, we can also observe system downtime after the failure of each device due to the delay induced by both the failure detection mechanism and the longer script

installation times. Nonetheless, this experiment allows us to verify that our solution can provide a decentralized orchestration capable of reallocating tasks when faced with device failures, and, therefore, answering affirmatively to the research questions **RQ1** and **RQ3**, and partially to **RQ2** since it is able to handle device failures but not appearances, as seen in Section 6.3.3.2 (p. 72).

Furthermore, the observed allocations during the experiment reveal the worst possible allocation sequence with our current solution. Following the example in Section 5.5, the order in which these devices failed — from Device 4, with the smallest identifier, to Device 2, with the highest identifier, alphabetically — resulted in all of the nodes being reallocated to Device 1, creating the most unbalanced allocation.

6.3.2.3 I2-VS2

The objective of this experiment is to validate the fallback mechanism and node operating modes introduced in Section 5.4 (p. 44). The flow and number of devices were purposely made smaller to make the results easier to read.

In this experiment, we have two devices: (1) Device 1 that can only run common Micropython nodes; and (2) Device 2 has a temperature sensor and can run the temperature node. Since the flow has two temperature nodes, these have been allocated to Device 2, and the remaining nodes can be distributed evenly. Additionally, the two *Weather API Nodes* have not been modified to generate Micropython code and thus have to be assigned to Node-RED.

As we can see in Figure 6.12 (p. 66), in the first 50s of the experiment, this is what happens when Node-RED is assigned two nodes, and Devices 1 and 2 are given three nodes each — 2 of those 3 being the temperature nodes for Device 2.

Upon the failure of Device 2, we can see that from its three nodes, two were reallocated to Node-RED and one to the other device. The nodes reallocated to Node-RED were the now inactive temperature nodes that need to be *replaced* by the Weather API nodes.

After receiving the *local operating mode* by the orchestrator, the temperature nodes send a message through their second output channel, triggering the Weather API nodes. It is important to note that since the temperature data is now being produced locally by Node-RED, the two nodes attached to them — that check if the temperature is above 30 degrees — will be receiving data through the events in Node-RED but, at the same time, running in Device 1. The *if* nodes are running in the *remote mode* (*cf.* Section 5.4, p. 44) and since they received a message through events, they forward it to the *input topic* of their respective device, maintaining the flow.

After Device 1 fails, there are no devices available, and Node-RED assumes all nodes. Given that the other nodes do not depend on specific hardware to be executed, Node-RED can run their normal *javascript* code.

At around 100s into the experiment, Device 1 is turned on again, and it is once again assigned the same nodes that it had, equally distributing the load accordingly to the device capabilities. As seen in Section 6.3.3.2 (p. 72), the orchestrator capability of handling the appearance of new devices after the first allocation has been made presents one of the major advantages of the centralized solution over ours. Device 1 is then also turned on, and, as expected, the allocation is the

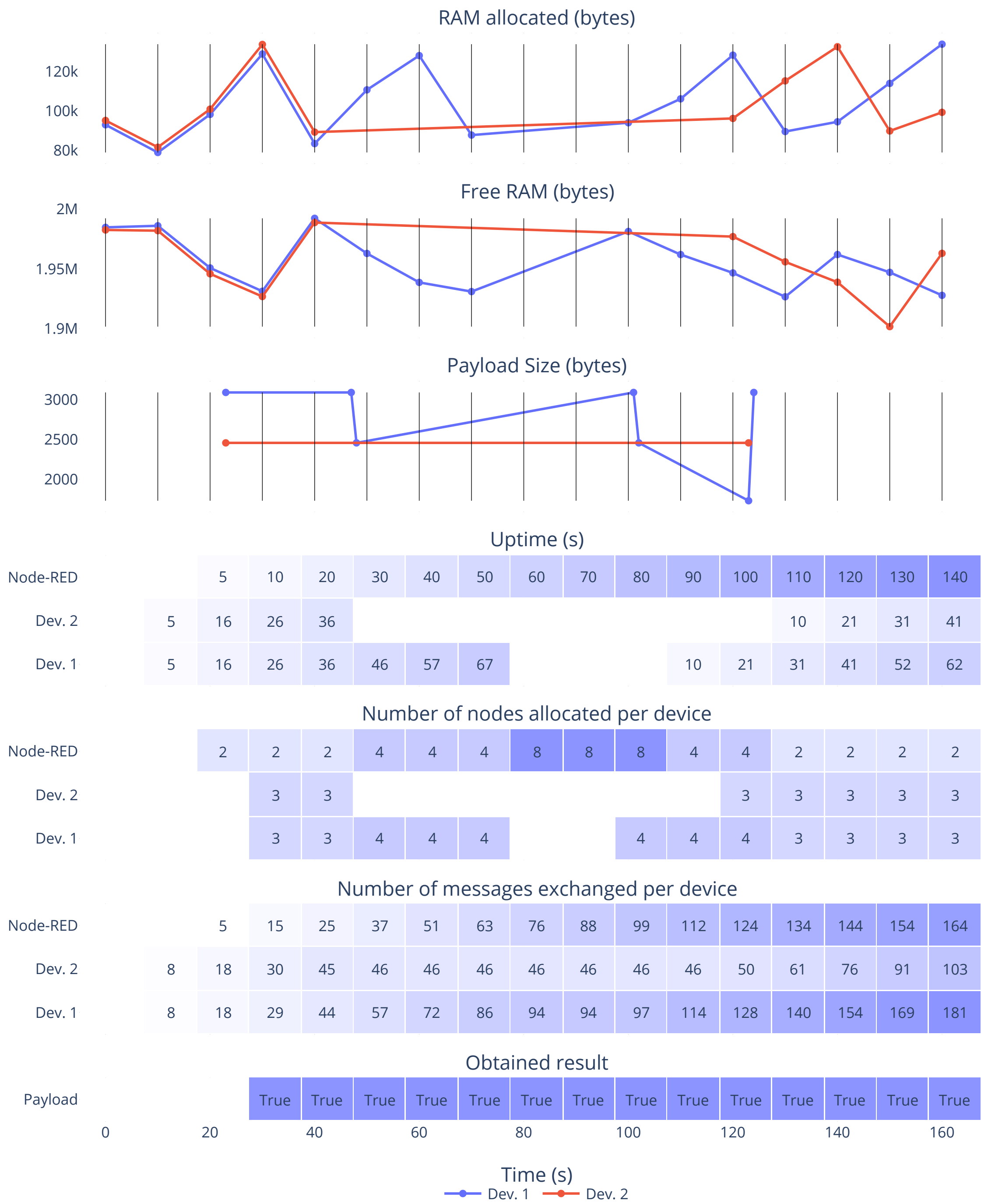

Figure 6.12: I2-VS2 measurements showing the introduced fallback mechanism with virtual devices, where Node-RED is capable of triggering local responses to the failure of devices, maintaining a functional system.

same as initially, having no system downtime recorded throughout the experiment. We can then conclude that the fallback mechanism is working as expected.

