FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Dynamic Real-Time IoT Orchestration

**Tiago José Viana Fragoso**


Mestrado Integrado em Engenharia Informática e Computação

Supervisor: André Restivo, Assistant Professor

Second Supervisor: Hugo Sereno, Assistant Professor

July 21, 2021

# Dynamic Real-Time IoT Orchestration

**Tiago José Viana Fragoso**

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Gil Gonçalves
External Examiner: Prof. Filipe Sousa
Supervisor: Prof. André Restivo

July 21, 2021

# Abstract

Internet-of-Things (IoT) is a network of heterogeneous Internet-connected devices and has been rapidly growing in device numbers, spreading across several application domains. As the number of devices grows, so does the data produced by them. Consequently, traditional IoT architectures where data produced by fog/edge devices (*i.e.*, sensors, actuators) is processed on the cloud become less suitable to handle the large amounts of data, incurring in bandwidth limitations, higher latency, and even privacy concerns as data may cross political boundaries. In the interest of mitigating these issues and leveraging the ever-growing computational capabilities at the edge of the network, the fog and edge computing paradigms have developed.

Fog/edge orchestration platforms for IoT have emerged as an implementation of the fog/edge computing paradigms and provide application partitioning and dynamic allocation of tasks to devices. As devices closer to the edge are, generally, more failure-prone, some parts of the application may be assigned to a device that fails, causing service disruption. Thus, fog/edge orchestration platforms must ensure the system's dependability, usually through fault-tolerance/self-healing mechanisms that aim to provide correct service in the presence of faults. However, available systems often overlook this need — either only offering a mechanism that re-distributes tasks when device failure is detected or offering no fault-tolerance at all.

In a previous work, a Node-RED-based real-time orchestration system for IoT was developed, NoRDOr. However, some limitations hinder the systems' resilience, such as (1) limited self-healing mechanisms that are not suitable to handle device instability and memory constraints, (2) (re-)orchestration deploys the entire system, not taking into account the current state, leading to greater orchestration overhead, and (3) certain task configurations lead the system to produce unbalanced assignments, which, in turn, may lead to considerable service disruption in case of device failure.

In this work, we address these limitations in order to achieve greater resilience by (1) employing probing mechanisms to leverage real-time information from devices, such as stability and resources, to produce orchestrations less susceptible to device failure, (2) altering the greedy algorithm to mitigate its task-order bias, thus generating balanced assignments, and (3) leveraging the current orchestration to produce minimal disturbance to the system when re-orchestrating, thus allowing it to provide correct service for a greater amount of time which may be crucial to ensure safety in critical systems.

To validate our solution, a real-time orchestration simulator was developed to easily create and measure reproducible scenarios that mimic real-world IoT orchestration ones, tightening the validation cycle of the proposed improvements. Then, the solution was ported to NoRDOr and tested in virtual and physical IoT devices, using smart-home scenarios, where several metrics were collected to evaluate the system's resilience. We concluded that leveraging runtime information from the devices would lead to a system better suited to provide correct service in the presence of faults.

**Keywords**: IoT, Edge computing, Node-RED, Self-healing, Fault-tolerance

# Resumo

A Internet-of-Things (IoT) é uma rede de dispositivos heterogéneos ligados à Internet e está a crescer rapidamente em número de dispositivos, alastrando-se a diversos domínios. Com o crescimento do número de dispositivos, a quantidade de dados por eles produzida também cresce. Consequentemente, as arquiteturas tradicionais de IoT, onde a maioria dos dados produzidos por dispositivos *fog*/*edge* são processadas na *cloud*, tornam-se menos adequadas para processar grandes quantidades de dados, devido a limitações de largura de banda, maior latência e transferências de dados entre fronteiras políticas. De modo a mitigar estes problemas e aproveitar o poder computacional disponível na periferia da rede, sugiram os paradigmas de *fog* e *edge computing*.

As plataformas de orquestração em *fog*/*edge* emergem como implementações dos paradigmas *fog*/*edge computing*, oferecendo partição de aplicações em tarefas e alocação dinâmica dessas tarefas aos dispositivos. Tais dispositivos na periferia da rede são, geralmente, mais suscetíveis a falhas. Assim, é possível que parte da aplicação estejam alocadas a dispositivos que falham, causando uma interrupção do bom funcionamento da aplicação. Deste modo, as plataformas de orquestração em *fog*/*edge* devem garantir a *dependability* do sistema, normalmente através de mecanismos de *self-healing*, que visam manter o funcionamento do sistema na presença de falhas. No entanto, as plataformas existentes negligenciam esta necessidade — apenas oferecendo um mecanismo de redistribuição das tarefas quando uma falha é detetada ou até nenhum mecanismo.

Num trabalho anterior, foi desenvolvida uma plataforma de orquestração para IoT em tempo-real, baseada em Node-RED, NoRDOr. Contudo, existem algumas limitações que prejudicam a resiliência do sistema, tal como (1) mecanismos limitados de *self-healing* que não são adequados para lidar com a instabilidade e restrições de memória dos dispositivos, (2) a re-orquestração implica re-alocar todas as tarefas do sistema, ignorando o estado atual, o que leva a um maior *overhead* de orquestração, e (3) certas configurações levam a que o sistema produza alocações desbalanceadas, que podem causar interrupções no funcionamento do sistema em caso de falha de um dispositivo.

Este trabalho foca-se em colmatar estas limitações, de modo a melhorar a resiliência do sistema, através de (1) utilização de mecanismos de *probing* para tirar partido da informação dos dispositivos em tempo-real, como estabilidade e recursos, e realizar orquestrações menos suscetíveis às falhas dos dispositivos, (2) alterações ao algoritmo *greedy* de forma a mitigar o impacto da ordem de processamento das tarefas, resultando em alocações mais balanceadas, e (3) utilizar a alocação atual de modo a reduzir as perturbações introduzidas no sistema aquando de uma re-orquestração, permitindo que o bom funcionamento seja garantido durante mais tempo, o que pode ser crucial para garantir a segurança de sistemas críticos.

Com o intuito de validar a solução proposta, um simulador de orquestração em tempo-real foi desenvolvido para criar e medir cenários de forma reproduzível, comprimindo o ciclo de validação. De seguida, a solução foi aplicada ao sistema NoRDOr e testada em dispositivos IoT virtuais e físicos, usando cenários de *smart-home*, onde várias métricas foram recolhidas para avaliar a resiliência do sistema. Foi possível concluir que as melhorias propostas, que utilizam informação dos dispositivos em *runtime*, resultam num sistema mais capaz de garantir o seu bom funcionamento na presença de falhas.

**Keywords**: IoT, Edge computing, Node-RED, Self-healing, Fault-tolerance

# Acknowledgements

Firstly, I would like to thank the supervisors of this dissertation, André Restivo and Hugo Sereno Ferreira, for providing guidance and insights, motivating me to produce the best work possible. Additionally, I would also like to thank João Pedro Dias, who was present throughout the whole process, and provided valuable advice and feedback.

Then, I would like to thank my friends and colleagues in *Road* and *LabES no Telegrama* that relentlessly heard my frustrations and echoed theirs, whilst offering advice and cheering me up through jokes and repeated questioning as to why we are writing a dissertation in the first place. A special thanks to Pedro Costa, that produced a work closely related to mine, for his availability and willingness to pair with me in finding solutions to common problems.

Finally, I would like to thank Beatriz Mendes, my girlfriend, for always being there for me, but especially for accompanying me in procrastinating during this dissertation.

Tiago Fragoso

*"Sent from my iPhone"*

Randall Munroe, in XKCD 1942

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| API | Application Programming Interface |
| CPU | Central Processing Unit |
| HTTP | Hypertext Transfer Protocol |
| IoT | Internet-of-Things |
| M2M | Machine-to-Machine |
| MQTT | Message Queuing Telemetry Transport |
| MTBF | Mean Time Between Failure |
| MSC | Minimum Set (of) Changes |
| RAM | Random Access Memory |
| VPL | Visual Programming Language |

# Chapter 1

# Introduction



This chapter describes the motivation and problem at hand, as well as the goals this project aims to achieve. Section 1.1 introduces the context of this work, while Section 1.2 (p. 2) introduces its motivation. Then, Section 1.3 (p. 2) defines the problem under study, and Section 1.4 (p. 2) details the goals this work aims to achieve and how they are evaluated. Finally, the document structure is presented in Section 1.5 (p. 4).

## 1.1 Context

Internet-of-Things (IoT) is a network of heterogeneous [6] Internet-connected devices that has applications in, for example, smart transportation, smart cities [32] and smart homes [5, 49]. The number of IoT device connections is expected to grow more than twofold, from 2018 to 2023, to almost 15 billion, according to Cisco [12]. With the increase in computational capabilities and the sheer number of IoT devices scattered across different applications, the amounts of data to be exchanged with the Cloud can incur in bandwidth limitations, higher latency, and privacy concerns. Further, as IoT embraces mission-critical domains (*e.g.*, Ambient Assisted Living), the dependability of theses systems becomes crucial [38, 23].

In order to mitigate this problem, two paradigms have emerged: Fog and Edge computing. Both aim to bring cloud computing benefits closer to the edge of the network [29]. The first aims to bring cloud capabilities to devices closer to the edge (*e.g.*, gateways) and provide computing and storage at an intermediate level between the cloud and the edge, reducing the dependency on cloud [50]. The latter aims to leverage edge devices' computational capabilities. This is motivated by the increase in computational and communication capabilities at the edge, as well as the change in the data flow. Edge devices, traditionally data consumers, are now both data consumers and data producers [44].

Although moving computation to the edge has advantages (*e.g.*, lower latency, privacy), it also comes with challenges as edge devices are highly heterogeneous, computationally constrained, and failure-prone. Fog/edge orchestration platforms (*cf.* Section 3.2, p. 10) have been developed as a

way to partition applications and distribute tasks through edge devices based on capabilities and constraints of each task and device, abstracting the edge device's run-time environment either by developing custom frameworks or using OS-level virtualization tools [47].

## 1.2 Motivation

As computation moves towards the edge of the network with the emergence of fog and edge computing paradigms, a pressing need for dependability concerns arises (*cf.* Section 2.2, p. 7).

Edge computing orchestration platforms partition large applications into tasks or sets of tasks that are then dynamically allocated to the connected devices to be executed in a distributed manner. Some of these devices may be assigned a crucial part of the application, such as processing data from a *door sensor* for a smart home's alarm system.

Particularly, when working with edge devices, reliability is not a given due to fragmentation in the IoT field (*i.e.*, high heterogeneity, different protocols, and standards), resource constraints, and hardware failures, meaning that, at a certain point, a subset of the application may be deployed in an unstable or faulty device [14, 4].

Hence, mechanisms must be employed to ensure the dependability of the system. In IoT environments, self-healing mechanisms are particularly suitable as they minimize the impact to the end-user by requiring no user interaction. In these platforms, it can be achieved by removing tasks from unstable or faulty devices and moving them to healthier devices in the system.

## 1.3 Problem

Although necessary, fault-tolerance is considerably limited in the available fog/edge orchestration platforms. Some offer a self-healing mechanism that detects device failure and re-distributes the tasks to other devices, while others do not provide any fault-tolerance mechanisms. Given the possible instability and resource constraints of edge devices, failing to employ comprehensive mechanisms that aim to ensure the system's dependability in such dynamic systems, can result in downtime.

Additionally, specific issues were found in a real-time orchestration system for IoT, NoRDOr [45, 46], that limit its ability to provide correct service in the presence of faults. In this work, we tackle these issues in pursuit of achieving a more resilient system.

## 1.4 General goals

The main goal of this dissertation is to improve the resilience of a real-time orchestration platform for IoT systems at the edge. In order to do this, we extend a previously existing system, NoRDOr [45, 46], that is comprised of (1) a modified version of Node-RED, enhanced with the ability to dynamically partition and allocate computational tasks to edge devices, and (2) a custom MicroPython framework that enables general-purpose code executing on edge devices. NoRDOr

author's choice for Node-RED as a base work was motivated by being open-source, its generalized popularity, and the widely adoption by the research community as a target/base for other experiments [8, 14, 52, 30].

To achieve greater resilience, the system should:

**Generate balanced assignments** so that the computation is distributed and device failure has a lower impact on the system.

To do this, we explore the possibility of processing the computational tasks to be assigned to the currently devices in a specific order, such that the algorithm can produce a balanced solution.

**Leverage device stability** so that the system can produce assignments less susceptible to device failure.

*Probing* mechanisms are introduced to allow the system to get real-time information from the devices and compute stability metrics derived from their stability or the stability of their assignment. Collecting this data makes it possible to leverage it on orchestration, generating assignments that make the system less susceptible to experience downtime due to device failure.

**Leverage current state upon re-orchestration** so that the set of changes to be deployed is reduced in order to minimize orchestration overhead.

In order to reduce the overhead caused by the re-assigning a new set of computational tasks to all devices, we minimize the set of changes that need to be deployed to devices on re-orchestration by focusing on tasks that were assigned to failed devices and attempt to deploy them while affecting the least number of healthy devices.

**Estimate memory limitations** so that the system can generate assignments within the memory limitations of each device without causing downtime.

Memory is a limited and dynamic resource at the edge of the network. Hence, edge devices report their memory resources when connecting to the system, allowing to estimate if a given assignment is within the memory limitations of each device.

In the interest of validating the proposed solutions, the improvements are iteratively tested on an *Orchestration simulator* before being extensively compared to the original system in a set of experiments comprised of virtual and physical IoT devices, where fault injection is used to simulate real-world scenarios, in pursuit of finding solid answers to each of the research questions of this dissertation.

## 1.5 Document structure

This chapter introduced the context and motivation for this work, as well as the problem under study and the general goals. This document is composed of eight more chapters, structured as follow:

- Chapter 2 (p. 5), **Background**, introduces key concepts and their relationships, which are required to comprehend this work fully;
- Chapter 3 (p. 9), **State of the Art**, describes the current state of the art on the topic of orchestration of distributed IoT systems at the fog/edge and self-healing mechanisms in IoT;
- Chapter 4 (p. 25), **Problem Statement**, presents the current issues this dissertation aims to tackle, the requirements a solution must fulfill, and the research questions it aims to answer;
- Chapter 5 (p. 29), **Orchestration simulator**, describes the scope, implementation details, and simulation lifecycle of the developed simulation tool;
- Chapter 6 (p. 37), **Orchestration strategies**, outlines the development methodology and the implementation details of the several iterations produced in order to reach the end solution;
- Chapter 7 (p. 61), **Reference implementation**, details the implementation of the developed improvements to the base system, as well as its limitations;
- Chapter 8 (p. 67), **Evaluation**, analyzes the results of the proposed system against the original system in a set of experimental tasks designed to highlight the issues under study and derives answers to the research questions of this work;
- Chapter 9 (p. 99), **Conclusions**, presents a summary of the developed work, outlines difficulties that slowed down development, and describes future research directions.

# Chapter 2

# Background



This chapter introduces key concepts and their relationships, which are required to comprehend this work fully. Section 2.1 defines Internet-of-Things (IoT) and IoT tiers, namely Cloud, Fog and Edge. Then, Section 2.2 (p. 7) presents the concepts of Fault-tolerance and Self-healing.

## 2.1 Internet-of-Things

As Whitmore *et al*. [53] pointed out, there is no universal definition for the Internet-of-Things (IoT), but at its core, it *"is a paradigm where everyday objects can be equipped with identifying, sensing, networking, and processing capabilities that will allow them to communicate with one another and with other devices and services over the Internet to accomplish some objective"*.

The IoT network is comprised of varied devices from everyday objects to sensors and actuators, powered by a diverse range of underlying technologies, thus creating a network of interconnected heterogeneous devices [15]. Its applications range from the transportation and logistics domain to healthcare to smart home/office environments [6].

Cloud computing solutions (*e.g.*, Amazon Web Services (AWS) [1] and Google Cloud Platform [2]) provide centralized, scalable, and high-performance computation capabilities that were suitable for storing and processing IoT data [21, 32]. However, as the number of devices increases [12], so does the amount of data produced and the geographical distribution, and it becomes increasingly difficult, costly, or even illegal to centralized data storage and processing in the cloud [21, 36]. Moreover, the emergence of delay-sensitive IoT applications motivates the division of IoT systems into three tiers [14, 54, 15]: **Cloud**, **Fog**, and **Edge**.

**Cloud:** High availability, scalability, and performance but also high latency. Composed of data centers and servers.

**Fog:** Lower performance than the cloud, higher heterogeneity, and geographical distribution but lower latency. Located in between cloud and edge and composed of gateways or on-premise servers.

**Edge:** High heterogeneity, computationally limited, and considerably low latency. Composed of sensors, actuators, embedded systems.

New paradigms have emerged in response to the needs of modern IoT networks, fuelled by the increase of computation capabilities of edge devices and their low latency: Fog computing and Edge computing.

### 2.1.1 Fog computing

Fog computing was introduced in 2012 [9] as an extension of cloud computing closer to the edge. This paradigm is still being extensively studied, and, thus, there are different definitions and approaches to its architecture [33]. However, authors converge on fog computing's essential characteristics [9, 29]: low latency, geographical distribution, heterogeneity, interoperability, and real-time processing.

The main goal of fog computing is bringing cloud computing benefits to the edge network and, thereby allowing for low-latency data management and processing. Some approaches employ high-performance servers as fog nodes, while others make use of networking devices (*e.g.*, gateways) that are closer to the edge devices. As a result, fog computing can be applied to different environments.

Although there are many benefits to Fog computing when compared to Cloud computing for IoT networks, it also faces some challenges [33]. Resource allocation is a particularly challenging one since fog nodes are distributed and heterogeneous; thus, efficient resource provisioning and allocation needs to be performed. Moreover, due to the heterogeneity of the edge network, not all devices are prepared to perform general-purpose computation.

### 2.1.2 Edge computing

Edge computing takes advantage of the computational resources of edge devices to process data. Driven by the increase in the numbers of devices and, consequently, the amount of data to be exchanged with the cloud, this paradigm is a distributed architecture that aims to reduce latency and response time [28].

As more devices become connected at the edge, different dynamics appear as well. Particularly, devices at the edge are moving from being data consumers to being both data consumers and producers [44], which increases the amount of data being sent to the cloud. Furthermore, since most data will be consumed at the edge, it does not need to be stored in the cloud.

Privacy concerns may also arise from cloud computing, where data sometimes travels through political boundaries, which may incur illegalities. This can be mitigated by using an edge computing architecture.

However, there are also some drawbacks to this paradigm, namely the computational constraints and heterogeneity of edge devices are challenging for developers. Moreover, reliability at the edge is limited. It may be hard to detect failures and the underlying cause since hardware issues may be involved.

## 2.2 Self-healing & fault-tolerance

In 2004, Avizienis *et al*. [7] sought to give clear definitions of relevant terms in the realm of dependability, *i.e*., *"the ability to deliver service that can justifiably be trusted"*. Some of the key definitions relevant to this work are introduced below.

**Correct service:** *"service implements the system function"*

**(Service) Failure:** *"an event that occurs when the delivered service deviates from correct service"*

**Error:** *"the part of the total state of the system that may lead to its subsequent service failure"*

**Fault:** the *"hypothesized cause of an error"*

**Fault-tolerance/Self-healing/Resilience:** *"avoid service failures in the presence of faults"*

**Safety:** *"absence of catastrophic consequences on the user(s) and the environment"*

The author then splits fault-tolerance into two components: (1) error detection and (2) recovery. The former occurs earlier than the latter, and it *"identifies the presence of an error"*. The recovery component is responsible for *"transforming a state that contains one or more errors (...) into a state without errors"*. Itself can also be subdivided into two components: (1) error handling, which *"eliminates errors from the system state"*, and (2) fault handling, which *"prevents faults from being activated again"*.

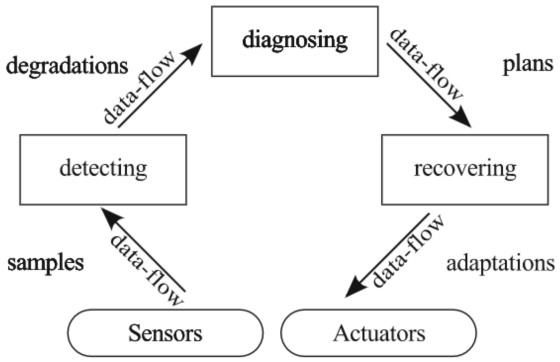


Figure 2.1: Self-healing loop [39].

Moreover, Ganek *et al*. [24] defined a **self-healing system** as one that is capable of maintaining correct service while detecting and isolating failed components and fixing them or introducing replacement components into the system. The authors also highlighted the need for the system to predict eventual problems and take action accordingly, such that the fault does not lead to service failure.

Psaier *et al*. [39] propose a three-stage self-healing loop (*cf.* Figure 2.1) comprised of (1) *detecting*, (2) *diagnosing*, and (3) *recovering* phases. In the *detecting* phase, the input is analyzed,

and detected errors are reported to the next phase. The *diagnosing* phase is responsible for finding the fault for a given error and deciding on the recovery policy to apply. The *recovering* phase is responsible for applying the recovery policy defined previously while ensuring that system constraints are respected. The three phases are thus connected by data flow.

# Chapter 3

# State of the Art



This chapter describes the state of the art for the topics of interest to this work: (1) distributed fog/edge orchestration, and (2) self-healing/fault-tolerance in IoT. First, the methodology is presented in Section 3.1. In Section 3.2 (p. 10), results for distributed fog/edge orchestration are presented and analysed. The same is done for self-healing/fault-tolerance in IoT in Section 3.3 (p. 19).

## 3.1 Methodology

An *ad hoc* methodology was used for this literature review. Several databases (*cf.* Section 3.1.2) were queried with the relevant terms for each research question (*cf.* Section 3.1.1) and the results were sorted and filtered using criteria such as the number of citation and the publication year. This was done using an iterative process (*cf.* Section 3.1.3).

### 3.1.1 Survey research questions

The following survey research questions were developed to guide this literature review:

**SRQ1:** *What solutions exist to orchestrate distributed IoT systems at the edge?*

**SRQ2:** *What self-healing/fault-tolerance mechanisms are employed in distributed IoT systems?*

### 3.1.2 Databases

Three of the most relevant digital libraries in Computer Science were chosen as the main source of the literature analyzed in this work: ACM DL, IEEE Xplore, and Scopus.

### 3.1.3 Process

An iterative process was adopted in order to increase the amount of relevant material examined. To do this, lessons learned from each iteration are fed into the next (*e.g.*, synonyms) so that the search space is as wide as possible while staying inside the scope. A brief overview of the process used is shown below.

1. Translate each research question into a query;
2. Submit query into library (sort based on citations, filter based on year);
3. Identify each result as relevant or not based on title, abstract, and conclusions;
4. Add relevant results to read-list if not present;
5. Read articles in the list, expanding select references;
6. Go back to 2., refining search terms based on previous results (*e.g.*, synonyms, missed terms).

## 3.2 Distributed fog/edge orchestration

Using the methodology described in Section 3.1 (p. 9) to attempt to respond to SQR1 (*cf.* Section 3.1.1, p. 9), the following query was elaborated:

```
(iot OR  internet-of-things OR "internet-of-things") AND (edge OR fog OR
    distributed OR decentralized OR decentralised)
```

From the pool of results, the following are considered to be the most relevant for this work.

**DNR**

In 2014, Blackstock *et al.* [8] and Giang *et al.* [25, 26] in subsequent works proposed an open-source edge computing platform – Distributed Node-RED (DNR) – that extends Node-RED to deploy application flows on devices ranging from the edge to the Cloud. The latest iteration of this software (DNR v3 [26]) was built with large-scale applications in mind, allowing developers to set context-aware constraints (*e.g.*, the physical location of the device) and making use of an external coordination layer for communication management, thereby supporting inter-component constraints. This is particularly important essential when building more complex systems (*e.g.*, smart cities).

The authors [26] explain the iterative process on which the system was built, outlining the challenges and problems motivating each new version and how they solved them.

First [8], Node-RED nodes were extended to support a *device Id*, specifying the device where each node should run. To allow for inter-device communication, each device subscribes and publishes the data in the corresponding MQTT topic of each node that is running in an external device.

In the second version [25], scalability issues of the previous implementation are addressed – namely, (1) the simplistic mapping of nodes to devices, and (2) the lack of an automated flow deployment system. To improve (1), the authors extended nodes with a *constraint* property that defines which device can run each node based on several aspects (*e.g.*, CPU

and memory resources available, physical location, etc.). Furthermore, the concept of *wire cardinality* is introduced as a way to clearly define the mapping of inputs and outputs of each node which is crucial in large-scale environments where nodes can be deployed in different devices at the same time. The naming of each MQTT topic is then based on the *wire cardinality* of each node. In order to solve (2), each device subscribes to an MQTT topic which contains the current state of the application flow – called *main flow* and, upon receiving a new *main flow*, subsequently parses it and decides if it should run a given sub-flow based on its constraints and the device's own capabilities.

In the latest version [26], another set of scalability issues are addressed – (1) as the system grows, individual devices might not be able to decide on their own if they should run a node, and (2) communication between different nodes needs to be coordinated. This led to the introduction of a coordination layer (*cf.* Figure 3.1, p. 12), composed of 3 components: the *Global coordinator*, which is a centralized instance with system-wide knowledge and, in each device, a *Flow coordinator*, responsible for synchronizing device context with the *Global coordinator* and a set of *Coordinator nodes* that coordinate node communication. A seen in Figure 3.1 (p. 12), there are four possible coordination states that these nodes can be in (1) NORMAL, where they pass the data through its outputs (requires that input and output nodes are deployed in the same device), (2) DROP, where the data is not forwarded but rather dropped (useful to avoid processing of duplicate data), (3) FETCH_FORWARD, where the node gets the input from an external device, and (4) RECEIVE_REDIRECT, where it forwards data to an external device.

The proposed solution is well suited for large-scale systems where geographic dispersion is high and has an important part in the computation distribution. However, requiring all nodes to run Node-RED limits the number of supported edge devices. Moreover, the authors fail to mention the system's fault-tolerance. This might be due to the *participatory model* followed in their approach, as stated in [27]. Using this approach, task completion is not guaranteed as devices contribute resources with some inconsistency. According to the authors, this approach is more suitable to highly dynamic systems compared to guaranteed task completion because the latter implies constant monitoring and orchestration of devices and can lead to considerable coordination overhead.

**DDFlow**

Noor *et al*. [34] introduced DDFlow – an extension to Node-RED runtime engine that allows for dynamic scaling and real-time adaptation in distributed IoT networks. The system is managed by a distributed coordinator responsible for mapping tasks to devices and monitoring the network for significant changes or device failure, which result in a re-orchestration. The aim of this solution is to provide developers with an extensible abstraction capable of producing a high-quality distributed application while minimizing end-to-end latency.

The authors extend Node-RED primitives such as *Nodes* and *Wires* in order to support dynamic scalability based on resources and constraints and communication cardinality. In

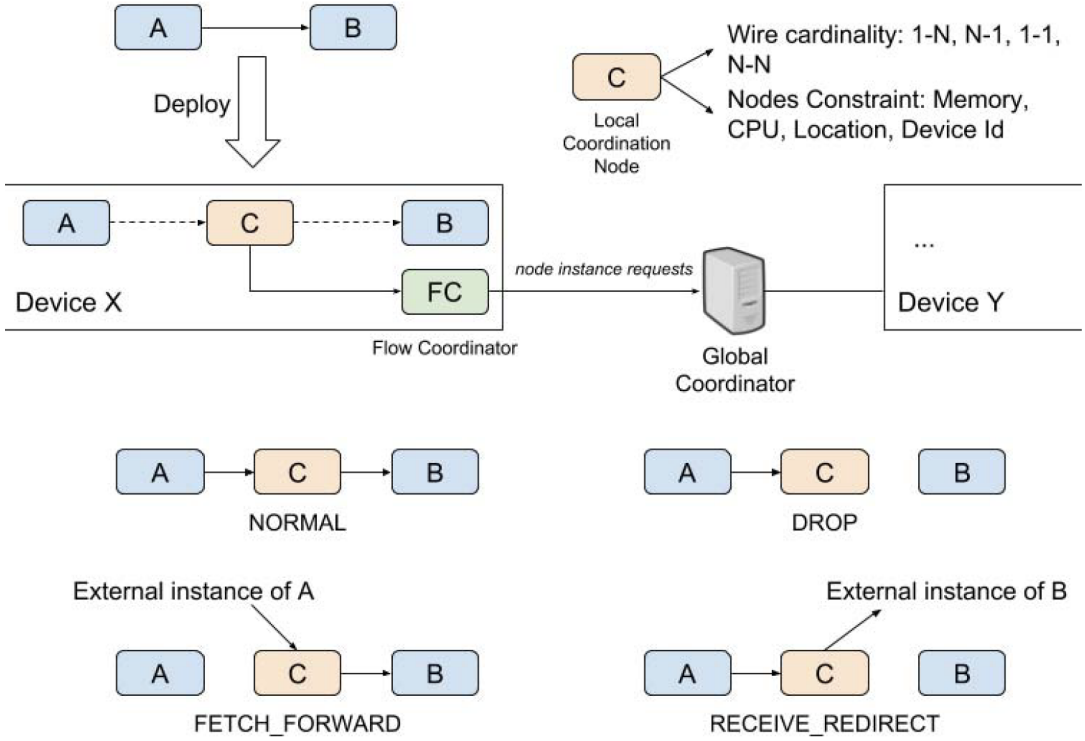


Figure 3.1: Exogenous Coordination in DNR [26].

DDFlow, a *Node* represents a task that is deployed to one or more devices in the network. Its (optional) inputs and outputs are in the form of a key-value dictionary and contain application data as well as metadata. Moreover, *Nodes* can be constrained either by *Region* (only selecting devices inside one or more physical locations) or *Device* (only selecting devices contained in a subset of all network devices). During deployment, the system will fulfill these constraints dynamically (e.g., when devices are mobile and change location frequently). Another key concept is *Wires*, which transport information from the output of a Node to the input of another, also in a key-value format. In order to support multiple deployments of the same task in different nodes, *Wires* can be of three different types: (1) Stream, where one input is mapped to one output, (2) Broadcast, where one output feeds multiple inputs, and (3) Unite, where multiple outputs feed one input.

As seen in Figure 3.2 (p. 13), DDFlow consists of two main components: *Coordinator* and *Device*. The *Coordinator* is a web server that may be replicated among several devices and manages the network. It's composed of three main components: (1) a *Web Interface* where developers can set up their applications, (2) a *Deployment Manager* which communicates with each device to deliver their task assignment, and (3) a *Placement Solver* that assigns nodes to devices. Each *Device* offers a set of *Services* – *i.e.*, implementations of each *Node* in DDFlow, which are communicated to the *Coordinator* by each device's *Device Manager* along with the current resources information (*e.g.*, CPU load) and capabilities. The *Device*

*Manager* is a web server that is also responsible for intra and inter-device communication.

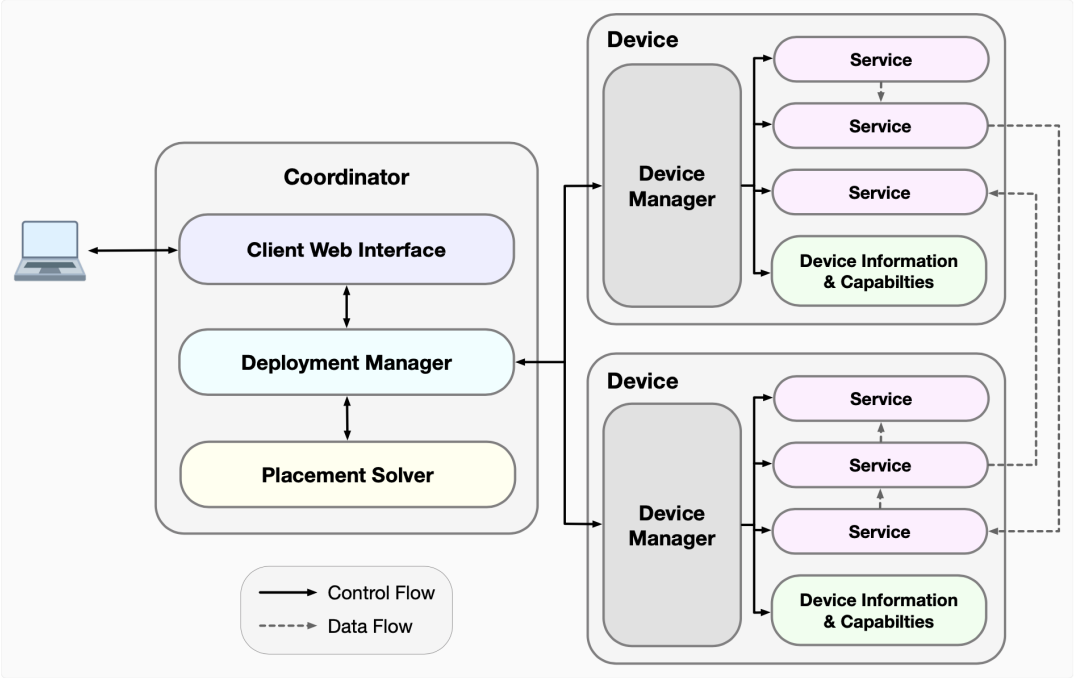


Figure 3.2: DDFlow component architecture [34].

To achieve dynamic adaptation and recovery, the *Coordinator* monitors the network periodically for environmental changes (*e.g.*, device failure, device overload, etc.). Upon detection of an error, the *Coordinator* preventively re-orchestrates the network by formulating the assignment as a linear programming problem that minimizes end-to-end latency and solving it while satisfying the application constraints (*e.g.*, physical location of devices) as well as network constraints (*e.g.*, connectivity between devices). The assignment is then propagated by the *Deployment Manager* to each device's *Device Manager*. Furthermore, the system can also adapt to network-level failures by switching from TCP/IP to Wi-Fi ad hoc mode.

The authors evaluated the system against device and network failures in a scenario consisting of image capture and processing. When compared to a static system (*i.e.*, not self-adapting), DDFlow performed significantly better, showcasing fault-tolerance by (1) preventive network re-orchestration (*i.e.*, before device failure), and (2) network adaptation upon wireless access point failure. In both cases, the static deployment crashed upon device or network failure.

The proposed solution would be a rather complete one if it were not for the fact that it is not open-source, which makes it impossible to confirm their findings. It is also important to mention that each device must have an implementation of tasks it can compute, which can be cumbersome in such heterogeneous IoT networks.

**FogFlow**

Cheng *et al*. [10, 11] proposed an open-source distributed execution framework, FogFlow, with the goal of dynamically orchestrating IoT networks. The framework is based on the data-flow programming model but requires developers to create dockerized applications called *operators*, representing the implementation of a certain task. These applications must be based on the Next Generation Service Interfaces (NGSI) standard, which defines a data model and communication interface for context-based communication. The former is composed of the entity description (*e.g.*, ID and type) and metadata (*e.g.*, location). The latter defines a lightweight interface with context-aware publish, query, and subscribe functionality (*e.g.*, query by location). In FogFlow [10], a service performed by an IoT device can be represented by (1) a *service topology* (*i.e.*, multiple operators linked together) which is deployed when consumers request data, or (2) a *fog function*, a more simplistic *service topology* with only one *operator* which is triggered as soon as input data becomes available.

As illustrated in Figure 3.3 (p. 15), the system is comprised of three different types of heterogeneous resources: (1) service management, (2) context management, and (3) data processing. The service management division is composed of: (1) the task designer, a Web-based interface that allows developers to monitor and develop flow-based applications to be deployed, (2) the docker image repository, where developers can submit their specific service implementations, and (3) the topology manager (TM) which is responsible for mapping the application flow to a concrete system orchestration. The context management division contains: (1) a set of IoT Brokers which communicate with workers providing context updates, queries, and subscriptions, (2) a centralized IoT Discovery component, responsible for handling the registration and discovery of context entities, and (3) a Federated Broker which extends the IoT Broker and serves a the bridge to exchange context information between different domains (*e.g.*, different smart cities). The data processing division includes a set of workers that perform computation based on their assigned tasks by the TM, communicated through a RabbitMQ broker. Each worker can deploy multiple tasks, being limited by the available docker images and their resources and context.

Developers can specify *hints* when creating *operators* in the task designer, which allows them to filter (*e.g.*, by location) and group (*e.g.*, by ID) input streams for a given operator and also set the cardinality of the input streams (*e.g.*, unicast, broadcast).

Once developers have submitted the application flow and developed the necessary docker images, the deployment is triggered. The TM executes a *Topology lookup* to find a possible processing path, then generates an *Execution plan* based on the topology and application flow, and, finally, deploys the tasks to workers.

