FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Orchestration for Automatic Decentralization in Visually-defined IoT

Ana Margarida Oliveira Pinheiro da Silva

WORKING VERSION

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Hugo Sereno Ferreira

Second Supervisor: André Restivo

June 27, 2020

Orchestration for Automatic Decentralization in Visually-defined IoT

Ana Margarida Oliveira Pinheiro da Silva

Mestrado Integrado em Engenharia Informática e Computação

June 27, 2020

Abstract

The Internet-of-Things (IoT) is an ever growing network of devices connected to the Internet. Such devices are heterogeneous in their protocols and computation capabilities. With the rising computation and connectivity capabilities of these devices, the possibilities of their use in IoT systems increases. Concepts like smart cities are the pinnacle of the use of these systems, which involves a big amount of different devices in different conditions.

There are several tools for building IoT systems; some of these tools have different levels of expertise required and employ different architectures. One of the most popular is Node-RED. It allows users to build systems using a visual data flow architecture, making it easy for a non-developer to use it.

However, most of these mainstream tools employ centralized methods of computation, where a main component — usually hosted in the cloud — executes most of the computation on data provided by edge devices, *e.g.* sensors and gateways. There are multiple consequences to this approach: (a) edge computation capabilities are being neglected, (b) it introduces a single point of failure, and (c) local data is being transferred across boundaries (private, technological, political...) either without need, or even in violation of legal constraints. Particularly, the principle of Local-First — *i.e.*, data and logic should reside locally, independent of third-party services faults and errors — is blatantly ignored.

Previous work attempt to mitigate some of these consequences, usually through tools that extend existing visual programming frameworks, such as Node-RED. They go as far as to propose a solution to decentralize flows and its execution in fog/edge devices. So far, achieving such decentralization requires that the decomposition and partitioning effort be manually specified by the developer when building the system.

Our goal is to extend Node-RED to allow automatic decomposition and partitioning of the system towards higher decentralization, by inferring computational boundaries. Furthermore, through automatic detection of abnormal run-time conditions, we also intend to provide dynamic self-adaptation. The prototype developed will be first validated with real devices and later with simulations.

As a result, we expect to achieve a more robust and efficient execution of IoT systems, by leveraging edge and fog computational capabilities present in the network, and improving overall reliability.

Keywords: Internet of Things, Visual Programming, Edge Computing

Resumo

A Internet-of-Things (IoT) é uma rede de dispositivos conectados à Internet em constante crescimento. Estes dispositivos são heterogéneos nos seus protocolos e capacidades de computação. Com o crescimento das capacidades de computação e conectividade destes dispositivos, as possibilidades do seu uso em sistemas IoT aumentaram. Conceitos como Cidades Inteligientes são o pináculo do uso destes sistemas, que envolvem um grande número de dispositivos diferentes em diferentes condições.

Existem várias ferramentas para construir sistemas IoT; algumas destas ferramentas requerem diferentes níveis de perícia e usam diferentes arquiteturas. Uma das ferramentas mais populares é Node-RED. Esta permite aos seus utilizadores construir sistemas usando uma arquitetura visual de *data flow*, tornando o processo mais fácil para um utilizador não programador.

No entanto, a maioria das ferramentas convencionais usam métodos centralizados de computação, onde um componente principal - normalmente alocado na *cloud* - executa a maioria da computação nos dados provenientes dos dispositivos *edge*, *e.g.* sensores e *gateways*. Com esta abordagem estão associadas multiplas consequências: (a) capacidades de computação de dispositivos *edge* estão a ser neglegenciadas, (b) introduz um único ponto de falha, e (c) data local está a ser transferida através de limites (privados, tecnológicos, políticos...) sem necessidade ou violando restrições legais. Especificamente, o princípio de *Local-First* - *i.e.*, dados e lógica devem residir localmente, independentemente de falhas e erros de serviços terceiros - é totalmente ignorado.

Trabalhos feitos até agora tentam mitigar algumas destas consequências, construindo ferramentas que estendem ferramentas existentes de programação visual, como Node-RED. Algumas propôe uma solução que consiste na descentralização de *flows* e a sua execução em dispositivos de *fog* e *edge*. Atualmente, para obter a este tipo de descentralização é necessário que o esforço de decomposição e partição seja manualmente efetuado pelo programador quando este constrói o sistema.

O nosso objetivo é estender a ferramenta Node-RED para permitir a decomposição e partição automática do sistema com o fim de obter uma maior descentralização. Para isso é necessário deduzir os limites de computação do sistema. Para além disso, também pretendemos que o sistema se adapte automaticamente às mudanças do ambiente, detectando automaticamente condições anormais em *run-time*. O protótipo construído será validado, numa primeira fase, com dispositivos reais e, mais tarde, com o uso de simulações.

Como resultado, esperamos construir uma execução de sistemas IoT mais robusta e eficiente, aproveitando as capacidades de computação presentes nos dispositivos *edge* e *fog* da rede, e melhorando a confiança e segurança do sistema.

Keywords: Internet of Things, Visual Programming, Edge Computing

Acknowledgements

TODO

Ana Margarida Silva

"Until I began to learn to draw,
I was never much interested in looking at art."

Richard P. Feynman

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
CPSCN	Cyber Physical Social Computing and Networking
CPU	Central Processing Unit
HTTP	Hypertext Transfer Protocol
IEC	International Electrotechnical Commission
ISO	Internation Organization for Standardization
IFTTT	*If This Then That*
IoT	Internet-of-Things
JSON	JavaScript Object Notation
MQTT	Message Queuing Telemetry Transport
MTTR	Mean Time To Recover
QoS	Quality of Service
RaaS	Robot as a Servive
IoIT	Internet of Intelligent Things
REST	Representational State Transfer
URL	Uniform Resource Locator
VPL	Visual Programming Language
WWW	*World Wide Web*

Chapter 1

Introduction

This chapter introduces the motivation and scope of this project, as well as the problems it aims to solve. Section 1.1 details the context of this project in the area it is based on. Section 1.2 defines the problem we aim to solve. Then, Section 1.3 explains the reason why this work and the area it belongs to is important and the goals of this dissertation are described in Section 1.4. Finally, Section 1.5 describes the structure of this document and what content it contains.

1.1 Context

The Internet-of-Things (IoT) paradigm states that all devices, independently of their capabilities, are connected to the Internet and allow for the transfer, integration and analytic of data generated by them [15]. This paradigm has several characteristics, such as the heterogeneity and high distribution of devices as well as their increasing connectivity and computational capabilities [5]. All these factors allow for a great level of applicability, enabling the realization of systems for the management of cities, health services, and industries [18].

The interest in Internet-of-Things has been growing massively, following the rising of connected devices along these past years. According to Siemens, there are around 26 billion physical devices connected to the Internet in 2020 and predictions are pointing at 75 billion in 2025 [4]. Although this allows for more opportunities, it is important to note that these devices are very different in their hardware and capabilities, which causes several problems in terms of developing the systems, as well as their scalability, maintainability, and security.

Visual Programming Languages (VPLs) allow the user to communicate with the system by using and arranging visual elements that can be translated into code [17]. It provides the user with an intuitive and straightforward interface for coding at the possible cost of losing functionality. There are several programming languages with different focuses, such as education, video game development, 3D building, system design, and even Internet-of-Things [53]. Node-RED[1] is one of the most famous open-source visual programming tools, originally developed by IBM's Emerging Technology Services team and now a part of the JS Foundation, which provides an environment for users to develop their own Internet-of-Things systems.