6.3.2.4 I2-PS2

Following I2-VS2, this experiment repeats the same process but makes use of 4 physical devices.

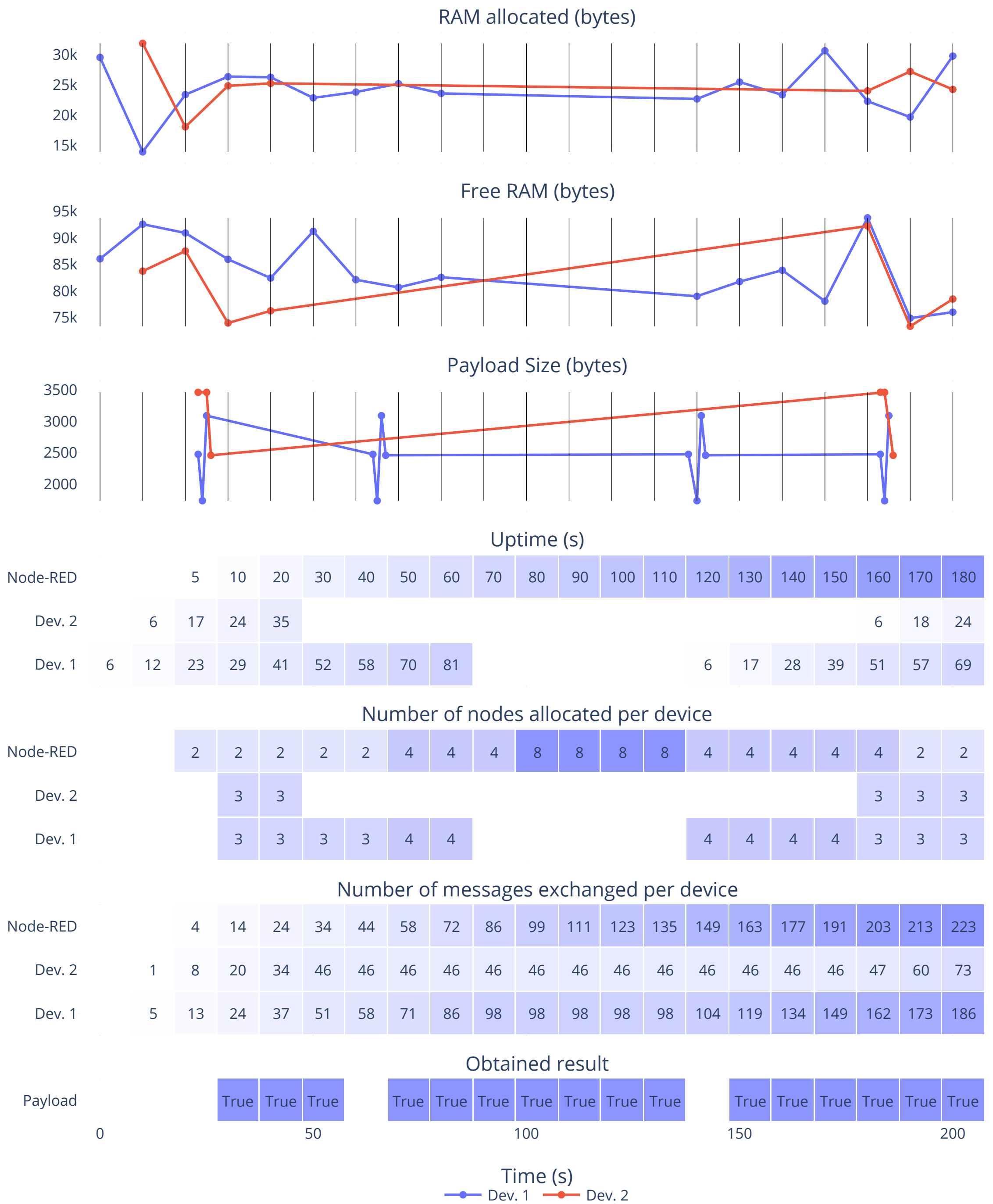

Figure 6.13: I2-PS2 measurements showing the introduced fallback mechanism with physical devices, where Node-RED is capable of triggering local responses to the failure of devices, maintaining a functional system.

As we can see in Figure 6.13 (p. 67), the system behaviour is very similar, with the only difference appearing in the system downtime. While no system downtime was detected in I2-VS2, in this experiment, upon the failure of Device 2 and the re-appearance of Device 1, the system was unable to send a message to the AC for 10 seconds. This increase in the system downtime was expected due to the introduced delay in detecting the failure by using physical devices, as seen in I1-PS1. Nonetheless, the results are positive, confirming that the fallback mechanism is also working as expected with physical devices.

6.3.2.5 I3-VS2

The objective of this experiment is to further test the fallback mechanism and its interaction with the failsafe mechanism (*cf.* Section 5.3, p. 41). Similar to I2-VS2, the two devices have different capabilities, where only Device 2 can run the two temperature nodes in the flow. To trigger the failsafe routine, we introduced a 25% possibility of provoking a memory error every 10 seconds in each device.

As seen in Figure 6.14 (p. 69), the first failsafe occurs at around the 30s mark, and it is of Device 2. As expected, the orchestrator allocates one less node to Device 2. Since that same node was reallocated to Node-RED and not to Device 1, we know that it was a temperature node, triggering the respective fallback node. This process repeats four more times until no node is assigned to Device 1, after failing with only one node.

During the experiment, nodes running locally on Node-RED had to communicate with nodes in devices and vice-versa. Since there was no downtime registered, this validates the *proxy mode*, where the output from the device is sent to the node, through a *proxy topic*, and forwarded to the next node through the native Node-RED event-based communication.

6.3.2.6 I3-PS2

Following I3-VS2, this experiment has the same objective but makes uses of physical devices. As can be seen in Figure 6.15 (p. 70), the orchestration results are identical with no system downtime detected.

With the introduction of the failsafe mechanism, the system is no longer relying on the LWT message to perform a new orchestration since that the device does not entirely fail and, therefore, does not disconnect itself from the broker. Consequently, it is not necessary to wait for one and a half time the Keep-Alive time for the broker to communicate the device failure, as previously observed in Section 6.3.2.2 (p. 63), and thus removing any significant delay between this event and the new re-orchestration. We can then conclude that our fallback mechanism is working as intended with both physical and virtual devices.