FogFlow is a scalable solution that takes advantage of parallel brokers to achieve low-latency discovery. However, its approach has some problems: (1) developers need to manually develop an implementation of a task and publish it to the docker image repository, which not only incurs in work for developers but also limits the device support, and (2) the use of

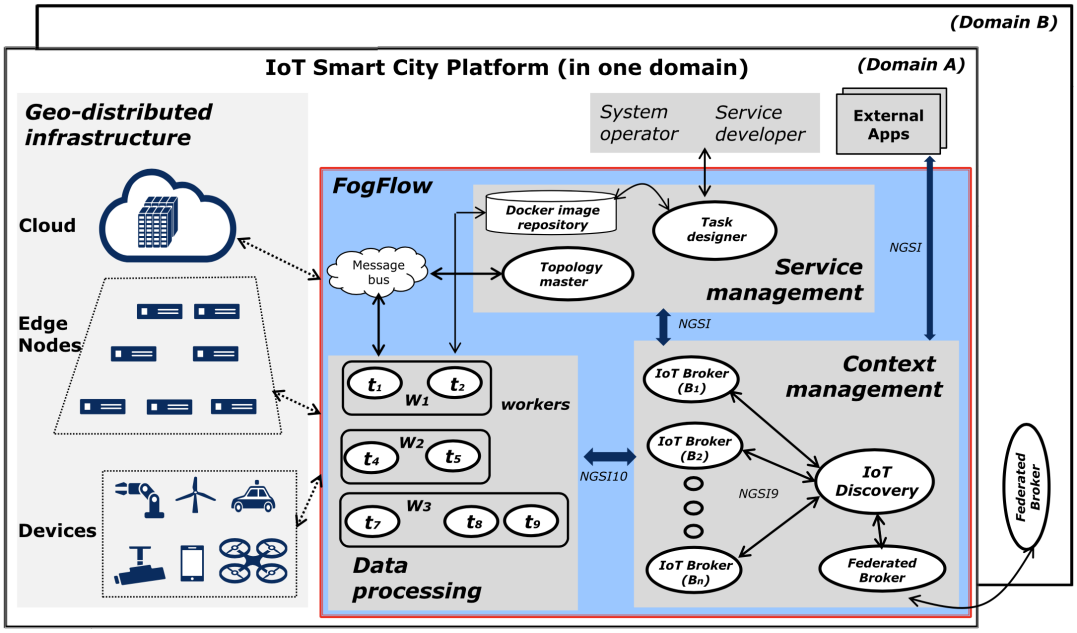


Figure 3.3: FogFlow system architecture [11].

NGSI and the lack of common support for this technology means that some edge devices need to communicate through a proxy device that implements this standard. Moreover, the authors fail to mention or evaluate the fault-tolerance of the system, only adding that support for fault-tolerance in extreme conditions is part of future work.

ECHO

Ravindra *et al.* [41] presented an open-source platform, ECHO, aimed at developing and deploying dynamic applications throughout the whole IoT tier (Cloud, Fog, Edge). It follows the data-flow programming model, supports hybrid data sources and the use of native runtime engines (*e.g.*, Tensorflow). A centralized *Platform Master service* is responsible for deploying data-flows based on the information present on the *Resource Directory*, which contains current information about the fog devices and their resources. Each fog device is managed by a *Platform service* that decides which parts of the data flow are run on each device.

However, the platform does not offer any self-adaptability functionality, as the operator must manually create the new deployment and trigger the re-orchestration in the system. Moreover, there is no mention of support for non-functional properties for automatic deployment, which have been proven essential in large-scale distributed fog systems [27, 34, 43].

**NoRDOr**

Silva *et al.* [45, 46] proposed a decentralized execution method for IoT systems with a twofold approach: (1) extend of Node-RED, to achieve automatic partitioning of application flows and dynamic deployment to edge devices, and (2) develop a MicroPython framework

for resource-constrained micro-controllers which allows them to run custom code. The extended version of Node-RED includes two new nodes: (1) *Registry*, which keeps a list of available devices in the system, and (2) *Orchestrator*, which decomposes the application flow into tasks and deploys them to edge devices. Additionally, Node-RED was also modified to use MQTT as the communication protocol so that nodes can be deployed externally; and, finally, some existing nodes were altered to generate MicroPython code, which in turn is run by edge devices. These devices run a custom framework that exposes endpoints for monitoring (*i.e.*, PING/ECHO) and executing a custom script. Furthermore, they announce themselves and their capabilities as they enter the system, allowing the *Orchestrator* to (re-)orchestrate the system dynamically.

Upon starting the system, the *Orchestrator* waits for device announcements for a short period of time before performing the first orchestration. After partitioning the application flow, a greedy algorithm is used to map nodes to devices. This algorithm takes into account each node's *Predicates* and *Priorities*, such that predicates are always fulfilled. Priorities are used alongside the number of nodes already deployed in each device to calculate the value of the heuristic used in the algorithm. After the assignment process is concluded, the MicroPython script for each node is generated and sent to the device. In case of a failed deployment (*e.g.*, device does not have enough memory), the device informs the orchestrator, and a re-orchestration is performed. A PING/ECHO strategy is employed to monitor each device's status, which can also trigger the orchestrator in case no response is sent from the edge device. Furthermore, upon crash, a device will restart and announce itself to the system again, and a re-orchestration is triggered. A sequence diagram representing the sequence of events in the system can be seen in (*cf.* Figure 3.4, p. 17).

According to the author, there are several known limitations in the present solution: (1) a re-orchestration does not minimize the set of changes to be performed, leading to greater overhead, (2) the mechanism for detecting node failures is too sensitive, *i.e.*, one failed PING request is assumed as device failure, which might not be the case in unreliable IoT devices, (3) the system halts during the assignment process, drastically reducing system availability, and (4) the assignment algorithm does not take advantage of intra-device communication by deploying connected nodes to the same device. In addition to the limitations identified by the author, we also found that the employed self-healing mechanisms that deal with general device failure and memory-driven device failure are rather basic, failing to handle situations of high device instability and requiring a trial-and-error cycle to prevent future memory errors.

**Seeger *et al.***

In 2018, Seeger *et al.* [43] proposed a system capable of running distributed and dynamic IoT orchestrations. Given the centralized approaches present on the market for automation in commercial buildings and the need for a technician to perform changes in the systems, the authors introduced a distributed approach that allows for sensors and actuators to be

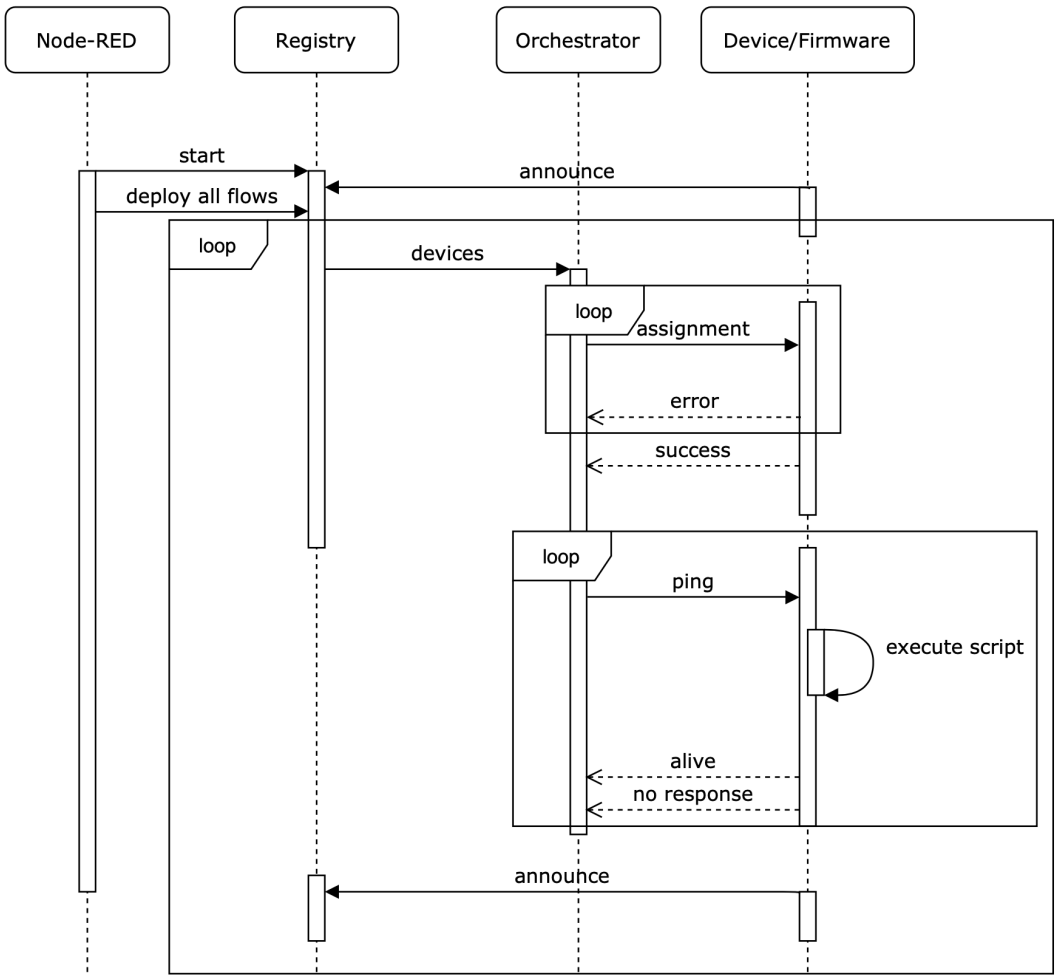


Figure 3.4: NoRDOr orchestration sequence diagram [45, 46]

orchestrated without the need for a centralized controller. The solution is based on a semantic model for IoT defined in a previous work by Thuluva *et al.* [51], called *Recipes*. This data-flow based model contains the concepts of: (1) recipes, which are templates for the configuration of *ingredients* and their *interactions*, (2) ingredients, which are placeholders for *offerings*, (3) interactions, which describe the flow of data between *ingredients*, and (4) offerings that describe service or device instances. Additionally, *offerings* are represented in an "offering description" format, containing information about its inputs and outputs, as well as how to access them.

In order to describe non-functional requirements for each *ingredient*, the authors introduced *offering selection rules* (OSR) that are evaluated against the non-functional properties of an offering. When instantiating a *recipe* (*i.e.*, replacing ingredients with offerings), OSRs must be respected. A recipe has many recipe runtime configurations (RRC) (*e.g.*, different orchestrations of the same recipe) and, each, multiple ingredient runtime configurations (IRC). Each IRC can contain multiple OSRs (*e.g.*, , restrictions on the location of each

device), and an RRC can contain one OSR, defining non-functional properties of the recipe (*e.g.*, cardinality). The relation diagram can be seen in Figure 3.5.

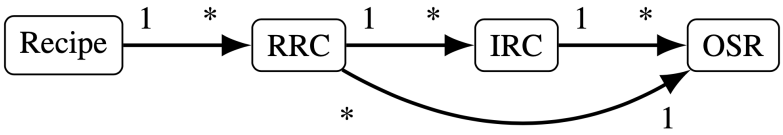


Figure 3.5: Relation of Recipes and OSRs [43].

The system's performance was evaluated by the time required to resolve OSRs, by varying the number of RRCs and OSRs, and was found to scale relatively well, completing the computation under one second up until 650 RRCs.

The authors also highlight some improvements to be made to the system – namely, (1) the current recipe model is limited to modeling REST services, and (2) the lack of fault-tolerance in the system, which was then explored in a later work [42].

The analyzed tools were characterized according to their support or approach to the following features:

**VPL:** makes use of Visual Programming Languages to design the application flow;

**Open-source:** allows for modification and redistribution;

**Device requirements:** IoT devices are considerably heterogeneous. Thus, it is relevant to analyze how broadly the solution can be applied;

**Automatic task allocation:** tasks are automatically allocated to devices based on the requirements of each task and capabilities of each device;

**Self-healing:** makes use of self-healing mechanisms.

Table 3.1: Fog/edge orchestration solution features

| Solution | VPL | Open-source | Edge device requirements | Automatic task allocation | Self-healing |
|---|---|---|---|---|---|
| DNR [8, 25, 26] | • | • | Must run Node-RED | • | - |
| DDFlow [34] | • | | -[4] | • | • |
| FogFlow [10, 11] | • | • | Must run Docker | • | - |
| 𝔼ℂℍ𝕆 [41] | • | • | Must run LXC | | |
| NoRDOr [45, 46] | • | • | Custom framework | • | Limited[2,3] |
| Seeger *et al.* [43] | | - | - | • | Limited[2] |

Bullets (•) mean *yes*, hyphens (-) mean *no information available*, and empty means *no*.
[1] Developers must create a docker image for each specific task and upload it to an image repository
[2] Performs full re-orchestration on (1) device failure or (2) device introduction
[3] System halts during orchestration
[4] Although no specific requirements are mentioned, edge devices must contain a set of services, *i.e.*, implementations of the tasks it can execute

From the analysis present in Table 3.1 (p. 18), we can conclude that the requirements for edge devices are varied, which is due to the high heterogeneity of devices in IoT networks. Because of this, some of the solutions [10, 11, 41] use OS-level virtualization to abstract the underlying heterogeneity. DNR [8, 25, 26] requires Node-RED on all devices, which excludes devices that are not running x86 or ARM-based Linux systems. Finally, Silva *et al*. [45, 46] developed a custom MicroPython framework for specific resource-constrained edge devices, and, to the best of our knowledge, DDFlow [34] had a similar approach, but the code is not open-source, contrary to most of the analyzed solutions.

Regarding task allocation, most technologies perform it automatically based on application-level constraints, except in 𝔼ℂℍ𝕆 which requires that developers manually deploy the system.

Finally, most solutions are limited in self-healing capabilities. Silva *et al*. [45, 46] and Seeger *et al*. [43] offer recovery from device failure and balancing upon device introduction, while DDFlow [34] also provides preventive balancing based on metrics such as CPU load but fails to provide further details on the implementation.

## 3.3 Self-healing & fault-tolerance in IoT

In this section, we explored self-healing/fault-tolerance mechanisms in IoT systems, using the following query:

```
(iot OR  internet-of-things OR "internet-of-things") AND (fault-tolerance OR fault-
    tolerant OR self-healing OR "run-time verification" OR "runtime verification")
```

Some authors [31, 3] present *Run-time verification* — *i.e.*, inspecting a running system by monitoring relevant events and metadata — as a good approach to ensure that IoT systems are reliable and compliant with the specification, given the heterogeneity of software and hardware in some systems. Leotta *et al*. [31] propose a run-time verification mechanism that (1) provides a monitoring mechanism, and (2) checks the event values against the previously formally defined system expected behavior using state machines. The authors implemented a new Node-RED node such that devices make a *check-event* POST request to an HTTP server, the monitor, which performs the verification against the model and communicates the result. However, this solution cannot be applied to dynamic edge computing scenarios where the system cannot be formally defined *a priori*.

Aligned with the run-time verification principle, but in the VPL realm, Dias *et al*. [17, 18] discussed the implementation of Self-Healing Extensions for Node-RED (SHEN) [20], a set of reusable Node-RED nodes that add run-time verification and self-healing capabilities. According to the authors, the motivation for the work was twofold: (1) the lack of IoT-specific testing systems, namely in popular IoT platforms such as Node-RED, and (2) the missing feedback loop between run-time verification and self-healing (taking action) in IoT systems.

A simplified version of Psaier's Self-healing loop [39] is used as a basis for the developed nodes in this work (*cf.* Figure 3.6). The *detection* nodes provide run-time verification by probing the system, *detecting* and *diagnosing* possible errors. Then, *maintenance* nodes provision *recovery* mechanisms. These mechanisms, as well as probing mechanisms, are also described in the form of patterns in a work by the same authors [19]. This paradigm of self-adaptive systems has been already explored in other contexts [22]. Although not designed for edge computing platforms, this work tackles some issues those platforms face (*e.g.*, proposes a redundancy mechanism for message brokers, thus removing a single point of failure).

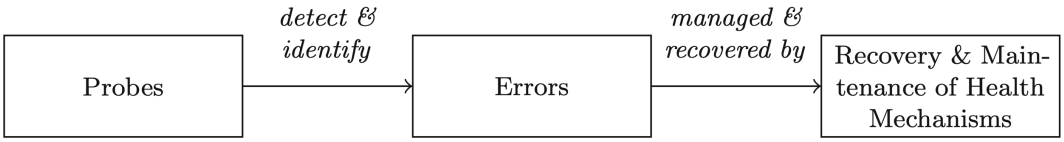


Figure 3.6: SHEN self-healing sequence [17].

Seeger *et al*. [42] identified the main steps of self-healing in IoT orchestration and presented a twofold approach to solving some of the challenges inherent to device failure and recovery: (1) detection of device failure, and (2) optimal allocation of tasks to edge devices. A policy-based failure detector based on the $\phi$-accrual principle of failure detection is proposed as a more suitable solution than existing ones due to the nature of edge IoT networks as it is (1) lightweight and thereby easily executable by distributed nodes with low computing capabilities, and (2) capable of detecting failures on unreliable and inconsistent networks.

However, the failure detection approach is limited to node crashes and fails to take into account other device degradation real-time metrics. The authors then introduce a heuristic-based approach to the challenge of assigning tasks to nodes, which resembles an NP-hard problem. Through this heuristic, they are able to reach a sub-optimal solution that can be deployed in real-time IoT systems, albeit with a considerable performance trade-off. Although the presented use case can be applied to a general IoT system as it minimizes the energy consumption of the network for a given distributed workflow, its definition of optimality might not be relevant for other IoT networks.

Another approach to run-time verification, namely Complex Event Processing (CEP), is used by other authors [3, 37] as a means to detect errors. CEP is an event processing technique that aims to extract patterns or events by combining data from multiple sources in (near) real-time. Aktas *et al*. propose a *Run-time verification module* that utilizes CEP to extract patterns based on the metadata sent by devices (*e.g.*, CPU load, memory load, bandwidth) and match them to a set of rules defined by the authors. These rules map a certain condition (*e.g.*, device exceeded a CPU load threshold in a given time period) to a self-healing action (*e.g.*, re-orchestrate network). The authors outline possible self-healing actions but do not implement them, as they are considered out-of-scope. Upon evaluation, the introduction of the run-time verification mechanism showed negligible overhead.

Power *et al*. [37] go one step further and combines CEP with a Machine Learning (ML) model to achieve reactive and proactive fault-tolerance in IoT, using a microservice-based approach. The *Real-time FT* microservice is responsible for communicating with edge devices and retrieving measures and possible errors. The incoming data is fed to a CEP component that uses pattern-matching between current values and previous ones to flag data as erroneous. Based on the extensiveness of the detected error (*e.g.*, erroneous data from 3 sensors), the system can also employ different recovery strategies. Furthermore, it can block data flagged as erroneous from going into the database. The *Predictive FT* microservice uses ML to consume the error and recovery data and attempts to generate fault patterns that allow the system to predict if, for a given state, it is likely that there will be errors in the future.

Although most of the solutions are centralized, it is also possible to perform distributed monitoring [35, 5]. Ozeer *et al*. [35] present a self-monitoring framework for dynamic Fog networks. *Fog nodes* monitor each other using and themselves and report the suspected failure to a *global manager* which can trigger re-orchestrations of the system. The system is suitable for stateful applications as it can re-construct and, thereby, recover the system state upon failure by the use of checkpoints and message logs.

Rivulet [5] is a fault-tolerant platform for IoT aimed at "smart-home" environments. The authors identified the common and varied failures of hubs, sensors, and actuators as the main driver for the platform, as well as the danger of event loss in critical scenarios (*e.g.*, elder-care apps, fire and intrusion sensors, etc.). In order to mitigate event loss, Rivulet employs two strategies: (1) *Gapless* delivery, where all processes query the available sensors and propagate the events using a lightweight ring protocol, and (2) *Gap* delivery, where only one process queries each sensor and propagates the events in a logical chain. The former comes at a greater guarantee that events are not lost, albeit also with a greater network overhead. Whereas the latter may lose some events if the process responsible for querying a sensor crashes but provides smaller network overhead.

When it comes to recovery mechanisms, Dias *et al*. [19] present them in the form of patterns. Most solutions offer a balancing mechanism, *i.e.*, moving load from one device to another. This is present in some of the previously analyzed edge computing platforms [46, 34, 42] and is achieved by re-orchestrating the network to cope with the current state in an attempt to maintain correct service.

Skarlat *et al*. [48] provide some insight into the difficulty of orchestrating a fog network by modeling and evaluating a fog computing system comprised of cloud and fog nodes, with the latter being organized in *fog colonies*. Only devices with relatively high computing resources, *fog cells*, are considered for computation tasks. Applications contain a set of services that can be time or resource-constrained (*e.g.*, CPU, RAM, storage demand). The authors identify the Fog Service Placement Problem (FSPP) as the placement of services in IoT nodes while satisfying Quality-of-Service (QoS) and time constraints. This is formulated as an Integer Linear Problem (ILP) and solved using IBM CPLEX. The ILP maximizes the utilization of the fog nodes, prioritizes applications with closer deadlines, and makes sure that the assigned services do not exceed the

node's capabilities.

The authors evaluate the system in terms of cost vs. a Cloud solution, with the latter proving considerably more costly (assuming ownership of the fog landscape). This is achieved while meeting application time constraints. Finally, the FSPP computational time is also evaluated with positive results – under one second even for large-scale applications.

Zhou *et al*. [55] present a fault-tolerant approach to IoT using virtual service composition. The authors mention the challenge of applying redundancy mechanisms in IoT, mostly due to cost and physical limitations (*e.g*., deploying two sensors in the exact place). As an alternative, they deploy virtual services that combine data from different sensors and apply regression models to infer an approximate value, similar to the DIVERSITY pattern [19].

From the work analyzed above, it is possible to conclude that self-healing in IoT is a diverse topic but not a very researched one [31]. Run-time verification has emerged as one of the most relevant approaches to self-healing in IoT, being used in several ways from state-machine modeling [31] to CEP [3, 37], sometimes used in combination ML to achieve not only fault detection but also fault prediction [37] Dias *et al*. [17] present a VPL solution for self-healing in Node-RED, using probing mechanisms to detect failures and applying recovery mechanisms. Seeger *et al*. [42] introduce a failure detection mechanism more suitable for edge networks and evaluate a heuristic-based approach to the node assignment problem in distributed IoT orchestrations, while Skarlat *et al*. [48] formulate the node assignment problem as an ILP and uses IBM CPLEX to solve it. Distributed monitoring is explored by Ozeer *et al*. [35], allowing edge devices to report a detected fault to the orchestrator. On the other hand, Ardekani *et al*. [5] propose a fault-tolerant "smart-home" IoT platform that ensures event delivery.

Dias *et al*. [19] present detection and recovery mechanisms in the form of patterns and apply some of them in [17], providing a VPL solution for self-healing in Node-RED. Zhou *et al*. [55] present virtual service composition as an alternative to redundancy in order to provide fault recovery in IoT networks. However, these solutions are not applied to the dynamic allocation of computing tasks – orchestration — across available devices but attempt to improve the already existent orchestration and processing solutions, thus not being suitable for our objectives.

## 3.4 Summary

Section 3.1 (p. 9) describes the methodology, including the research questions guiding this work, the queried databases, and the process used.

Section 3.2 (p. 10) presents the analyzed solutions for the orchestration of distributed IoT networks at the fog/edge. Each tool is summarized, and its main goal, characteristics, and drawbacks are described. From the analysis of the results, we can conclude that most solutions use VPLs, but implementation and requirements for edge devices are varied due to the high heterogeneity of IoT networks. Most solutions also provide automatic deployment, usually through the use of application-level constraints specified by the developer when building the application

model. Finally, we can also conclude that self-healing capabilities are rather limited in the analyzed solutions. Only one of them employed preventive mechanisms, but it is closed-source, and the implementation description is not very clear. Two other frameworks were found to have self-healing capabilities but to a limited extent.

Section 3.3 (p. 19) describes self-healing approaches in IoT systems, highlighting the run-time verification approach and included detection mechanisms such as probing and CEPs and recovery mechanisms such as balancing and isolation.

It was found that there is a clear gap when it comes to self-healing in IoT orchestration, as most solutions lack comprehensive self-healing mechanisms. We consider these to be crucial for IoT systems at the edge of the network, as these devices may be considerably unreliable Section 2.1.2 (p. 6), and their failure hinders the resilience of the system.

# Chapter 4

# Problem Statement



This chapter presents the problem under study in this dissertation, research questions, and validation methods. Section 4.1 describes the limitations derived from the current state of the art. Section 4.2 (p. 26) details a set of requirements for the system to be developed. Then, Section 4.3 (p. 27) sets the bounds of this work and Section 4.4 (p. 27) describes the hypothesis and research questions to be explored. Finally, Section 4.5 (p. 28) outlines the development and evaluation methodology to be followed in this dissertation.

## 4.1 Current issues

Section 3.2 (p. 10) presents several solutions that aim to orchestrate distributed IoT systems at the fog and edge. However, these tools lack comprehensive mechanisms to ensure dependability.

We focus on one particular tool — NoRDOr [45, 46] due to its rather complete feature set. It is an open-source system that integrates with Node-RED, meaning it can be used by non-developer end-users. Furthermore, it offers automatic partitioning and allocation of tasks, based on a set of requirements for each task and a set of capabilities for each device. Although limited, it also provides some fault-tolerance, re-orchestrating when device failure is detected, and providing a mechanism to deal with memory limitations.

Nevertheless, it fails to provide fault-tolerance in scenarios where devices are unstable or memory-constrained, ignoring the first and providing a flawed mechanism to handle the second. Moreover, the system's state is not leveraged upon re-orchestration, deploying the same or an entirely different script to each device every time. We further identify the specific issues found in the NoRDOr system:

**Inconsistent state** Many orchestrations can be triggered concurrently, sometimes producing different assignment outputs, leaving that the system in an incoherent state;

**Script re-deployment** Scripts are re-deployed to devices even if they are already assigned the same set of nodes, causing needless orchestration overhead and unnecessary data transfers;

**Unbalanced assignments** Certain scenarios may lead to an unbalanced assignment due to the underlying greedy algorithm used for assignment, meaning the failure of one device can cause a considerable disruption;

**Device (in-)stability** In the likely event that devices fail, the system does not leverage this information to improve future orchestrations, decreasing the system's resilience as nodes are assigned to unstable devices;

**Re-orchestration overhead** Whenever a new orchestration is computed, the system does not take the current assignment into account, performing a full re-orchestration that provokes orchestration overhead;

**Memory limitations** The mechanism for handling the devices' memory limitations relies on a trial-and-error mechanism that may require multiple re-orchestrations until devices' memory limitations are correctly estimated;

**Device-Node failures** In cases where a Device-Node pairing causes the device to fail, the system resorts to assuming it is memory limited instead of attempting to identify and mitigate the pairing.

The presented issues contribute to lower system resilience, as the failure of devices may highly impair the correct service. In this work, we focus on solving the aforementioned issues in the interest of achieving a fault-tolerant IoT orchestration platform for distributed task execution.

## 4.2 Desiderata

This work proposes a system capable of addressing the limitations stated above drawn from the NoRDOr system. Such a system should satisfy the following *desiderata*:

**D1: Generate balanced assignments** The orchestrator must be able to generate balanced assignments, given the currently available devices and the tasks' requirements;

**D2: Leverage device stability** The orchestration mechanism must take into account the device stability when calculating an assignment;

**D3: Minimal perturbation of the system** The orchestration mechanism must take into account the current assignment to minimize the disturbance introduced into the system;

**D4: Memory limitations** The orchestrator must be able to generate an assignment within the memory limitations of each device;

**D5: Detect device-node failure** The orchestrator must be able to identify and mitigate device-node pairings that cause device failure.

By generating balanced assignments, we refer to evenly distributing the tasks through the available devices that may change dynamically. By memory limitations, we are referring to RAM limitations. RAM is a limited resource at the edge and changes dynamically in runtime, meaning that it can considerably impact the amount of tasks a device can execute.

## 4.3 Scope

The scope of this work is to develop a prototype of a platform for real-time orchestration of IoT at the edge that ensures dependability in dynamic scenarios. This is to be developed on top of existing distributed orchestration platforms; thus, the development of the underlying platform itself is considered out-of-scope. Additionally, the focus of this work is the orchestration strategy, meaning that the distributed task execution mechanism is out-of-scope.

The target audience of the developed platform is IoT developers or system integrators that wish to leverage the computational capabilities of existing IoT devices while ensuring system dependability.

## 4.4 Hypothesis & research questions

Given a distributed heterogeneous IoT system, with dynamic task allocation, orchestration, the central research question this work intends to answer is:

> *"What mechanisms can be employed in dynamically orchestrated IoT systems that improve the system's dependability?"*

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

> *"Runtime information from dynamically orchestrated IoT systems can be used to achieve greater system resilience."*

In order to validate our hypothesis, we defined the following research questions that guide this work, in pursuit of fulfilling our *desiderata*:

**RQ1:** If we change the order in which tasks are processed, can we generate balanced assignments in dynamic settings?

**RQ2:** If we measure device stability and leverage it to make decisions on orchestration, can we produce solutions that are less susceptible to device failure?

**RQ3:** If we leverage the current assignment on re-orchestration, can we generate solutions that result in less orchestration overhead?

**RQ4:** If we leverage the real-time information on device resources to make decisions on the orchestration, can we produce solutions that are less susceptible to memory limitations?

**RQ5:** If we measure the stability of device-node pairings, can we leverage it on orchestration to identify and mitigate device-node pairings that cause device failure?

## 4.5 Methodology

To validate whether the proposed system fulfills the *desiderata* and provides solid answers to the *research questions*, we designed test scenarios and controlled experiments using both virtual and physical devices. Each of the experiments will be compared with a baseline of the original NoRDOr system and aims to validate one or more of the developed improvements.

However, not only is creating reproducible scenarios hard when working with cyber-physical real-time systems, but it is also challenging to collect real-time metrics that are agnostic to underlying device characteristics. In the interest of running a tighter validation cycle and streamlining the implementation of the proposed system, we developed a software-only *Orchestration Simulator*, where the improvements were tested before being ported into the NoRDOr architecture.

## 4.6 Summary

The current issues in existing edge computing orchestration platforms are introduced in Section 4.1 (p. 25), derived from the gap identified in the literature. A set of requirements for the system that the system must meet is detailed in Section 4.2 (p. 26), which summarises the characteristics required to address the limitations identified. The focus of this work is the orchestration strategy in an edge computing orchestration platform, and it is clearly defined in Section 4.3 (p. 27). Furthermore, the underlying platform is also excluded from the scope of this dissertation. As described in Section 4.5, it will be developed and validated in a tight validation cycle using an *Orchestration simulator*. Then, we transition to a system with physical and virtual IoT devices where we simulate failures and different system configurations in an attempt to prove the enhanced resilience of the system and provide answers to the questions singled out in Section 4.4 (p. 27).

# Chapter 5

# Orchestration simulator



This chapter presents an overview of the *Orchestration simulator*. First, the motivation for the development of this tool is described in Section 5.1. Then, a set of requirements that the system should fulfill is outlined in Section 5.2, and its boundaries are defined in Section 5.3 (p. 30). The implementation details are presented in-depth in Section 5.4 (p. 30), and an overview of the simulation lifecycle is described in Section 5.5 (p. 33). Finally, the systems' limitations are detailed in Section 5.6 (p. 33).

## 5.1 Motivation

Performing experiments with easily reproducible scenarios in cyber-physical real-time systems is not a trivial task, as the devices are susceptible to hardware and connectivity limitations, making it hard to ensure that only the relevant variables are in play.

In addition to low reproducibility, crafting complex scenarios and tweaking parameters is also cumbersome, as it may require building and deploying custom versions of the software [16, 13]. Consequently, reaching satisfactory solutions may prove to be a prolonged process without a tool that tackles these issues.

## 5.2 Desiderata

**D1: Real-time orchestration** The simulator must be able to dynamically allocate tasks to a set of devices in real-time;

**D2: Reproducible scenarios** The simulator must provide an interface to define and run reproducible scenarios, comprised of a set of tasks and devices;

**D3: Configurable devices** Each device must be configurable in terms of its resources and behaviors;

**D4: Orchestration strategies** The simulator must provide a modular interface to create different orchestration strategies;

**D5: Metrics** It must be possible to see real-time metrics and collect them to formatted files;

**D6: Extensibility** The simulator must be easily extended to include more complex concepts.

## 5.3 Scope

This simulator aims to mimic the functionality of a real-time orchestration platform for IoT systems. Its scope is limited to the orchestration component, *i.e.*, the functional component of distributed code execution is out-of-scope. The orchestration component includes the monitoring of devices by the orchestrator, the assignment process, and devices' behaviors.

This simulator is targeted at IoT developers that wish to test orchestration strategies in a controlled setting.

## 5.4 Implementation

The simulator was built using *Node.js*[1], and provides a Command Line Interface (CLI) as the entry-point, which allows the user to specify what previously defined *Scenario* to run, as well as define logging parameters, a time limit for the experiment and which orchestrator version to use.

A high-level view of the different components in the system is shown in Figure 5.1 (p. 31), and each is presented in detail in the next sections.

### 5.4.1 Scenario

The `Scenario` is the entry point class to the system. It is defined by its name — for logging purposes — but, most importantly, it contains a set of nodes and devices. The `run` method starts an experiment, *i.e.*, a single run of a scenario, by setting up the `Orchestrator`, the `Devices`, and the set of `Nodes`. Additionally, it also sets up the `Metrics` object that takes care of all the logging.

During the experiment, its only responsibilities are (1) increasing the experiment clock — the `Scenario` is the source of truth in this regard, and (2) optionally, running a custom scenario behavior, which allows for the definition of any custom function that is not easily encapsulated into individual device behaviors, *e.g.*, restarting devices in a specific order throughout the experiment.

[1] Node.js: https://nodejs.org/en/

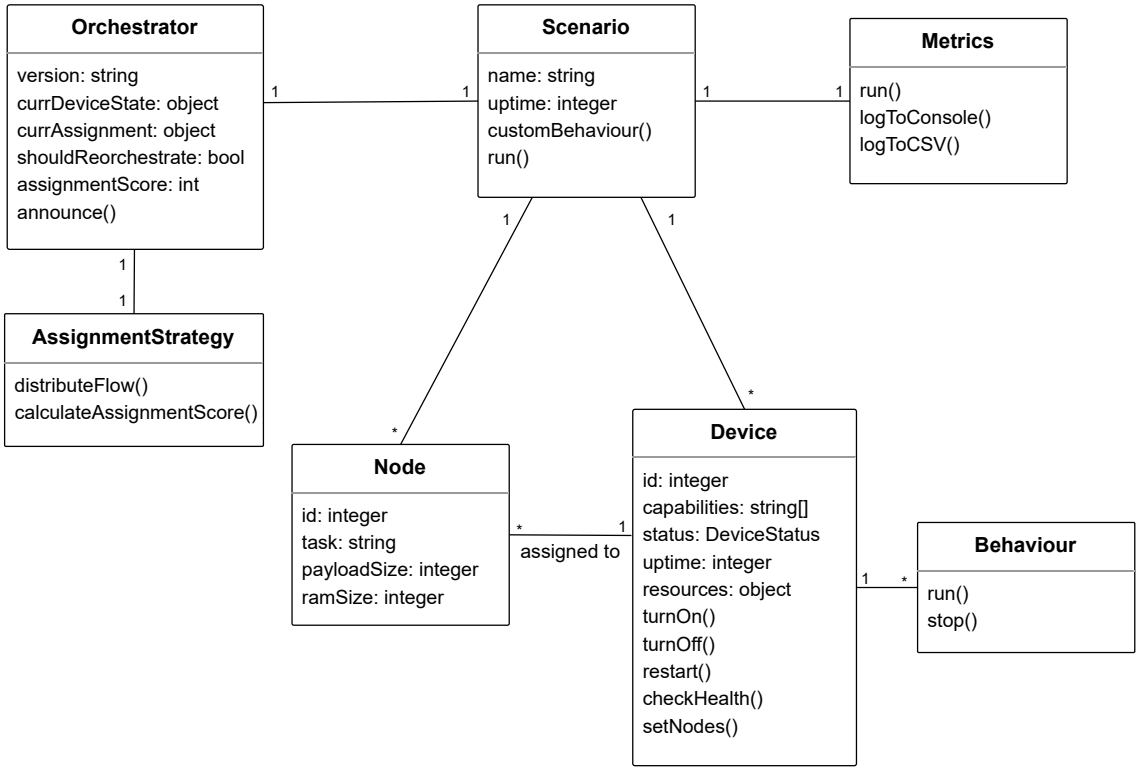


Figure 5.1: Orchestration simulator class diagram

### 5.4.2 Node

A `Node` is a simplified version of a *Node* in NoRDOr. It is also identified by an `id`, but instead of the *predicates* concept, it only has one associated `task`. A device can execute the node if its `task` is included in the device's capabilities. Additionally, each `Node` has a configurable `payload size` and `RAM size`, both defaulting to a value of 2.

### 5.4.3 Device

A `Device` has similar properties to the real devices in the NoRDOr system. It is identified by its `id`, has a set of `capabilities` — *i.e.*, the set of tasks it is able to execute, a `DeviceStatus` — `ON/OFF`, a set of `resources` — including static resources like `flash size` and `total RAM`, both defaulting to a value of 99999, and dynamic resources like `allocated RAM`, and an internal clock, in order to report their `uptime`. To abstract different device behaviors, each device can have a set of synchronous and asynchronous behaviors. Each synchronous behavior is executed in the run cycle of the device through the `run` method that also increments its clock. On the other hand, asynchronous behaviors run concurrently, allowing the user to define behaviors without relying on the device clock.

For the `Orchestrator` to communicate with a `Device`, the latter offers the `setNodes` and `checkHealth` methods. The first provides the interface through which the `Orchestrator`

deploys an assignment to the device — equivalent to the `POST /execute` request in NoRDOr, while the latter sends a report of its current state to the `Orchestrator` — equivalent to `GET /health`. The return value of these methods is actually a `Promise` that is resolved in a small random amount of seconds to mimic the real-world latency of making an HTTP request. As this promise resolves, the `Orchestrator` receives a `Response` object, which contains a `status` and `data`. Similarly to HTTP requests, the status indicates the result of the request — in this case, these can be `SUCCESS`, `FAIL` or `MEM_ERROR`, with the last only being used in the `setNodes` method.