Non-functional attributes in a system are very important, specially attributes such as resiliency, fault-tolerance, and self-healing in Internet-of-Things systems. All these attributes mean that when an error or problem occurs, the system can adapt and overcome them dynamically and automatically.

Node-RED, mentioned above, is a centralized system, as well as most of the visual programming environments applied to IoT. A centralized architecture has a central instance that executes all computational tasks on the data provided by the other devices in the network. On the other hand, in a decentralized architecture the central instance, if it exists, partitions the computational tasks in independent blocks that can be executed by other devices. In IoT, these decentralized architectures are mentioned in Fog and Edge computing.

1.2 Problem Definition

Most mainstream visual programming tools focused on Internet-of-Things, Node-RED included, have a centralized approach, where the main component executes most of the computation on data provided by edge devices, e.g. sensors and gateways. There are several consequences to this approach: (a) computation capabilities of the edge devices are being ignored, (b) it introduces a single point of failure, and (c) local data is being transferred across boundaries (private, technological, political...) either without need or even in violation of legal constraints. The principle of Local-First [39] - i.e, data and logic should reside locally, independent of third-party services faults and errors - and NoCloud [52] - i.e, on-device and local computation should be prioritized over cloud service computation - is being ignored.

Besides being a single point of failure, centralized systems can be less efficient than decentralized ones and in this context, it might be the case, since there are computation capabilities that aren't being taken advantage of.

Chapter 4 expands on the problem definition, explaining it in bigger detail, defining its scope, desiderata, use cases and research questions.

[1] https://nodered.org/

1.3 Motivation

Internet-of-Things is a rapidly growing concept that is being applied to several areas, such as home automation, industry, health, city management, and many others. Given the number of existing systems with different protocols and architectures, it becomes difficult for a user to build a system that is in accordance with standards [3].

With the appearance of visual programming languages focused in IoT, more specifically Node-RED, users can build their own systems in an easier and streamlined way, removing the overhead of learning advanced programming concepts and protocols. These tools must be resilient, in order to withstand flaws and non-availability of devices as well as failure in the network. However, the majority of these tools are centralized, including Node-RED, and this type of architecture hinders the resiliency of the system. Given the existence of only one unit that executes most or all the processing of data, if this device fails, the system becomes nonfunctional. A possible solution would be increasing the redundancy of the system, creating more than one instance of the main unit. However, this approach has several costs, not only monetary but also in the increase in complexity.

1.4 Goals

The main goal of this dissertation is to leverage the computation capabilities of the devices in the network, increasing efficiency, fault-tolerance, resiliency and scalability in an Internet-of-Things system.

To achieve this goal, a prototype will be developed, extending or rewriting Node-RED, which enables IoT devices to communicate their "computational capabilities" back to the orchestrator. In its turn, the orchestrator is able to partition the computation and send "tasks" to the nodes, which are the devices in the network, leveraging their computation power and independence.

As a secondary goal, several other challenges will be tackled, viz: (i) computational capabilities of the devices in the network, (ii) detecting non-availability and using alternative computation resources, and (iii) exploring different alternatives of leveraging current IoT devices, including using firmwares that allow the execution of programs written in Lua, Javascript, Python, etc., amongst others.

1.5 Document Structure

Chapter 2 introduces the background information and explanation about concepts necessary for the full understanding of this dissertation. Chapter 3 describes the state of the art regarding the ecosystem of this project's scope, including a Systematic Literature Review on the state of the art of visual programming applied to the Internet-of-Things paradigm. Chapter 4 presents the problem this dissertation aims to solve, as well as the approach taken to solve it. Finally, Chapter 7 concludes this dissertation with a reflection on the future contributions of this project.

Chapter 2

Background

This chapter describes the necessary foundations regarding visual programming tools for the Internet-of-Things context. Section 2.1 describes the background of the Internet-of-Things paradigm and important concepts in that area, with a description of IoT architectures, including Fog and Edge, in Section 2.1.1. Section 2.3 describes the Node-RED programming tool and its architecture and uses. Finally, Section 2.2 mentions visual programming languages, their uses as well as their benefits and drawbacks.

2.1 Internet-of-Things

Internet-of-Things paradigm was defined by the committee of the International Organization for Standardization and the International Electrotechnical Commission [1] as:

> *"An infrastructure of interconnected objects, people, systems and information resources together with intelligent services to allow them to process information of the physical and the virtual world and react."*

This paradigm is built upon the network of heterogeneous devices interconnected between themselves, people and the environment. According to Buuya [35], the applications of IoT systems can be divided into four categories: (i) Home at the scale of an individual or home, (ii) Enterprise at the scale of a community, (iii) Utilities at a national or regional scale and (iv) Mobile, which is spread across domains due to its large scale in connectivity and scale.

However, one might think that IoT only relates to machines and interactions between them. Most of the devices we use in our day-to-day - mobile phones, security cameras, watches, coffee machines - are now computation capable of making moderately complex tasks and are constantly generating and sending information. This relates to the *human-in-the-loop* concept, where humans and machines have a symbiotic relationship [48].

2.1.1 IoT architectures

Internet-of-Things systems deal with big amounts of data from different sources and have to process it in efficient and fast ways. Typical IoT systems are composed of three tiers, which are:

- **Cloud Tier**, which is composed of data centers and servers, normally running remotely. It is characterized by having high computation power and latency.
- **Fog Tier** is composed of gateways and devices that are normally between the cloud servers and the edge devices. This tier has less latency than the cloud, more heterogeneity, and geographical distribution.
- **Edge Tier** contains all the edge devices (sensors, embedded systems, light sources, etc). Since its devices have smaller computational capabilities, this tier is the one with smaller computation power but with less latency value.

These tiers can also be called Application Layer, Network Layer, and Perception Layer [44], respectively, which is compatible with the characterizing mentioned above. However, not all devices in each tier map to the respective layer. One example is a third-party service that gives readings. It can be contained in the Perceptive Layer, but it is not included in the Edge Tier.

New paradigms of computing appeared related to each of these tiers. The majority of IoT systems use a Cloud Computing architecture, where it takes advantage of centralized computing and storage. This approach has several benefits, such as increased computational capabilities and storage, as well as easier maintenance. However, it comes with several problems such as (a) high latency and (b) high use of bandwidth, due to the need to send the data generated from the sensors to the centralized unit [40]. Systems that only use cloud computing are not scalable, specially real-time applications, which are sensitive to increased latency. With the increasing computation capabilities of edge devices and the requirements of reduced latency, two new paradigms appeared - Fog and Edge Computing.

2.1.1.1 Fog Computing

Nowadays, with the improvement of wireless networks and the hardware and software of mobile devices, there is a possibility to take advantage of these variables in the computational execution of IoT systems. This will allow for devices in the network to communicate and share resources between them, reducing latency. The central instance, which in the paradigm before executed all the computation, now serves as a scheduler and state manager of the communication between the devices, occasionally providing necessary resources if needed. The paradigm described before is called Fog Computing, which aims to bring computing closer to the perception tier, bringing the cloud near the edge of the network [41]. It focuses on distributing data throughout the IoT system, from the cloud to the edge devices, making the system a distributed one.