6.3.3 Limitations

The results and discussion of the experiments regarding the limitations of our solution are presented next. Since these experiments do not depend or are affected by any distinct characteristic

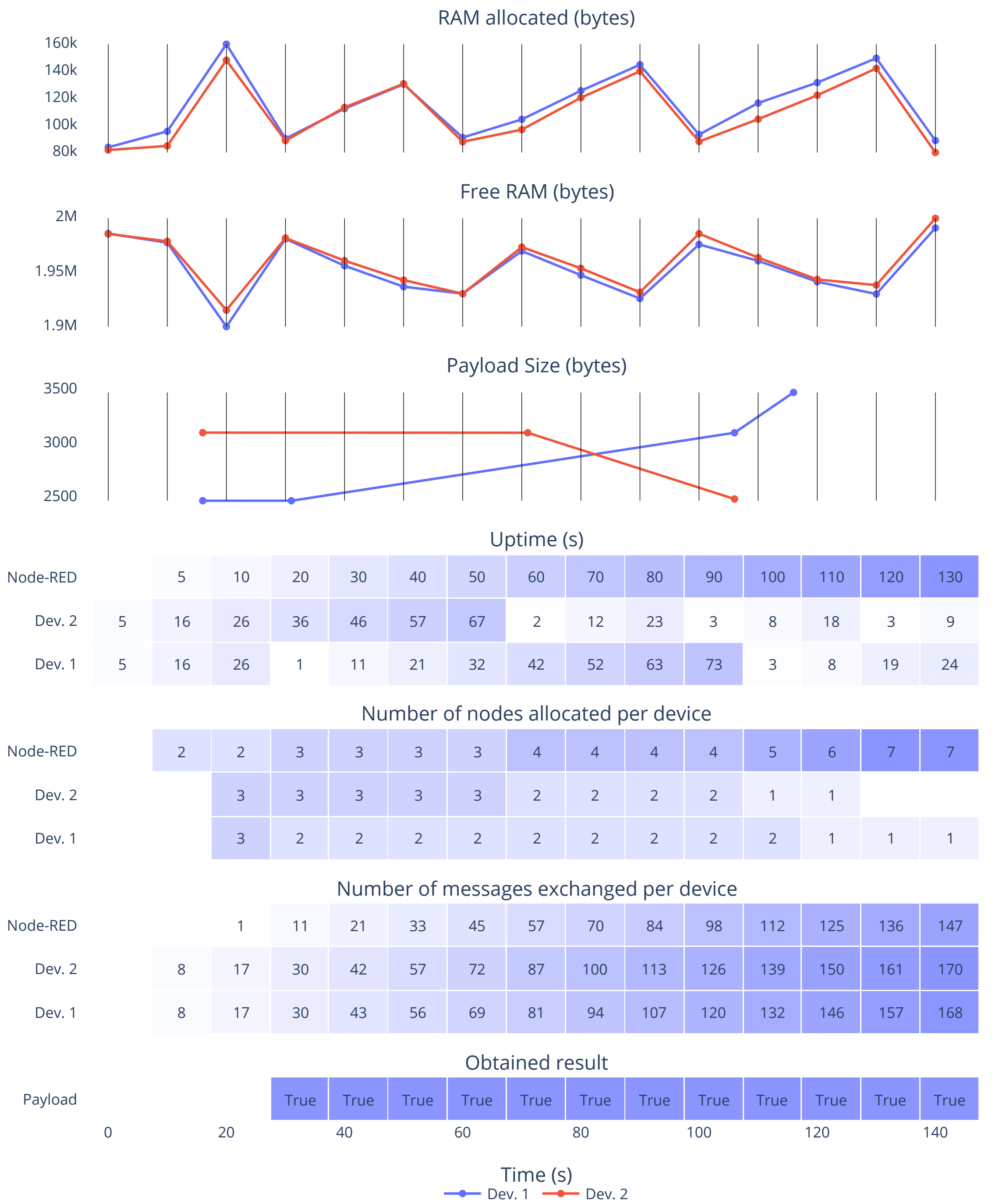

Figure 6.14: I3-VS2 measurements showing the interaction of the fallback and failsafe mechanisms with virtual devices. Node-RED assigns fewer nodes to each device after each failsafe, thus either triggering the fallback nodes or assuming the task.

of the physical devices that may have over the virtual ones, we only used virtual devices due to the simpler setup.