A `Behaviour` encapsulates a common behavior of a `Device`. This is usually related to triggering a device failure for different reasons but could be extended to simulate resource degradation, for example.

In `asynchronous behaviors`, the `Device` executes the behavior's `run` method to start the behavior `interval` when the device boots up and the `stop` method when it turns off.

In the case of `synchronous behaviors`, the `Device` executes the behavior's `run` method on every tick of its clock.

Currently, the following behaviors are implemented:

**`DeviceNodeFail`** Synchronous behavior that restarts the device if it is assigned a specific node;

**`ProbabilisticFail`** Asynchronous behavior that restarts the device given a certain probability;

**`RecurrentFail`** Asynchronous behavior that restarts the device at a given interval;

**`SingleFail`** Asynchronous behavior that turns off or restarts the device once.

Each of these behaviors is highly customizable, from the interval at which they run to the time the device should take to restart.

### 5.4.4 Orchestrator

The `Orchestrator` contains the core logic of the system. It's responsible for assigning `Nodes` to `Devices`, as well as monitoring the latter and performing re-orchestrations as needed. It combines the functionality from the *Registry* and *Orchestrator* nodes in NoRDOr, becoming the source of truth for the device state. Additionally, it stores the `nodes` it should distribute, the `current assignment`, and its `score`.

In order to allow `Devices` to announce themselves to the `Orchestrator`, the latter provides the `announce` method that devices use as if they were announcing themselves to the MQTT topic in NoRDOr, *i.e.*, they call this method but expect no response from it.

The `Orchestrator` also runs a periodic health check that calls the `checkHealth` method on each device so that it can keep a quasi-real-time record of the device's state, as well as update any derived metrics relevant to the system.

Finally, the assignment logic is extracted using the STRATEGY design pattern by providing the `Orchestrator` with a `version` that can be mapped to a specific `AssignmentStrategy` at runtime. This allows the developers to create different strategies and compare them in various scenarios without using manually versioned code.

Each `AssignmentStrategy` can employ a different orchestration mechanism, as long as they provide the `distributeFlow` method that is called by the `Orchestrator`. The *superclass* implements some common methods, such as the `calculateAssignmentScore` method that all *subclasses* can use. Each *subclass* must also take care of deploying the assignment to the devices.

### 5.4.5 Metrics

The `Metrics` object provides the logging capabilities to both the console and CSV files. The object has access to the `Orchestrator` and `Devices` and can derive metrics as the experiment progresses by accessing the other objects. This means that the precision of the metrics is considerably higher than in a real system that relies on larger intervals and latency to collect information on the physical devices. Some of the collected metrics include device uptime, number of assigned nodes, and node uptime.

## 5.5 Simulation lifecycle

A simulation starts when the `run` method of the `Scenario` is executed. This triggers a set of method calls on other objects in the system — `Metrics`, `Orchestrator`, and all `Devices`, that run their initial setup. After this, the `Scenario` updates its experiment clock, resorting to the `setInterval` timing method.

Similarly, the `Metrics` object performs two operations every second: (1) update the metrics it keeps on devices and nodes with the latest information, and (2) log this information to the console and to CSV files.

The `Orchestrator` object permanently runs 2 parallel intervals. The first, every 500ms, checks if the system should trigger a re-orchestration, which may be due to a new device in the system, a failed device, or because the current assignment is incomplete. The second runs every 3s to check the health of every device.

Finally, each `Device` announces itself to the `Orchestrator` when it turns on, letting it know that it is available to be assigned nodes. Conversely, the `Orchestrator` can assign a set of nodes to the device. Additionally, every second, the device updates its uptime and runs all synchronous behaviors. In parallel, all the asynchronous behaviors execute their tasks at their defined intervals.

A visual representation of the events comprised in a simulation can be seen in Figure 5.2 (p. 34).

## 5.6 Known limitations

The simulator was deliberately designed to be a simplified and easier way to test real scenarios that offer low reproducibility. However, in doing so, the system may behave differently to a real

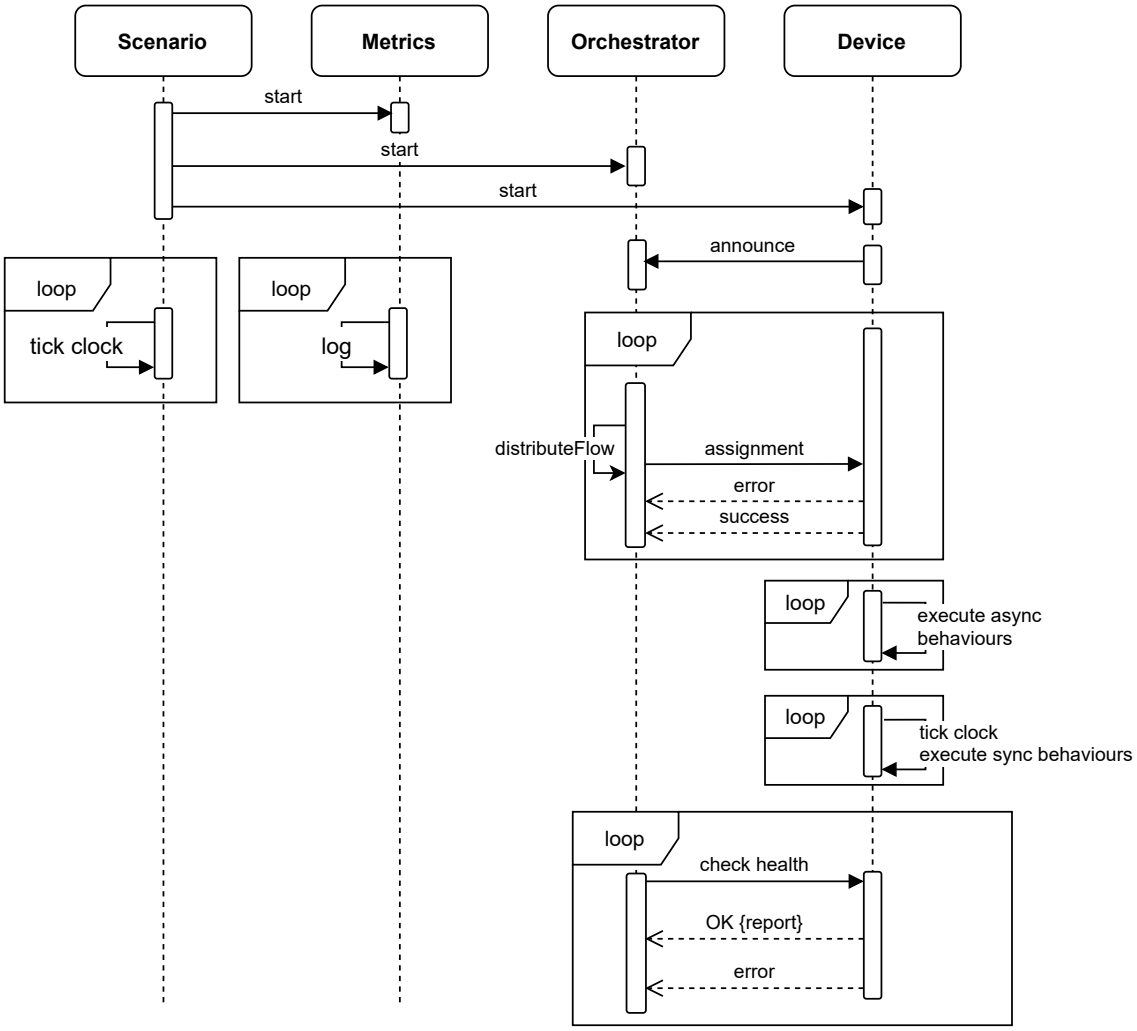


Figure 5.2: Simulation sequence diagram

system due to the lack of real concurrency — the system runs on the *Node.js* runtime environment, which is single-threaded.

Moreover, the simulator does not completely mimic the flows from NoRDOr as the `Node` is isolated and lacks the concept of input and output nodes. Nevertheless, we expect this to be easily implemented if need be due to the high extensibility of the system.

## 5.7 Summary

This chapter describes the implementation of an *Orchestration simulator* which is a *Node.js* application that allows the creation and simulation of reproducible scenarios for dynamic task allocation in IoT environments.

In Section 5.1 (p. 29), the motivation for this tool is described. Then, in Section 5.2 (p. 29),

a set of requirements the simulator must fulfill are presented, having been derived from the requirements of NoRDOr, and from the reproducibility issues that led to the development of this simulator.

The system focuses on the orchestration component — dynamic allocation of tasks through the available devices but does not include the task execution component, as outlined in Section 5.3 (p. 30).

In Section 5.4 (p. 30), we present an overview of the system and its main components, as well as a detailed explanation of the responsibility and core concepts of each one. We also highlight the fact that different iterations of the assignment strategy can be chosen at runtime due to the use of a STRATEGY design pattern, enabling developers to test different solutions without having to manually replace the code.

The lifecycle of a simulation and the associated interval-based events are presented in Section 5.5 (p. 33), alongside a sequential representation of them.

Finally, in Section 5.6 (p. 33), we describe the limitations of the developed system, such as the lack of pure concurrency that can impair the simulator's resemblance to a real-world system.

# Chapter 6

# Orchestration strategy



In this chapter, the several iterations that took place during the improvement of the orchestration strategy are outlined. First, the methodology and an overview of each iteration are presented in Section 6.1. Then, in Section 6.2 (p. 38) the basic behavior of the simulator is validated. Finally, each improvement is described in detail in Sections 6.3 (p. 38) to 6.9 (p. 54).

## 6.1 Methodology

When developing the solutions to tackle the current problems with the NoRDOr system (*cf.* Section 4.1, p. 25), we leveraged the *Orchestration simulator* to test and tweak the possible solutions quickly. This allowed us to validate quickly, although with lower confidence, the improvements by crafting and running a scenario in both the current and previous version. Since we took an iterative approach, adding cumulative changes to the system, the ability to separate each version and compare it with previous versions in deterministic scenarios, allows us to (1) evaluate the impact of previous improvements on each of the tackled problems, and (2) clearly measure the impact of a specific improvement, due to the baseline already containing the previous ones.

For each version, we craft one or more scenarios that highlight the problems we wish to tackle from the previous version. Then, we proceed to apply the proposed changes and compare the results of the same scenarios with the baseline version.

The several improvements to be made to the original system in order to fulfill the Desiderata and answer the Research Questions (*cf.* Chapter 4, p. 25) led to a total of five versions of the

orchestration strategy, excluding the baseline version ported from NoRDOr. In this chapter, this version is referred to as **v0**, while the enhanced versions are referred to as **v1**, **v2**, **v3**, **v4**, and **v5**.

Each version contains all the improvements to previous versions with one or more additional improvements, as shown in Table 6.1.

Table 6.1: Improvements by simulator version

| **Improvement** | **v0** | **v1** | **v2** | **v3** | **v4** | **v5** |
|---|---|---|---|---|---|---|
| General enhancements | | • | • | • | • | • |
| Device stability | | • | • | • | • | • |
| Orchestration score | | • | • | • | • | • |
| Sorted nodes | | • | • | • | • | • |
| RAM monitoring | | | • | • | • | • |
| Backtracking | | | | • | • | • |
| Minimum set of changes | | | | | • | • |
| Device-node stability | | | | | | • |

## 6.2 Base strategy

In the interest of validating the basic behavior of the simulator, this version, **v0**, is ported from NoRDOr, with the least possible modifications.

We design 2 Sanity Checks to ensure that the orchestration and re-orchestration mechanisms are aligned with the expectations. For both experiments, we use a scenario comprised of 36 nodes and 4 devices, where any device can be assigned any node.

In the first, **SIM-SC1**, we shut down one device at a time and expect that nodes are re-distributed through the available devices, maintaining a balanced assignment. As shown in Figure 6.1 (p. 39), the expectations are completely met.

In the second, **SIM-SC2**, we restart one device at a time and expect that nodes are re-distributed both when devices disappear but also when they re-appear. Figure 6.2 (p. 40) shows that the orchestration mechanism is indeed triggered in both cases. However, when Devs. 3 and 4 are restarted, the Number of nodes graph does not show this clearly, even though it can be seen in the Payload graph. We attribute this to the discretization of time and the need to aggregate data, losing some precision in the case of the heatmap graphs.

## 6.3 Initial improvements

We start by identifying some lower-priority problems that do not necessarily contribute to our hypothesis (*cf.* Section 4.4, p. 27), and solve them in version **v1**.

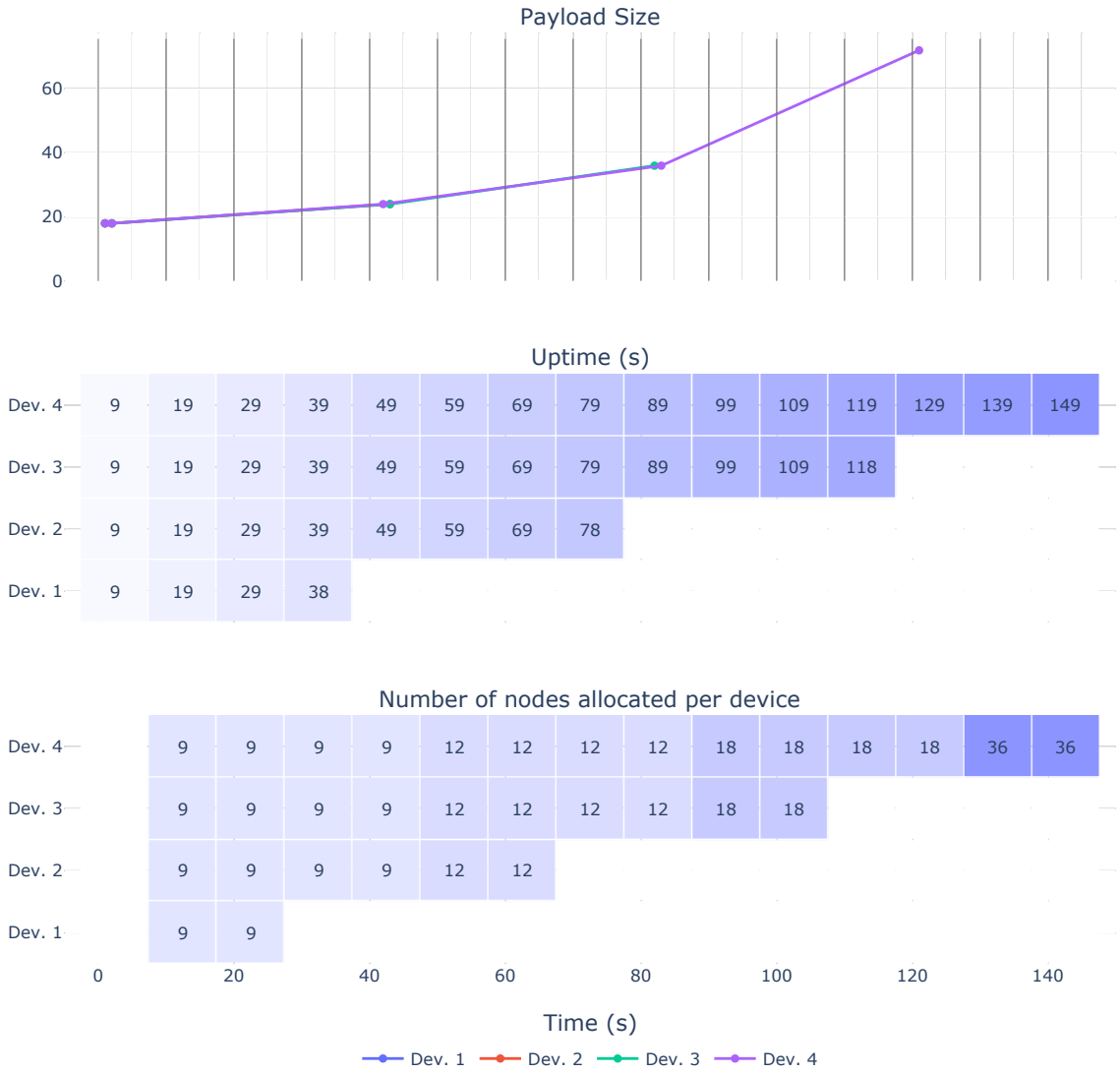


Figure 6.1: SIM-SC1 using **v0**

### 6.3.1 Identified problems

Firstly, it is possible for different calls of the `distributeFlow` method to be running concurrently, leading to an inconsistent state. Additionally, the same method can incur in recursive calls when a deployment fails, making it harder to measure its performance. Finally, the deployment mechanism does not take into account if the assignment it is deploying to a certain device has not changed, causing unnecessary network traffic and orchestration overhead.

### 6.3.2 Solutions

Firstly, we extract the re-orchestration trigger, `distributeFlow`, to a single execution point. When a device enters or leaves the system, a `shouldReorchestrate` boolean flag is set to

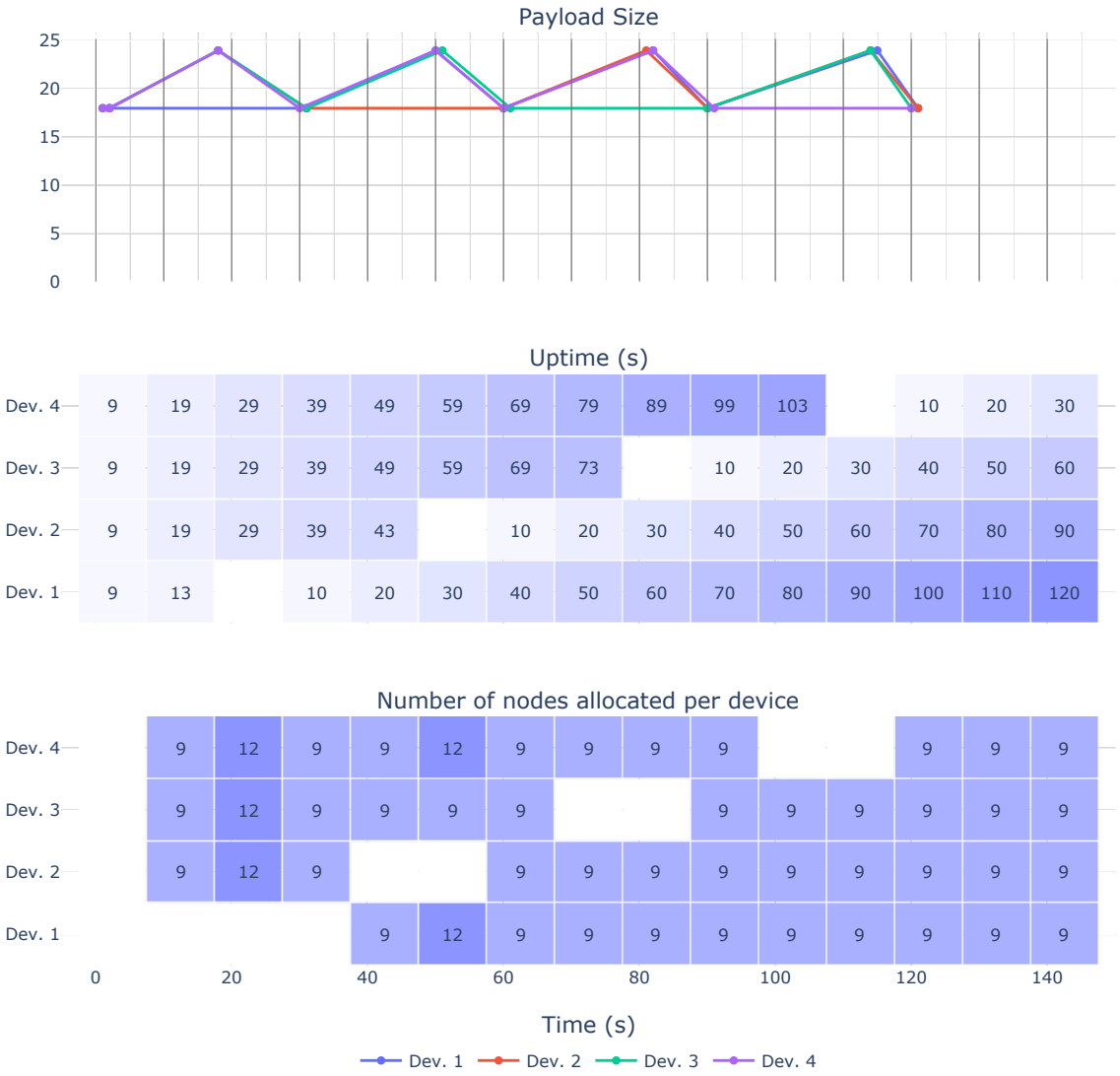


Figure 6.2: SIM-SC2 using **v0**

`true` and is set to `false` when a successful orchestration is completed. Additionally, we carry the `assigning` boolean flag from the original system that signals that the `distributeFlow` method is running. Combining these two flags, every 500ms, the system checks if it should re-orchestrate, and if it is not currently assigning, calls the `distributeFlow` method. These changes ensure that there are no concurrent orchestrations running and that there is no recursion on the orchestration method itself. In addition to this change, we also prevent a script from being deployed if the device is already executing those same nodes. This is achieved by comparing the set of nodes resulting from the new assignment and the ones the device is currently executing.

## 6.4 Handling device instability

Version **v1** also introduces a mechanism for leveraging device stability on orchestration.

### 6.4.1 Identified problem

If a device is behaving incorrectly, causing it to fail repeatedly or failing more often than would be considered acceptable, the system should converge to a state where that device is not assigned any nodes, if possible. Because there is no such mechanism, whenever a device either re-connects or disconnects, the system will perform a re-orchestration, even if that device has shown unstable behavior in the past.

In order to confirm our expectations and validate a possible solution, we design a scenario, **SIM-A**, comprised of 36 nodes and 4 devices, 2 of which fail every 5s, taking 15s to restart. We expect the system to re-orchestrate whenever the devices fail and re-connect.

Figure 6.3 (p. 42) shows that our assumptions hold. Whenever a device re-connects, the system generates a new assignment, including the failing devices, leading to a countless number of re-orchestrations. As the device fails shortly after, another re-orchestration needs to happen for the system to return to a correct state. It is also possible to notice that during the whole experiment, the 36 nodes are only being executed in 3 separate instances — since data is aggregated by 10s intervals, this does not mean it only happened for 3s.

### 6.4.2 Solution

To measure device stability, we resort to the *Mean Time Between Failures* (MTBF) metric. This can be calculated by dividing the total number of operational time (seconds in this case) by the total number of failures, as shown in (6.1). A lower MTBF means that the device has failed more often and can be considered unstable, and vice-versa.

$$MTBF = \frac{Total\_uptime}{\#\ failures} \tag{6.1}$$

Each device's MTBF is updated on 2 separate occasions: (1) on health check — if the device is alive, the total uptime is updated, and consequently the MTBF, and (2) on announce — the number of fails is incremented, and the MTBF re-calculated. This is possible because the proposed system replaces the PING pattern with the HEALTH CHECK pattern, where the device sends a report of its state instead of simple `OK` to signal it is alive. In this report, each device sends its current uptime, which is kept in the *Orchestrator*, along with the total uptime for that device.

The **device stability** metric is then included in the greedy heuristic when calculating the score for each device. Since the MTBF value is not upper bound, we normalize it by applying 2 operations, as show in (6.2). Firstly, the MTBF value is divided by $\alpha$, which is the `STABILITY_FACTOR` (in seconds) configured for the system. If the MTBF value is greater than this factor, the device is considered stable; otherwise, it is a value between 0 and 1, with 0 meaning highly unstable and 1 meaning stable. Afterward, because this value can still be greater than 1 — if the MTBF is greater

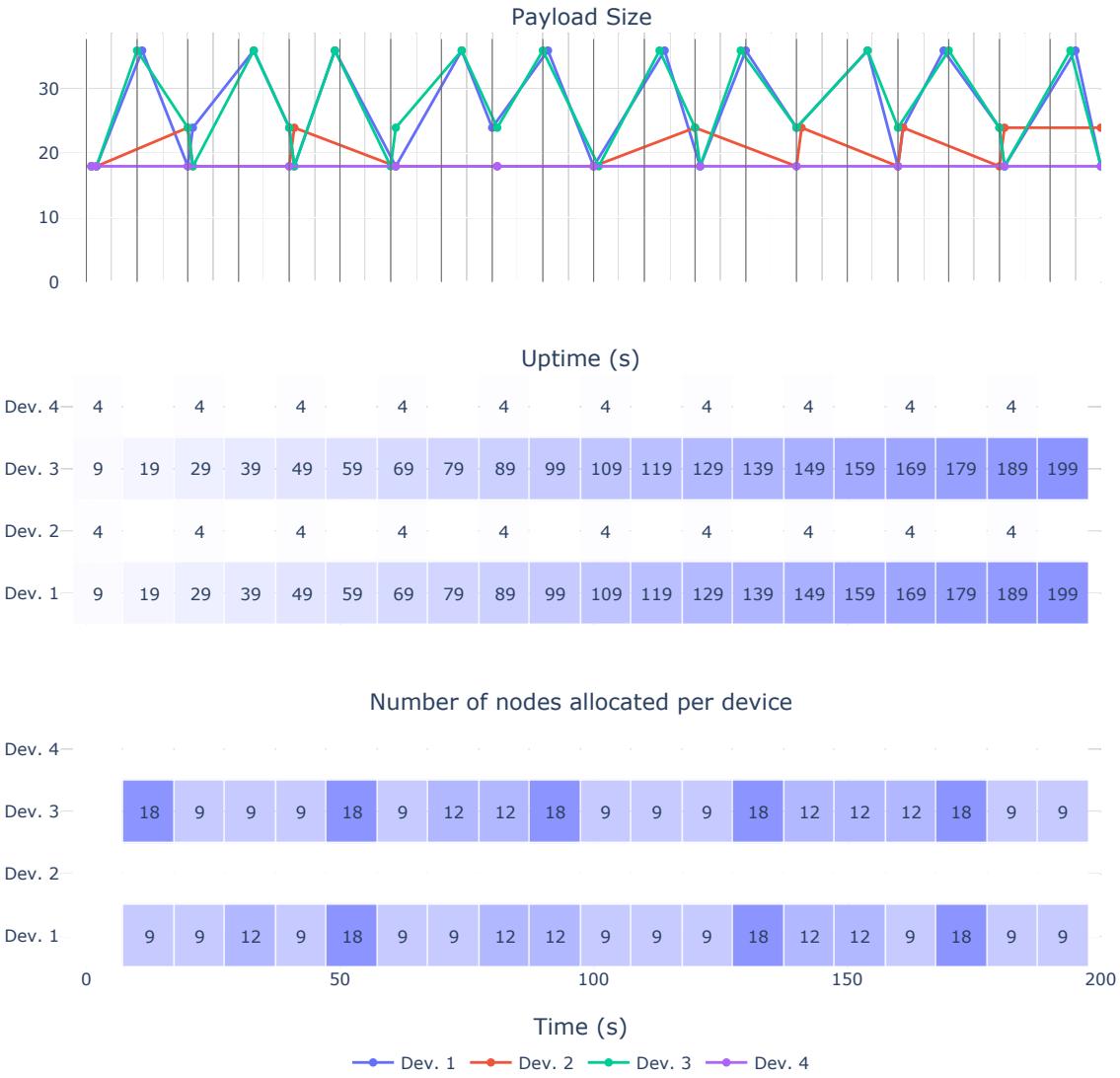


Figure 6.3: SIM-A using **v0**

than `STABILITY_FACTOR`, we take the *minimum* between the calculated value and 1, effectively setting the upper bound to 1. On the other hand, if the device has not failed at all, the MTBF would be *infinity*, so we set the *stability* to 1.

$$stability = \begin{cases} min(\frac{MTBF}{\alpha}, 1) & \text{if device.failures} > 0 \\ 1 & \text{otherwise} \end{cases} \tag{6.2}$$

The greedy heuristic, $h$, for each device $d$, becomes $h = 0.3 \cdot d.vacancy + 0.7 \cdot d.stability$, where *vacancy* is the inverse of the number of nodes that the device has been assigned so far — this was ported from the NoRDOr system, where it is also used to ensure the balancing as the assignment progresses. This means that less stable devices receive fewer nodes and that stability takes precedence over balancing.

In addition to modifying the heuristic, we should also prevent an assignment from being deployed if it is worse than the previous. Thus, we introduce the concept of **assignment score** — a floating-point number between 0 and 1 for a successful assignment; -1 otherwise. Using the score to decide if an assignment should be deployed allows us to prevent the system from constantly deploying a new assignment when an unstable device connects. The score is calculated independently from the greedy algorithm, as it should objectively assess the quality of a full assignment, and it encompasses the concepts of *device stability* and *balancing*.

To compute the score, as shown in (6.3), we start by calculating the *mean* of nodes per device — this represents the **expected**, not the **observed** value. We then calculate the *total absolute deviation* for the observed values. Because we want to normalize this value to be between 0 and 1, and since we know the largest possible *deviation*, we divide the *total absolute deviation* by the largest deviation possible — when all nodes are assigned to one device. At this point, the more unbalanced the system is, the greater the balancing value is, so we reverse it.

Additionally, we calculate the *average node stability* by computing the total stability and dividing it by the number of nodes. Since each *device stability* value, as shown in (6.2), is bound between 0 and 1, dividing the total by the number of nodes means that the *average node stability* will also be between 0 and 1. Finally, the *assignment score* is computed as the *arithmetic mean* of the *stability* and *balancing* values if the assignment is complete (all nodes are assigned); or -1, otherwise.

$$
\begin{aligned}
mean &= \frac{\#\,nodes}{\#\,devices} \\
deviation &= \sum_{d \in devices} |d.\#nodes - mean| \\
balancing &= 1 - \frac{deviation}{(\#\,devices - 1) \cdot mean \;+\; (\#\,nodes - mean)} \\
stability &= \frac{1}{\#\,nodes} \sum_{d \in devices} d.stability \;\cdot\; d.\#nodes \\
score &= \begin{cases} -1 & \text{if incomplete assignment} \\ 0.5 \cdot balancing \;+\; 0.5 \cdot stability & \text{otherwise} \end{cases}
\end{aligned}
\tag{6.3}
$$

The *assignment score* is stored whenever an assignment is successfully deployed, and when a new one is computed, it is only deployed if the score is greater than the current one. However, if a device is found to have failed, the current score is set to -1 to allow for **any** complete assignment to be deployed, restoring the correct service.

After implementing these changes, we re-test scenario **SIM-A** with version **v1** and expect that the system does not re-orchestrate as the unstable devices keep re-connecting. Figure 6.4 (p. 44) indeed shows that our assumptions are correct. By looking at the Payload graph, we can see that in the first assignment, all devices are assigned nodes, but as Devs. 2 and 4 fail shortly after receiving the assignment, it is not observable in the Number of nodes graph. As Devs. 2 and 4 restart, the system does not assign them any nodes because they are considered unstable; thus, the

new assignment does not have a higher score than the previous one.

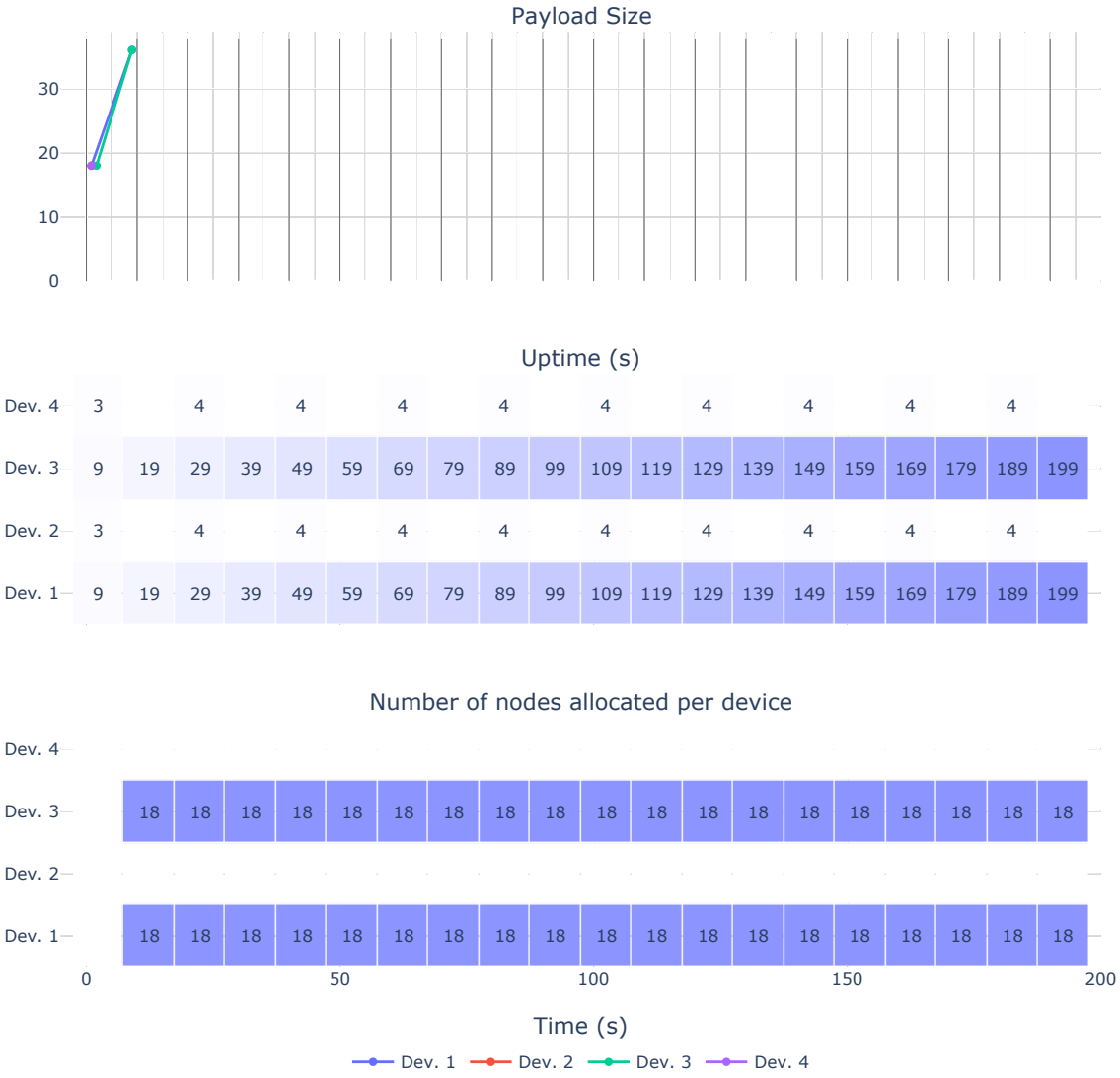


Figure 6.4: SIM-A using **v1**

Comparatively to **v0**, **v1** yields a *average node uptime* almost 15% higher, meaning that the system is providing correct service for a considerable greater amount of time, as shown in Figure 6.5 (p. 45).

## 6.5 Generating balanced assignments

The final improvement introduced in version **v1** is a strategy that aims to provide balanced assignments in dynamic settings.

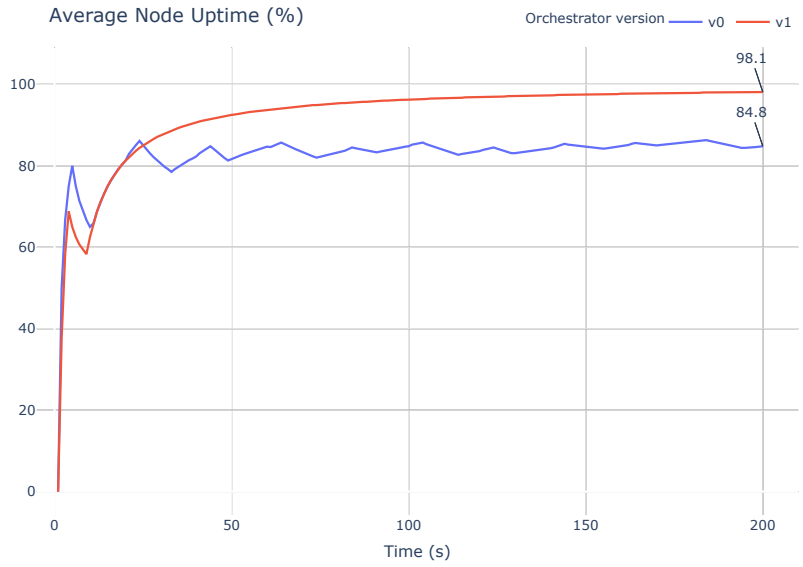


Figure 6.5: SIM-A average node uptime comparison between versions **v0** and **v1**

### 6.5.1 Identified problems

The greedy algorithm always processes nodes in the same order, meaning that there are some cases where it can lead to an unbalanced assignment. If a large number of specific nodes (only executable by a subset of devices) is processed last, then the assignment will always be unbalanced. Because the greedy algorithm solves for locally optimal solutions, it does not consider that the last nodes that it will process will all be assigned to a smaller number of devices. Additionally, the devices available to the system change dynamically, meaning that a balanced assignment for a set of devices might not be as balanced for a different one.

As a practical example, a scenario, **SIM-B**, is comprised of 10 nodes and 2 devices. The first 6 nodes can be executed by any device, while the other 4 need to be executed by Dev. 1. The system should assign 3 nodes to each of the devices after processing the first 6 nodes, and then assign the last 4 nodes to Dev. 1, resulting in 7 nodes assigned to Dev. 1 and 3 nodes assigned to Dev. 2. In larger, more complex scenarios, this could possibly result in a highly unbalanced assignment.