According to Buuya [15], Fog Computing has several advantages: (i) reduction of network traffic by having edge devices filtering and analyzing the data generated and sending data if necessary, (ii) reduced communication distance by having the devices communicate between them without using the cloud as middleman, (iii) low-latency by moving the processing closer to the data source rather than communicating all the data to the cloud for it to be processed, and (iv) scalability by reducing the burden on the cloud, which could be a bottleneck for the system.

It is possible to see an example of the architecture of an IoT system using the Fog paradigm in Figure 2.1. The Fog Computing connects the cloud to the edge devices, normally with the use of access points and gateways.

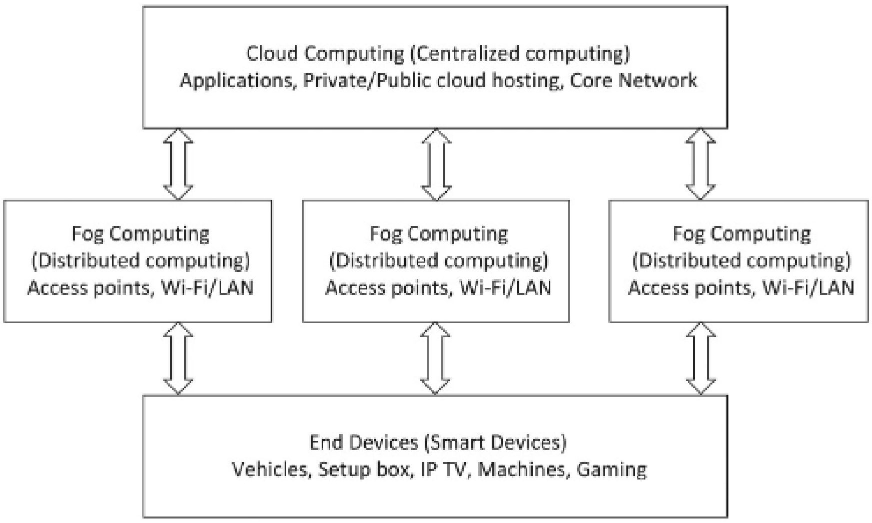

Figure 2.1: Fog Computing Architecture [15]

Despite all the advantages, Fog Computing has several requirements and difficulties. To make a successful and efficient distribution of computation and communication, it requires knowledge about the resources of the connected devices. The complexity is also bigger than Cloud Computing since it needs to work with heterogeneous devices with different capacities.

2.1.1.2 Edge Computing

Edge Computing, also known as Mist Computing, is a distributed architecture that uses the devices' computational power to process the data they collect or generate. It takes advantage of the Edge tier, which contains the devices closer to the end-user - smartphones, TVs, sensors, etc. The goal of this paradigm is to minimize the bandwidth and time response of IoT systems while leveraging the computational power of the devices in them. It reduces bandwidth usage by processing data instead of sending it to the cloud to be processed, which is also correlated to reduced latency since it does not wait for the server response. In addition to these advantages and related to their cause, Edge Computing also prevents sensitive data from leaving the network, reducing data leakage and increasing security and privacy [43, 58].

In this paradigm, each device serves both as a data producer and a data consumer. Since each device is constrained in terms of resources, this brings several challenges such as system reliability and energy constraints due to short battery life and overall security. Other issues consist of the lack of easy-to-use tools and frameworks to build cloud-edge systems, non-existent standards regarding the naming of edge devices and the lack of security edge devices have against outside threats such as hackers [57].

There is some confusion in the research community regarding the concepts of Fog and Edge computing. The publication from Iorga et. al. [37] was used to inspire the definitions of these terms in this dissertation. Edge Computing focuses on executing applications in constrained devices, without worrying about storage or state preservation. On the other hand, Fog Computing is hierarchical and includes devices with more capabilities, capable of control activities, storage, and orchestration.

2.2 Visual Programming Languages

Visual Programming, as defined by Shu, consists of using meaningful graphical representations in the process of programming [59]. With this definition, we can consider Visual Programming Languages (VPLs) as a way of handling visual information and interaction with it, allowing the use of visual expressions for programming. According to Burnet and Baker [13], visual programming languages are constructed to *"improve the programmer's ability to express program logic and to understand how the program works"*.

There are several applications of visual programming languages in different areas, such as education, video game development, automation, multimedia, data warehousing, system management, and simulation, with this last area being the area with most use cases [53].

Visual programming languages have several characteristics, such as a concrete process and depiction of the program, immediate visual feedback and requires the knowledge of fewer programming concepts (e.g. pointers, memory allocation, etc) [13].

VPLs were categorized by Downes [12] based on their visual paradigms and architecture:

- **Purely Visual Languages**, where the creation is made using only graphical elements and the subsequently debugging and execution is made in the same environment.
- **Hybrid text and visual systems**, where the programs are created using graphical elements but their executions is translated into text language.
- **Programming-by-example systems**, where a user uses graphical elements to teach the system.

- **Constraint-oriented systems**, where the user translates physical entities into virtual objects and applies constraints to them, in order to simulate their behavior in reality.
- **Form-based systems**, which were based on the architecture and behavior of spreadsheets.

The categories mentioned can be present in a single system, making them not mutually exclusive.

2.3 Node-RED

Node-RED [30] is a programming tool applied to the development of Internet-of-Things systems. It was first developed with to manipulate and visualize mappings between MQTT topics in IBM's Emerging Technology Services group. It then expanded into a more general open-source tool, which is now part of the JS Foundation.

It is a web-based tool consisting of a run time built with the Node.js framework and a browser-based visual editor. This tool provides the end-user with a simple interface to connected devices and APIs, using a flow-programming approach. Programs are called *flows*, built with *nodes* connected by wires. Each node corresponds to an action, such as input, output, data processing, etc.

The Node-RED interface has three components: (1) Palette, (2) Workspace and (3) Sidebar. The Palette contains all the nodes installed and available to use, divided into categories. They can be used by dragging them into the workspace and additional features for each node are accessible by double-clicking them. The Workspace is where the flows are created and modified. It is possible to have several *flows* and *sub-flows* accessible with the use of tabs. Lastly, the Sidebar contains information about the nodes, the debug console, node configuration manager and the context data. Figure 2.2 showcases the visual interface of Node-RED and its elements.

Figure 2.2: Node-RED environment

One example of a *flow* can be seen in picture 2.3, where a request is being made in intervals of 5 minutes to an HTTP URL that returns a CSV with the feed of significant earthquakes in the last

7 days. The data from the CSV is then printed to the debug console and, if the magnitude is equal or bigger than 7, the message "PANIC!" is printed to the console.

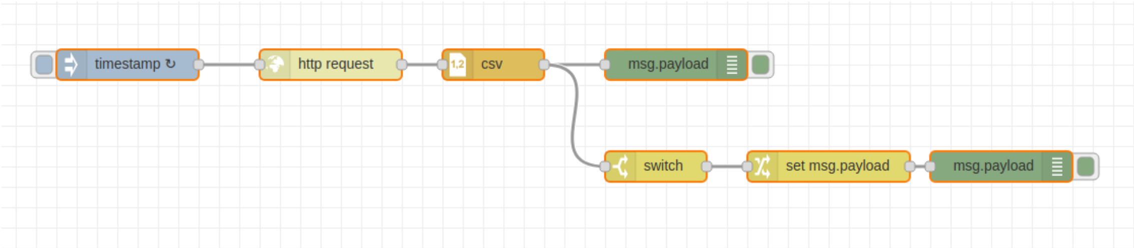

Figure 2.3: Example of a Node-RED flow

Regarding the architecture of Node-RED, the `Node` base class is a subclass of Node.js event APIs `EventEmitter`. This class implements an observer design pattern that maintains a subscriber list of all the nodes connected to it by *wires* and emits events to them. When a node finishes processing data from external sources or another node, it calls the methods `send()` with a Javascript object. In its turn, this method calls the `EventEmitter emit()` method that sends named events to the subscribed nodes.