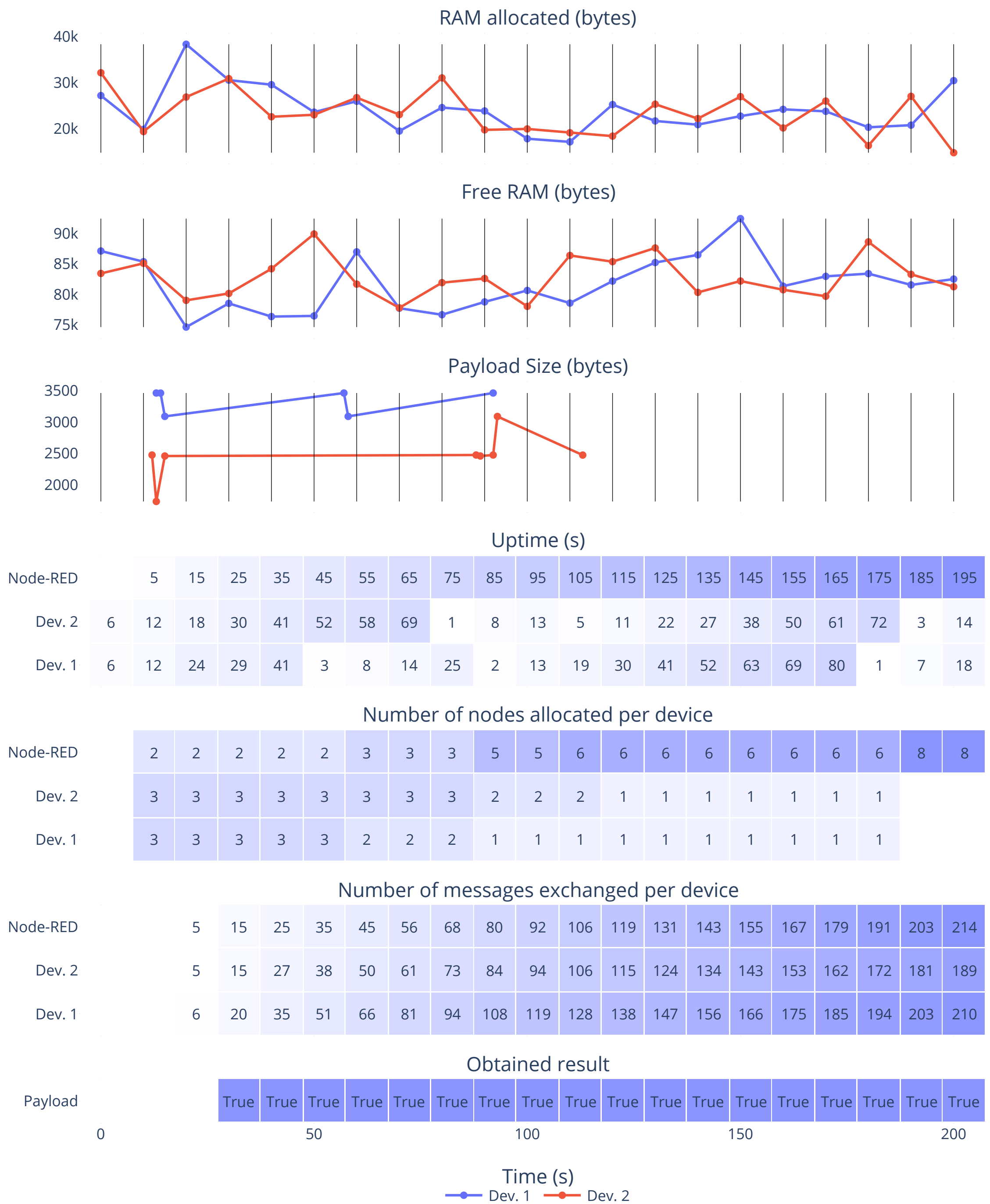

Figure 6.15: I3-PS2 measurements showing the interaction of the fallback and failsafe mechanisms with physical devices. Node-RED assigns fewer nodes to each device after each failsafe, thus either triggering the fallback nodes or assuming the task.

6.3.3.1 L1-VS3

This experiment makes use of FLW4 (*cf.* Figure 6.4, p. 52) and five virtual devices (*cf.* Section 6.1.1, p. 49), each with a different capability equal to "devX", where X is a number between

0 and 4. Each node in the flow also has a pair of priorities equal to “dev1 devX”. Since Device 1 has the most required capability of the devices, it will always score better in complying with each node priorities, which is the most valued heuristic followed by the equal distribution of nodes.

As the number of nodes assigned the Device 1 increases, the orchestrator will try to balance the distribution of nodes when possible. However, since the decentralized orchestration is based on pre-computed allocations, it does not consider any changes in the system after the orchestrator has died, being unable to account for the rising load induced in Device 1. Thus, the objective of this experiment is to compare the different allocations made by the centralized and decentralized mechanisms and validate the limitations presented in Section 5.5.

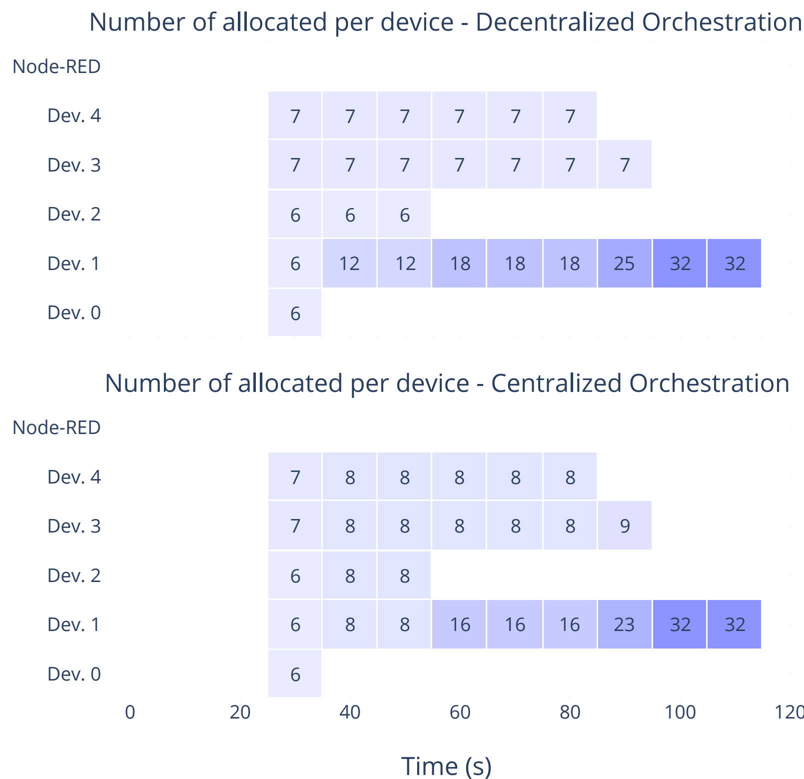

Figure 6.16: L1-VS1 measurements comparing the allocations of the centralized and decentralized orchestrations. The centralized version presents a more balanced distribution of the nodes across the devices.

As we can observe in Figure 6.16, the initial allocation of nodes is, as expected, identical in both parts since no change was made to flow or devices. Upon the failure of Device 0, while the orchestrator evenly distributed its nodes among the remaining four devices, the decentralized orchestration transferred them to Device 1, resulting in a more unbalanced distribution. This inequality is not due to the problem presented in Section 5.5 regarding the alphabetical order of devices. That problem is only present in scenarios where all devices share the same capabilities, which is not the case. Instead, this allocation result is due to the *desirable* “dev1” capability of Device 1 that allows it to better comply with the priorities of the nodes. Since the algorithm

favours complying with priorities over the equal distribution in the number of assigned nodes, it rewards the device with a higher score.

Around 60s, when Device 2 fails, the result is similar in both parts when all nodes are reallocated to Device 1. Again, this highlights the overall preference of the algorithm to comply with the nodes' priorities than to distribute the tasks evenly.

Later, at around 90s, when Device 4 fails, we can observe a difference of one node in the reallocation result. Without Node-RED, all of the eight nodes are reallocated to Device 1, further unbalancing the distribution of nodes. With Node-RED, one of the eight nodes is instead assigned to the other device alive, Device 3. Although Device 1 can better comply with the priorities of the nodes previously held by Device 4, the difference between the number of assigned nodes of Device 1 and 3 becomes significant enough for the orchestrator to try balancing it. This highlights the advantage of considering the current number of allocated nodes and re-calculating the scores in run-time. Finally, upon the failure of the last device, all nodes are assigned to Device 1, the last device alive.

We can then conclude that, while only marginally, the centralized orchestration better distributes the nodes among devices.

6.3.3.2 L2-VS1

The objective of this experiment is to validate the inability of the decentralized orchestration mechanism to account for the appearance of new or recently failed nodes.