As shown in Figure 6.6 (p. 46), the system behaves exactly as expected, producing an unbalanced assignment.

### 6.5.2 Solution

To solve the highligted balancing issues, we take a simple approach — sort the nodes by their specificity, which is given by the number of devices that can currently execute it. This is done before the start of the assignment process in order to ensure that the specificity is limited to the currently available devices.

Using scenario **SIM-B** described above, we run a new simulation using **v1**. We expect that this change results in the following: (1) the 4 specific nodes are assigned to Dev. 1 (the only one

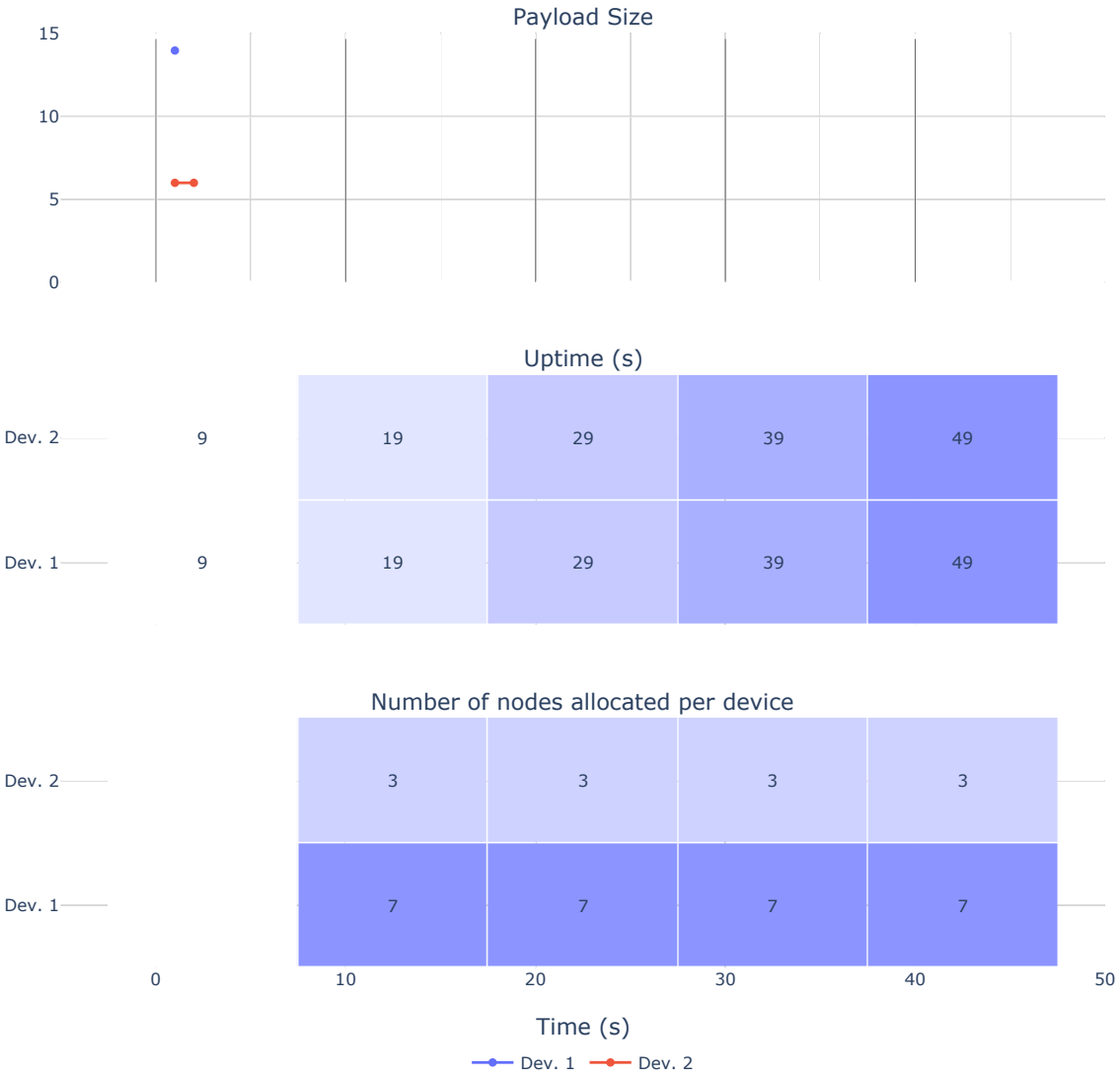


Figure 6.6: SIM-B using **v0**

that can execute them), and (2) the following 6 general nodes are distributed 1 to 5 for Devs. 1 and 2, respectively, because of the balancing heuristic, resulting in a total of 5 nodes for each device. Figure 6.7 (p. 47) shows that the behavior is as expected.

## 6.6 Dealing with memory limitations

Version **v2** introduces changes to the mechanism responsible for preventing device failure caused by memory limitations.

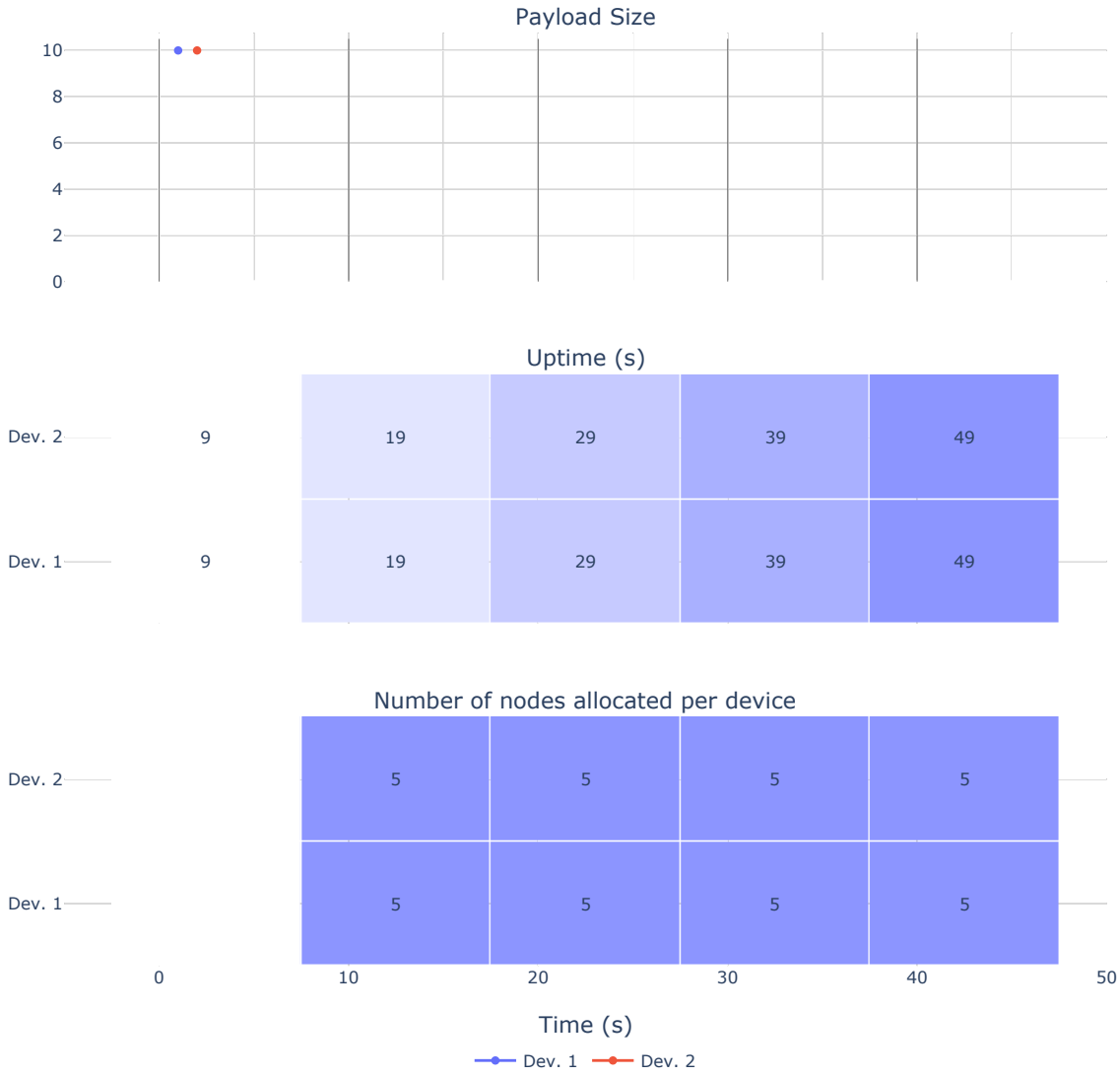


Figure 6.7: SIM-B using **v1**

### 6.6.1 Identified problem

The NoRDOr system uses the concept of `memoryErrorNodes` to deal with memory limitations. This is an integer value associated with each device that should represent the maximum amount of nodes it can execute. Whenever a device fails with a memory error — either on deployment or by announcing a failure — its `memoryErrorNodes` value is set to the number of nodes it was executing. This behavior can be both over-restrictive and not restrictive enough. Since different nodes have a different impact on RAM usage, and some specific nodes can even cause a specific device failure, the `memoryErrorNodes` may be decremented so much to the point that we are no longer leveraging the device's capabilities. On the other hand, it can also take a trial-and-error cycle to get to a reasonable `memorryErrorNodes` value because the system will keep trying to

assign `memorryErrorNodes` − 1 nodes to the device until it no longer crashes.

In order to highlight this limitation, we design a scenario, **SIM-C**, where 2 of the 4 devices are RAM-constrained. There is a total of 36 nodes to be distributed, and any device can execute any node. Due to the impact of the *device stability* feature (*cf.* Section 6.4, p. 41), we also restart one of the non-constrained devices, forcing the system to perform a re-orchestration. We expected the system to clearly show several attempted deployments to the constrained devices, as the system adjusted to the correct `memoryErrorNodes` value.

Figure 6.8 (p. 49) shows that our assumptions hold but not as clearly as expected. Looking at the Payload graph for the first assignment, it is noticeable that there are several assignment attempts to Devs. 1 and 3, which are RAM-constrained. Because they repeatedly fail so early, their stability becomes very low, causing the system not to assign them any nodes. When Dev. 2 is restarted at around 60s, Devs. 1 and 3 are still not assigned any nodes, presumably due to its still low stability — they fail multiple times in the early instances. At around 140s, when Dev. 2 is restarted again, the Devs. 1 and 3 are already considered more stable, and the system attempts to distribute them nodes — we can clearly see that the payload sent to these devices is lower than the ones attempted at the beginning of the experiment, but still too big for the devices, causing them to fail again and becoming too unstable to be considered on orchestration. At around 210s, Devs. 1 and 3 are, finally, assigned nodes without crashing. They are not kept at their RAM limit, possibly due to their still low stability causing the system to assign them fewer nodes.

### 6.6.2 Solution

By leveraging the ability to gather real-time data from the devices (from the health check), we can replace the `memoryErrorNodes` with the RAM size of each device. Since the orchestrator knows the RAM size of each node and each device, it can perform a perfect assignment in this regard. Specifically, we exclude a device from being assigned a certain node if adding it to its current assignment would exceed the device's RAM size.

In the same scenario **SIM-C**, **v2** performs a perfect orchestration, assigning the highest possible amount of nodes to each of the devices in the system (*cf.* Figure 6.9, p. 50).

## 6.7 Exploring the solution space

In version **v3**, we introduce a backtracking mechanism to mitigate some downfalls of the greedy algorithm.

### 6.7.1 Identified problem

As Silva *et al.* [45, 46] pointed out, the greedy algorithm can lead to some problems — namely, memory constraints may cause impossible orchestrations when the only device capable of executing a certain node is already assigned nodes that keep it at its memory limit. This happens because the algorithm employs neither a backtracking mechanism nor a RAM-related heuristic.

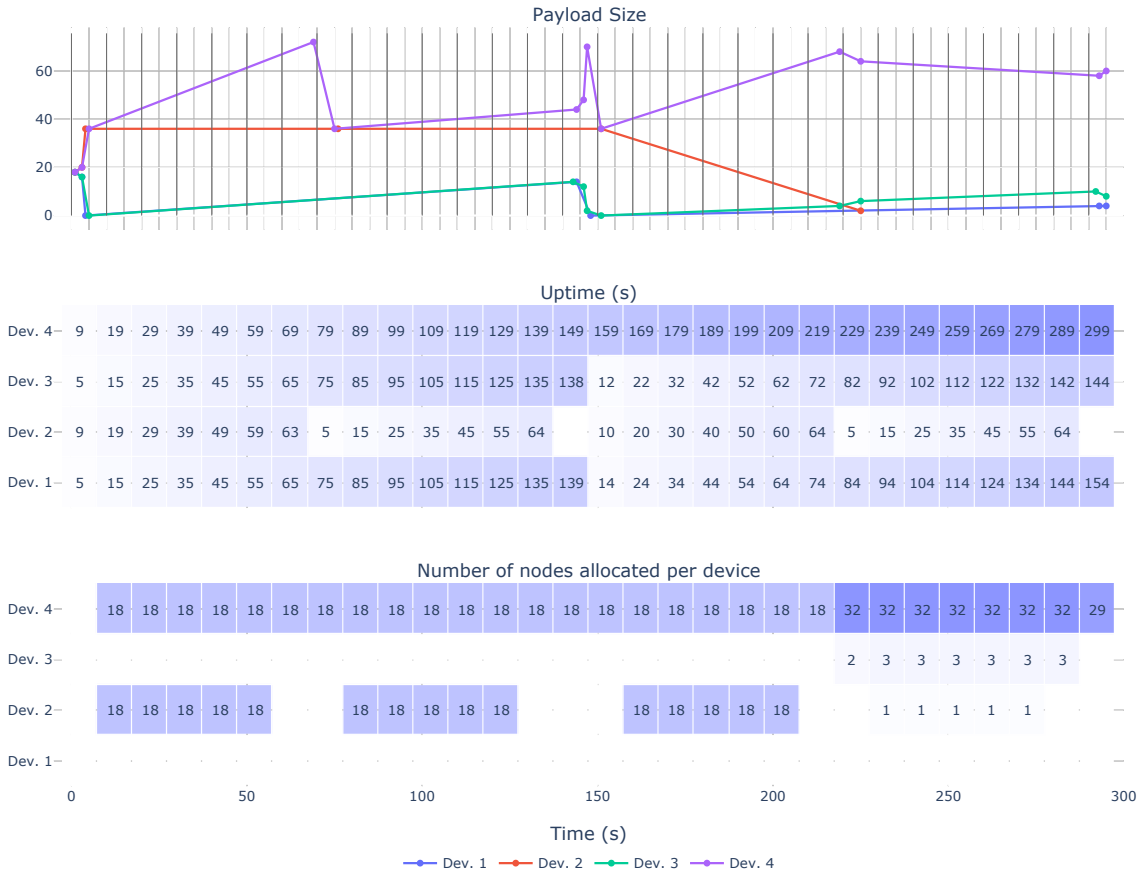


Figure 6.8: SIM-C using **v1**

### 6.7.2 Solution

In order to mitigate this problem, we introduce a backtracking mechanism in the algorithm that will attempt to select the best device for each node, backtracking if the current solution fails, and attempting the next best device. Specifically, we replace the `getBestDevice` method, responsible for finding the best device for a specific node, with the `getDeviceScores` that returns a sorted list of the devices that can execute a specific node. Additionally, we encapsulate the backtracking behavior in a recursive method, `generateAssignment`, that makes use of the `getDeviceScores` to compute the assignment and returns the full assignment when a solution is found, as shown in Algorithm 1 (p. 51).

We opt not to include a RAM component in the greedy heuristic because it's not clear that the system would benefit from it to the detriment of the current stability and balancing-based heuristic. Thus, we stick to excluding a device from the assignment if it will not be able the node being tested (*cf.* Section 6.6, p. 46). This means that even though we are using backtracking to fight the RAM limitations, we are not actually solving for the best RAM allocation, which may be considered illogical.

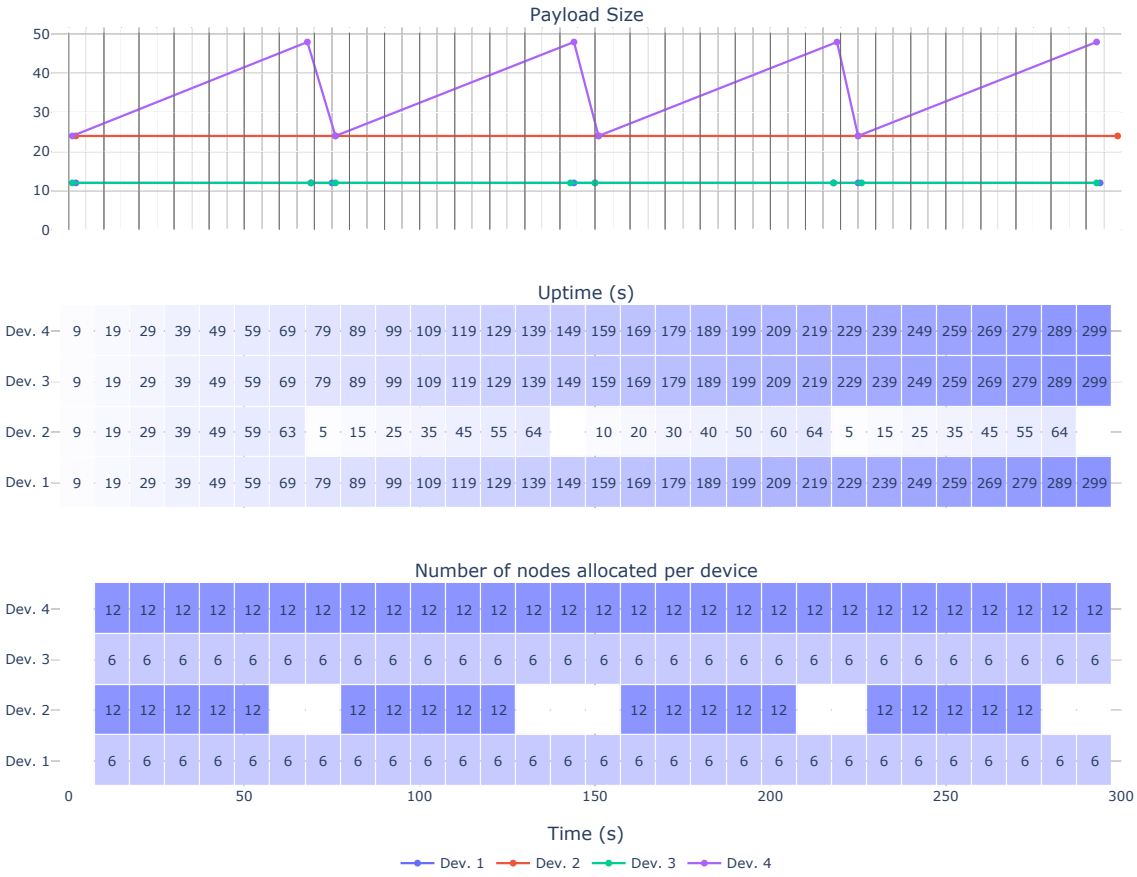


Figure 6.9: SIM-C using **v2**

Because of this limitation, we noticed that when testing highly constrained scenarios with a considerable amount of nodes, the backtracking mechanism is impractical because of the brute-force approach. We do not consider this to be a very likely scenario, so we opted to maintain the backtracking mechanism for its benefits and add an *early stopping* strategy that interrupts the backtracking if it takes longer than 3 seconds to find a solution.

## 6.8 Minimizing the disturbance on re-orchestration

Version **v4** introduces a mechanism to leverage the current assignment on re-orchestration as a means to minimize orchestration overhead.

### 6.8.1 Identified problem

Currently, when a new orchestration is generated, the current one, if valid, is not taken into account. Deploying a new assignment to a device is costly because it needs to stop executing the current nodes, delete the script and re-install the new script. Although this is not necessarily very time-consuming, it may have a functional impact on the system, as critical data from sensors could

**Algorithm 1:** `generateAssignment()`
Recursive greedy assignment algorithm with backtracking mechanism

```
input : nodes: Node[],
        index: integer,
        devices: Device[],
        assignment: Map<Device, Node[]>
output: Map<Device, Node[]>

begin
    if index ≥ nodes.length then
        return assignment
    end
    node ← nodes[index]
    devices ← getDeviceScores(node, devices, assignment)
    for device ∈ devices do
        newAssignment ← assignment ∪ {device → node}
        next ← generateAssignment(nodes, index + 1, devices, newAssignment)
        if next ≠ NULL then
            return next
        end
    end
    return NULL
end
```

be lost when a device is re-assigned nodes. Additionally, it also adds entropy to the system as it increases the chance of causing a failure in the system, *e.g.*, device-node fail (*cf.* Chapter 6.9, p. 54).

In the interest of validating our solution to this problem, we set up a scenario, **SIM-D**, comprised of 35 nodes and 4 devices, where any device can execute any node. One of the devices is turned off shortly after, and the system re-orchestrates. We expect that, in **v3**, a full re-orchestration takes place, re-assigning all nodes. As expected, Figure 6.10 (p. 52) shows that when Dev. 3 fails at around 30s, all of the nodes are re-orchestrated, assigning 12 nodes to Devs. 1 and 2, and 11 to Dev. 4.

### 6.8.2 Solution

In order to address this issue, the current orchestration must be leveraged in order to mitigate the impact of the re-orchestration. Because the NoRDOr system requires the deployment of the full script for each device, *i.e.*, it is not possible to add or remove a specific node from a device, and due to the fact that changing the code execution component is out-of-scope (*cf.* Section 4.3, p. 27), we attempt to minimize the set of changes on re-orchestration by minimizing the number of affected nodes.

Specifically, we define the **Minimum Set of Changes** (MSC) as the set of changes to the assignment that minimize the number of nodes that need to stop during the deployment of the assignment. In case the re-orchestration is triggered due to device failure, this is done in 4 steps:

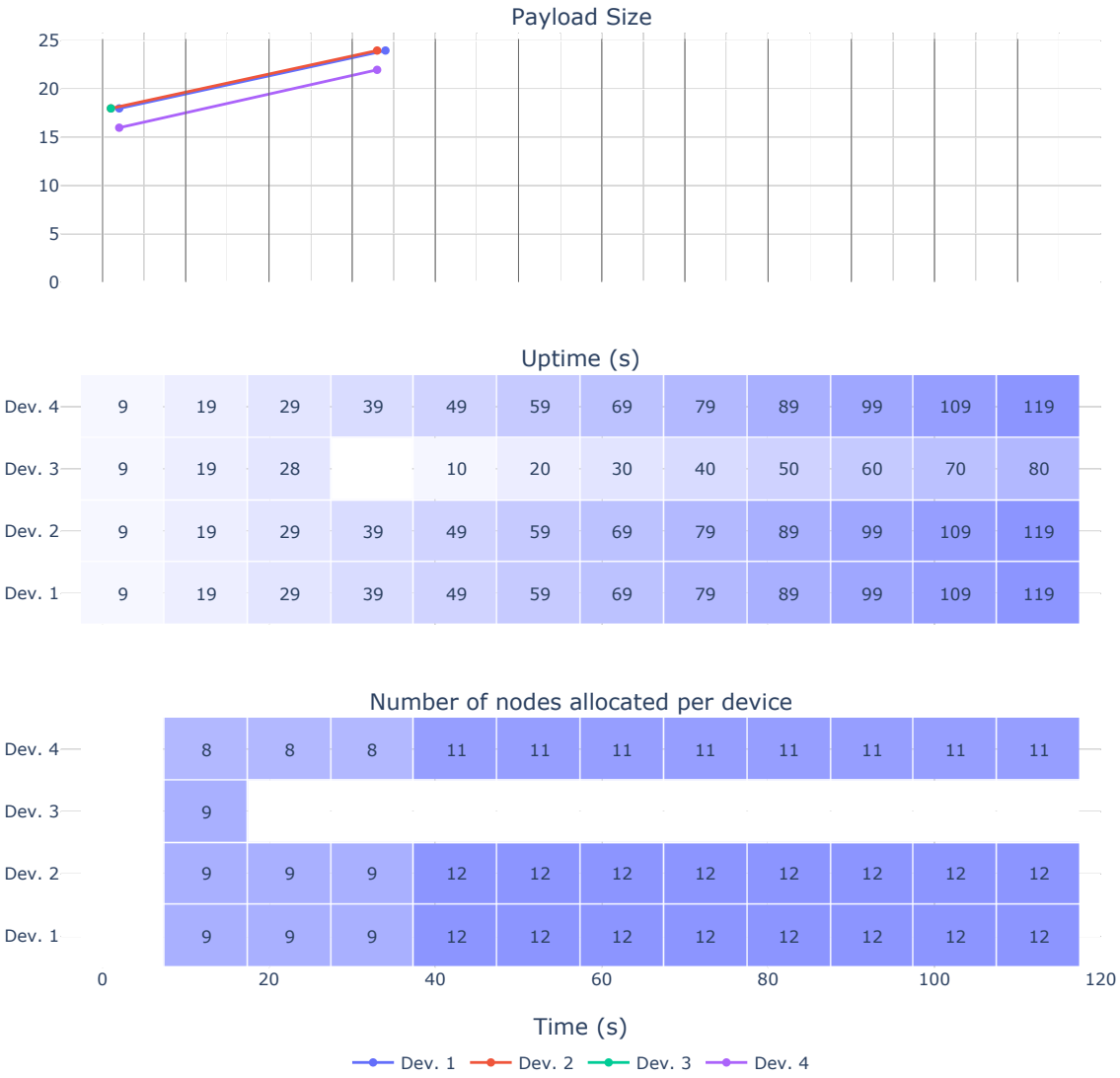


Figure 6.10: SIM-D using **v3**

1. Keep track of failed devices, when checking the health of devices or when a new device announces itself;
2. When assigning, only consider the nodes currently assigned to failed devices;
3. Take into account the number of affected nodes when choosing in which device to deploy each node;
4. In case an assignment cannot be reached, perform a full re-orchestration.

In case the orchestration is triggered by a new device announcing itself, we compare not only the new and old assignment scores but also take into account the number of affected nodes.

We start by adding a *Set* of failed devices that the *Orchestrator* keeps. When the `distributeFlow` method is called, we compute the nodes that were assigned to every failed device. If there is any, we attempt to re-distribute only these nodes. If this fails — either by the algorithm not finding a solution or by deploy failure — the system will try to perform a full re-orchestration. A simplified version of `distributeFlow` highlighting this mechanism is shown in Algorithm 2.

**Algorithm 2:** `distributeFlow()`
Entry-point method to trigger re-orchestration (simplified)

```
input : currentAssignment: Map<Device, Node[]>,
        failedDevices: Device[],
        α = 0.5,
        β = 0.5
begin
    failedNodes ← currentAssignment.filterKeys(failedDevices).values().flatten()
    failedDevices.clear()
    if failedNodes.length > 0 then
        assignment ← generateAssignment(failedNodes, 0, devices, currentAssignment)
        if assignment ≠ NULL and deploy(assignment) then
            setCurrentAssignment(assignment)
            return
        end
    end
    assignment ← generateAssignment(allNodes, 0, devices, emptyAssignment)
    if assignment ≠ NULL then
        newScore ← α· assignment.score + β· affectedNodesScore
        if newScore > currentAssignment.score then
            if deploy(assignment) then
                setCurrentAssignment(assignment)
            end
        end
    end
end
```

In addition to these changes, two more are necessary to implement our solution: (1) add a new component to the greedy heuristic that minimizes the number of affected nodes, and (2) take into account the number of affected nodes when comparing the old and new assignment scores.

Regarding the heuristic change, we add a new component to the existing ones (*cf.* Section 6.4, p. 41): *changeIndex* — the inverse of the number of affected nodes. The greater the *changeIndex*, the fewer nodes need to be stopped when the new script is deployed, which is considered to be favorable for the system. In order to calculate this value, at every step of the assignment algorithm, the number of affected nodes is either the number of nodes that the device is currently executing or 0 if it has already been assigned a new node during the current assignment process. This means that if we assign 2 nodes to a device, the number of affected nodes for the

first node will be the current number of nodes it is executing, but will be 0 for the following node because deploying 2 nodes results in the same number of affected nodes as deploying 1 node. The calculation of these values is shown in (6.4). The greedy heuristic, $h$, for each device $d$, becomes $h = 0.15 \cdot d.changeIndex + 0.15 \cdot d.vacancy + 0.7 \cdot d.stability$.

$$\begin{aligned} affectedNodes &= \begin{cases} 0 & \text{if device.newNodes} \neq \text{device.nodes} \\ device.nodes & \text{otherwise} \end{cases} \\ changeIndex &= (affectedNodes + 1)^{-1} \end{aligned} \tag{6.4}$$

In addition to changing the heuristic, another change is necessary to complete the feature. If the system applied the MSC when a device failed but performed a full re-orchestration when it re-connected, due to a better *assignment score* (*cf.* Section 6.9), the purpose of MSC would be defeated. In order to mitigate this behavior, we make a slight adjustment to how the comparison works. Instead of comparing the new and old assignment scores, the new score is calculated as the arithmetic mean between the new *assignment score* and the *affectedNodesScore*, as shown in (6.5).

$$\begin{aligned} affectedNodesScore &= 1 - \frac{\#\, affected\_nodes}{\#\, nodes} \\ newScore &= 0.5 \cdot affectedNodesScore + 0.5 \cdot score \end{aligned} \tag{6.5}$$

Although the operational impact of MSC is impossible to evaluate in this simulator due to the lack of the code execution component, we validate the orchestration strategy nonetheless. Using the scenario **SIM-D**, we repeat the experiment using **v4**, expecting that when Dev. 3 fails, only its nodes are re-assigned to the device with the least assigned nodes. Figure 6.11 (p. 55) shows that our assumptions hold. When Dev. 3 fails, its 9 nodes are re-assigned to Dev. 4 because it affects the least number of nodes, as it is only executing 8 nodes, compared to the remaining devices' 9 nodes.

## 6.9 Handling faulty assignments

In the final version, **v5**, we extend the device stability to device-node pairings in the interest of reducing device failure caused by faulty assignments.

### 6.9.1 Identified problem

There may be cases where a specific device-node pairing causes the device to crash. The system should be able to mitigate this situation by attempting not to assign this device-node combination.

In order to assess how the system responds to device-node pairings that cause device failure, we design a scenario, **SIM-E**, with 12 nodes and 4 devices, where each device fails if executing a specific node (one of the devices fails if executing any of 2 specific nodes). We then restart

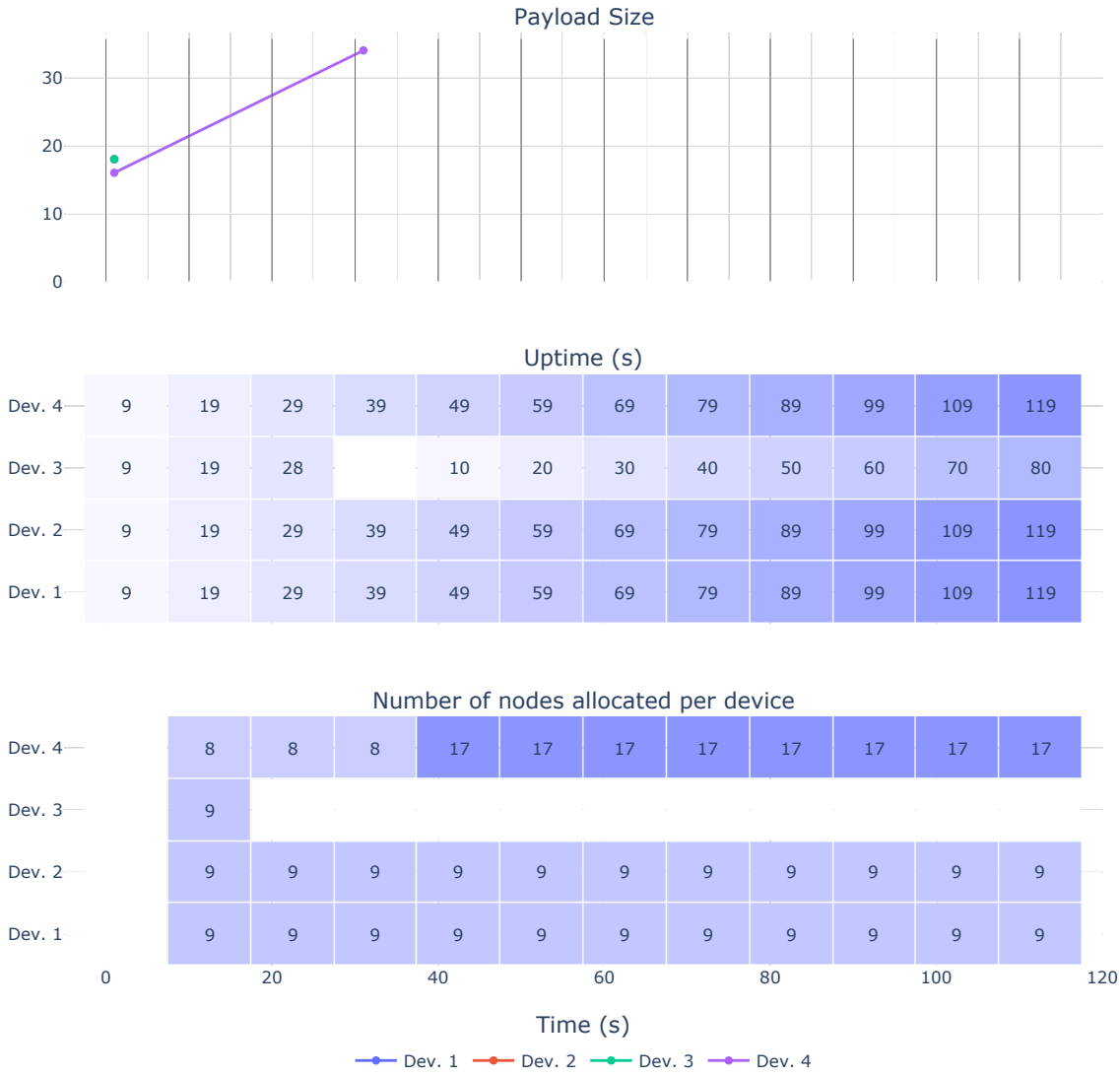


Figure 6.11: SIM-D using **v4**

all 4 devices in a random order every 40s, to allow the nodes to rotate to different devices. We expect that the system will keep assigning the nodes that cause devices to fail because there is no mechanism to prevent it.

Figure 6.12 (p. 56) shows that for most of the re-orchestrations, devices are assigned that cause them to fail. It is possible to clearly observe this behavior between 130 and 180s, where devices seem to have restarted multiple times due to being assigned a faulty node — their uptime does not increase steadily as it should.

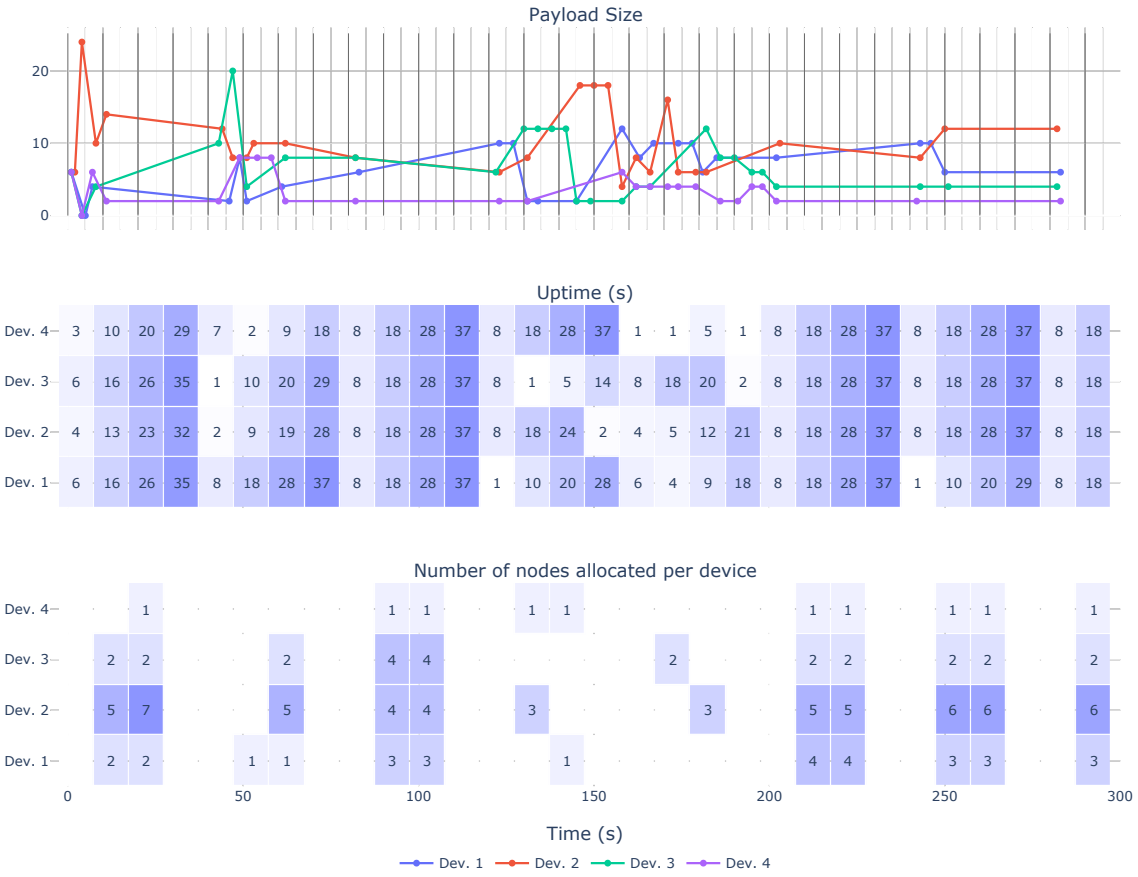


Figure 6.12: SIM-E using **v4**

### 6.9.2 Solution

Similarly to the *device stability* concept introduced in Section 6.4 (p. 41), we introduce a *device-node stability* metric, which is calculated as the MTBF of each device-node pairing. At every health check, for each node that each device is executing, its total number of *ticks* is incremented. If the device is found to have failed or has reconnected, the total number of fails for each of the nodes it was executing is incremented. Contrary to the *device stability*, we measure it in *ticks* instead of seconds because it is unknown for how long each device has been executing each node. Additionally, this metric is not as precise as the *device stability* because even though only one node might have caused the failure, all of the nodes that were being executed will have their *device-node stability* decreased. Nevertheless, as time progresses, we expect that as the system collects more data, the metric will become more precise.