Being open-source, Node-RED takes advantage of a large community that contributes with new nodes and improvements to the tool. It is the most popular open-source visual programming tool for IoT, with more than 9,300 stars on Github.

2.4 Summary

This chapter introduces two areas that are fundamental to the understanding of this dissertation. Internet-of-Things is defined, as well as its use cases and categories. Fog and Edge computing paradigms are explained, which will be mentioned throughout this document. Node-RED is introduced as a visual programming tool for IoT and its architecture is explained. Finally, a definition and categorization of visual programming languages are introduced and explained.

Chapter 3

State of the Art

This chapter describes the state of the art in visual programming tools in the Internet-of-Things context, as well as decentralized methods of work distribution in flow-based architectures. Section 3.1 presents a systematic literature review on the topic of visual programming tools applied to the Internet-of-Things paradigm, which aims to answer the research questions defined in section 3.1.1.1. Section 3.1.2 contains the results of the Systematic Literature Review, as well as their categorization. Section 3.1.3 contains the additional tools found in a survey and their analysis. The discussion and analysis of the tools found as well as the answering of the research questions made previously are made in Section 3.1.4. The Systematic Literature Review conclusions are presented in Section 3.1.5. Lastly, Section 3.2 contains the state of the art of visual programming tools applied to IoT that implement a decentralized architecture.

3.1 Systematic Literature Review

A Systematic Literature Review was made to gather information on the state of the art of visual programming applied to the Internet-of-Things paradigm. The goal of a systematic literature review is to synthesize evidence with emphasis on the quality of the it [50].

3.1.1 Methodology

During this Systematic Literature Review, a specific methodology was followed to reduce bias and produce the best results [50]. We started by defining the research questions to be answered as well as choosing data sources to search for publications.

3.1.1.1 Research Questions

In this Systematic Literature Review we intend to answer the following questions:

SRQ1: What relevant VPLs applied to IoT orchestration exist? Internet-of-Things is a paradigm with several years, and its integration with visual programming languages makes their development easier for the end-user. The tools that integrate these two paradigms are useful and reduce the overhead of programming or prototyping IoT systems.

SRQ2: What is the tier and architecture of the tools found in RQ1? IoT systems can belong to one or more of tiers - Cloud, Fog and Edge as well as implement a centralized or decentralized architecture. A visual programming tool applied to IoT orchestration can be used to facilitate the development of systems that operate on these tiers. Each tier and type of architecture offers vantages and disadvantages, which are important to understand the usages and characteristics of a system.

SRQ3: What was the evolution of VPLs applied to IoT orchestration over the years? To understand the field of visual programming tools applied to IoT, more specifically its orchestration, it is important to perceive its evolution.

3.1.1.2 Databases

The publications retrieved during this research were retrieved from the following databases, which are considered good and reliable sources:

- IEEE
- ACM
- Scopus

3.1.1.3 Search Process

To obtain results from the databases chosen, a research question was written with the union of the keywords "visual programming", "node-red", "dataflow" and the intersection with the keyword "Internet-of-Things".

```
((vpl OR visual programming OR visual-programming) OR (node-red OR node red OR
    nodered) OR (data-flow OR dataflow)) AND (IoT OR Internet-of-Things OR
    internet-of-things)
```

The search was performed in October of 2019 and the results produced are the ones present in Table 3.1.

Database	Total Results	Extracted Results
IEEE	420	379
ACM	294,259	2021
Scopus	553	500

Table 3.1: Systematic Literature Review search results per database

3.1.1.4 Inclusion Criteria

To be included in the results, all publications should respect the inclusion criteria. If one of the criteria were not checked, the publication would not be included in the results. The inclusion criteria are the following:

1. On the topic of visual programming in Internet-of-Things;
2. Research findings include sufficient explanation;
3. Publication year in the range between 2008 and 2019.

3.1.1.5 Exclusion Criteria

In addition to the inclusion criteria, all publications were analyzed in their compliance with the exclusion criteria. If any publication failed to comply with at least one of the exclusion criteria, it would not be included in the results. The exclusion criteria are the following:

1. Has less than two (non-self) citations when more than five years old;
2. Presents just ideas, tutorials, integration experimentation, magazine publications, interviews or discussion papers;
3. Presents a tool or framework that doesn't support the orchestration of multiple devices;
4. Not in English.
5. In a format other than camera ready (PDF)

3.1.1.6 Quality Assessment

To classify if a publication is relevant to the research field, 4 assessments were made to better facilitate the process. The quality assessments are the following:

Quality Assessment Query	Quality Indicator (0-2)
Is the publication relevant to us?	BARELY-PARTIALLY-SATISFACTORILY
Are research objectives included adequately in the publication?	NO-PARTIALLY-YES
Are challenges and limitations well defined?	NO-PARTIALLY-YES
Is the proposed contribution well described?	NO-PARTIALLY-YES

Table 3.2: Parameters for measuring the quality of a publication

Each assessment was posed in the form of a question, and to each question, there were three possible answers, with a numeric value each. If a publication didn't address the assessment the value would be 0, if the assessment was partially addressed the value would be 1. If the assessment was successfully satisfied, the value would be 2. In the end, the sum of all the assessments would represent the quality of the publication.

3.1.1.7 Evaluation Process

The evaluation process of the publications followed six steps with specific purposes:

1. **Range:** Publications are evaluated on date range, between 2008 and 2019;
2. **Relevance:** Title and abstract are examined for relevance concerning the defined research field;
3. **Inclusion:** Publications are evaluated against inclusion and exclusion criteria. Any publications not meeting the full inclusion criteria or failing to comply with any exclusion criteria are discarded;
4. **Specificity:** The publications are analyzed to confirm if it relates closely enough to the defined research field;
5. **Data:** Selected publications are examined for information related to the research questions and their contribution details;
6. **Publication quality:** Publications are appraised using quality criteria defined in Table 3.2.

The results from the evaluation process can be seen in Table 3.3.

Step	Nº of publications	Nº of excluded publications
Search	2698	N/A
Duplicates	2626	72
Exclusion/Inclusion criteria (Titles and Abstracts)	65	2561
Exclusion/Inclusion criteria (Introduction and Conclusion)	22	43

Table 3.3: Publications per step

3.1.2 Results

After analyzing the 22 publications, we organized them by categories. [53] is a survey and the remaining 21 were frameworks or tools.

Regarding the survey, publication [53] makes an in-depth review of 13 visual programming languages in the field of IoT, comparing them under four attributes: (1) programming environment, (2) license, (3) project repository and (4) platform support. The author concluded with some advantages of using visual programming languages, such as the ease of visualizing programming logic, useful for rapid development and less burden on handling syntax error. However, some negative aspects were also mentioned, such as the large amount of time required for building simple IoT applications.