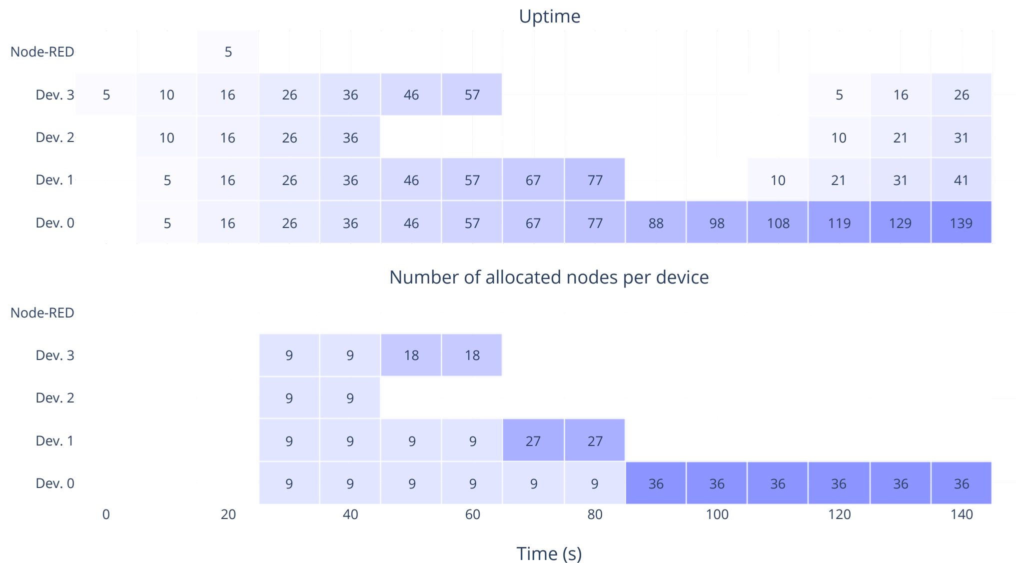

Figure 6.17: L2-ES1 measurements showing the inability of the decentralized orchestration to account for device appearances due to the pre-computation of the allocations.

As we can observe in Figure 6.17 (p. 72), Node-RED is only up enough time for the first allocation to be made and sent to each device. Then, only Device 1 is kept alive while the remaining are turned off sequentially and turned back on after.

After Devices 2, 3, and 4 are turned off, Device 1 is responsible for all the nodes, exactly like the result seen in Section 6.3.1.1. At around 110s into the experiment, Device 2 is turned back on. Since Node-RED is no longer operational and the decentralized orchestration does not handle appearances of devices, the number of allocated nodes per device remains the same, with no nodes attributed to Device 2. The same happens for Devices 3 and 4, resulting in a scenario where four devices are fully operational, but all nodes are only being executed by one device.

Currently, the decentralized mechanism cannot perform a re-calculation of scores upon detecting new devices in a decentralized manner, ignoring these events completely. Thus, our solution fails to comply with the second part of **RQ2**, providing only a partial answer as we have seen in Section 6.3.2.1 (p. 61). In this scenario, the centralized solution would perform much better, as we've seen in Section 6.3.2.3.

6.4 Replication Package

As mentioned in Section 5.1 (p. 37), the underlying platform developed by Silva *et al.* [60] completely replaced the Node-RED event-based communication with a publish-subscribe model with MQTT. This resulted in a platform that only operates in their own modified version of Node-RED. In our work, we isolated and extracted these modifications, allowing the nodes to communicate through both mediums and provide a platform that can be easily installed in vanilla Node-RED. The previous experiments allowed us to validate our work, and thus we proceeded to create a Node-RED package.

The package was created following the Node-RED packaging practices[1] and is available in the *npm* registry: https://www.npmjs.com/package/node-red-contrib-decentralized-computation. The package contains the flows presented in Section 6.1.2 (p. 50) that can be used to replicate our scenarios.

6.5 Summary

Considering the obtained results, we can then conclude that the following regarding the *desiderata* presented in Section 4.2 (p. 32): **D1** was fulfilled by the introduction of the `Task Repository` and the generation of a single script per node; **D2** was partially achieved, with a decentralized mechanism capable of handling device failures but not new appearances; and no advancements were made towards **D3**.

Our system does not fulfil **D2** completely in two ways: (1) it fails to handle the appearance of devices and tasks; and (2) it initially depends the central orchestrator for the pre-computation of the allocations. Nonetheless, the literature review made in Section 3.2 (p. 26) regarding the

[1] Node-RED Packaging, https://nodered.org/docs/creating-nodes/packaging.html

current state of decentralized task allocation methods provides a good basis and starting point for the development of the solution.

Chapter 7

Conclusions

7.1	Conclusions	75
7.2	Contributions	77
7.3	Challenges	77
7.4	Future Work	78

This chapter presents an overview of the work developed during this dissertation. Section 7.1 presents the key conclusions of this work, revisiting the hypothesis and research questions that guided our work. Then, Section 7.2 presents the main contributions of our work and Section 7.3 highlights the main challenges during the implementation. Finally, Section 7.4 analyses the possible key points to be tackled by future work.

7.1 Conclusions

The Internet of Things paradigm has been steadily growing in the past years, supported by the increasing number of IoT devices with rising computational capabilities. Following this trend, IoT systems became more complex and challenging to develop on. One of the solutions created to ease the development process was the adoption of visual programming tools in IoT that provide an intuitive and visual way of defining systems. However, the majority of these tools provide only a centralized architecture where the main component — the orchestrator and processing unit — performs all the computation on the data provided by edge devices. This architecture introduces several limitations that may hinder the system: (1) the orchestrator presents, in itself, a single point of failure (SPoF); (2) the rising capabilities of devices are being wasted; and (3) the data is unnecessarily transferred across different boundaries.

With the emergence of Fog and Edge computing, there has been a shift in computing and storage towards the edge of the network and closer to the user, thus making better use of the capabilities of each device. However, moving the computing towards the edge requires a mechanism capable of decomposing and orchestrating the IoT system into smaller units that can be allocated to the edge devices. Furthermore, it requires complex monitoring systems that can handle the failure of devices that are now responsible for a specific part of the system logic and no longer just a data source.

The systematic literature review showed that the majority of the tools used to create an IoT system currently still either follow a centralized architecture or provide a centralized solution to the challenges introduced by the Fog and Edge paradigms. The tasks of decomposing, orchestrating

and monitoring the distributed systems have been placed once again on the orchestrator, rendering him a SpoF even in an environment where all the devices are actively contributing.

In our work, we address these problems by providing a decentralized orchestration strategy with minimal dependency on a central orchestrator that offers better results than a centralized orchestration when the orchestrator fails, as initially set by our hypothesis:

> *"Given an IoT system with multiple connected heterogeneous devices, a weak decentralized orchestration mechanism offers better results than a strong centralized orchestration when faced with failures from the orchestrator."*

The experiments presented in Chapter 6 (p. 49) were made to: (1) validate our hypothesis (*cf.* Section 4.4, p. 33); (2) assess to which degree the desiderata was achieved; (3) validate the new features of the system; and (4) present the current limitations of our solution.

The *Sanity Checks* performed (*cf.* Section 6.3.1, p. 55) showed us that our system could comply with the most basic scenarios, presenting no issues in decomposing and allocating the flow to several devices.