When calculating the greedy heuristic for each node and for each device, instead of only using the *device stability*, we also take into account the *device-node stability*. Although the heuristic stays the same as the one presented in **v4** (*cf.* Section 6.8, p. 50), the calculation of *stability* component changes, as shown in (6.6). Similarly to *device stability*, we normalize the MTBF

value to a value between 0 and 1. If the pairing device-node has not failed, the MTBF value would be infinity; thus, it is set to 1. Otherwise, we divide the MTBF by $\alpha$ which is analogous to the `STABILITY_FACTOR` concept used for *device stability* (*cf.* Section 6.4, p. 41). Because we are combining both *stability* values, the `STABILITY_FACTOR` should be consistent. However, since the one in *device stability* is measured in seconds, and device-node MTBF is measured in *ticks*, we convert the `STABILITY_FACTOR` to its corresponding value in *ticks*. Thus, $\alpha$ is equal to the value of `STABILITY_FACTOR` in *ticks*, as shown in (6.6).

$$\begin{aligned} stability &= 0.5 \cdot deviceStability + 0.5 \cdot \begin{cases} 1 & \text{if device-node has not failed} \\ min(\frac{MTBF}{\alpha}, 1) & \text{otherwise} \end{cases} \\ \alpha &= \frac{\texttt{STABILITY\_FACTOR}}{health\_check\_interval} \end{aligned} \tag{6.6}$$

When then run the same scenario **SIM-E** and expect the system to identify the device-node pairings that are causing failures as the experiment progresses, making it less likely for devices to fail due to being assigned a faulty node.

Upon analyzing Figure 6.13 (p. 58), it is possible to notice that the system seems to have adapted better to the situation, clearly showing that recurrent assignments that cause device failure occur less often. However, in the last two orchestrations, at around 250 and 290s, the system still assigns nodes that cause Devs. 2 and 3 to fail. Although it is unclear how positive the impact of *device-node stability*, it is possible to conclude that it does not completely solve the identified problem.

## 6.10 Summary

This chapter describes the iterative methodology and the several steps that were taken to reach the final solution.

In Section 6.1 (p. 37), we outline the methodology for developing each improvement using the *Orchestration simulator* (*cf.* Chapter 5, p. 29), consisting of a 4-step process: (1) identify one or more current issues, (2) create a scenario that reproduces these issues, (3) craft improvements to mitigate the issues, and (4) re-evaluate the scenario given the new improvements and compare the results. An overview of the changes introduced in each version is also presented.

Section 6.2 (p. 38) validates the basic behvaior of the simulator. Then, Sections 6.3 (p. 38) to 6.9 (p. 54) describe, or each iteration, the identified problems and proposed solutions, alongside a discussion of the results drawn from the set of experiments that aim to validate each solution. A total of 5 iterations were developed, resulting in 5 *Orchestrator* versions, from **v1** to **v5**.

Version **v1** contains the most changes when compared to the previous version, as it is the first to deviate from the original system. In this version, a set of general enhancements are introduced that aim to solve some undesirable characteristics found in NoRDOr. Then, we present a *device stability* metric, and an *assignment score* value, that are leveraged on orchestration to mitigate the

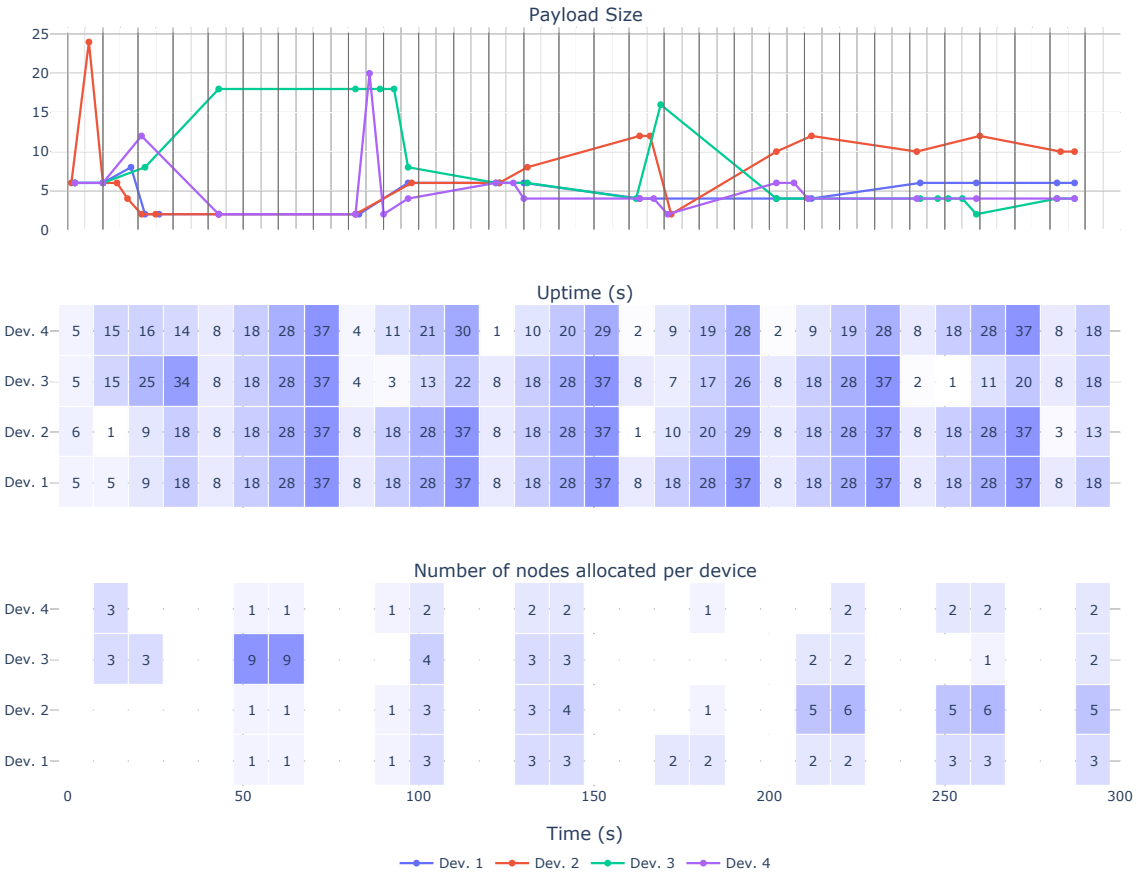


Figure 6.13: SIM-E using **v5**

impact of unstable devices on the system's resilience. Lastly, we changed the processing order of the nodes in the system in the interest of producing a balanced assignment in dynamic scenarios where some nodes strictly need to be executed in a subset of devices.

In the next version, **v2**, the original mechanism for handling memory limitations is replaced with a RAM monitoring solution that aims to eliminate the need for a trial-and-error cycle that could cause devices to fail repeatedly due to memory limitations.

Version **v3** addresses a key drawback of the greedy algorithm used for assignment. The lack of backtracking or heuristics to solve for memory limitations would prevent the system from reaching a valid assignment in cases where some devices are memory-constrained. Hence, we introduce a backtracking mechanism coupled with an *early stopping* strategy to prevent the system from halting during the greedy algorithm's brute-force attempt to find a solution in scenarios comprised of highly constrained devices.

In version **v4**, the current assignment is leveraged to produce the new one in the interest of reducing the orchestration overhead. The *Minimum Set of Changes* (MSC) assignment aims to reduce the number of nodes that need to be stopped (while the device re-installs the script) on re-orchestration by only deploying the nodes assigned to failed devices preferably to the healthy

devices executing the least number of nodes.

Finally, version **v5** extends the concept of *device stability* to the device-node pairings in order to identify and mitigate scenarios where a specific device-node pairing causes device failure.

In the next chapter, we port these improvements to the original system and provide possible limitations of the solution.

# Chapter 7

# Reference implementation



This chapter presents the reference implementation of the proposed improvements to the NoRDOr system. Firstly, the implementation is detailed in Section 7.1, and, then, its limitations are outlined in Section 7.2 (p. 65).

## 7.1 Implementation

The last version of the simulator, **v5** (*cf.* Section 6.9, p. 54), is ported into the NoRDOr system in the interest of validating the solution with real-world scenarios.

The system can be divided into four agents: (1) **Node-RED** (editor), (2) **Registry** node, (3) **Orchestrator** node, and (4) **Device** as shown in Figure 7.1 (p. 62).

Slight changes are required in the *device firmware*, but most of the changes occur in the Node-RED component of the system, encompassing the *Node-RED editor*, *Orchestrator*, and *Registry* nodes.

### 7.1.1 Device firmware

The NoRDOr Micropython firmware is altered to meet the new system's requirements, namely the communication mechanisms. Firstly, in addition to the `GET /ping` route used for the PING mechanism, a `GET /health` is added to serve the new HEALTH CHECK mechanism. This new route returns a *JSON* output containing its current state — including metrics such as uptime, assigned nodes, and available resources. Then, the `announce` message that each device uses to let the *Orchestrator* know it is online and wants to connect to the system is extended to include the available RAM. Because we cannot assign, or remove, individual nodes to/from a device, but rather the full assignment, the *Orchestrator* relies on the value of available RAM that the device reports when connecting instead of looking at real-time RAM data. This approach is taken under the assumption that when the device connects and is not executing any script, its available RAM size will be close to the available size when it deletes the script to start executing a new one.

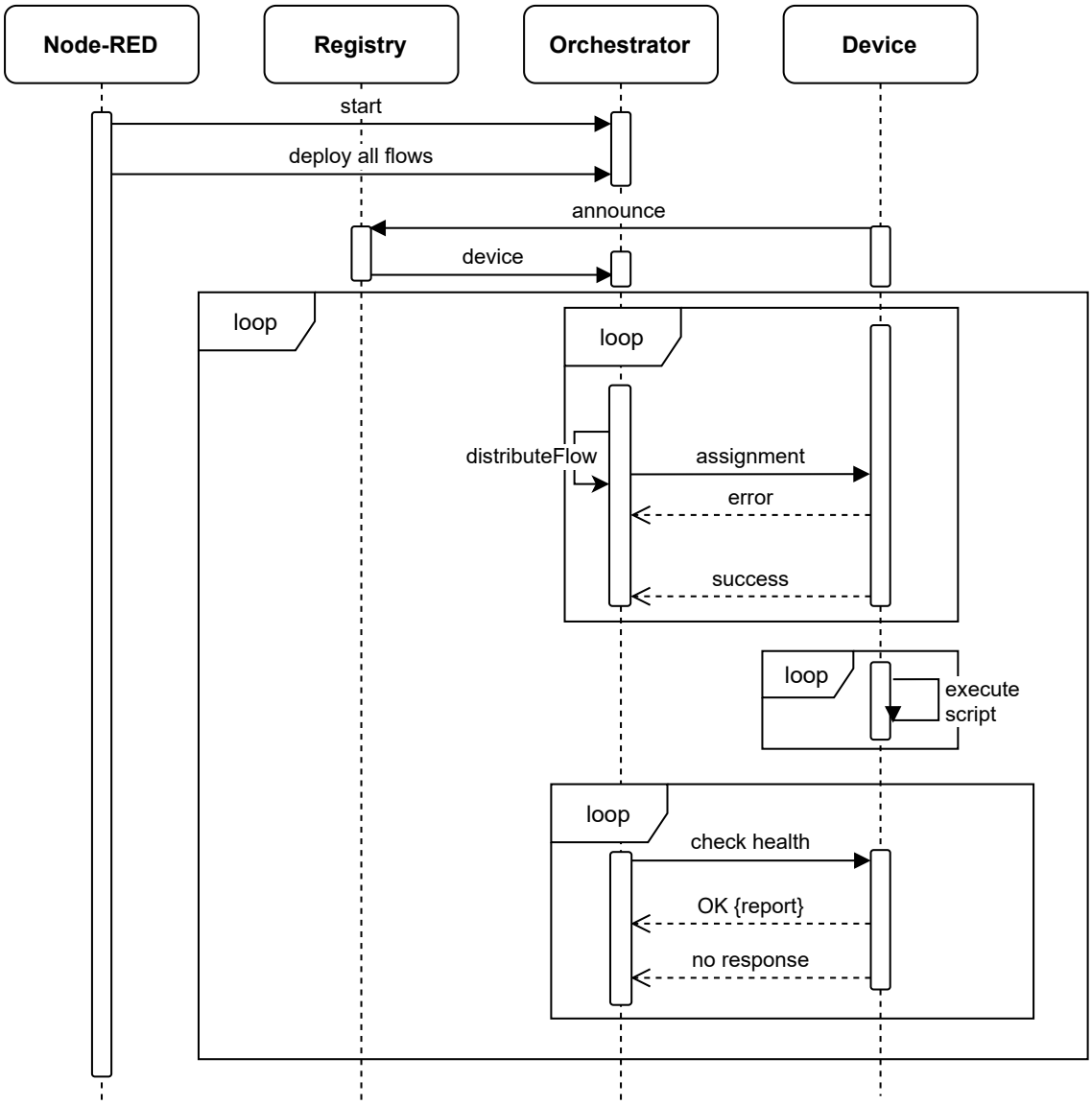


Figure 7.1: Proposed system sequence diagram

### 7.1.2 Node-RED

The Node-RED editor is responsible for deploying the flows created by the user. We assume this happens once and sets up the nodes that run on Node-RED — *Registry* and *Orchestrator*.

#### 7.1.2.1 Registry node

In the original system, the *Registry* node is responsible for keeping a record of all devices and handling (re-)connections. However, it is the *Orchestrator* that handles the PING mechanism, meaning that when it finds a failed device, the device state is updated on the *Orchestrator* but not on the *Registry*. In fact, this behavior may lead to the system assuming that some failed devices

are available when performing an orchestration, requiring it to re-orchestrate when the deployment fails.

In the proposed system, the *Registry* node acts as a *parser* — when a device publishes its announcement message to the MQTT topic, the registry processes it and sends a formatted *JavaScript Object* to the *Orchestrator* node, where the state of all devices is kept and updated.

#### 7.1.2.2 Orchestrator node

The *Orchestrator* node is the core component of the system. It is responsible for keeping the orchestration state (including the list of devices), perform the necessary orchestrations, and periodically check the health of the devices in the system. Apart from absorbing part of the *Registry* node's responsibilities, the *Orchestrator* node suffered the most changes.

**Health check**

As mentioned in Section 6.4 (p. 41), the PING mechanism is replaced with a HEALTH CHECK that allows the *Orchestrator* to gather real-time information about each device. The implementation is similar since only the route needed to be changed. Additionally, when the device responds to the HTTP request, the system updates not only the device status (to `ON/OFF`) but also other stability metrics.

**RAM heuristic**

Most of the improvements developed using the *Orchestration simulator* are ported with minimal changes to NoRDOr, with one notable exception — the *RAM heuristic*.

In the simulator, each node has a corresponding RAM size, and each device reports the exact available RAM. However, this is not applicable in a real-world scenario, as the available RAM for each device is dynamic, and statically calculating the expected RAM impact of a *code script* is not trivial.

As this is not the main focus of this work, we tackle this problem by attempting to find a simple but effective heuristic that would remove the limitations of the `memorryErrorNodes` implementation (*cf.* Section 6.6, p. 46). Upon analyzing the available characteristics from the *code script* — lines of code and size (in bytes), a correlation was found between the size of the *script* and the devices' available RAM before and after installing the *script*. There is a correlation of approximately 2, thus, we define a *RAM heuristic*, $h$, as $h(assignment) = 2 \cdot assginment$.scriptSize, with all values being in *bytes*.

At every step of the assignment algorithm, we check if the available RAM the device reported when it connected to the system is larger than the expected RAM impact of the *script*. It is worth noting that we only measured the initial impact — when the script is installed, not during the execution of the *script*.

**Orchestration strategy**

The triggers for computing an orchestration — `distributeFlow` — are twofold: (1) the

current assignment is invalid ($score = -1$) — this can be due to a correct device having failed or simply because the assignment is impossible, or (2) a device has (re-)connected. The re-orchestration flow has two possible paths (*cf.* Algorithm 2, p. 53). If the system has found a failed device and that device had assigned nodes, then it will try to assign those nodes to minimize the correct nodes that need to be stopped. However, if the system is healthy and a device connects, a full re-orchestration takes place — it is only deployed if the new score is higher than the current assignment (*cf.* Section 6.8, p. 50).

The `distributeFlow` method encompasses not only the assignment generation but also the deployment — its call graph is shown in Figure 7.2.

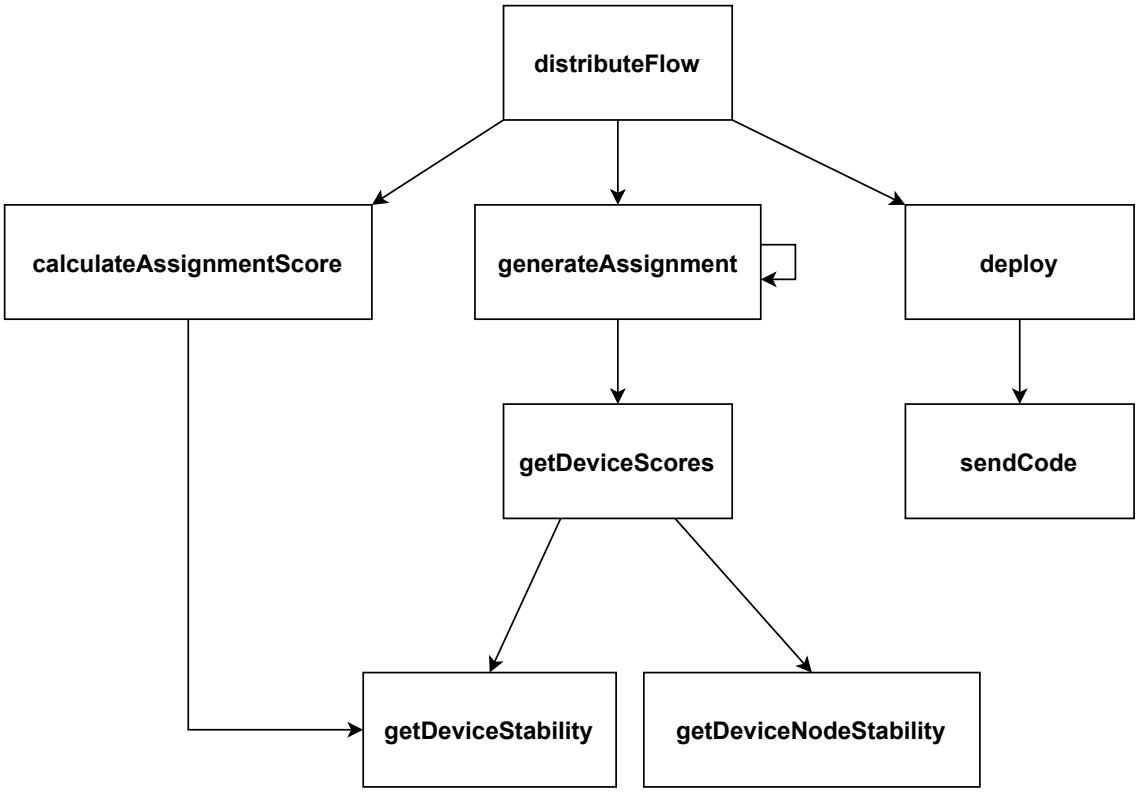


Figure 7.2: `distributeFlow` call graph

To generate the assignment, the system makes use of the recursive greedy algorithm with a backtracking mechanism — `generateAssignment` (*cf.* Algorithm 1, p. 51). The method that computes the greedy heuristic for each node is `getDeviceScores`. It can be considered the core of the assignment algorithm; thus, we provide a full pseudo-code version of it in Algorithm 3 (p. 65).

The method outputs a list of devices for every node, sorted by their *fitness* to execute the node. The devices in the system are filtered by 3 conditions: (1) available RAM to execute the node, according to the *RAM heuristic*, (2) are currently connected, and (3) are capable

of executing the node. The latter is true if the set of predicates for a given node is contained in the set of capabilities announced by a device.

The *fitness* of each device is calculated using a weighted average with 3 predefined weights. The *changeIndex* represents the inverse of the impact this assignment has on the device (*cf.* Section 6.8, p. 50) and has a weight of 0.15. With the same weight, the *vacancy* represents the inverse of the number of nodes the device is being assigned and is responsible for keeping the assignment balanced. Lastly, the *stability* is a compound metric, with a weight of 0.7, comprised of the *device stability* and *device-node stability* (*cf.* Section 6.9, p. 54).

**Algorithm 3:** `getDeviceScores()`
Greedy heuristic for node assignment

**input :** node: Node,
devices: Device[],
$\alpha = 0.15$,
$\beta = 0.15$,
$\gamma = 0.7$,
$\delta = 0.5$,
$\varepsilon = 0.5$
**output:** Device[]

**begin**
electible $\leftarrow$ $\{d \in \text{devices} \mid \text{hasMem}(d) \wedge \text{isAvailable}(d) \wedge \text{isCapable}(d)\}$ orderedBy fitness($d$)
**where**
hasMem($d$) $\leftarrow$ $d$.availableRAM >= 2$\cdot$ $\{d$.nodes $\cup$ node$\}$.scriptSize
isAvailable($d$) $\leftarrow$ $d$.status = ON
isCapable($d$) $\leftarrow$ node.predicates $\subseteq$ $d$.capabilities
fitness($d$) $= \alpha \cdot d$.changeIndex + $\beta \cdot d$.vacancy + $\gamma\cdot$ stability
**where**
changeIndex $\leftarrow$ (#$d$.affectedNodes + 1)$^{-1}$
vacancy $\leftarrow$ (#$d$.nodes + 1)$^{-1}$
stability $\leftarrow$ $\delta \cdot d$.stability + $\varepsilon\cdot$ getDeviceNodeStability($d$, node)
**end**

For the deployment component, the system employs the `deploy` and `sendCode` methods that handle the response and generate and make the HTTP request, respectively.

## 7.2 Known limitations

Comparatively, with the NoRDOr system, the new orchestration strategy introduces some limitations — the concept of *node priorities* is not taken into account, and nodes are only deployed to devices, never to the Node-RED runtime. Since these features do not contribute to our research

questions (*cf.* Section 4.4, p. 27), they were removed for simplicity. However, they should be easily plugged back into the system.

Regarding the developed improvements, there are also some limitations to be pointed out.

Firstly, in the *assignment score* (*cf.* Section 6.4, p. 41), the *total absolute deviation* is used as a metric to define the balancing of the system due to the ease of normalization of its value. However, there is one key drawback with this metric — it is not a **squared** deviation. This means that there are some cases where the distribution of nodes is slightly different, but the balancing has the same value due to a lack of relevance given to more deviated values.

Secondly, the *RAM heuristic* used is too simplistic — RAM usage is a dynamic metric, and statically calculating the expected RAM impact of a *code script* is not a trivial problem. Further research should be conducted in this direction in order to perform a better estimation of the RAM usage for a MicroPython script.

Furthermore, the greedy algorithm proved to be not very scalable, prompting us to include an *early stopping* mechanism (*cf.* Section 6.7, p. 48) to prevent the system from incurring in errors, such as a *stack overflow* error. As the system becomes more complex and more conditions are considered in the assignment process, it should be considered the possibility of replacing the greedy approach with a better-suited one.

We revisit these topics and provide possible solutions in Section 9.4 (p. 101).

## 7.3 Summary

This chapter describes the reference implementation of the improvements described in Chapter 6 (p. 37) ported into the NoRDOr system.

In Section 7.1 (p. 61), an overview of the changes the system suffered is presented, alongside an overview of the sequence of relevant events on orchestration. In addition to the specific implementation details of each improvement already described in Chapter 6 (p. 37), we detail some key changes that took place when implementing the changes in NoRDOr — namely the changes to the *device firmware* that took place mainly in the communication mechanisms and the shift in responsibility of the *Registry* node. A full overview of the final orchestration strategy is then presented, combining all the concepts introduced in each different iteration.

Then, in Section 7.2 (p. 65), we explore the limitations of the proposed solution that may impair its success. Specifically, we identified possible improvements to the *assignment score* and *RAM heuristic* that could be explored in future work.

# Chapter 8

# Evaluation



This chapter outlines the experimental process utilized for the evaluation of our solution. Section 8.1 presents the different experimental setups used throughout the experiments. Then, Section 8.3 (p. 70) describes the experiments to be conducted, detailing the flow and devices to be used in each one. Section 8.4 (p. 71) presents and analyzes the obtained results. Finally, Section 146 (p. 93) re-visits the research questions and draws conclusions from the results of the experiments.

## 8.1 Experimental setups

In order to evaluate if the proposed improvements reach their expected goals, we tested several scenarios against the baseline original NoRDOr system.

### 8.1.1 System

The experiments are performed in a Macbook Pro 2016, with an Intel(R) Core(TM) i5-6267U CPU @ 2.90GHz and 8GB of RAM running macOS Big Sur 11.3. Using Docker 20.10.6 to containerize (1) the modified version of Node-RED (NoRDOr), which is an extension of Node-RED 1.0.6, (2) a Eclipse Mosquitto MQTT Broker 2.0.11, and (3) a InfluxDB 1.8. Both virtual IoT devices and physical IoT devices are used in the experiments — the first for its flexibility in dynamically introducing different parameters to the system; the latter for its proximity to a real-world scenario of resource-constrained IoT devices, resulting in the following setups:

**ES1** 4 virtual IoT devices, each running in a Docker container, using the Unix-port of the Micropython 1.14 framework (slight variation for **ES1-B** and **ES1-E**).

**ES2** 4 Espressif Systems ESP32 devices, running a custom firmware generated using the ESP32-port of the Micropython 1.14 framework, connected to the same Wi-Fi network.

### 8.1.2 Flows

We use two different flows in the experiments, both similar to what would be expected of the system in a home/small office environment, with an additional custom scenario for a specific experiment.

**FS1** A room has one sensor that measures temperature and humidity every 5 seconds. Based on the combination of these readings, the system may trigger the AC unit to turn on or off and control its humidification mode. This results in 13 nodes to be distributed through the devices (*cf.* Figure 8.1).

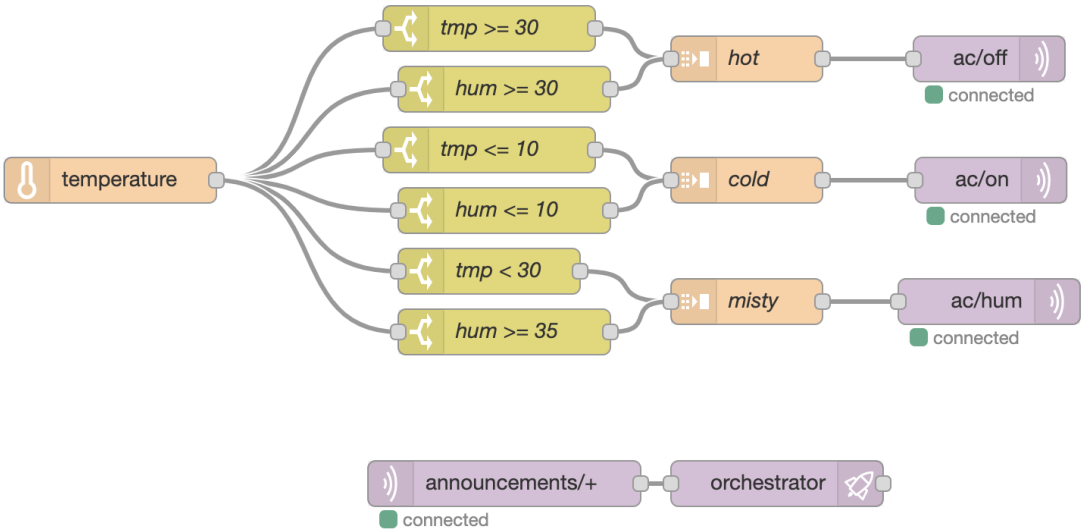


Figure 8.1: Flow Setup 1

**FS2** A room has three sensors that measure temperature and humidity every 5 seconds. Based on the combination of these readings with the other sensors' readings, the system may trigger the AC unit to turn on or off and control its humidification mode. This results in 36 nodes to be distributed through the devices (*cf.* Figure 8.2, p. 69).

**FS3** In addition to FS2, the house has 3 door sensors which report the status of each door (*open* or *closed*) every 2 seconds. If any of these sensors report that the door is *open*, the alarm is triggered. This results in 45 nodes to be distributed through the devices (*cf.* Figure 8.3, p. 70).

## 8.2 Metrics

In this section, we outline how and which metrics are collected during the experiments and their importance in the assessment of the system's performance [40].

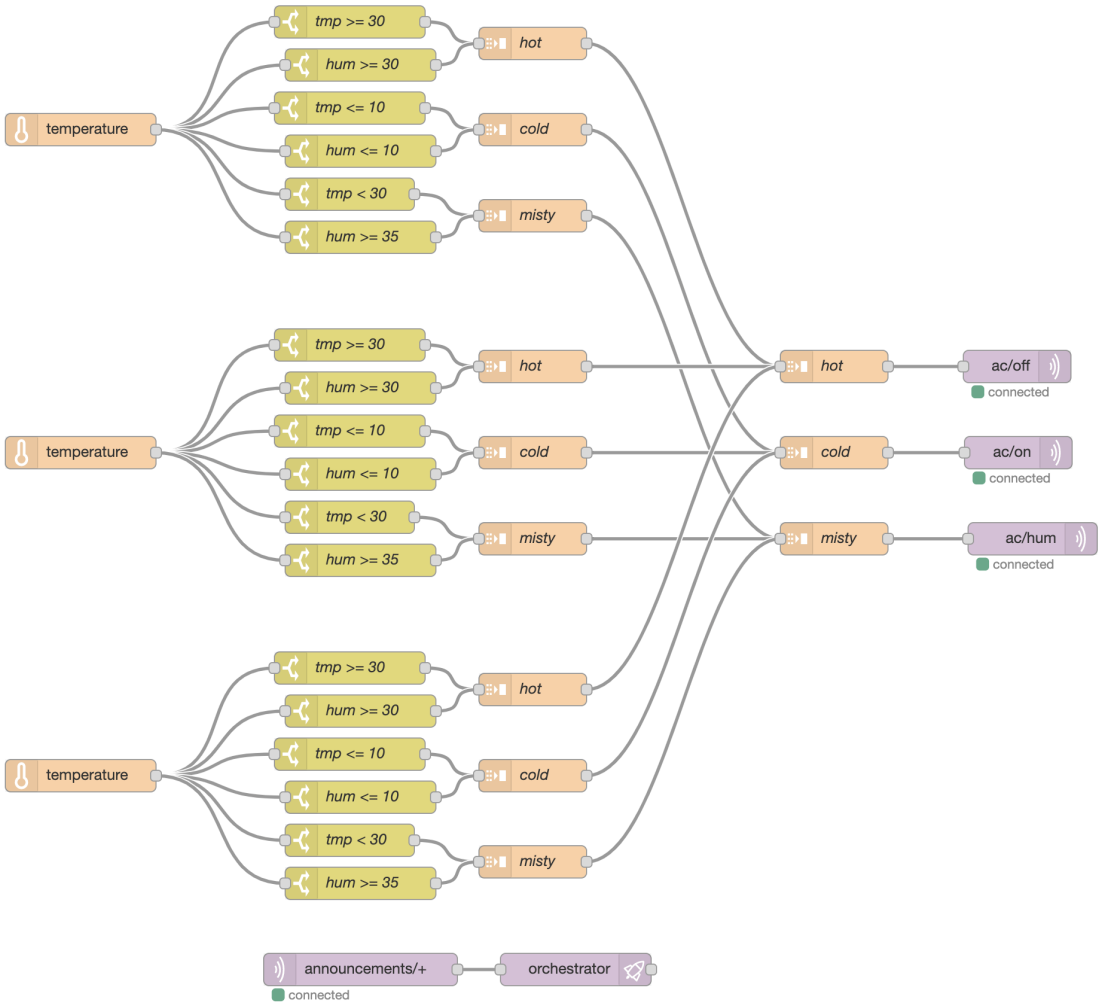


Figure 8.2: Flow Setup 2

Every 5 seconds, each device in the system publishes its state to a topic in the Mosquitto MQTT broker, which is then piped to the InfluxDB. These metrics are the following:

**Free RAM (bytes)** Allows us to evaluate the memory load of each device.

**Uptime (s)** Allows us to identify which devices are running at each point of the experiment.

**Number of assigned nodes** Allows us to assess how balanced the system is and if all of the nodes are being executed.

Additionally, another two metrics are collected at different rates. Every second, each device publishes the nodes it is currently executing, which allows us to calculate the **Average Node Uptime** of the system. Lastly, the **Payload size** that each device receives is published whenever code is deployed to the device.

**Average Node Uptime (%)** Allows us to assess the system's resilience as it measures the percentage of correct service.

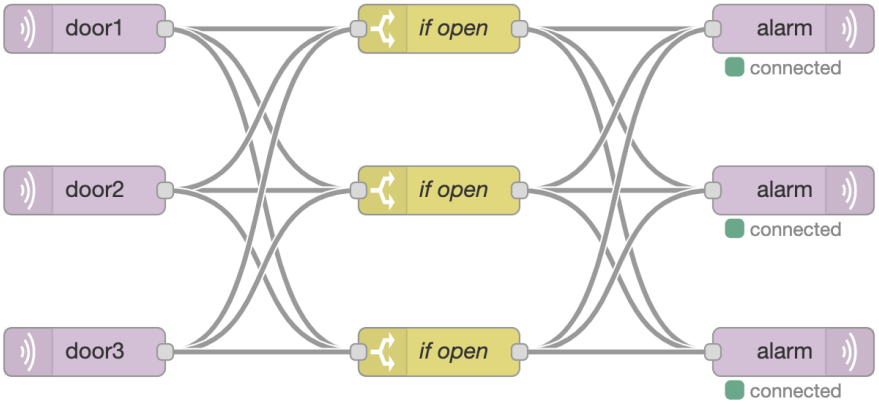


Figure 8.3: Flow Setup 3 (partial)

**Payload size (bytes)** Allows us to pinpoint when code is distributed to each device and to identify possible memory errors.

## 8.3 Experiments overview

For every experiment, the results of the proposed system are evaluated against the results of the original NoRDOr system.

### 8.3.1 ES1 experiments

**ES1-SC1** Using **FS2**, turn off one device at a time, forcing the system to re-distribute the nodes to different devices.

**ES1-SC2** Using **FS2**, restart one device at a time, forcing the system to handle the disappearance and appearance of devices.

**ES1-A** Using **FS2**, 2 of the 4 devices recurrently fail seconds after entering the system, forcing the system to handle the quick disappearance and appearance of the same devices.

**ES1-B** Using **FS3**, devices are modified to take a longer time to re-install a script. Then, 3 of the 9 devices are turned off, one at a time, forcing the system to re-orchestrate. During the re-installation of the script, a message is injected into the system, allowing us to assess if the message is processed.

**ES1-C** Using **FS2**, devices are kept at a lower RAM limit than normal, and 1 device is turned off after a short period of time, forcing the system to perform a careful re-orchestration taking into account the limited resources.

**ES1-D** Using **FS1**, devices are modified to crash if a certain node is assigned to them and are restarted in a different order throughout the experiment, forcing the system to perform a re-orchestration taking into account possible device-node failures.

**ES1-E** Using a modified **FS1**, where 6 nodes require a specific capability from devices, and only 2 devices in the system, where only 1 of them has that same capability, the system is forced to find a balanced orchestration.

### 8.3.2 ES2 experiments

**ES2-SC1** Using **FS2**, turn off one device at a time, forcing the system to re-distribute the nodes to different devices.

**ES2-SC2** Using **FS2**, restart one device at a time, forcing the system to handle the disappearance and appearance of devices.

**ES2-A** Using **FS2**, 2 of the 4 devices recurrently fail seconds after entering the system, forcing the system to handle the quick disappearance and appearance of the same devices.

## 8.4 Discussion

In this section, we present the results of each experiment in the form of graphs containing the relevant metrics for each case and interpret the results and limitations of the proposed changes. For each Experimental Setup (ES), there are two Sanity Checks (SC) that mimic the most simplistic behaviors of the system. These experiments allow us to confirm that the basic re-orchestration mechanism works as expected but also demonstrate some of the proposed improvements that affect the entire orchestration mechanism. The remaining experiments represent more specific behaviors or system configurations that warrant a more complex orchestration mechanism.