The remaining 21 articles are frameworks or tools of visual programming applied to IoT. One of the tools is repeated in two papers, which showcases its evolution. The frameworks are:

1. **Belsa et al.** [8] is a solution for connecting devices from different IoT platforms, using Flow-Based Programming with Node-RED. Its motivation is based on the limitation imposed by the IoT platform on communication between components and extensibility. This limits the possibility to interact with other platforms' services. To validate their solution, they implemented a use case in the domain of transportation and logistics, with a service that uses five different types of applications. The developed tool offers access to available services in a centralized visual framework, where end-users can use them to build more complex applications.

2. **Ivy** [24] proposes the next step forward regarding visualization applied to IoT with a visual programming tool that allows its users to link devices, inject logic and visualize real-time data flows using immersive virtual reality. It provides the end-users with an immersive virtual reality that allows them to visualize the data flow, access to debugging tools and real-time deployment. Each programming construct called node - data flow architecture - has a distinct shape and color, which facilitates the understanding of the system being built or debugger for the user. The experiences made to validate the prototype were positive, with the participants being receptive to Ivy and indicating use cases for it.

3. **Ghiani et al.** [31] proposition is to build a collection of tools that allow non-developer users to customize their Web IoT applications using trigger-actions rules. The proposed solution provides a web-based tool that allows users to specify their trigger-action rules using *IFTTT*, as well as a context manager middleware that can adapt to the context and events of the devices and apply rules to the system. To validate the developed tool, an example home automation application that displays sensor values and directly controls appliances were built. The results were for the most part positive, and the issues found are related to usability and visual clues.

4. **ViSiT** [2] uses the jigsaw puzzle metaphor to allow its end-users to implement a system of connected IoT devices. It provides a web-based visual tool connected with a web-service

that, given a jigsaw representation, generates an executable implementation. Their goal is achievable by adapting model transformations used by software developers into understandable metaphors for non-developers to use. They validated the developed tool with a usability evaluation, which was overall positive, with a great percentage considering the tool useful and providing real-life scenarios where they could implement it.

5. **Valsamakis and Savidis** [66] propose a framework for Ambient Assisted Living (AAL) using IoT technologies, which allows for customized automation. It uses visual programming languages to facilitate their end-users - carers, family, friends, elderly - to build and modify automation. They built a visual programming framework that introduces smart objects grouping in tagged environments and real-time smart-object registration through discovery cycles. It runs on typical smartphones and tablets and is built in Javascript, allowing it to run in browsers. Their future work focuses on integrating different visual programming paradigms to fully accomplish the requirements of the end-user.

6. **WireMe** [49] is an intuitive solution for building, deploying and monitoring IoT systems, built with non-developer end-users in mind but also extensible for advanced users to built over it. The developed solution makes use of Scratch, a visual programming interface, to provide its users with a customizable dashboard where they can monitor and control their IoT system as well as program automation tasks. It has a Main Control Unit responsible for communicating the device's status to the dashboard via MQTT, which is programmable using their visual interface and Lua programming language. Their tool was validated by students around 16 years old and engineering students without programming experience. The results were not totally positive, with some students not being able to create the required simple logic. Future work consists of improving programming blocks to become more intuitive.

7. **VIPLE** [21], Visual IoT/Robotics Programming Language Environment, is a new visual programming language and environment. It provides an introduction to topics such as computing and engineering and tools for more practical domains like software integration and service-oriented computing. It focuses on complex concepts such as robot as a service (Raas) units and Internet of Intelligent Things (IoIT) while studying the programming issues of building systems classified as such. The developed tool is extremely powerful and has been tested and used in several universities since 2015.

8. **Smart Block** [7] is a block-based visual programming language and visual programming environment applied to IoT systems, that allows non-developer users to build their systems more easily. Their solution is specific to the home automation domain, like Smart Things. The language was designed using IoTa calculus, used to generalize Event-Condition-Action rules for home automation. The environment was built using a client-side Javascript library called Blockly, which allows for the creation of visual block languages. Future work for this

project consists of supporting device grouping and security by expanding custom blocks, as well as extending the tool for other domains besides home automation.

9. **PWCT** [29] is a visual programming language applied to build IoT, Data Computing and Cloud Computing systems. Its goal consists of reducing the cost of development of these types of systems by providing an easy and more productive development tool. The language was meant to compete with text-based languages such as Java and C/C++. It makes use of graphical elements to replace code and has 3 main layers: (1) the VPL layer, composed of graphical elements, (2) the middleware layers, responsible for connecting the VPL layer to the system's view, which is the (3) System Layer, responsible for dealing with the source code generated by the first layer. The created solution received positive feedback from the community, with more than 70,000 downloads and 93% of user satisfaction.

10. **DDF** [34] is a Distributed Dataflow (DDF) programming model for IoT systems, leveraging resources across the Fog and the Cloud. They implemented a DDF framework extending Node-RED, which originally is a centralized framework. Their motivation comes from the possibility to develop applications from the perspective of Fog Computing, leveraging these devices for efficiency and reduced latency, since there is a big amount of resources such as edge devices and gateways in IoT systems. They evaluated their prototype using a small scale evaluation, which was positive. The results showed that their DDF framework provides an easy alternative for designing and developing distributed IoT systems, despite having some open issues such as not having a distributed discovery of devices and networks.

11. **GIMLE** [64], Graphical Installation Modelling Language for IoT Ecosystems), is a visual language that uses visual elements to model domain knowledge using significant ontological requirements. The goal of this language is to fill the gap of modeling requirements on the physical properties of IoT installations by proposing a new process for configuring industrial installations. It makes use of flow-based and domain-based visual programming to isolate the requirements' logical flow from their details. The developed tool supports reuse within the models, which is valuable due to the repetitive nature of industrial installations but it still needs to clarify its scope within the current practice and its use in production settings.

12. **DDFlow** [47] is a macro-programming abstraction that aims to provide efficient means to program high quality distributed apps for IoT. The authors refer a lack of solutions for complex IoT systems programming, causing developers to build their own systems, which leads to a lack of portability/extensibility and results in a lot of similar systems that do the same thing, but are "different" because they were created by different programmers. Developers use Node-Red to specify the application functionalities and DDFlow handles scalability and deployment. The authors describe DDFlow's goal as to allow developers to formulate complex applications without having to care about low-level network, hardware and coordination details. This is done by having the DDFlow accompanying runtime dynamically scaling and mapping the resources, instead of the developer. DDFlow gives developers the

possibility to inject custom code on nodes and have custom logic if the available nodes are not enough for some tasks.

13. **Kefalakis et al.** [38] proposition consists of a visual environment that operates over the OpenIoT architecture and allows for the development of IoT applications with reduced programming effort. Modeling IoT services with the developed tool is made by specifying a graph that corresponds to an IoT application, which can be validated and its code generated and performed over the OpenIoT middleware platform. It aims to fill the gap of tools that provide support for the development and deployment of integrated IoT applications. The approach taken presents several advantages: (1) it leverages standards-based semantic model for sensor and IoT context, making it easier to be widely adopted, (2) it is based on web-based technologies which open the possibilities of applications from developers and (3) it is open source.