The *Improvements* (*cf.* Section 6.3.2, p. 61) confirmed the correct behaviour of our decentralized orchestration mechanism by showing the successful re-orchestration of the devices without the interference of the Node-RED instance. The devices were able to detect failures from their peers and obtain the required tasks while only initially depending on a central orchestrator for the pre-computation of the allocations. In the previous platform, the system could not adapt itself to any changes in run-time if the central orchestrator failed. As we have shown, our system introduced different mechanisms that now allow the system to react and adapt after the orchestrator has failed — provided it was alive for enough time to complete to pre-compute the allocations — offering a functional system when a centralized orchestration could not, thus validating our hypothesis. Furthermore, these experiments also allowed us to validate the additional modifications we made to create an isolated package that anyone could easily install in the vanilla version of Node-RED (*cf.* Section 6.4, p. 73).

Finally, the *Limitations* (*cf.* Section 6.3.3, p. 68) showed the current shortcomings of our decentralized orchestration strategy. Currently, the system can only adapt to failures and not new appearances of the devices, leading to scenarios where devices that announced themselves after the flow has started will never be allocated a task. Moreover, the allocation provided by the decentralized mechanism is not as balanced as the centralized due to it being unable to account for the run-time changes in the number of allocated nodes to devices.

Considering these results, our system enables devices to obtain their assigned tasks without depending on the central orchestrator by introducing a `Task Repository` — answering **RQ1** — and a weak decentralized orchestration, with minimal dependency on the central orchestrator, capable of reallocating those tasks across devices following pre-computed allocations — partially answering **RQ2**. Additionally, it presents a failure detection mechanism based on LWT messages sent by the MQTT Server that does not depend on a central orchestrator, thus answering **RQ3**.

However, our solution fails to completely answer to **RQ2** by presenting a decentralized orchestration that can handle device failures but is unable to account for new devices appearances.

Furthermore, we can similarly conclude that *desiderata* **D1** was fulfilled, **D2** was partially achieved, and no advancement was made towards **D3** since our the solution is dependant on MQTT for the communication between devices and nodes (*cf.* Section 6.5, p. 73).

In conclusion, we validated our hypothesis by presenting a weak decentralized orchestration on which we decided to violate the second, third and fourth capabilities of a strong orchestration (*cf.* Section 4.4, p. 33). However, we recognize that there might exist other decentralized weak orchestrations that violate a different set of capabilities and offer a better solution than our own. We believe that our work presents a good basis for future work on the development of different orchestrations.

7.2 Contributions

Our work resulted in the following contributions to the software engineering state of the art and the Node-RED community:

Systematic Literature Review on decentralized VPLs in IoT: The current state of the art of decentralized architectures and mechanisms used in visual programming tools for IoT systems was analyzed. We further analyzed the current state of the art of decentralized task allocation algorithms and consensus mechanisms.

Decentralized orchestration strategy for IoT systems: We present a decentralized orchestration strategy based on the pre-computation of the allocations capable of detecting and dealing with device failures only requiring a central orchestrator to perform the pre-computation.

Reference implementation: We provide an implementation of our decentralized strategy as a Node-RED package which was published to the npm registry[1].

7.3 Challenges

Throughout the implementation of our solution, we came across multiple challenges that slowed the development process.

The process of extracting and isolating the changes from the previous platform on which we built our work introduced new and unexpected problems that required us to implement additional mechanisms to maintain correct behaviour.

The MicroPython firmware proved to be problematic due to the differences between the UNIX and ESP-32 ports. The MQTT client in both ports provided different interfaces and features, requiring two versions in order to carry out the experiments with virtual and physical devices.

[1]Node-RED package, https://www.npmjs.com/package/node-red-contrib-decentralized-computation

7.4 Future Work

The mechanisms developed for our solution currently contain limitations that have been present in Section 5.5 (p. 45) and discussed in Section 6.3.3 (p. 68). In this section, we revisit those limitations and present new ones, providing possible solutions for future work.

The current decentralized orchestration mechanism is only equipped to deal with devices failures, ignoring any new devices' appearances that might occur. This is due to the inability to recalculate the scores that the orchestrator has distributed in a decentralized fashion. One possible solution is to discard the scores created by the orchestrator and make use of an auction algorithm paired with a consensus mechanism, as seen in [15]. In the auction phase, each device could place a bid for each task (accordingly to its capabilities), and the consensus mechanism provided a way of settling the auction and deciding the winner for each task in a decentralized manner. Besides providing a mechanism to deal with the appearances of devices, this mechanism could also provide a solution for the unbalanced allocations created in the scenarios explained in Section 5.5 (p. 45) and Section 6.3.3.1 (p. 70) by taking into consideration the current number of assigned nodes each device has when placing a bid.

The communication between devices and nodes is totally dependant on the availability of the MQTT broker. This issue can be tackled by introducing an alternative communication mechanism. The firmware is already capable of communicating through HTTP, which is how each device obtains the tasks from the `Task Repository`. The firmware could be updated to detected failures from the MQTT server and default to HTTP communication (provided the nodes in Node-RED are also updated). Furthermore, given that the availability of the network itself can be problematic in IoT systems, a new firmware based on the ESP-IDF framework[2] could be developed since it already contains multiple mechanisms to handle network failures and low coverage[3].

Lastly, the work here presented can be combined with the advancements being made in fault-tolerance for IoT systems, namely within Node-RED, to improve the overall system dependability [27, 29, 28].

[2] ESP-IDF Framework, https://www.espressif.com/en/products/sdks/esp-idf
[3] ESP-IDF Networking API, https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/

References

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

[2] Sarah A. Al-Qaseemi, Hajer A. Almulhim, Maria F. Almulhim, and Saqib Rasool Chaudhry. IoT architecture challenges and issues: Lack of standardization. In *FTC 2016 - Proceedings of Future Technologies Conference*, pages 731–738. Institute of Electrical and Electronics Engineers Inc., jan 2017.

[3] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 01(1):11–33, 2004.

[4] Amazon Web Services. AWS IoT Greengrass - Amazon Web Services. Available at https://aws.amazon.com/greengrass/. Accessed in February 2021.

[5] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, oct 2010.

[6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. Understanding the Internet of Things: definition, potentials, and societal role of a fast evolving paradigm. *Ad Hoc Networks*, 56:122–140, 2017.

[7] Sanjeev Baskiyar and N. Meghanathan. A Survey of Contemporary Real-time Operating Systems. *undefined*, 2005.

[8] Martin Bauer, Ernö Kovacs, Anett Schülke, Naoko Ito, Carmen Criminisi, Laurent Walter Goix, and Massimo Valla. The context API in the OMA next generation service interface. In *2010 14th Int. Conference on Intelligence in Next Generation Networks: "Weaving Applications Into the Network Fabric", ICIN 2010 - 2nd Int. Workshop on Business Models for Mobile Platforms, BMMP 10*, 2010.

[9] Michael Blackstock and Rodger Lea. Toward a distributed data flow platform for the Web of Things (Distributed Node-RED). In *ACM International Conference Proceeding Series*, volume 08-October, pages 34–39, New York, New York, USA, oct 2014. Association for Computing Machinery.