It is also worth mentioning that all of the experiments were conducted using the value of the `STABILITY_FACTOR` (*cf.* Section 6.4, p. 41) as 60s, meaning that for all purposes, a device is considered *stable* if its Mean Time Between Failure (MBTF) is higher than 60s. For short experiments, this showed to perform well; however, for real-world scenarios, this value may be increased or decreased as the developers see fit.

### 8.4.1 ES1 experiments

#### 8.4.1.1 ES1-SC1

This experiment is intended to validate the re-orchestration mechanism that should ensure that when one device fails, its nodes must be re-assigned to the other correct devices.

Since there are 36 nodes to be distributed, we expect the original NoRDOr system to assign 9 to each of the 4 devices and that as devices fail, all nodes are re-distributed, maintaining a balanced assignment. Figure 8.4 (p. 72) shows that this is indeed the case. When the first assignment occurs

around 10s, each device gets 9 nodes. Around 40s, Dev. 1 is turned off, and the system performs a full re-orchestration, assigning 12 nodes to each of the 3 available devices. Around the time of this assignment, Dev. 2 is turned off; thus, the graph for the number of nodes does not show the 12 nodes assigned to this device. The system performs a new orchestration to cope with this failure at 80s, assigning 18 nodes to the remaining devices. At around 85s, Dev. 3 is turned off, and all 36 nodes are assigned to Dev. 4.


Figure 8.4: ES1-SC1 in NoRDOr

Comparatively, we expect the new system to have a slightly different behavior in terms of balancing due to the *MSC assignment* (*cf.* Section 6.8, p. 50) — as devices fail, only their nodes are re-distributed, and this is done in a way that minimizes the number of affected nodes, *i.e.*, the nodes should be assigned to the devices with the least number of assigned nodes. The results in Figure 8.5 (p. 73) match our expectation. The first assignment occurs at around 10s and results

in a balanced orchestration, with 9 nodes running in each device. When Dev. 1 is turned off at around 40s, its 9 nodes are re-assigned to Dev. 2, totaling 18 nodes. When this device is turned off at around 60s, the 18 nodes are then re-assigned to Dev. 3, which is turned off a few seconds later at around 80s. Finally, the 27 nodes from Dev. 3 are re-assigned to the remaining device.

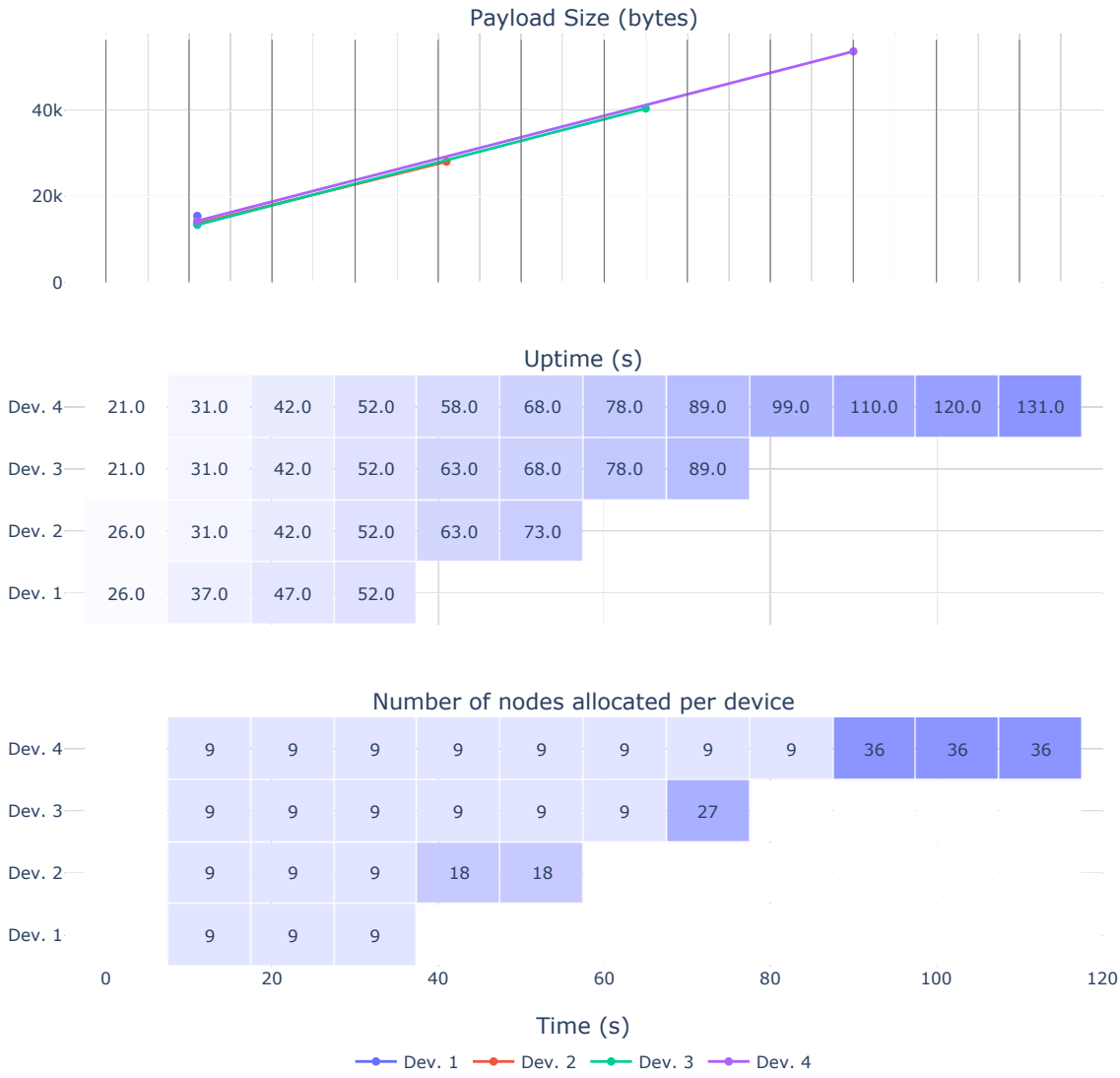


Figure 8.5: ES1-SC1 in proposed system

#### 8.4.1.2 ES1-SC2

Contrary to the previous experiment, devices are restarted instead of turning off for the remainder of the experiment. This allows us to evaluate how the system deals with a device disappearing and then re-appearing.

It is expected that, in both systems, the first assignment yields 9 nodes for each device. In NoRDOr, we expect that every time a device is turned off, all of the nodes are re-distributed to the remaining devices, and as soon as it is turned back on, another re-orchestration will occur, re-distributing the nodes through the 4 devices. Because the devices are manually turned off, the NoRDOr system does not consider it a failure, and thus, devices that have failed can still be assigned the same number of nodes.

By taking a look at the number of nodes graph in Figure 8.6, we can confirm that when a device is turned off, 12 nodes are allocated to each device, and when it is turned back on, the system assigns 9 nodes to each device. This leads to several full re-orchestrations in the system and, consequently, a considerable number of payloads being sent.

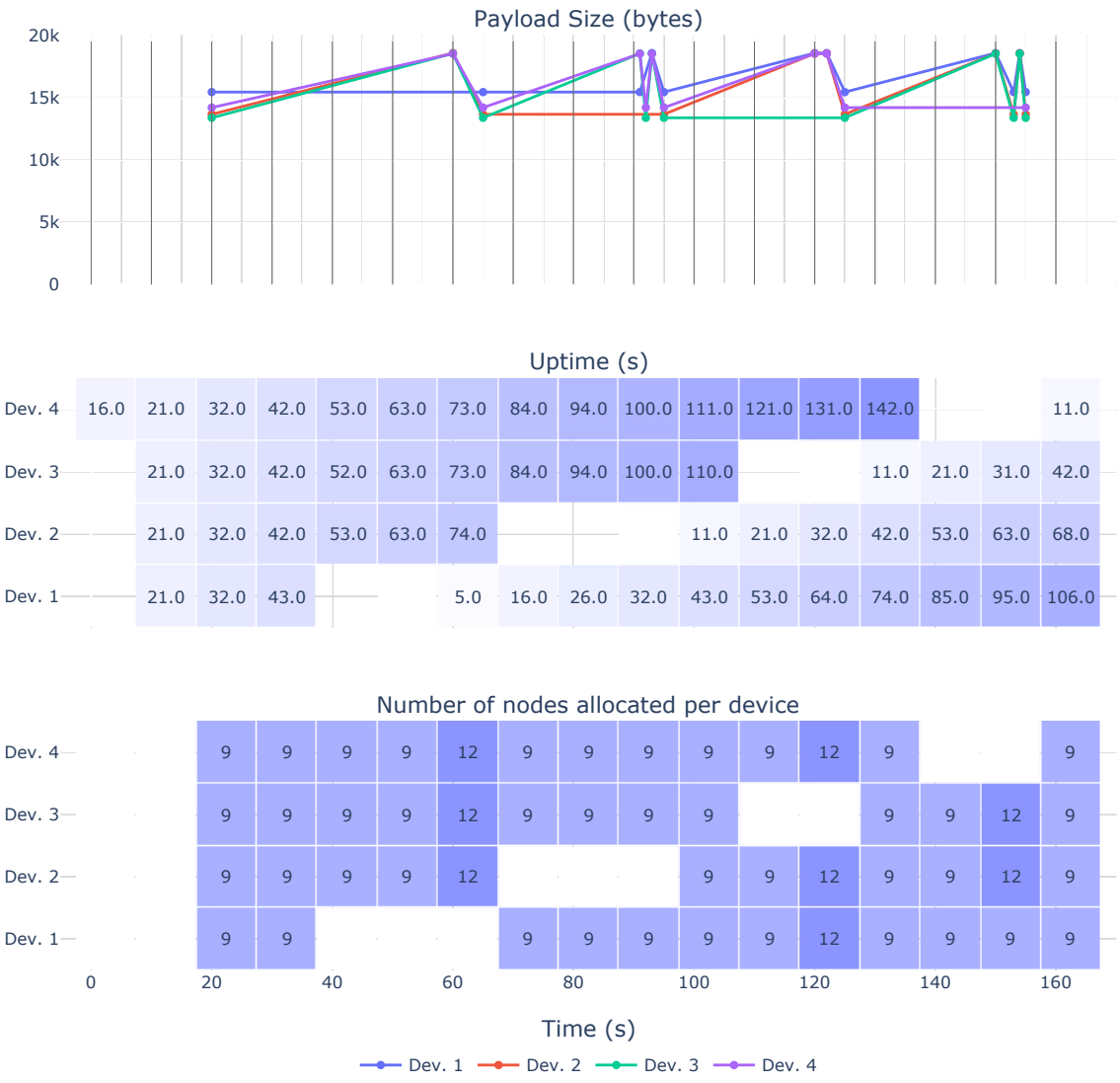


Figure 8.6: ES1-SC2 in NoRDOr

On the other hand, we expect the proposed system to use the *MSC assignment* to assign the

nodes of failed devices to other available devices instead of performing a full re-orchestration. Moreover, when a device is turned back on, it is expected that the system will only re-orchestrate if it finds a new orchestration with a better score than the previous.

When analyzing the results in Figure 8.7 (p. 76), we noticed that when the first device, Dev. 1, is turned off, its nodes are re-assigned to Dev. 2, leaving it with 18 nodes, and when Dev. 1 is turned back on, it is not assigned any nodes. However, for the following device restarts, the system has a slightly different behavior. When Dev. 2 is turned off, its 18 nodes are assigned to Dev. 1 because it had 0 assigned nodes, meaning it was essentially a *free* assignment, *i.e.*, it did not affect any running nodes. When the Devs. 3 and 4 are restarted, we see a similar behavior — the system takes advantage of the last device that has joined the system and has 0 nodes assigned to it to deploy the failed nodes.

It is also worth noting the sheer difference in payloads sent compared to the NoRDOr system. This is justified by the MSC assignment and the introduction of a conditional deployment mechanism that will not send the payload if the device is already executing it.

#### 8.4.1.3 ES1-A

The instability of IoT devices can considerably hinder the system's resilience, as many nodes can be assigned to unstable devices. Hence, the *device stability* metric is introduced in Section 6.4 (p. 41) as a counter-measure, providing the system with an actionable stability score for each device that can be leveraged in the assignment mechanism. In the interest of evaluating the effectiveness of this measure, we test both systems in a scenario where some devices are highly unstable.

It is expected that, in the original system, every time a device re-enters the system, a full re-orchestration is triggered, even if that device has failed recurrently.

Upon analysing Figure 8.8 (p. 77), we can conclude that our assumption was correct. During the first assignment, all devices are available, and 9 nodes are assigned to each device. Devs. 2 and 3 are turned off shortly after that, leaving the system only executing a total of 18 nodes. When the system notices that the devices have failed, a new assignment is calculated for the 2 available devices — this is observable in the payload graph at around 32s. However, right after this, the 2 failing devices re-connect, and a new assignment is calculated, which includes them. It is unclear if they failed before confirming the reception of the assignment, but it is likely that at least one of them died after confirming the reception of 12 nodes around 50s. From 60s to 80s, there is a period of stability in the system, with the 2 correct devices executing 18 nodes each. After this, there seems to be a similar situation as the previous assignment, causing the system to run only part of the 36 nodes until the end of the experiment.

In the proposed system, we expect to not encounter the same problem as in the NoRDOr system due to the inclusion of the concept of the *device stability* metric in the orchestration mechanism. The failing devices should not be assigned any nodes when re-connecting to the system because they are considered unstable.


Figure 8.7: ES1-SC2 in proposed system

Figure 8.9 (p. 78) shows, indeed, that our assumption holds. In the first assignment, around 10s, all the devices are included, causing the system to distribute 9 nodes to each device. However, Devs. 3 and 4 fail shortly after being assigned the nodes, and the system moves the failed nodes to Dev. 1. The failing devices re-connect several times, but a new orchestration is never triggered because of their low stability. In order to ensure that this is not happening due to the system's reluctance of re-orchestrating without a device failure, Dev. 2 is turned off at around 100s. Then, its nodes are not distributed to the failing devices, which both are online and have 0 nodes assigned. Instead, they are assigned to Dev. 1 because of its greater stability, despite its already high number of nodes. In this case, the system is running all 36 for most of the experiment.

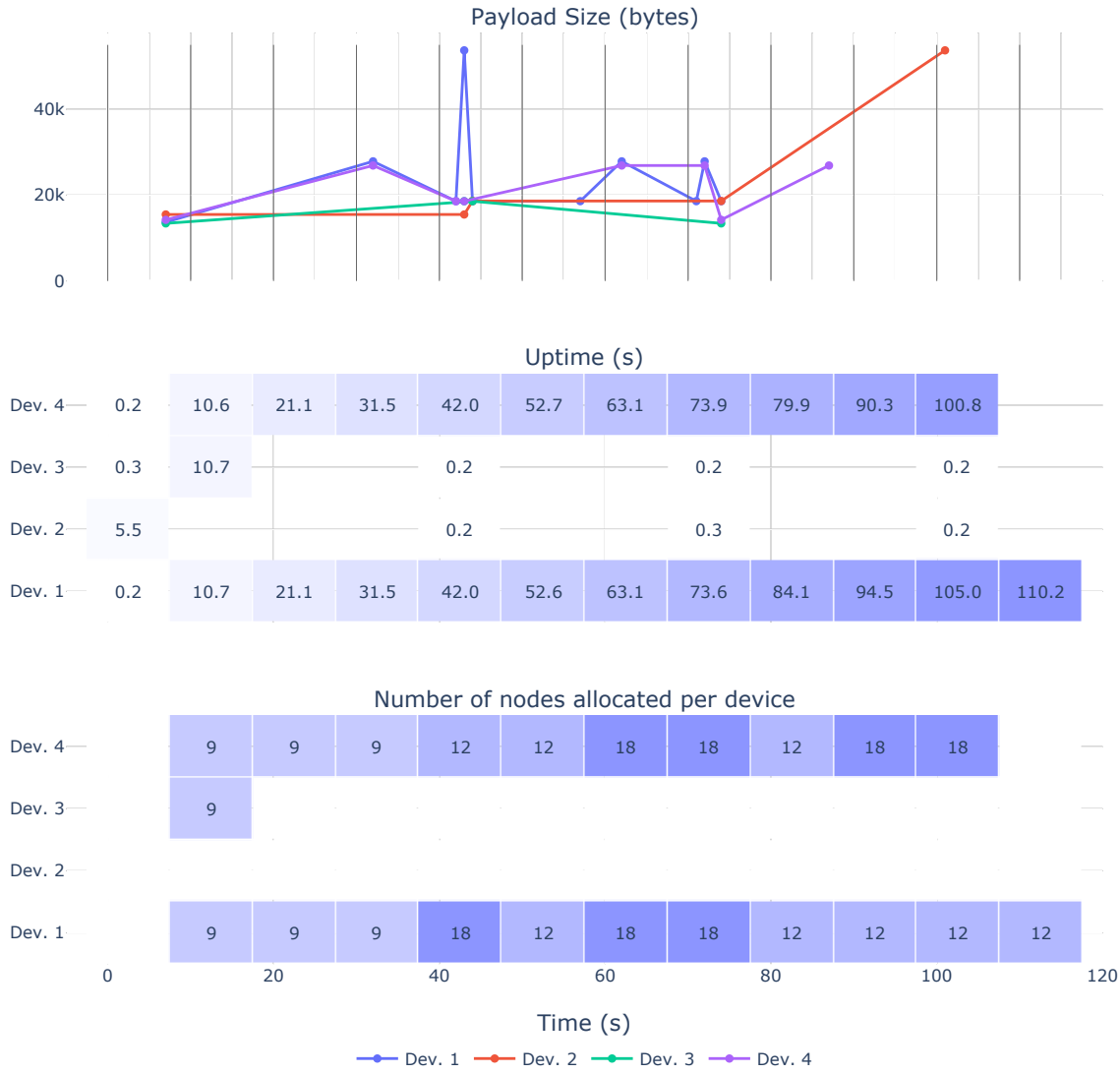


Figure 8.8: ES1-A in NoRDOr

#### 8.4.1.4 ES1-B

Some IoT systems may be critical systems — the loss of one measurement can have a considerable impact. The designed scenario — **FS3** (*cf.* Figure 8.3, p. 70) — mimics a critical door alarm system comprised of 3 door sensors. These sensors are represented by *MQTT IN* nodes, and the measurements are manually injected. If any of these door sensors receives the message `"open"`, then the alarm is triggered. The logic nodes — *if* nodes that check if the door is open — and the alarm nodes — *MQTT OUT* nodes that trigger the alarm — are replicated and strongly connected to increase the likelihood that these are deployed to correct devices.

In this experiment, only 3 of the 9 devices can execute the door-alarm nodes, and every 50s, one of the other devices is turned off, causing the system to re-orchestrate. The devices' firmware

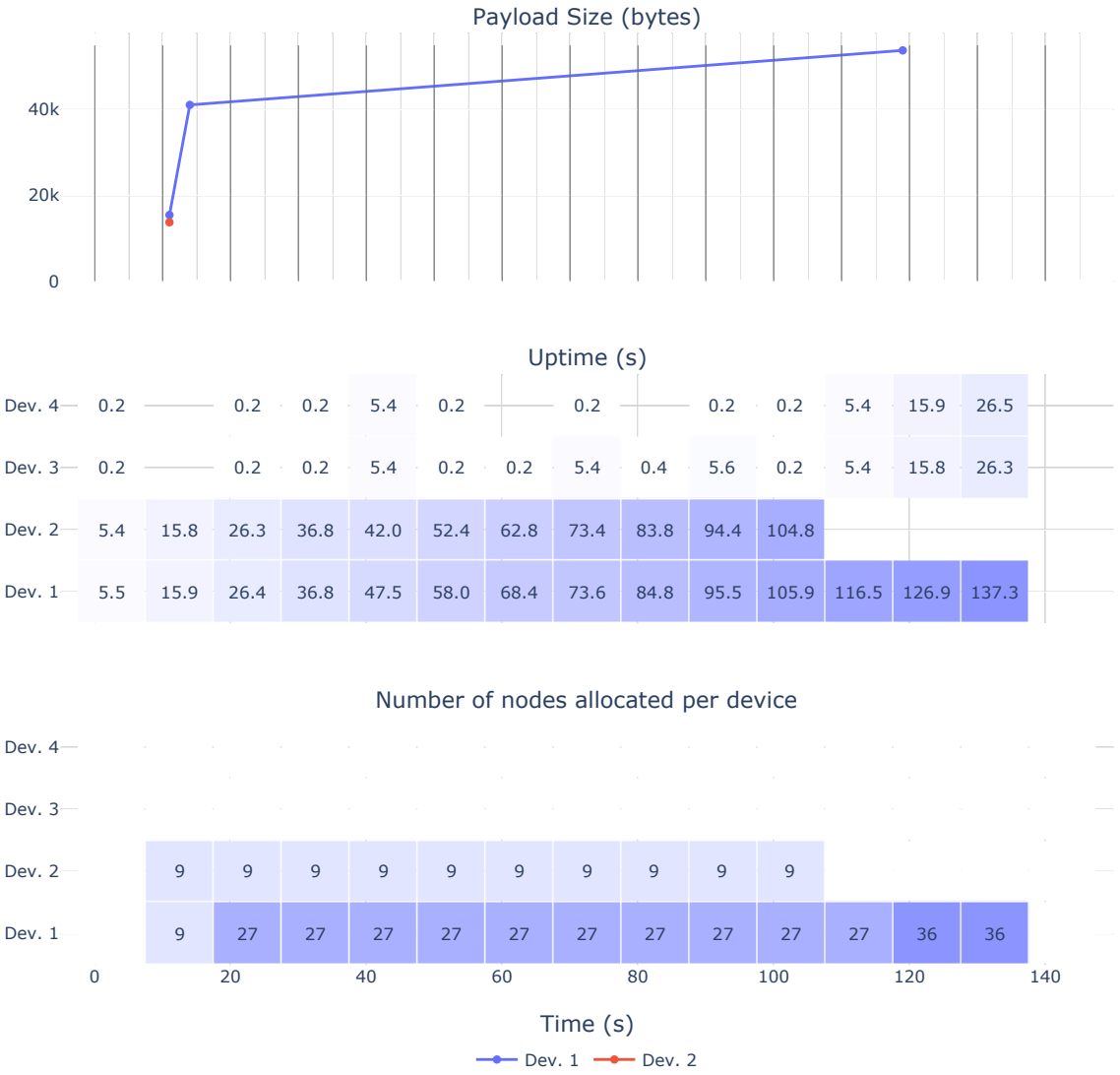


Figure 8.9: ES1-A in proposed system

was modified to take 10s to re-install a script, allowing us to simulate real-time measurements entering the system during this time. Particularly, we inject a `"open"` message to each of the door-alarm nodes when devices are re-installing the script.

With this experiment, we intend to evaluate the impact of a full re-orchestration on critical systems, compared to the impact of a *MSC orchestration*, by monitoring the relevant topics and assessing if the alarm is triggered when a door sensor reports itself as open. We represent this data in the *Door sensors and Alarm output* graph — where 1 represents the `"open"` message for doors or that the alarm was triggered, and 0 represents either a `"closed"` message for doors or that the alarm was not triggered.

We expect that, in the NoRDOr system, because the system performs a full re-orchestration every time, causing all devices to re-install the script, some messages may be lost. This should be

visible when monitoring the alarm that is not expected to be triggered for all the "open" messages injected in the door-alarm topics.

In fact, Figure 8.10 shows that none of the injected "open" messages triggered the alarm. When Dev. 6 fails at around 80s, its 5 nodes are re-distributed to 5 other devices, but all devices re-install the script, causing the injected message in Door 1 to not trigger the alarm. Similarly, at 130 and 180s, the re-installation causes all devices to stop executing their nodes, and, thus, they miss the "open" messages, causing the alarm never to be triggered.

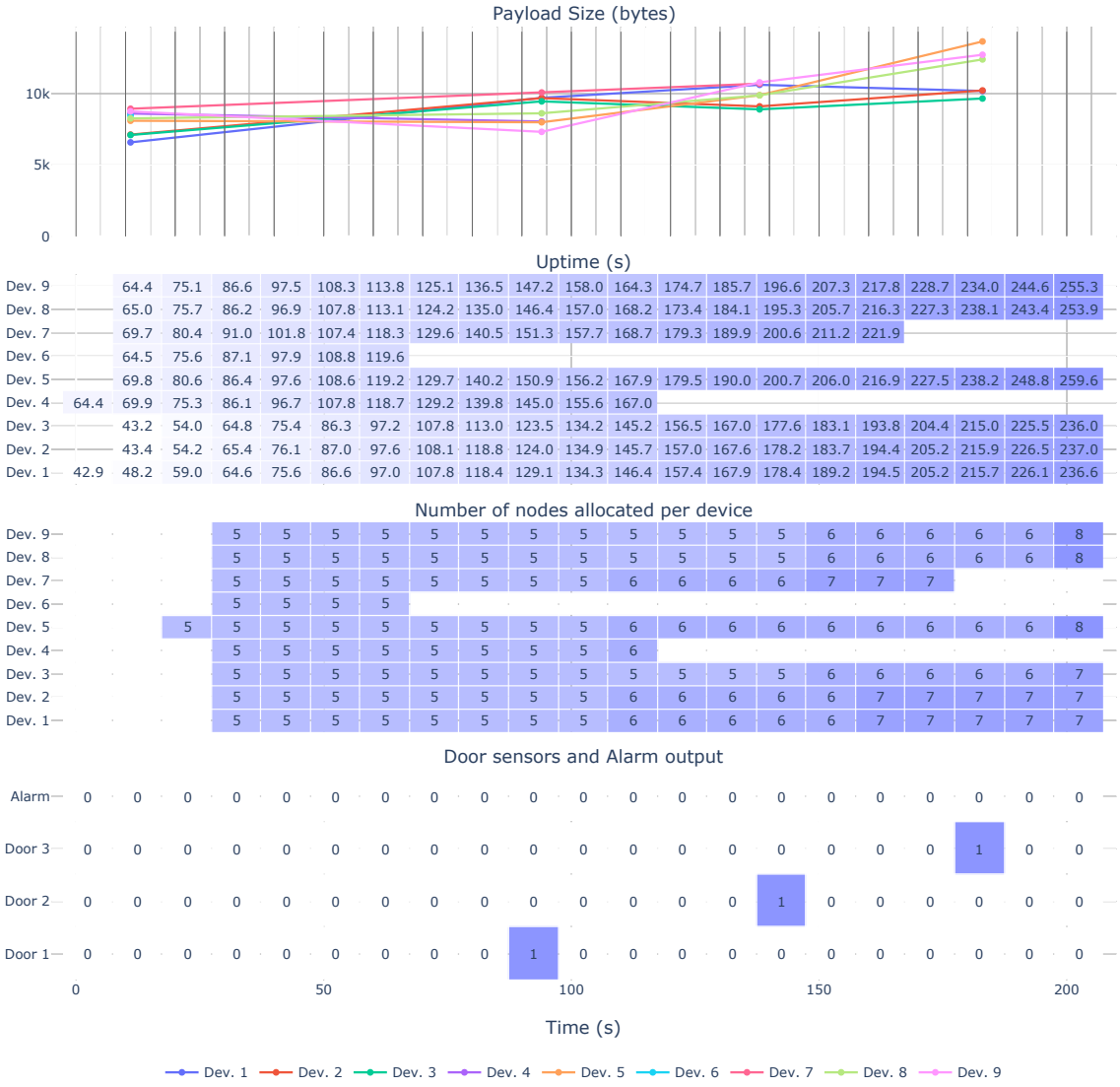


Figure 8.10: ES1-B in NoRDOr

On the other hand, we expect that the proposed system will be able to process the door-alarm "open" messages because when one device fails, only its nodes are deployed, and the number of devices needing to re-install the script is minimized. It is still possible that not all messages are

caught, in case that the devices that are chosen to be assigned the failing nodes are, in fact, the ones with door-alarm sensors.

Figure 8.11 (p. 81) shows that all of the injected `"open"` messages are caught, and the alarm is always triggered. At around 60s, when Dev. 4 fails, its 5 nodes are re-distributed to Dev. 7. During this re-installation, Door 1 is reported as open, and the alarm is triggered. At around 120s, Dev. 5 fails, and its nodes are re-distributed to Dev. 6. Door 2 is reported as open, and the alarm is also triggered — the graph shows some latency between the door-alarm sensor and the alarm, but it is considered to be within the acceptable range. Finally, at around 170s, Dev. 9 fails, and its nodes are re-distributed to Dev. 8. Again, Door 3 is reported as open, and the alarm is triggered. Because the alarm was triggered on all three occasions, we conclude that the devices containing door-alarm senors were never re-assigned nodes and were able to execute their function for the whole experiment.

#### 8.4.1.5 ES1-C

Memory limitations in IoT devices are common and should be considered by the system when generating its assignment in order to minimize the possibility of causing a device failure due to memory.

In the interest of comparing the approach of both systems to the RAM limitations of edge devices, we design an experiment where all devices are RAM-limited — 3 of them are limited to 50kB of RAM, while the fourth is limited to 20kB. This limitation strictly applies to installing the code script and not to its execution due to technical limitations of the UNIX-port of the MicroPython framework.

When one of the less RAM-constrained devices is turned off, we expect the systems to re-orchestrate, taking into consideration the limitations of the remaining devices, and pay special attention to the more RAM-constrained one.

The original system is expected to perform a full re-orchestration when one device is turned off, maintaining the balance in the assignment, which may attribute a larger amount of nodes to the constrained device than it can handle, causing it to crash.

Figure 8.12 (p. 82) shows that after the original assignment of 9 nodes to each device, Dev. 4 is turned off at around 40s, causing the system to re-orchestrate. When this happens, Dev. 3 — the more RAM-constrained device fails repeatedly. This is likely due to its RAM limitations, as we can see that the payload data points for this device firstly increase, causing it to fail, and then start to decrease as the system employs the `memoryErrorNodes` metric to determine the device's limitations. As a result of these failures, the system generates 2 extremely unbalanced assignments that cause Devs. 1 and 2 to crash also due to memory limitations. When then turn Dev. 3 back on, at around 90s, the system returns to the initial assignment of 9 nodes per device. Finally, we turn off Dev. 2 at around 100s, causing the system to assign 12 nodes to each of the available devices, leading to yet another crash by Dev. 3 due to memory limitations. It is unclear why the `memoryErrorNodes` mechanism did not prevent this assignment as it had previously determined it would lead to device failure.

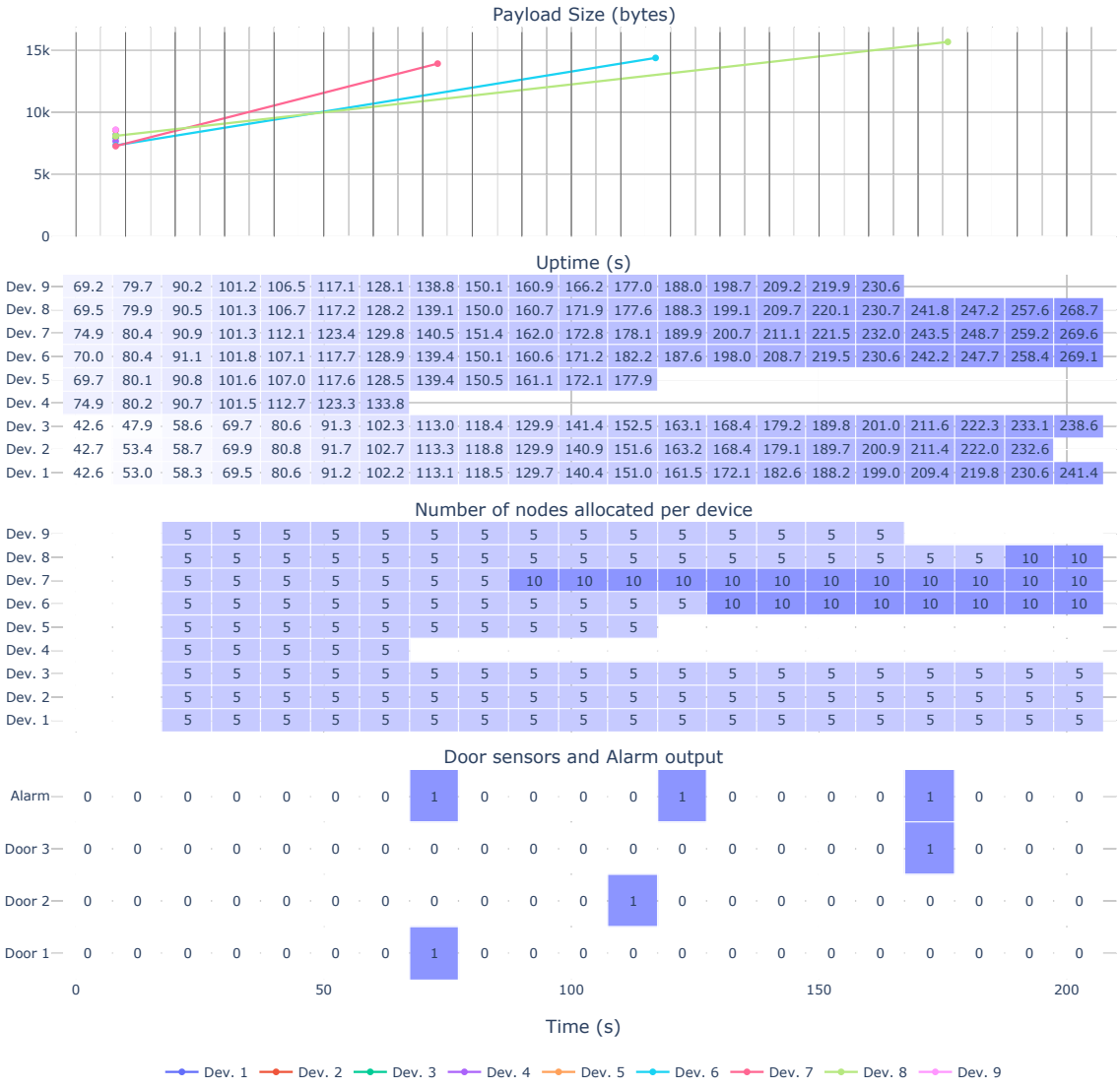


Figure 8.11: ES1-B in proposed system

On the contrary, we expect the proposed system to not provoke any memory errors in the constrained devices, as it leverages the available RAM size they have reported when connecting to the system.

Upon analysing Figure 8.13 (p. 83), we can see that the first assignment is not balanced, as the more RAM-constrained device is only assigned 6 nodes. This means that even though the RAM heuristic is working, it is likely too restrictive, as we saw the same device handling another 3 nodes without any memory limitations when using the original system. When we turn off Dev. 4, at around 30s, its 10 nodes are not deployed using the *MSC assignment* as it would cause devices to crash. The system generates an assignment where Devs. 1 and 2 get 15 nodes each, and Dev. 3 maintains its assignment, allowing the system to provide correct service continuously. When Dev. 2 is turned off at around 90s, its 15 nodes are re-assigned to Dev. 4, which is already back in the


Figure 8.12: ES1-C in NoRDOr

system.

We can conclude that monitoring the device's RAM is an effective measure to prevent failures due to memory limitations, providing the system with information that it can leverage on orchestration, thus increasing the system's resilience. On the other hand, it is also noticeable that the developed heuristic needs tuning as it may be over-restrictive.

#### 8.4.1.6 ES1-D

IoT devices are likely to fail due to memory limitations or hardware failures which may be triggered by a specific node being assigned to them. In the interest of evaluating the effectiveness of

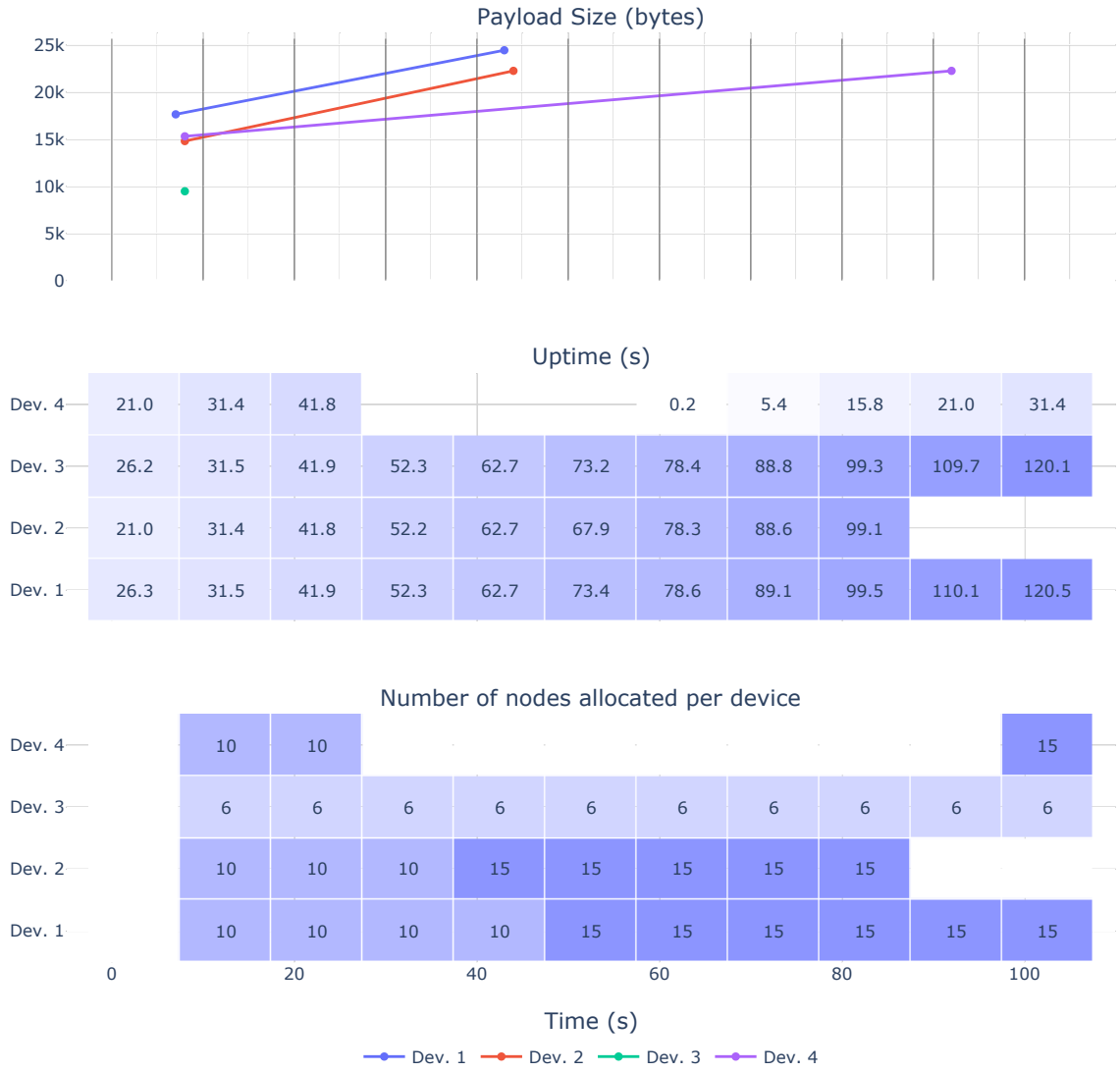


Figure 8.13: ES1-C in proposed system

the *device-node stability* metric introduced in Section 6.9 (p. 54), we test if the system can identify and mitigate the specific device-node pairings that cause them to crash.