14. **Eterovic et al.** [27] proposes an IoT visual domain-specific modeling language based on UML, with technical and non-technical users in mind. The authors defend that, with the evolving nature of IoT, the future end user will be a common person, with no programming knowledge. To solve the problems this future brings, it is important to build a visual language easy enough to be understood by non-technical people but expansible enough to represent complex systems. To evaluate the proposed solution, they invited 11 users of different levels of UML expertise to model a simple IoT system with the developed language. The System Usability Score was positive, as well as the Tasks Success Rate. Despite the positive score, some future actions would be the testing of the language with a more complex task as well as the integration of advanced UML notations.

15. **FRED** [11] is a Frontend for Node-RED, a development tool that makes it possible to host multiple Node-RED processes. It can be used to connect devices to services in the cloud, manage communication between devices, creating new web app applications and APIs and event integrate services. To provide all these features, FRED allows the running of flows for multiple users, in which all flows get fair access to resources such as CPU, memory, storage, as well as secure access to flow editors and the flow runtime. The authors concluded that FRED is useful for users learning about Node-RED and allows to rapidly prototype cloud-hosted applications.

16. **WoTFlow** [10] is proposed as a cloud-based platform that aims to provide an execution environment for multi-user cloud environments and individual devices. It aims to take advantage of data flow programming, which allows parts of the flow to be executed in parallel in different devices. Based on this, the tool will take advantage of the ability to split and partition the flows and distribute them by edge devices and the cloud. The state of the developed tool was in the early stages, with future expansions based on the use of optimization heuristics, automatic partitioning based on calculated constraints, security and privacy.

17. **Besari et al.** [56] [9] proposes an IoT-based GUI that aims to control sensors and actuators in an IoT system using an android application, in which the users use a visual programming language to configure and interact with the IoT system. The system was tested with a Pybot, which is a robot that is programmable similarly to an IoT system, with sensors and actuators. After testing and evaluating the system, the authors came to a score of 72.917 (out of 100) for the Pybot software, which is considered "GOOD". The overall acceptability of the system was "ACCEPTABLE", which led the authors to consider the application accepted by users.

18. **CharIoT** [63] is a programming environment that promises its end-users a solution to unify and support the configuration of IoT environments. It provides three blocks of support: capturing higher-level events using virtual sensors, construction of automation rules with a visual overview of the current configuration and support for sharing configuration between end-users using a recommendation mechanism. To enable the capturing of higher-level events, it was developed two types of virtual sensors. The programmed virtual sensor provides more accessible and understandable abstractions (defining that a room is "cold" if the temperature is below 20^{o}C). The demonstrated virtual sensors are more complex, requiring the user to provide a demonstration of the occurrence and lack of occurrence of the event (for example, the event of someone knocking on the door and the absence of someone knocking on the door). This last one requires the training of a Random Forest classifier. This programming environment is similar to IFTTT but goes one step further, with smarter event capturing and reusing of configurations, allowing the end-user to build faster and more robust IoT installations.

19. **Desolda et al.** [22] proposition uses a tangible programming language that allows non-programmers to configure the behavior of the smart objects to create and customize the smart environments. The main goal was to create, with the developed technology, a scenario of a smart museum. The authors defend that the personalization of a smart environment cannot be limited by the synchronization of smart devices and it may require experts to build the narrative of them, much like a museum said that. With this in mind, they introduced custom attributes to assign semantics to involved objects, in order to empower and simplify the creation of event-condition-action rules. In conclusion, this is ongoing research focused on developing new technology with an interaction paradigm that supports domain experts' input in the creation of smart environments. In addition, the fact that this technology uses expensive material (tabletop surface as a digital workspace) does not allow a regular user to use it as stated in the introduction.

20. **Eun et al.** [28] proposes an End-User Development (EUD) tool that allows users to develop their personal applications. It uses the dataflow approach, which allows for a more generalized programming experience as well as the facility to build more complex programs with simple modules. The proposed tool has three main components: Service Template Authoring Tool, Service Template Repository, and Smartphone Application. The first one

allows the end-user to build more complex methods using atomic templates (components with simple functionality, like opening a curtain if it receives a command). The Service Template Repository contains the proprietary atomic templates as well as ones built by the user. Lastly, the Smartphone Application runs and manages the applications built by the user, as well as their requirements and dependencies. The developed EUD tool was compared with *IFTTT* and Zapier, other tools focused on end-user development. *IFTTT* and the developed tool are more similar, focusing on consumer development, IoT and Home, with Zapier focusing on business. Both Zapier and *IFTTT* use the Trigger-Action paradigm (TAP), which differs from the dataflow paradigm used in this paper's tool.

The mentioned frameworks and tools were divided into the following categories, according to several characteristics:

Scope Some tools have specific use cases in mind (*e.g.* smart cities, Home automation, industry, etc). Therefore, knowledge of the scope of a tool is useful to assess if it solves a problem or fills a specific gap in the literature. Example values consist of *smart cities, home automation, education, industry* or *many* if there is more than one.

Architecture Visual programming tools applied to the Internet-of-Things can have a centralized or decentralized architecture, based on their use of Cloud, Fog or Edge Computing architecture. Possible values are *Centralized*, *Decentralized* and *Mixed*.

License The license of software or tool is essential in terms of its usability. Normally, an open-source software reaches a bigger user base and allows them to expand and contribute to it. Possible values are the name of the tool license or N/A if it does not have one.

Tier IoT systems, as explained in Section 2.1.1 is composed of three tiers - *Cloud*, *Fog* and *Edge*. A tool can interact in several of these tiers, which shapes the features it contains and how it is built.

Scalability Defines how the tool or framework scales. It can be calculated based on metrics used to test the performance of the system. In this case, we considered scalability in terms of number and different types of devices supported. Possible values are *low*, *medium*, *high* or N/A, in case there is no sufficient information.

Programming According to Downes and Boshernitsan [12] and also mentioned in Section 2.2, visual programming languages can be classified in five categories: (1) Purely Visual languages, (2) Hybrid text and visual systems, (3) Programming-by-example systems, (4) Constraint-oriented systems and (5) Form-based systems. These classifications aren't mutually exclusive. It is important to know which type, so that might be possible to assess the type of experience the tool provides to the user and its architecture.

Web-based Defines if the visual programming language and/or environment can be used in a browser. It is useful in terms of the usability of the tool.

Tool	Scope	Architecture	License	Tier	Scalability	Programming	Web-based
Belsa et al. [8]	Several	Centralized	-	Cloud	High	Hybrid text and visual system	•
Ivy [24]	Several	Centralized	-	Cloud	Medium[7]	Purely visual language	
Ghiani et al. [31]	Home Automation	Centralized	-	Cloud	-	Form-based programming	•
ViSiT [2]	Several	Centralized	-	Cloud	High	Hybrid text and visual systems	•
Valsamakis and Savidis [66]	Ambient Assisted Living	Centralized	-	Cloud	-	Hybrid text and visual system	•
WireMe [49]	Education, Home Automation	Centralized	-	Cloud	-	Hybrid text and visual system	
VIPLE [21]	Education	Centralized	-	Cloud	-	Hybrid text and visual system	
Smart Block [7]	Home Automation	Centralized	-	Cloud	-	Hybrid text and visual system	•
PWCT [29]	Several	Centralized	GNU GPL v2.0	-[1]	High	Hybrid text and visual system	
DDF [34]	-	Decentralized	Apache 2.0	Fog	High	Hybrid text and visual system	•
GIMLE [64]	Industry	Centralized	-	Cloud	High	Hybrid text and visual system	•
DDFlow [47]	Security	Decentralized	-	Fog and Edge	-	Hybrid text and visual system	•
Kefalakis et al. [38]	-	Centralized	LGPL V3.0[3]	Cloud	-	Hybrid text and visual system	
Eterovic et al. [27]	Home Automation	-[4]	-	-	-	Hybrid text and visual system	-
FRED [11]	Several	Centralized	-[5]	Cloud	High	Hybrid text and visual system	•
WoTFlow [10]	-	Decentralized	-	Fog and Edge	-	Hybrid text and visual system	•
Besari et al. [9] [56]	Education	Centralized	-	Cloud	-	Hybrid text and visual system	
CharIoT [63]	Home Automation	Centralized[6]	-	Cloud and Edge[6]	High[6]	Form-based programming	•
Desolda et al. [22]	Smart Museums	-	-	-	-	Hybrid text and visual system	
Eun et al. [28]	Home Automation	Centralized	-	-	-	Form-based programming	•