[10] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *MCC'12 - Proceedings of the 1st ACM Mobile Cloud Computing Workshop*, pages 13–15, New York, New York, USA, 2012. ACM Press.

[11] Margaret M. Burnett and David W. McIntyre. Visual Programming. *Computer*, 28(03):14–16, mar 1995.

[12] Yinong Chen and Gennaro De Luca. VIPLE: Visual IoT/Robotics programming language environment for computer science education. In *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016*, pages 963–971. Institute of Electrical and Electronics Engineers Inc., jul 2016.

[13] Bin Cheng, Ernoe Kovacs, Atsushi Kitazawa, Kazuyuki Terasawa, Tooru Hada, and Mamoru Takeuchi. FogFlow: Orchestrating IoT services over cloud and edges. *NEC Technical Journal*, 13(1):48–53, 2018.

[14] Bin Cheng, Gürkan Solmaz, Flavio Cirillo, Ernö Kovacs, Kazuyuki Terasawa, and Atsushi Kitazawa. FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities. *IEEE Internet of Things Journal*, 5(2):696–707, apr 2018.

[15] Han-Lim Choi, Luc Brunet, and Jonathan P How. Consensus-Based Decentralized Auctions for Robust Task Allocation. *IEEE TRANSACTIONS ON ROBOTICS*, 25(4):912–926, 2009.

[16] Han Lim Choi, Andrew K. Whitten, and Jonathan P. How. Decentralized task allocation for heterogeneous teams with cooperation constraints. In *Proceedings of the 2010 American Control Conference, ACC 2010*, pages 3057–3062, 2010.

[17] Abhimanyu Chopra, Hakan Aydin, Setareh Rafatirad, and Houman Homayoun. Optimal allocation of computation and communication in an IoT network. *ACM Transactions on Design Automation of Electronic Systems*, 23(6):1–22, dec 2018.

[18] Cisco. Global Cloud Index Projects Strong Multicloud Traffic Growth | The Network.

[19] L. Coetzee, D. Oosthuizen, and Buhle Mkhize. An Analysis of CoAP as Transport in an Internet of Things Environment. *undefined*, 2018.

[20] Giuseppe Colistra, Virginia Pilloni, and Luigi Atzori. Task allocation in group of nodes in the IoT: A consensus approach. In *2014 IEEE International Conference on Communications, ICC 2014*, pages 3848–3853. IEEE Computer Society, 2014.

[21] Gennaro De Luca and Yinong Chen. Semantic Analysis of Concurrent Computing in Decentralized IoT and Robotics Applications. In *Proceedings - 2019 IEEE 14th International Symposium on Autonomous Decentralized Systems, ISADS 2019*. Institute of Electrical and Electronics Engineers Inc., apr 2019.

[22] Donato Di Paola, David Naso, and Biagio Turchiano. Consensus-based robust decentralized task assignment for heterogeneous robot networks. In *Proceedings of the American Control Conference*, pages 4711–4716. Institute of Electrical and Electronics Engineers Inc., 2011.

[23] João Pedro Dias, F. Couto, A. C. R. Paiva, and Hugo Sereno Ferreira. A brief overview of existing tools for testing the internet-of-things. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 104–109, April 2018.

[24] Joao Pedro Dias, Joao Pascoal Faria, and Hugo Sereno Ferreira. A reactive and model-based approach for developing internet-of-things systems. In *Proceedings - 2018 International Conference on the Quality of Information and Communications Technology, QUATIC 2018*, pages 276–281. Institute of Electrical and Electronics Engineers Inc., dec 2018.

[25] João Pedro Dias and Hugo Sereno Ferreira. State of the software development life-cycle for the internet-of-things. *CoRR*, abs/1811.04159, 2018.

[26] Joao Pedro Dias, Hugo Sereno Ferreira, and Tiago Boldt Sousa. Testing and deployment patterns for the internet-of-things. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, EuroPLop '19, New York, NY, USA, 2019. Association for Computing Machinery.

[27] João Pedro Dias, Bruno Lima, João Pascoal Faria, André Restivo, and Hugo Sereno Ferreira. Visual self-healing modelling for reliable internet-of-things systems. In Valeria V. Krzhizhanovskaya, Gábor Závodszky, Michael H. Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João Teixeira, editors, *Computational Science – ICCS 2020*, pages 357–370, Cham, 2020. Springer International Publishing.

[28] Joao Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Empowering visual internet-of-things mashups with self-healing capabilities. In *2021 IEEE/ACM 2nd International Workshop on Software Engineering Research Practices for the Internet of Things (SERP4IoT)*, 2021.

[29] Joao Pedro Dias, Tiago Boldt Sousa, André Restivo, and Hugo Sereno Ferreira. A pattern-language for self-healing internet-of-things systems. In *Proceedings of the 25th European Conference on Pattern Languages of Programs*, EuroPLop '20, New York, NY, USA, 2020. Association for Computing Machinery.

[30] P. J. Escamilla-Ambrosio, A. Rodríguez-Mota, E. Aguirre-Anaya, R. Acosta-Bermejo, and M. Salinas-Rosales. Distributing computing in the internet of things: Cloud, fog and edge computing overview. *Studies in Computational Intelligence*, 731:87–115, 2018.

[31] Hugo Sereno Ferreira, Tiago Boldt Sousa, and Angelo Martins. Scalable integration of multiple health sensor data for observing medical patterns. In *International Conference on Cooperative Design, Visualization and Engineering*, pages 78–84. Springer, 2012.

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

[33] Nam Ky Giang, Rodger Lea, Michael Blackstock, and Victor C.M. Leung. Fog at the edge: Experiences building an edge computing platform. *Proceedings - 2018 IEEE International Conference on Edge Computing, EDGE 2018 - Part of the 2018 IEEE World Congress on Services*, pages 9–16, 2018.

[34] Nam Ky Giang, Victor C.M. Leung, Makoto Kawano, Takuro Yonezawa, Jin Nakazawa, Rodger Lea, and Matt Broadbent. CityFlow: Exploiting Edge Computing for Large Scale Smart City Applications. In *2019 IEEE International Conference on Big Data and Smart Computing, BigComp 2019 - Proceedings*, page 8679234. Institute of Electrical and Electronics Engineers Inc., apr 2019.

[35] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. 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):1275–1296, sep 2017.

[36] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *MCC 2013 - Proceedings of the 2nd, 2013 ACM SIGCOMM Workshop on Mobile Cloud Computing*, pages 15–20, New York, New York, USA, 2013. ACM Press.

[37] National Intelligence Council. Six Technologies With Potential Impacts on US Interests Out to 2025. Technical report, National Intelligence Council (U.S.), 2008.

[38] Srdjan Krco, Boris Pokric, and Francois Carrez. Designing IoT architecture(s): A European perspective. In *2014 IEEE World Forum on Internet of Things, WF-IoT 2014*, pages 79–84. IEEE Computer Society, 2014.