Every 50s, we restart all devices in a random order in an attempt to rotate the node assignments, allowing devices to come into contact with more nodes and allowing the system to collect data on which assignments cause instability for each device.

We expect that the NoRDOr system keeps assigning the faulty parings as it possesses no mechanism to detect or prevent these faults. Figure 8.14 (p. 84) shows that in all assignments, the devices were assigned nodes that caused them to crash. This is observable by the growth in uptime from one data point to the next — since data is aggregated for every 10s, when the uptime data does not increase by $\tilde{1}$0s in the next data point, we know that the device has restarted in that time.

On the other hand, the changes introduced in the proposed system should mean that as the

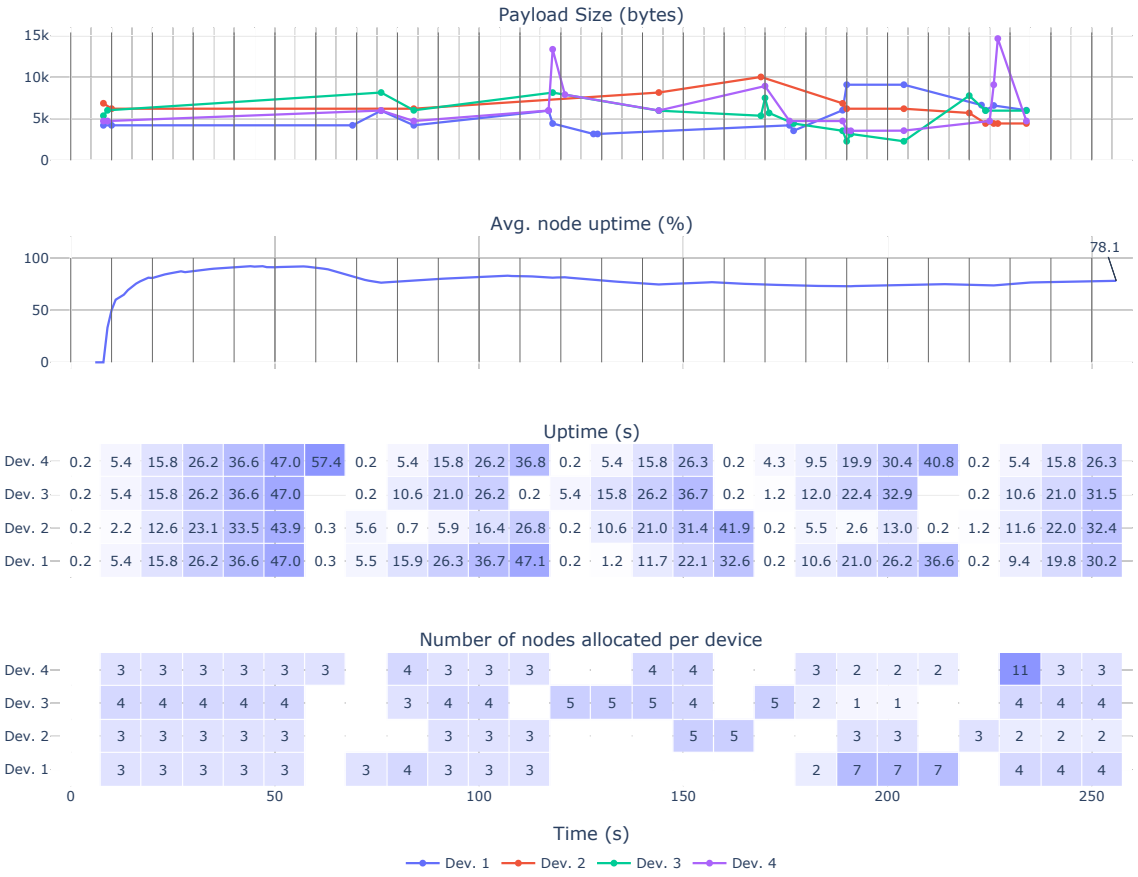


Figure 8.14: ES1-D in NoRDOr

experiment progresses, the system gathers more data about which device-node pairings cause failure and should be able to prevent it from happening so often. However, as the results show in the simulator experiments (*cf.* Section 6.9, p. 54), a noticeable improvement is unclear. Figure 8.15 (p. 85) confirms the latter assumption. Devices are assigned nodes that cause them to fail in most of the re-orchestrations. It is expected that, in the last assignment, at around 220s, the system should have gathered enough data to prevent these faulty pairings, but, in fact, devices are still assigned those nodes and failed shortly after. More noticeably, Dev. 1 is assigned the same nodes multiple times that cause it to fail more than once.

We can conclude that the benefits of the *device-node stability* metric are very reduced if any. As the Average Node Uptime of both systems is roughly the same, this feature seems not to favor neither hinder the system's resilience.

#### 8.4.1.7 ES1-E

In complex scenarios, we may want or need to restrict the executing of some nodes to some devices — this may be due to specific sensors being connected to these devices or because the location of the device has an impact on the nodes it can execute. So, there may be cases where a big portion of

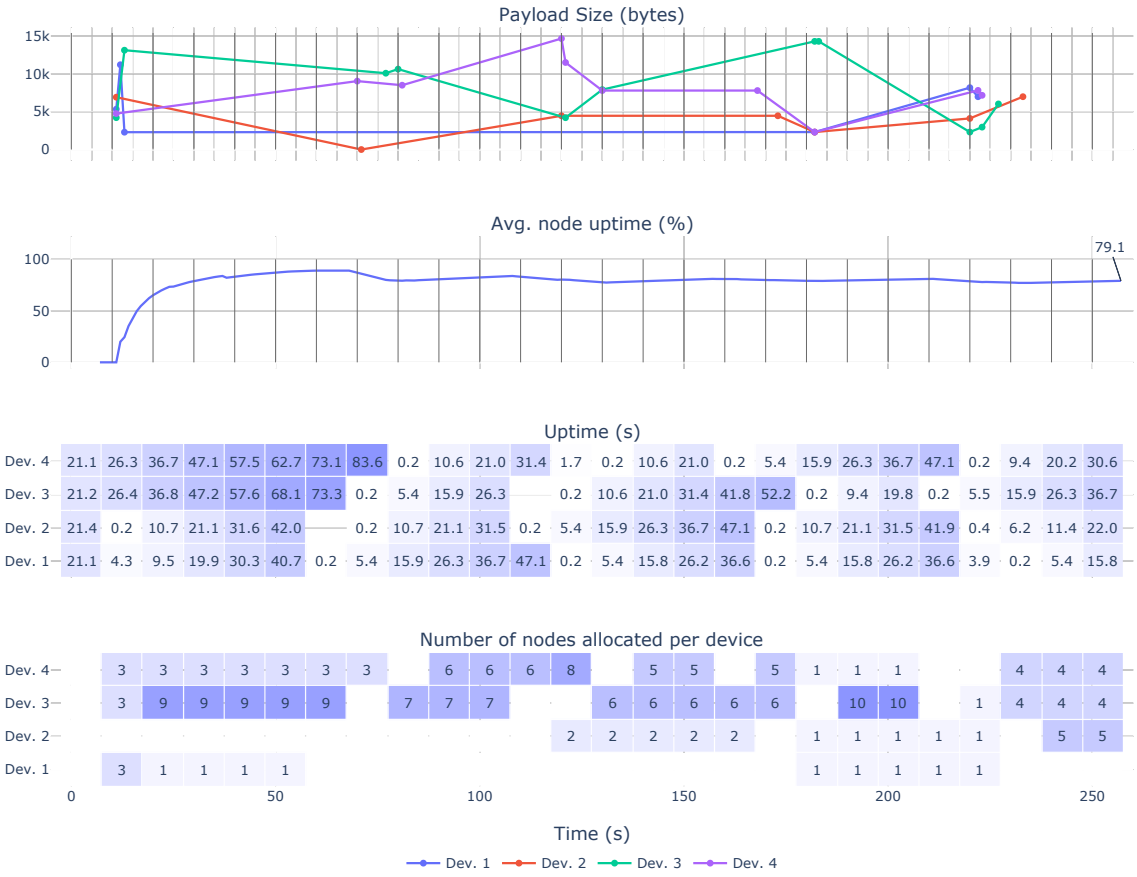


Figure 8.15: ES1-D in proposed system

nodes need to be executed in a certain subset of devices. Additionally, the available devices change dynamically, meaning that a balanced assignment depdents on the currently availabe devices.

This experiment aims to assess the impact of the order in which nodes are processed on the balancing of the system. In order to do this, we modified the last 6 nodes in the **FS1** (*cf.* Figure 8.1, p. 68) flow to require the capability *c1*, which only 1 of the 2 devices possesses.

We expect that the original NoRDOr system generates an unbalanced assignment by processing the less specific nodes first and then being forced to assign the last 6 more specific nodes to the same device.

Figure 8.16 (p. 86) shows that the system generated a quite unbalanced assignment of 9 to 4 nodes. This was likely caused by the following: (1) the first 7 general nodes are distributed 3 to 4 to Devs. 1 and 2, respectively, and (2) the remaining 6 nodes can only be deployed to Dev. 1, so it gets a total of 9 nodes.

In the proposed system, it is expected that this unbalance does not happen due to the processing order of nodes — sorted in ascending order by their specificity, *i.e.*, the number of currently available devices that can execute them.

As is shown in Figure 8.17 (p. 87), the nodes are assigned 7 to 6, which is as balanced as

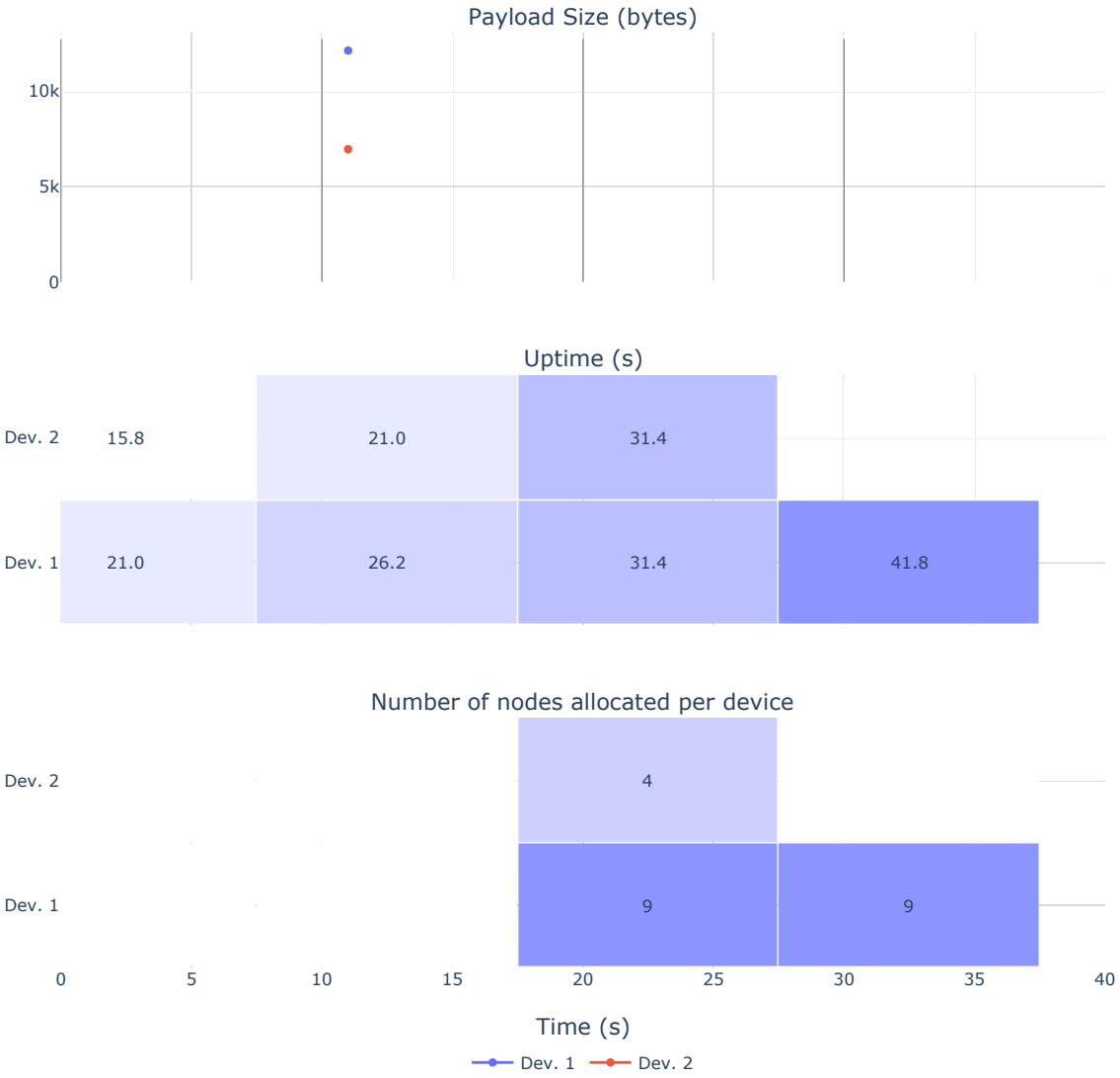


Figure 8.16: ES1-E in NoRDOr

possible. The system first processes the 6 specific nodes, assigning them to Dev. 1, then proceeds to the 7 general nodes. Most likely, the first 6 of these are assigned to Dev. 2, and the last one is assigned to Dev. 1. Even though the *MSC assignment* leads to an unbalanced orchestration when performing a full re-orchestration, a balanced one is considered to be preferable.

## 8.4.2 ES2 experiments

### 8.4.2.1 ES2-SC1

Similarly to **ES1-SC1** (*cf.* Section 8.4.1.1, p. 71), the aim of this experiment is to validate the re-orchestration mechanism which is triggered as each device fails.

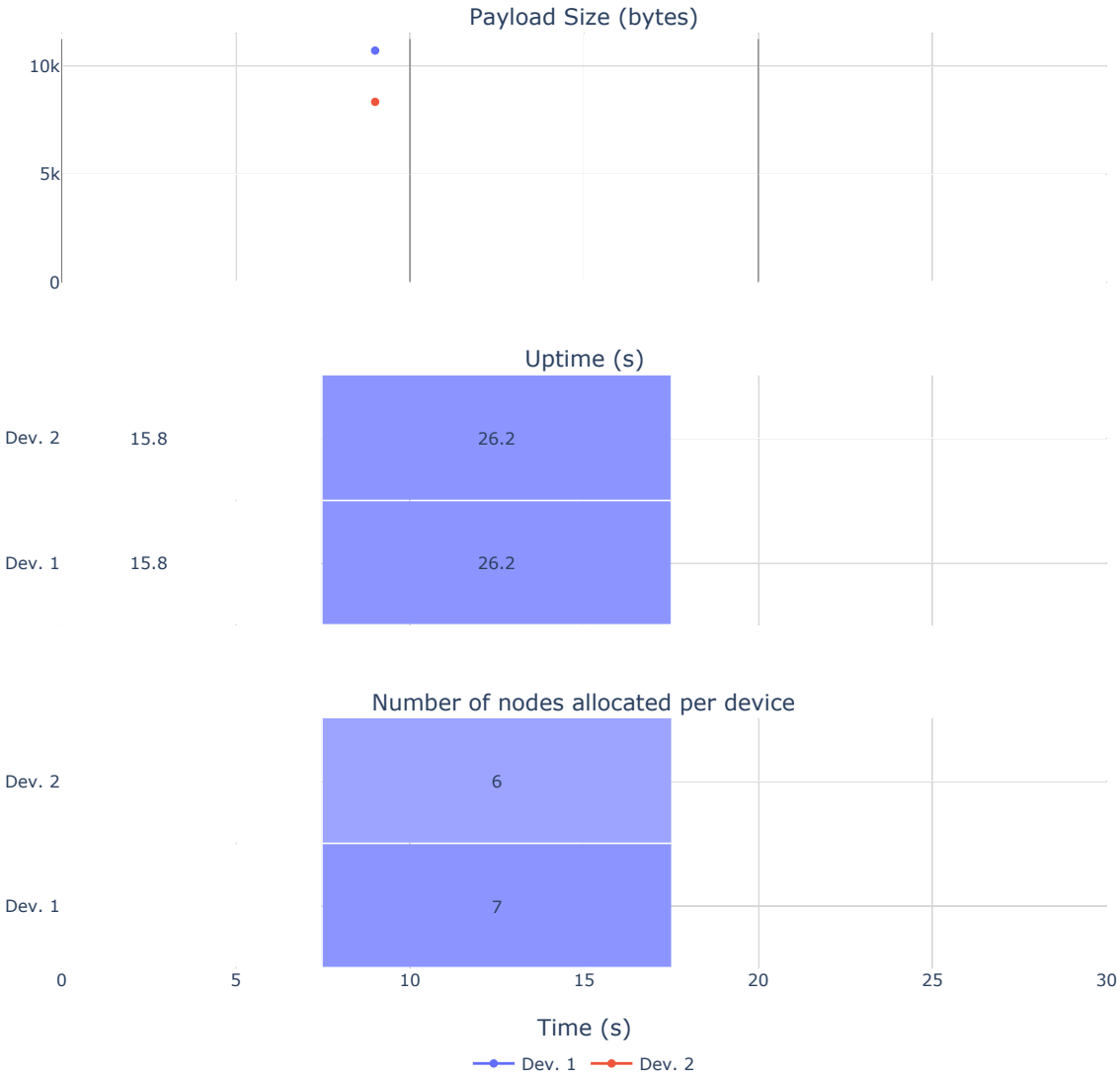


Figure 8.17: ES1-E in proposed system

Naturally, the expectations are also aligned with the ones expressed in **ES1-SC1**: the original NoRDOr system should re-orchestrate when each device is turned off and reach a balanced assignment every time, whereas the proposed system should use the *MSC mechanism* and deploy the failed nodes to one of the available devices on re-orchestration, resulting in a less balanced assignment.

After analyzing the results of the experiment shown in Figure 8.18 (p. 88), we can conclude that our expectations are met, as the system re-orchestrates when each device fails and generates balanced assignments every time. The first assignment yields 9 nodes for each of the 4 devices, the second yields 12 nodes for each of the 3 devices at around 40s, the third yields 18 for each of the 2 devices at around 70s, and, finally, the fourth should yield 36 nodes to the remaining device. However, we can see that when the, presumably, 36 nodes are deployed to Dev. 4, it crashes due

to a memory error. Since the NoRDOr system does not take into account the RAM of the devices in the system but rather assumes they can execute an unlimited amount of nodes until they fail for the first time, this is not entirely unexpected.

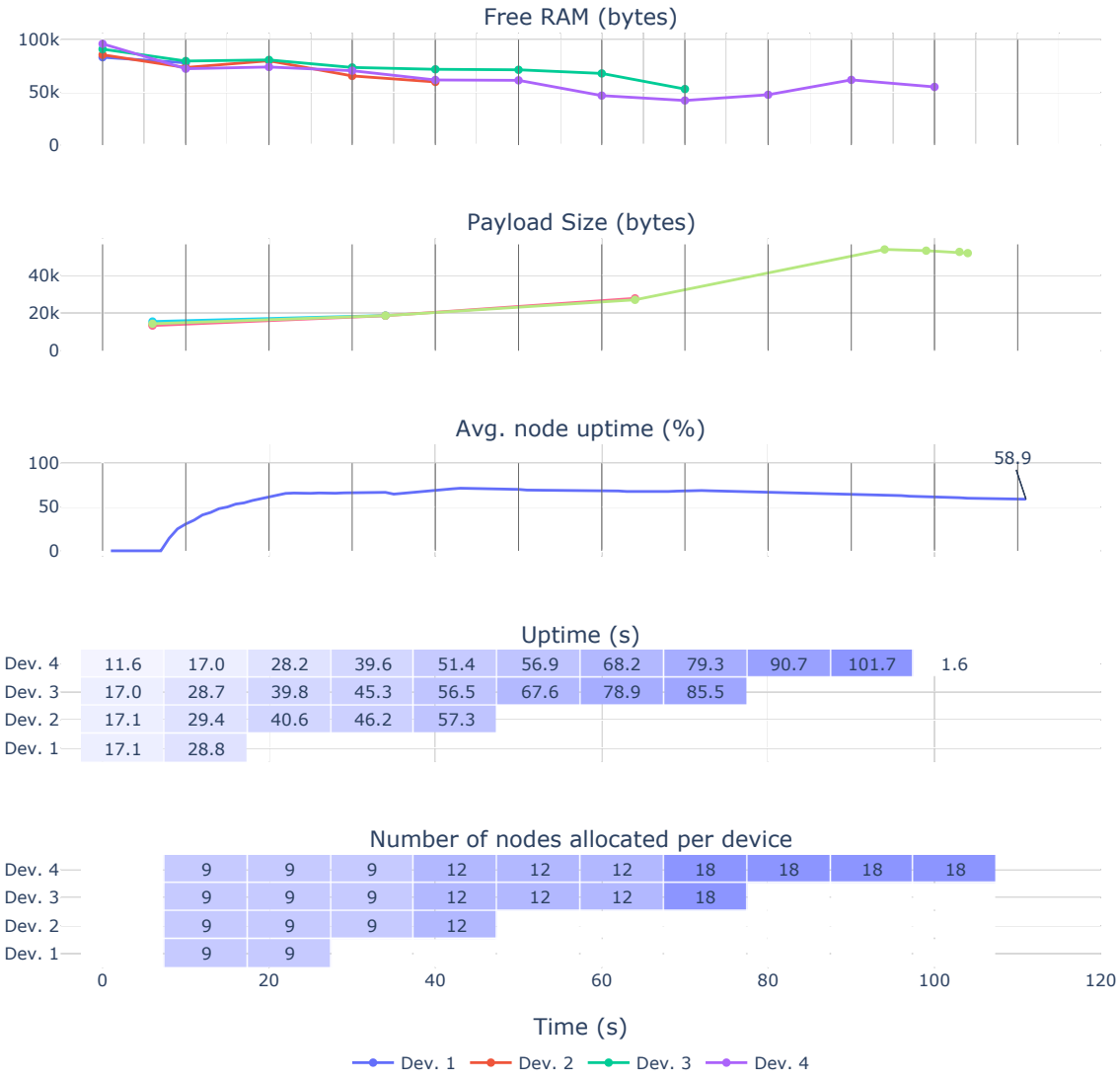


Figure 8.18: ES2-SC1 in NoRDOr

Correspondingly, the proposed system also meets our expectations, however, with slight differences due to the RAM limitations. It is possible to see the *MSC mechanism* in action at around 50s, assigning the 9 nodes from Dev. 1 to Dev. 2, and at around 70s, re-assigning the 18 nodes from Dev. 2 to Devs. 3 and 4. We attribute the fact that not all nodes were distributed to Dev. 3 to the *RAM heuristic* (*cf.* Section 7.1, p. 61), causing the 18 to be split to Devs. 3 and 4. Accordingly, the system does not perform the last orchestration of 36 nodes to Dev. 4 because the RAM heuristic signals the device cannot handle all of the nodes.

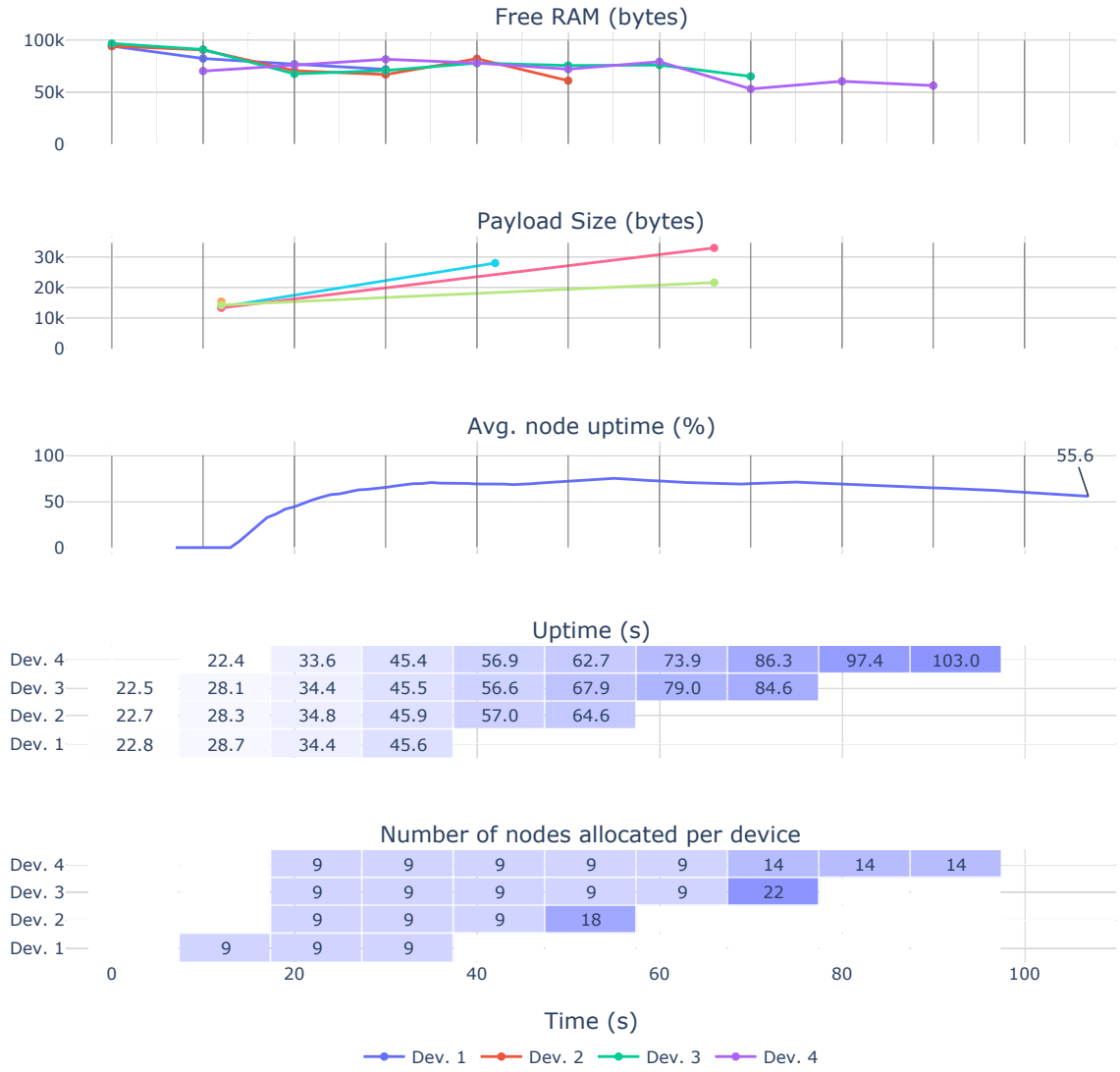


Figure 8.19: ES2-SC1 in proposed system

It is also worth noting that the Average Node Uptime of the NoRDOr system is slightly higher than the one of the proposed system at the end of the experiment, around 3% higher. After analyzing the data, it is possible to conclude that in the proposed system, the devices that are turned off are always the ones executing the most number of nodes, meaning that the *MSC assignment* negatively impacts these cases. In order to assess if the inverse yields a higher node uptime, we repeated the experiment but turned off the devices in another order.

Figure 8.20 (p. 90) shows that if the devices that are turned off are not the ones executing the most nodes, the Average Node Uptime is actually around 9% higher than the original system at the end of the experiment, meaning that the positive impact is slightly higher than the negative impact it produced in the opposite case.

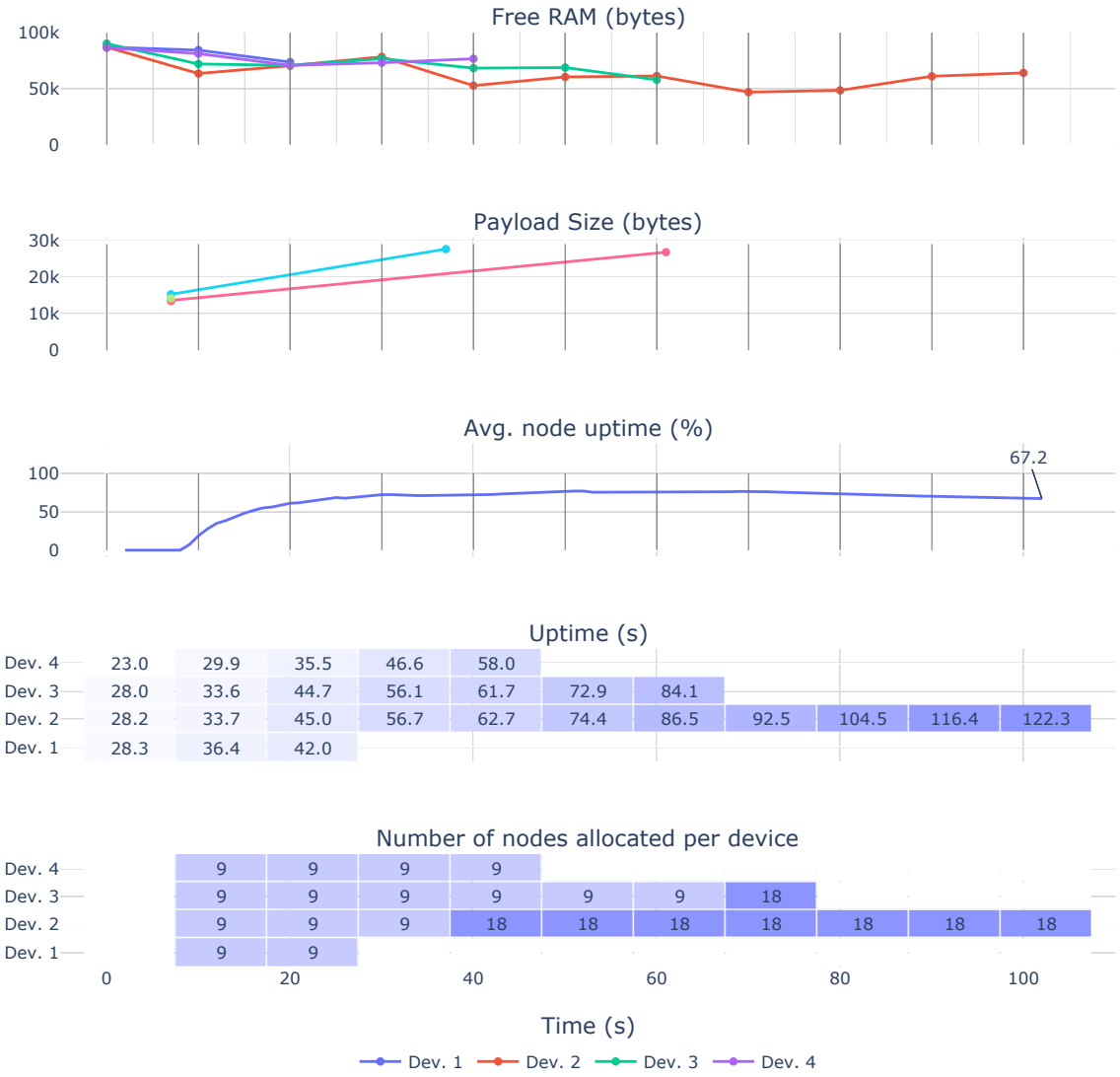


Figure 8.20: ES2-SC1_2 in proposed system

#### 8.4.2.2 ES2-SC2

This experiment, like **ES1-SC2** (*cf.* Section 8.4.1.2, p. 73), intends to validate the re-orchestration mechanism when devices disappear and re-appear in the system.

Consequently, we expect the same results as the ones observed in **ES1-SC2**. While the NoR-DOr system should re-orchestrate when devices fail and re-appear, keeping the number of nodes balanced throughout the devices, the proposed system should become slightly unbalanced when the first device is turned off but should also leverage the re-appearing devices to distribute the failed nodes as the experiment progresses.

As expected, Figure 8.21 (p. 91) shows that when Dev. 1 is turned off around 30s, all of the nodes are re-assigned, distributing 12 nodes to each of the available devices. However, Dev. 2

is turned off around 50s, leaving only the other 2 devices running 12 nodes each. The next re-orchestration, around 90s, yields an unexpected result — 9 nodes to Dev. 4, while Devs. 1 and 3 are executing 12 nodes. We are unsure as to what causes this behavior, but it might be due to 2 orchestrations running concurrently, each assuming a different number of available devices. The system eventually reaches a complete orchestration of 12 nodes per each of the 3 available devices at around 120s. Right after this, Dev. 4 is turned off the system, and the system seems to have performed, again, concurrent orchestrations that have led to inconsistent assignment results. Comparatively to the corresponding experiment **ES1-SC2**, the results are slightly less positive, which may be related to the increased network latency of working with physical devices.

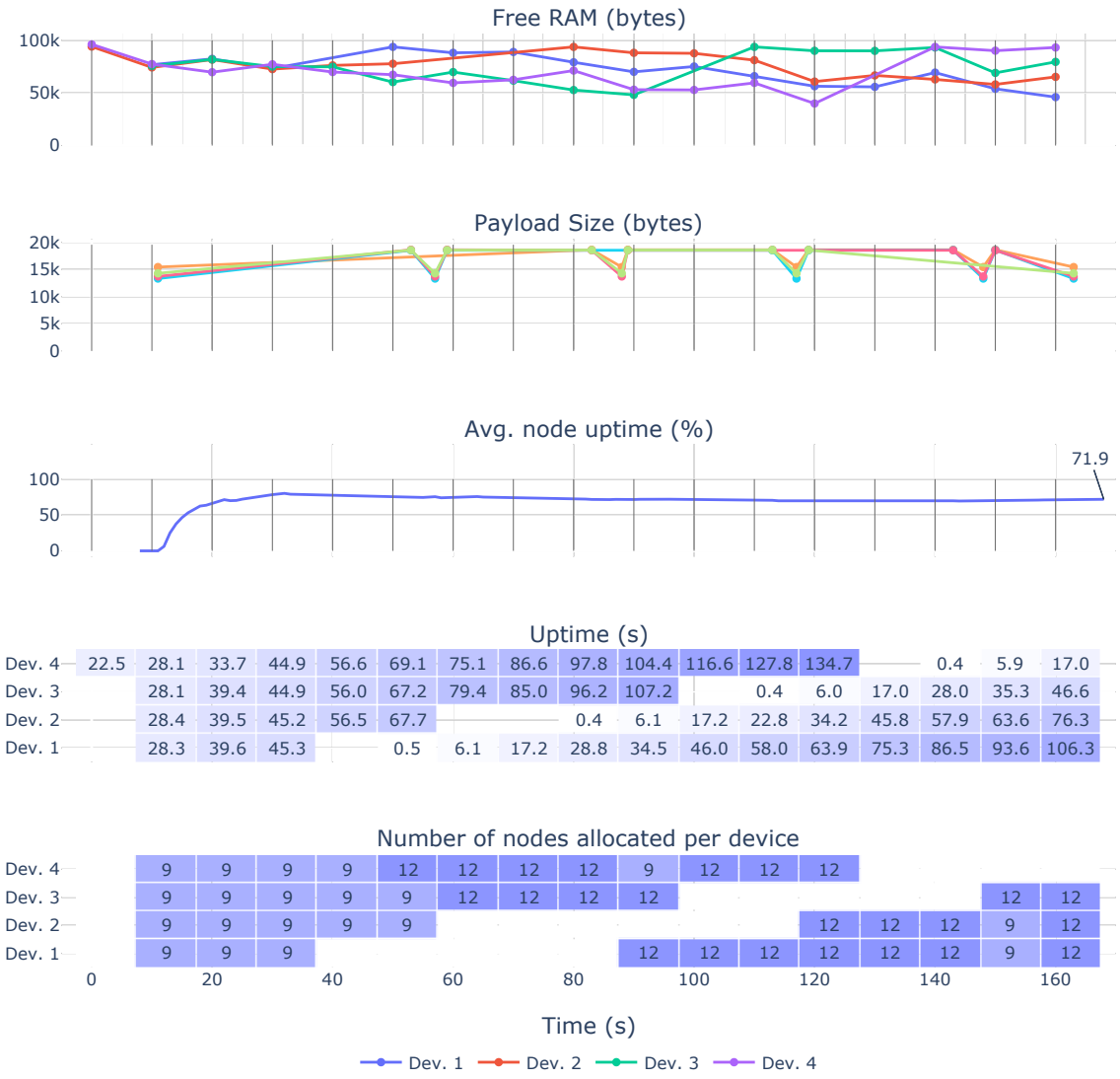


Figure 8.21: ES2-SC2 in NoRDOr

On the other hand, the proposed system works exactly as in **ES1-SC2**. As shown in Figure 8.22 (p. 92), the first re-orchestration accumulates 18 nodes in Dev. 2, which is turned off right

after being assigned the nodes. The system then assigns its 18 nodes Dev. 1, which has 0 assigned nodes. The following re-orchestrations also follow the same logic, taking advantage of the last re-connected device, which has 0 assigned nodes.