Table 3.4: Small circles (•) mean *yes*, hyphens (-) means *no information available*, empty means *no* and asterisk (*) means more than one. The ⋆ symbol represents certainty in the evaluation made.

[1] Used for several purposes, didn't specify the tier it is located in regarding IoT.
[2] Since it uses Node-RED, this information was based on its architecture.
[3] Under the same license of OpenIoT.
[4] No information given regarding the architecture of the environment created, only the VPL.
[5] No information about license is given, but further research discovered that it has paid plans and no source code available.
[6] CharIoT uses the Giotto stack, https://iotexpedition.org/about.html, from where we retrieved this information.
[7] Certainty regarding this information is low.

3.1.3 Expanded Search

The results of the Systematic Literature Review are disclosed in Section 3.1.2. However, some tools were found in non-systematic surveys [53] that are not present in the results mentioned. Possible reasons for this divergence may consist of:

1. tools not having academic publications associated with them, making it impossible to be returned as results of searches to the publication databases mentioned in Section 3.1.1.2. One example is *Node-RED* [30].

3.1.3.1 Expanded Results

The results from the found survey [53] were analyzed. The retrieved tools were assessed against the evaluation process defined in Section 3.1.1.7 and characterized by the categories mentioned in Section 3.1.2. Using the methodology described, the results are:

1. **Node-RED** [30] is a visual programming environment applied to the IoT paradigm. It makes use of flow-based development and supports a wide range of devices and APIs. Due to being open-source and extendable, its large community contributes with features that enrich the tool, some of them talked about in Section 3.1.2 (e.g. *FRED* [11] and *DDF* [34]).

2. **NETLab Toolkit** [65] is a visual environment that makes use of *drag-and-drop* actions to allows its users to build IoT applications. It provides a web interface to connect sensors, actuators, and others for the development of quick prototypes.

3. **NooDL** [46] is a platform that provides a visual programming interface for prototyping applications. It allows for the creation of interfaces, using live data and supporting several types of hardware. Although it is not specific to IoT, NooDL covers the programming of IoT systems. It makes use of MQTT broker agents for connecting devices and visual paradigms such as *nodes*, *connections*, and *hierarchies* to allow the user to build its system.

4. **DGLux5** [23] for DSA is a *drag-and-drop* visual language and environment that allows its users to build applications tailored for Distributed Services Architecture (DSA) IoT middleware. It provides a dashboard for analyzing and controlling device data in real-time, as well as building the system only using visual elements.

5. **AT&T Flow Designer** [6] is a visual tool incorporated in a cloud development environment, applied to the development of IoT systems. Its visual paradigm is similar to Node-RED, with the notion of *nodes* and *wires*. This tool provides an easy iteration and improvement of a product, as well as an easy deployment.

6. **GraspIO** [42] is a Graphical Smart Program for Inputs and Outputs that contains a block *drag-and-drop* visual paradigm that allows its users to build applications for the *Cloudio* hardware. It offers a Cloud Service that connects and manages all *Cloudio* devices, making them available at the user mobile device.

7. **Wyliodrin** [67] is a browser-based visual programming environment that allows the development of IoT systems of several devices, such as Raspberry Pi, Arduino, Intel Galileo, Intel Edison, and others. It provides a *drag-and-drop* environment, as well as support for text-based languages. A dashboard for visualizing the data collected is provided.

8. **Zenodys** [16] provides a *drag-and-drop* interface to build application backends as well as user interfaces. Its computing engine can run in several types of devices, from the cloud to chips, devices and distributed computers. Zenodys contains a visual debugger as well as support for text-based programming and code generation.

Tool	Scope	Architecture	License	Tier	Scalability	Programming	Web-based
Node-Red [30]	Several	Centralized	Apache 2.0	Cloud and Edge	High	Hybrid text and visual system	•
NETLab Toolkit [65]	-	-	GNU GPL	Edge[2]	-	Hybrid text and visual system	•
NooDL [46]	Several	-	NooDL End User License[1]	Cloud[2]	-	Hybrid text and visual system	
DGLux5 [46]	Several	-	DGLux Engineering License	Cloud and Fog[2]	High[2]	Purely visual language	
AT&T Flow Designer [6]	Several	-	GNU GPL3	Cloud[2]	High[2]	Hybrid text and visual system	•
GraspIO [42]	Education	-	BSD	Cloud[2]	-	Purely visual language	
Wyliodrin [67]	Several	-	GNU GPL3	All[2]	-	Hybrid text and visual system	•
Zenodys [16]	Several	-	GNU GPL3	Cloud[2]	High[2]	Hybrid text and visual system	•

Table 3.5: Characterization of VPls applied to IoT from survey [53]. Small circles (•) mean *yes*, hyphens (-) means *no information available*, empty means *no* and asterisk (*) means more than one.

[1] Available at https://www.noodl.net/eula
[2] Certainty regarding this information is low.

3.1.4 Analysis and Discussion

The tools presented in this Systematic Literature Review passed the evaluation process defined in Section 3.1.1.7. Tools that only supported one device were left out, as well as tools that extended a VPL applied to IoT.

3.1.4.1 Evolution Analysis

To understand the evolution of visual programming languages applied to IoT, the publication years of the tools found in Section 3.1.2, as well as the launch years of the survey tools of Section 3.1.3, were grouped. Figure 3.1 contains the evolution, where it can be observed that there was a bigger amount of work related to this topic in the years 2017 and 2018. The year 2019 still doesn't have conclusive data.