[39] André Sousa Lago, João Pedro Dias, and Hugo Sereno Ferreira. Managing non-trivial internet-of-things systems with conversational assistants: A prototype and a feasibility experiment. *Journal of Computational Science*, 51:101324, 2021.

[40] Somayya Madakam, R. Ramaswamy, and Siddharth Tripathi. Internet of Things (IoT): A Literature Review. *Journal of Computer and Communications*, 03(05):164–173, 2015.

[41] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. Fog Computing: A Taxonomy, Survey and Future Directions. *Internet of Things*, 0(9789811058608):103–130, nov 2018.

[42] McKinsey. Unlocking the potential of the Internet of Things | McKinsey.

[43] Microsoft. IoT Edge | Microsoft Azure. Available at https://azure.microsoft.com/de-de/services/iot-edge/. Accessed in February 2021, 2020.

[44] Aleksandar Milinković, Stevan Milinković, and Ljubomir Lazić. Choosing the right RTOS for IoT platform. *INFOTEH-JAHORINA Vol. 14,*, 14(March 2015):7, 2016.

[45] Jozef Mocnej, Winston K.G. Seah, Adrian Pekar, and Iveta Zolotova. Decentralised IoT Architecture for Efficient Resources Utilisation. *IFAC-PapersOnLine*, 51(6):168–173, jan 2018.

[46] Muhammad Mudassar, Yanlong Zhai, Lejian Liao, and Jun Shen. A Decentralized Latency-Aware Task Allocation and Group Formation Approach with Fault Tolerance for IoT Applications. *IEEE Access*, 8:49212–49223, 2020.

[47] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, mar 1990.

[48] Duong Tung Nguyen, Long Bao Le, and Vijay K. Bhargava. A Market-Based Framework for Multi-Resource Allocation in Fog Computing. *IEEE/ACM Transactions on Networking*, 27(3):1151–1164, jun 2019.

[49] Joseph Noor, Hsiao Yun Tseng, Luis Garcia, and Mani Srivastava. DDFlow: Visualized declarative programming for heterogeneous IoT networks. In *IoTDI 2019 - Proceedings of the 2019 Internet of Things Design and Implementation*, pages 172–177, New York, NY, USA, apr 2019. Association for Computing Machinery, Inc.

[50] Samodha Pallewatta, Vassilis Kostakos, and Rajkumar Buyya. Microservices-based IoT application placement within heterogeneous and resource constrained fog computing environments. In *UCC 2019 - Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 71–81, New York, NY, USA, dec 2019. Association for Computing Machinery, Inc.

[51] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. In *Information and Software Technology*, volume 64, pages 1–18. Elsevier, aug 2015.

[52] D. Pinto, João Pedro Dias, and Hugo Sereno Ferreira. Dynamic allocation of serverless functions in iot environments. In *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 1–8, October 2018.

[53] Sandra Prescher, Alan K Bourke, Friedrich Koehler, Angelo Martins, Hugo Sereno Ferreira, Tiago Boldt Sousa, Rui Nuno Castro, António Santos, Marc Torrent, Sergi Gomis, et al. Ubiquitous ambient assisted living solution to promote safer independent living in older adults suffering from co-morbidity. In *2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 5118–5121. IEEE, 2012.

[54] Carlo Puliafito, Enzo Mingozzi, Francesco Longo, Antonio Puliafito, and Omer Rana. Fog computing for the Internet of Things: A survey. *ACM Transactions on Internet Technology*, 19(2):1–41, apr 2019.

[55] RabbitMQ. RabbitMQ – Messaging that just works. Available at https://www.rabbitmq.com/. Accessed in February 2021, 2009.

[56] Antonio Ramadas, Gil Domingues, Joao Pedro Dias, Ademar Aguiar, and Hugo Sereno Ferreira. Patterns for Things that Fail. In *Proceedings of the 24th Conference on Pattern Languages of Programs*, PLoP '17. ACM - Association for Computing Machinery, 2017.

[57] Andreas Reiter, Bernd Prunster, and Thomas Zefferer. Hybrid Mobile Edge Computing: Unleashing the Full Potential of Edge Computing in Mobile Device Use Cases. In *Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017*, pages 935–944. Institute of Electrical and Electronics Engineers Inc., jul 2017.

[58] Jan Seeger, Rohit Arunrao Deshmukh, and Arne Broring. Running distributed and dynamic IoT choreographies. In *2018 Global Internet of Things Summit, GIoTS 2018*. Institute of Electrical and Electronics Engineers Inc., nov 2018.

[59] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, oct 2016.

[60] Margarida Silva, Joao Pedro Dias, Andre Restivo, and Hugo Sereno Ferreira. Visually-defined real-time orchestration of iot systems. In *Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, MobiQuitous '20. Association for Computing Machinery, 2020.

[61] Margarida Silva, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. A review on visual programming for distributed computation in iot. In *Proceedings of the 21st International Conference on Computational Science (ICCS)*. Springer, 2021.

[62] Christopher Simpkin, Ian Taylor, Daniel Harborne, Graham Bent, Alun Preece, and Ragu K. Ganti. Dynamic Distributed Orchestration of Node-RED IoT Workflows Using a Vector Symbolic Architecture. In *Proceedings of WORKS 2018: 13th Workshop on Workflows in Support of Large-Scale Science, Held in conjunction with SC 2018: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 52–63. Institute of Electrical and Electronics Engineers Inc., feb 2019.

[63] Danny Soares, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Programming iot-spaces: A user-survey on home automation rules. In *Proceedings of the 21st International Conference on Computational Science (ICCS)*. Springer, 2021.

[64] Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. Patterns for software orchestration on the cloud. In *Proceedings of the 22nd Conference on Pattern Languages of Programs*, pages 1–12, 2015.

[65] Statista. • Number of IoT devices 2015-2025 | Statista. Available at https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/. Accessed in February 2021.

[66] D. Torres, J. P. Dias, A. Restivo, and H. S. Ferreira. Real-time feedback in node-red for iot development: An empirical study. In *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–8, 2020.

[67] Krishnamurthy Vidyasankar. Distributing computations in fog architectures. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 3–8, New York, NY, USA, jul 2018. Association for Computing Machinery.

[68] Andrew K. Whitten, Han Lim Choi, Luke B. Johnson, and Jonathan P. How. Decentralized task allocation with coupled constraints in complex missions. In *Proceedings of the American Control Conference*, pages 1642–1649. Institute of Electrical and Electronics Engineers Inc., 2011.

[69] Yang Yang. Multi-tier computing networks for intelligent IoT, jan 2019.

[70] Takuro Yonezawa, Tomotaka Ito, Jin Nakazawa, and Hideyuki Tokuda. SOXFire: A Universal Sensor Network System for Sharing Social Big Sensor Data in Smart Cities. In *Proceedings of the 2nd International Workshop on Smart, SmartCities 2016*. Association for Computing Machinery, Inc, dec 2016.