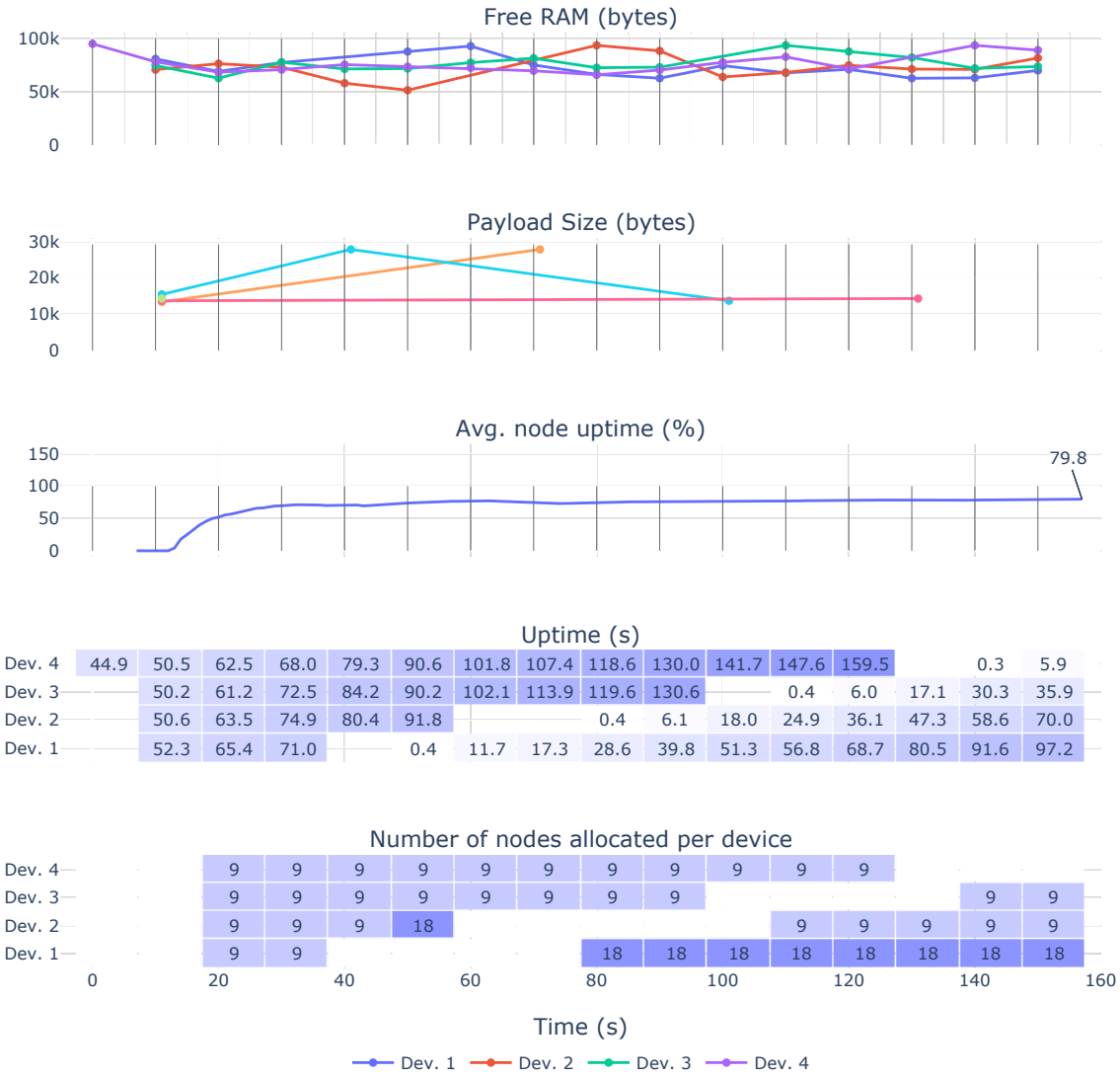


Figure 8.22: ES2-SC2 in proposed system

It is also possible to notice an 8% increase in Average Node Uptime in the proposed system. We attribute this to the unexpected assignments we outlined in the NoRDOr system, as well as the same *MSC* advantage we explored in **ES2-SC1** (*cf.* Section 8.4.2.1, p. 86) — in cases where the failing nodes have a lower number of nodes assigned, *MSC* leads to a higher Average Node Uptime.

#### 8.4.2.3 ES2-A

Similarly to **ES1-A** (*cf.* Section 8.4.1.3, p. 75), this experiment aims to evaluate the system's capability to adapt to unstable devices.

It is expected that the original NoRDOr system re-orchestrates every time a device disconnects or connects, regardless of how unstable it is. On the other hand, the proposed system should handle device instability by not assigning them nodes when they re-connect.

After analyzing Figure 8.23 (p. 94), we can conclude that, indeed, the system behaves as expected. The first assignment occurs almost at 20s and assigns 9 nodes to each of the 4 devices. Because Devs. 1 and 2 keep failing, they do not report their assigned nodes. At around 30s, the system assigns 14 nodes to Devs. 3 and 4. It is unclear why they are assigned this amount of nodes. By taking a look at the payload graph, we can see that Dev. 1 was also assigned some nodes, but it failed before reporting the number of nodes assigned. After this, the system keeps re-orchestrating on every re-connection, leaving 9 nodes to each of the correct devices for the remainder of the experiment.

Figure 8.24 (p. 95) also shows that the expected behavior is indeed the observed one. The first assignment at around 15s yields 18 nodes to each of the correct devices, and this remains constant throughout the experiment. As the failing devices re-connect, a different assignment is never deployed because of their lower stability. This behavior yields a much greater Average Node Uptime — almost 40% higher at the end of the experiment.

## 8.5 Hypothesis evaluation

The evaluation process was intended to provide answers to the research questions and to validate the hypothesis defined in Section 4.4 (p. 27). Thus, we re-visit them, summarize our findings and provide a clear answer to each one:

**RQ1:** If we change the order in which tasks are processed, can we generate balanced assignments in dynamic settings?

As pointed out by Silva *et al.* [45, 46] and shown in **ES2-SC1** (*cf.* Section 8.4.2.1, p. 86), a balanced system is more resilient because device failures cause less node downtime, thus providing correct service for a greater amount of time. However, since devices can leave and enter the system, it is imperative that we are able to generate balanced assignments given the currently available devices in the system.

Sorting the nodes by their specificity related to the currently available devices proved to solve the highlighted balancing issues, allowing the greedy heuristic to generate balanced assignments in dynamic situations. Thus, we consider the answer to this research question to be affirmative — processing the nodes in a different order can produce more balanced solutions.

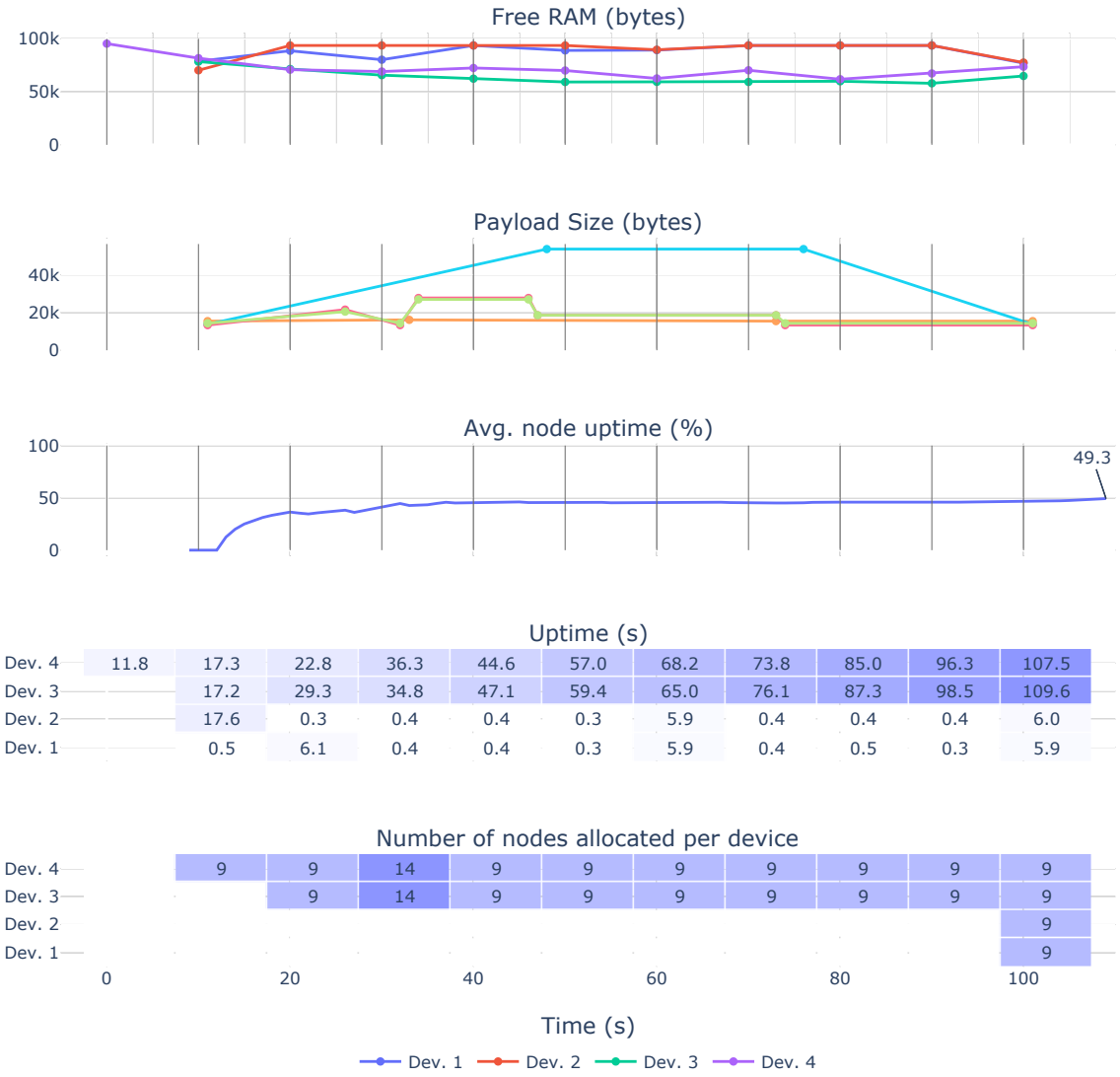


Figure 8.23: ES2-A in NoRDOr

**RQ2:** If we measure device stability and leverage it to make decisions on orchestration, can we produce solutions that are less susceptible to device failure?

The instability of edge devices is likely, and if unaccounted for, can considerably hinder the system's resilience, as unstable devices can cause node downtime.

By employing a probing mechanism, we use the Mean Time Between Failure (MTBF) of each device to calculate a *device stability* metric that allows the system to consider assigning fewer nodes to less stable devices. Additionally, we also leverage this metric to introduce the concept of *assignment score*, which is independent of the assignment algorithm, enhancing the system with the ability to decide if a re-orchestration benefits the current state of the system.

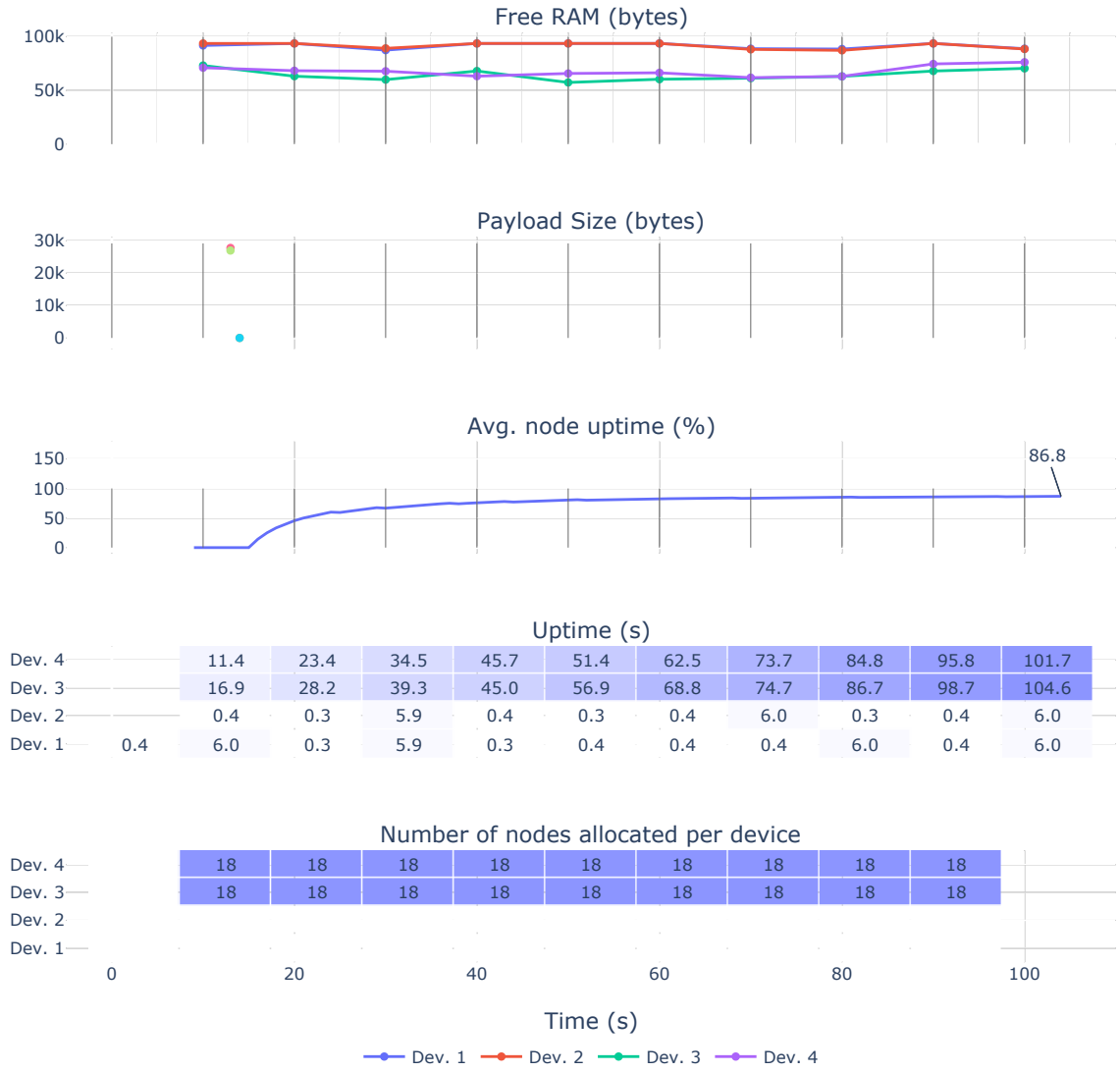


Figure 8.24: ES2-A in proposed system

**ES2-A** (*cf.* Section 8.4.2.3, p. 93) shows that node uptime almost doubled in the proposed system that measures node stability and assigns fewer nodes to unstable devices. Hence, we consider that measuring device stability and leveraging it on orchestration can prevent the system from incurring in assignments where unstable devices have assigned nodes, providing an affirmative answer to this research question.

**RQ3:** If we leverage the current assignment on re-orchestration, can we generate solutions that result in less orchestration overhead?

Re-distributing all nodes every time the system re-orchestrates is a limitation that can prove costly to the system — it adds orchestration overhead and entropy to already possibly unstable devices. However, it may be hard to measure the impact of this behavior.

Due to the underlying code execution mechanism, deploying a node to a device requires the deployment of the full code script containing every node it must execute. Consequently, devices need to delete the current script and install the new one. To tackle this issue, we introduce the *Minimum Set of Changes* (MSC) concept that aims to minimize the number of affected nodes — nodes that need to be stopped while the device re-installs the script — on re-orchestration, by taking into account the current assignment. It does so at the expense of the assignment balancing, which may be contradictory to the purpose of **RQ1**.

Given the results of the experiments, we can conclude that there are two clearly opposite forces in the system. While on the one hand, the *MSC* can produce a better result in critical systems, as shown in **ES1-B** (*cf.* Section 8.4.1.4, p. 77), it produces mixed results in **ES2-SC1** (*cf.* Section 8.4.2.1, p. 86). While balancing makes the system more resilient, in general, because the failure of a device has a lower impact than in a system where a device can be executing a high portion of nodes, it may also come at the expense of safety in critical systems.

Although leveraging the current assignment to produce the new orchestration results in less orchestration overhead, it is also clear that this solution warrants further research. The current execution mechanism sets some boundaries on other possible solutions to be explored in this regard, but changes to this mechanism would permit a deeper exploration of this topic. We provide possible research directions in Section 9.4 (p. 101).

**RQ4:** If we leverage the real-time information on device resources to make decisions on the orchestration, can we produce solutions that are less susceptible to memory limitations?

Memory constraints limit the number of nodes each device can be assigned and may cause the device to crash if unaccounted for. Furthermore, available RAM is a dynamic resource that cannot be statically assumed.

Due to the underlying code execution component, devices need to be assigned a full script on orchestration, meaning that the real-time RAM usage is not of much use to the *Orchestrator*. On the other hand, the device's *idle* RAM usage that it reports when it connects to the system should be similar to its available RAM when no script is being executed. Thus, the *Orchestrator* can leverage this value, using a heuristic that estimates if the RAM usage of a set of nodes is likely to crash the device and stop its deployment.

After analyzing the experiments, we can conclude that the proposed system can effectively estimate the RAM usage of a given script and prevent memory errors, making the system more resilient. We did, however, find that the developed heuristic might be overly restrictive at times and requires further tuning to better estimate the target value. We consider that the answer to this research question is affirmative but acknowledge the solution may be flawed.

**RQ5:** If we measure the stability of device-node pairings, can we leverage it on orchestration to identify and mitigate device-node pairings that cause device failure?

The execution of certain nodes by some devices can lead to failure, either by memory errors or hardware failure. Failing to identify these pairings causes the system to keep assigning them, and thus, devices may fail more often than expected.

In order to identify these pairings and mitigate them, we follow the same approach as with device stability — using the MTBF of a device-node pairing — resulting in the *device-node stability* metric. However, even if only one node causes the device to fail, all the nodes they were executing are considered to have caused the failure as it is currently impossible to discern what node was the root cause of the issue.

By observing the result of the experiments, we can conclude that our approach did not have a clear positive impact in mitigating this problem. Nevertheless, it is also important to mention that it did not have a negative impact on the system as well. What this means is that it is possible that the metric proves more useful as the system evolves and gathers more information on different pairings. Hence, we cannot provide a positive answer to this research question.

Finally, after answering each individual research question, we can evaluate the hypothesis claimed by this work:

> *"Runtime information from dynamically orchestrated IoT systems can be used to achieve greater system resilience."*

Considering the answers to the research questions, we can conclude that a greater system resilience can be achieved in the NoRDOr system by (1) employing probing mechanisms to leverage real-time information from devices, such as stability and resources, to generate better orchestrations, (2) altering the greedy algorithm to mitigate is node-order bias, thus generating balanced assignments dynamically, and (3) leveraging the current orchestration to produce minimal disturbance to the system when re-orchestrating, thus allowing it to provide correct service for a greater amount of time which may be crucial to ensure safety in critical systems. Thus, we consider our hypothesis to be validated as the runtime information gathered from devices led to a more resilient system.

## 8.6 Summary

In this chapter, the results drawn from the evaluation process are presented. Section 8.1 (p. 67) starts by defining the system configuration for the experiments, then Section 8.2 (p. 68) describes each collected metric and its purpose, and Section 8.3 (p. 70) summarizes the setup and high-level event sequence for each experiment.

In Section 8.4 (p. 71), we present the results of each experiment and provide a detailed analysis of them, highlighting the impact of the improvements compared to the original system. In Section 146 (p. 93), we re-visit the research questions and hypothesis driving this dissertation (*cf.*

Section 4.4, p. 27), and sum up the conclusions taken from the aforementioned results: (a) the proposed system achieves greater resilience by leveraging real-time information about edge devices to measure their stability, which it then uses to make decisions on assignment, (b) changing the order in which nodes are processed by the greedy algorithm can lead to more balanced assignments in dynamic settings, (c) the *Orchestrator* can take advantage of the RAM measurements reported by the device on orchestration to produce solutions that are less susceptible to memory limitations, (d) an assignment that minimizes the number of affected nodes by leveraging the current assignment can be beneficial in critical system but produces mixed results in more general scenarios, and (e) using a *device-node stability* metric based on the MTBF of device-node pairings **does not** produce solutions less susceptible to device failures due to faulty nodes. Additionally, we identify opposite forces in the developed solution that lead the system to highly balanced solutions in some cases and highly unbalanced solutions in others. We explore the benefits and drawbacks of prioritizing each direction but ultimately highlight the need for further research.

# Chapter 9

# Conclusions



This chapter presents an overview of this dissertation and outlines future research topics. In Section 9.1, we present the key conclusions and remarks of this work. Then, the contributions that resulted from this dissertation are introduced in Section 9.2 (p. 100). The main difficulties experienced during this work are outlined in Section 9.3 (p. 101). Finally, possible research directions drawn from this dissertation and presented in Section 9.4 (p. 101).

## 9.1 Conclusions

With the steady increase in the number of Internet-of-Things devices and a clear boost in their computational capabilities, new applications are also surging. Some of these new applications can no longer rely on the traditional cloud architecture of large-scale IoT systems due to the amount of data produced and geographical distribution, making it increasingly difficult, costly, or even illegal to perform centralized data storage and processing in the cloud.

Fog and Edge computing paradigms emerge as a response to these issues, having its main premise be the bringing IoT computation closer to the edge, allowing for low-latency and lower-cost systems while also setting clear boundaries for data transfers. However, they, too, come with drawbacks. Fog nodes are distributed and heterogeneous; thus, efficient resource provisioning and allocation need to be performed. Additionally, the computational constraints and heterogeneity of edge devices are challenging for developers, coupled with the lack of reliability due to hardware failures.

During the analysis of the state of the art, many edge orchestration platforms were found that leverage the edge devices computational capabilities by providing the user with: (1) ease of development, through the use of Visual Programming Languages (VPL), (2) general-purpose frameworks for heterogeneous edge devices capable of custom code execution, (3) automatic decomposition and allocation of tasks, and (4) self-healing mechanisms to improve the resilience of the system. However, we found that the fault-tolerance capabilities are usually overlooked, offering little to no dependability improvements.

The developed solution addresses many of the issues found in a particular open-source edge orchestration platform for IoT, NoRDOr [45] as an attempt to improve the system's dependability. Particularly, a probing mechanism was implemented, allowing the system to monitor the device's stability and resources. Having these real-time metrics, the *Orchestrator* can perform better assignments by considering deploying fewer nodes to more unstable devices and is also able to assess the device's memory capabilities and adjust the number of nodes to deploy.

In addition to improving the monitoring of devices, the system must also leverage the current assignment as a means to minimize re-orchestration overhead. Hence, a new assignment heuristic was developed that aims to reduce the number of affected nodes in each deployment by only assigning the currently failed nodes to the least possible number of devices.

Performing experiments with easily reproducible scenarios in cyber-physical real-time systems is not a trivial task, as the devices are susceptible to hardware and connectivity limitations, making it hard to ensure that only the relevant variables are in play. In the interest of reproducing the issues found in NoRDOr, and as a way to tighten the validation cycle, an *Orchestration simulator* application was developed. Its architecture mimics that of the NoRDOr system, containing a set of devices that execute code, a set of nodes (computational tasks) to be executed, and an *Orchestrator* component that manages the assignment of tasks to devices. More importantly, it allows for the definition of reproducible scenarios, controlling the nodes and their requirements and the devices' resources and behaviors.

After reproducing the issues and iteratively validating the solutions in the simulator, the proposed system changes were ported into the NoRDOr system and tested both on virtual (*Dockerized*) and physical IoT devices. In these experiments, we injected faults and limitations in the devices and compared the original system with the proposed one, analyzing their ability to cope with the faults and converge to a stable orchestration. The proposed system showed to not only perform as well as NoRDOr in some scenarios but also to be able to maintain correct service in more unstable situations that the original system could not handle.

Considering the results of the experiments, we can conclude that the improvements resulting from this dissertation make the system more able to cope with the instability and limitations of heterogeneous edge device devices and, thus, more resilient.

## 9.2 Contributions

The work developed in the course of this dissertation resulted in the following contributions:

**Orchestration simulator** A modular and extensible simulator that streamlines the evaluation of a task assignment algorithm in the realm of real-time orchestration platforms.

**Set of improvements on the orchestration strategy of IoT systems** A set of proposed improvements to NoRDOr, a real-time IoT orchestration system, that improve its dependability by leveraging runtime information.

## 9.3 Difficulties

During the development of our solution, some difficulties arose that delayed the process — namely when porting the solution to NoRDOr and experimenting on the system.

When performing changes to the devices' MircoPython framework, we came across several limitations that would result in odd behavior. Notably, the existence of multiple MQTT clients in the same device proved to cause inconsistent behavior, failing to report metrics, or failing to connect to the *Registry* node.

Reproducing the same scenario in both systems was also challenging as it mostly required manually stopping a virtual or physical device, only relying on the experiment clock. In some cases, changes to the devices' firmware were also necessary to simulate a specific behavior which meant re-building the firmware and restarting the whole system repeatedly.

## 9.4 Future work

The proposed improvements contain implementation limitations, as described in Section 7.2 (p. 65) but also introduces problems that warrant further research.

Firstly, and most pressingly, we identified two different forces that contribute to the system's resilience in opposite ways — balancing and minimizing the set of changes. While the first improves the general resilience of the system, the latter emerges a better solution for critical systems where safety is a priority. Given the current code execution mechanism — where devices cannot be assigned or removed a single node, but rather their entire node assignment, the balance of these forces lies in the use of hyper-parameters. A welcome change that could allow for a more suitable minimum set of changes assignment would be the modification of the code execution mechanism, allowing each device to be assigned a full set of nodes but also individually assigned or removed a single node. However, even with this change, further research is still warranted in order to assess the impact of adding or removing a single node from a healthy device, as it adds entropy to the system, which can be an undesirable force in a critical system.

Secondly, the monitoring of the stability of device-node pairings proved to be non-beneficial, as it is currently impossible to discern what node introduced the fault, meaning that the metric is not precise enough. A possible solution would be to enhance the devices' firmware with the ability to report which node caused the failure, using an *exception handling* mechanism. However, this solution is not perfect as (1) there may be cases where the fault is so strong it causes a complete device shutdown/restart, thus losing the ability to report the cause of failure, and (2) a specific device-node pairing may not be the cause of the problem but rather a set of various nodes assigned to the device.

Finally, the greedy assignment algorithm proved to be not very scalable in complex scenarios with a high level of memory limitations. As the system grows, and as more metrics are used to generate the best possible assignment, the greedy algorithm should eventually be replaced by a

better suitable alternative, such as modeling the problem as an Integer Linear Problem (ILP) and using optimization techniques to solve it, as proposed by Skarlat *et al*. [48].

# References


[1] Amazon Web Services (AWS) - Cloud Computing Services. Available at https://aws.amazon.com/. Accessed in January 2021.

[2] Google Cloud: Cloud Computing Services. Available at https://cloud.google.com/. Accessed in January 2021.

[3] Mehmet S. Aktas and Merve Astekin. Provenance aware run-time verification of things for self-healing internet of things applications. *Concurr. Comput. Pract. Exp.*, 31(3), 2019.

[4] M. Aly, F. Khomh, Y. Guéhéneuc, H. Washizaki, and S. Yacout. Is fragmentation a threat to the success of the internet of things? *IEEE Internet of Things Journal*, 6(1):472–487, 2019.

[5] Masoud Saeida Ardekani, Rayman Preet Singh, Nitin Agrawal, Douglas B. Terry, and Riza O. Suminto. Rivulet: a fault-tolerant platform for smart-home applications. In K. R. Jayaram, Anshul Gandhi, Bettina Kemme, and Peter R. Pietzuch, editors, *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017*, pages 41–54. ACM, 2017.

[6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Comput. Networks*, 54(15):2787–2805, 2010.

[7] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[8] Michael Blackstock and Rodger Lea. Toward a distributed data flow platform for the web of things (distributed node-red). In *Proceedings of the 5th International Workshop on Web of Things, WoT 2014, Cambridge, MA, USA, October 8, 2014*, pages 34–39. ACM, 2014.

[9] Flavio Bonomi, Rodolfo A. Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In Mario Gerla and Dijiang Huang, editors, *Proceedings of the first edition of the MCC workshop on Mobile cloud computing, MCC@SIGCOMM 2012, Helsinki, Finland, August 17, 2012*, pages 13–16. ACM, 2012.

[10] 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.

[11] 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 Things J.*, 5(2):696–707, 2018.

[12] CISCO. Cisco {Annual} {Internet} {Report} (2018–2023) {White} {Paper}. Technical report, 2020.

[13] 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.

[14] João Pedro Dias, João Pascoal Faria, and Hugo Sereno Ferreira. A reactive and model-based approach for developing internet-of-things systems. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 276–281, September 2018.

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

[16] 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.

[17] 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 Harold Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João Teixeira, editors, *Computational Science - ICCS 2020 - 20th International Conference, Amsterdam, The Netherlands, June 3-5, 2020, Proceedings, Part V*, volume 12141 of *Lecture Notes in Computer Science*, pages 357–370. Springer, 2020.

[18] 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.

[19] João Pedro Dias, Tiago Boldt Sousa, André Restivo, and Hugo Sereno Ferreira. A pattern-language for self-healing internet-of-things systems. In *EuroPLoP '20: European Conference on Pattern Languages of Programs 2020, Virtual Event, Germany, 1-4 July, 2020*, pages 25:1–25:17. ACM, 2020.

[20] J.P. Dias. jpdias/node-red-contrib-self-healing: Replication package for ICCS 2020., April 2020.

[21] Erik Elmroth, Philipp Leitner, Stefan Schulte, and Srikumar Venugopal. Connecting fog and cloud computing. *IEEE Cloud Comput.*, 4(2):22–25, 2017.

[22] Hugo Sereno Ferreira, Ademar Aguiar, and João Pascoal Faria. Adaptive object-modelling: Patterns, tools and applications. In *2009 Fourth International Conference on Software Engineering Advances*, pages 530–535. IEEE, 2009.

[23] 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.

[24] Alan G. Ganek and Thomas A. Corbi. The dawning of the autonomic computing era. *IBM Syst. J.*, 42(1):5–18, 2003.

[25] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C. M. Leung. Developing iot applications in the fog: A distributed dataflow approach. In *5th International Conference on the Internet of Things, IOT 2015, Seoul, South Korea, 26-28 October, 2015*, pages 155–162. IEEE, 2015.

[26] Nam Ky Giang, Rodger Lea, Michael Blackstock, and Victor C. M. Leung. Fog at the edge: Experiences building an edge computing platform. In *2018 IEEE International Conference on Edge Computing, EDGE 2018, San Francisco, CA, USA, July 2-7, 2018*, pages 9–16. IEEE Computer Society, 2018.

[27] Nam Ky Giang, Rodger Lea, and Victor C. M. Leung. Developing applications in large scale, dynamic fog computing: A case study. *Softw. Pract. Exp.*, 50(5):519–532, 2020.

[28] Salam Hamdan, Moussa Ayyash, and Sufyan Almajali. Edge-computing architectures for internet of things applications: A survey. *Sensors*, 20(22):6441, 2020.

[29] Michaela Iorga, Larry Feldman, Robert Barton, Michael J Martin, Nedim S Goren, and Charif Mahmoudi. Fog computing conceptual model. 2018.

[30] 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.

[31] Maurizio Leotta, Davide Ancona, Luca Franceschini, Dario Olianas, Marina Ribaudo, and Filippo Ricca. Towards a runtime verification approach for internet of things systems. In Cesare Pautasso, Fernando Sánchez-Figueroa, Kari Systä, and Juan Manuel Murillo Rodriguez, editors, *Current Trends in Web Engineering - ICWE 2018 International Workshops, MATWEP, EnWot, KD-WEB, WEOD, TourismKG, Cáceres, Spain, June 5, 2018, Revised Selected Papers*, volume 11153 of *Lecture Notes in Computer Science*, pages 83–96. Springer, 2018.

[32] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE Internet Things J.*, 4(5):1125–1142, 2017.

[33] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. Fog computing: A taxonomy, survey and future directions. In Beniamino Di Martino, Kuan-Ching Li, Laurence T. Yang, and Antonio Esposito, editors, *Internet of Everything - Technology, Communications and Computing*, pages 103–130. Springer, 2018.

[34] Joseph Noor, Hsiao-Yun Tseng, Luis Garcia, and Mani B. Srivastava. Ddflow: visualized declarative programming for heterogeneous iot networks. In Olaf Landsiedel and Klara Nahrstedt, editors, *Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI 2019, Montreal, QC, Canada, April 15-18, 2019*, pages 172–177. ACM, 2019.

[35] Umar Ozeer, Xavier Etchevers, Loïc Letondeur, François-Gaël Ottogalli, Gwen Salaün, and Jean-Marc Vincent. Resilience of stateful iot applications in a dynamic fog environment. In Henning Schulzrinne and Pan Li, editors, *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, MobiQuitous 2018, 5-7 November 2018, New York City, NY, USA*, pages 332–341. ACM, 2018.

[36] 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.

[37] Alexander Power and Gerald Kotonya. A microservices architecture for reactive and proactive fault tolerance in iot systems. In *19th IEEE International Symposium on "A World of Wireless, Mobile and Multimedia Networks", WoWMoM 2018, Chania, Greece, June 12-15, 2018*, pages 588–599. IEEE Computer Society, 2018.

[38] 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.

[39] Harald Psaier and Schahram Dustdar. A survey on self-healing systems: approaches and systems. *Computing*, 91(1):43–73, 2011.

[40] 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.

[41] Pushkara Ravindra, Aakash Khochare, Sivaprakash Reddy, Sarthak Sharma, Prateeksha Varshney, and Yogesh Simmhan. ECHO: an adaptive orchestration platform for hybrid dataflows across cloud and edge. *CoRR*, abs/1707.00889, 2017.

[42] Jan Seeger, Arne Bröring, and Georg Carle. Optimally self-healing iot choreographies. *ACM Trans. Internet Techn.*, 20(3):27:1–27:20, 2020.

[43] Jan Seeger, Rohit Arunrao Deshmukh, and Arne Bröring. Running distributed and dynamic iot choreographies. In *2018 Global Internet of Things Summit, GIoTS 2018, Bilbao, Spain, June 4-7, 2018*, pages 1–6. IEEE, 2018.

[44] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet Things J.*, 3(5):637–646, 2016.

[45] Margarida Silva. Orchestration for Automatic Decentralization in Visually-defined IoT. Master's thesis, University of Porto, 2020.

[46] 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.

[47] 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.

[48] Olena Skarlat, Matteo Nardelli, Stefan Schulte, and Schahram Dustdar. Towards qos-aware fog service placement. In *1st IEEE International Conference on Fog and Edge Computing, ICFEC 2017, Madrid, Spain, May 14-15, 2017*, pages 89–96. IEEE Computer Society, 2017.

[49] 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.

[50] 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.

[51] Aparna Saisree Thuluva, Arne Bröring, Ganindu P. Medagoda, Hettige Don, Darko Anicic, and Jan Seeger. Recipes for iot applications. In Simon Mayer, Stefan Schneegass, Bernhard Anzengruber, Alois Ferscha, Gabriele Anderst-Kotsis, and Joe Paradiso, editors, *Proceedings of the Seventh International Conference on the Internet of Things, IOT 2017, Linz, Austria, October 22-25, 2017*, pages 10:1–10:8. ACM, 2017.

[52] 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.

[53] Andrew Whitmore, Anurag Agarwal, and Li Da Xu. The internet of things—a survey of topics and trends. *Information systems frontiers*, 17(2):261–274, 2015.

[54] Yang Yang. Multi-tier computing networks for intelligent iot. *Nature Electronics*, 2(1):4–5, 2019.

[55] Sen Zhou, Kwei-Jay Lin, Jun Na, Ching-Chi Chuang, and Chi-Sheng Shih. Supporting service adaptation in fault tolerant internet of things. In *8th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2015, Rome, Italy, October 19-21, 2015*, pages 65–72. IEEE Computer Society, 2015.