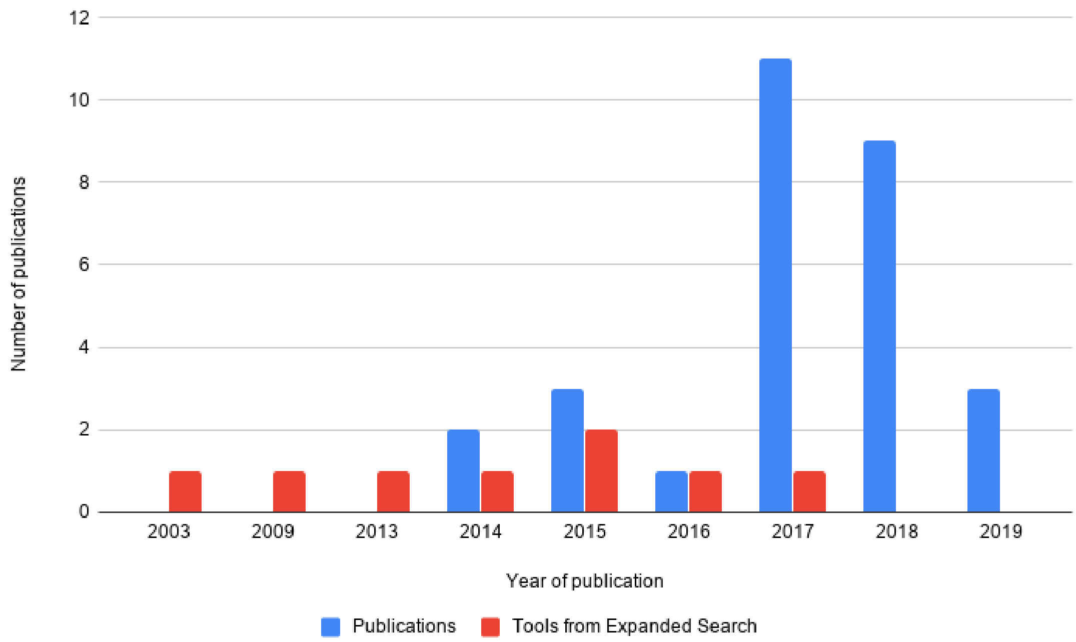

Figure 3.1: Publications and tools of VPL tools applied to IoT per year

3.1.4.2 Result Analysis

Scope Most of the tools found have several scopes, such as education, industry or home automation. From the 28 tools, 6 were specific to Home Automation, 4 to Education, 3 to specific domains, 1 for industry and the remainder had a wide range of use cases.

Architecture From the 28 tools found, 16 tools have a centralized architecture, 3 are decentralized and the remaining 9 didn't have enough information to reach a conclusion.

License Most of the tools didn't mention a license and the ones who did were in its majority open source (e.g. GNU GPL2, GNU GPL3, Apache 2.0 and LGPL3).

Tier From the 28 tools, 4 did not specify the tier they are situated in. From the remaining 24, 16 are situated in the Cloud tier, 2 are in the Fog and Edge, 2 in the Cloud and Edge, 1 in the Edge tier, 2 are work both in the Cloud and Edge tiers and, lastly, 1 in all the tiers.

Scalability The majority of tools analyzed don't have scalability metrics analyzed, more specifically the number of devices supported by them. The ones that do have high scalability, which concludes that that result is analyzed when the tool has support for it.

Programming From the 28 analyzed tools, 22 employ a hybrid text and visual system visual programming paradigm, while 3 use a purely visual and the other 3 a form-based one.

Web-based The majority of tools analyzed are web-based, being accessible with the use of a browser. Only one tool didn't provide an environment, only a specification of a visual programming language.

3.1.4.3 Research Questions

The research questions presented in Section 3.1.1.1 served as a way of directing the research of this Systematic Literature Review and obtain answers to relevant questions regarding the available tools that apply visual programming languages to the IoT domain. These answers are:

SRQ1: What relevant VPLs applied to IoT orchestration exist? From the analyzed tools in Section 3.1.2 and 3.1.3, there are around 28 visual programming tools applied to IoT orchestration.

SRQ2: What is the tier and architecture of the tools found in RQ1? Tables 3.4 and 3.5 give an overview of the characteristics of all the tools found. In these tables and subsequent analysis in Section 3.1.4.2 it is concluded that the majority of the tools have a centralized architecture and work in the Cloud tier.

SRQ3: What was the evolution of VPLs applied to IoT orchestration over the years? As it can be observed in Section 3.1.4.1 and more specifically in Figure 3.1, there are visual programming tools applied to the orchestration of IoT since 2003, and in 2017 and 2018 there was a bigger number of publications with a focus on building these type of tools.

3.1.5 Conclusions

In this Systematic Literature Review, 2698 publications were analyzed from IEEE, ACM and Scopus databases. This resulted in 21 visual programming tools applied to the Internet-of-Things. A survey made on VPLs applied to IoT found during the research process resulted in 8 more tools, making a total of 29.

The results show that there is a significant number of tools that allow its end-users to build IoT systems using visual programming, in several different scopes. The majority of these tools have a centralized architecture and operate in the Cloud tier. Despite the good amount of tools, most of them don't have their source code accessible nor have a license. The results from the expanded search are more positive in this aspect, with the majority of them being open-source, such as Node-RED [30], NETLab Toolkit [65] and others. However, this poses a problem, since there is a clear lack of open source tools.

In summary, the majority of tools found don't possess a license, employ a centralized architecture, operate in the Cloud tier and use a hybrid text and visual programming system. This propels the possibility of building, as future work, a visual programming tool applied to IoT that is (1) open-source, (2) has a decentralized architecture and (3) also operates in the Fog and/or Edge tiers.

3.2 Decentralized Architectures in Visual Programming Tools applied to the Internet-of-Things paradigm

Section 3.1 mentions some tools that aim to offer a decentralized solution to visual programming environments applied to Internet-of-Things systems [34] [47] [10].

3.2.1 DDF

The work made by in WoTFlow [10], DDF [34] and subsequent works [32] [33] consists of a system built on the Node-RED framework and focused on the use case of Smart Cities. Their goal is to make a tool more suitable for the development of fog-based applications that are dependent on the context of the edge devices they operate on.

In DDF [34], they started by extending Node-RED and implementing D-NR (Distributed Node-RED), which contains processes that can run across devices in local networks and servers in the Cloud. The application, called flow, is built in the visual programming environment, which is running in a development server. All the other devices running D-NR subscribe to an MQTT topic that contains the status of the flow. When a flow is deployed, all devices running D-NR are notified and subsequently analyze the given flow. Based on a set of constraints, they decide which nodes they may need to deploy locally and which sub-flow (parts of a flow) must be shared with other devices. Each device has a set of characteristics, from its computational resources such as bandwidth, available storage to its location. The developer can insert constraints into the flow, by specifying which device a sub-flow must be deployed in or the computational resources needed. However, these constraints and the characteristics of each device must be manually inserted in the system by developers or system operators.

Subsequent work to the previously mentioned tool focused on support for the Smart Cities domain. In the publication of 2018 [32], the problems addressed were the deployment of multiple instances of devices running the same sub-flow was a developed feature, as well as the support for more complex deployment constraints of the application flow. With this, the developer can specify requirements for each node on device identification, computing resources needed (CPU and memory) and physical location. In addition to these improvements, the coordination between nodes in the fog was tackled by introducing a coordinator node. This node is responsible for synchronizing the context of the device with the one given by the centralized coordinator. In Figure 3.2 it is possible to see the four possible states of a coordinator node: (1) NORMAL, where the node passes the data to its output, (2) DROP, in which the node does not pass the data to other node and instead drops it, (3) FETCH_FORWARD, where the node gets the input from an external instance of its supposed input and (4) RECEIVE_REDIRECT in which the node sends the data to an external instance of its output node.

In more recent work [33], support for CPSCN (Cyber-Physical Social Computing and Networking) was implemented, making it possible to facilitate the development of large scale CPSCN applications. To make this possible, the contextual data and application data were separated, so that the application data is only used for computation activities and the contextual data is used to coordinate the communication between those activities.

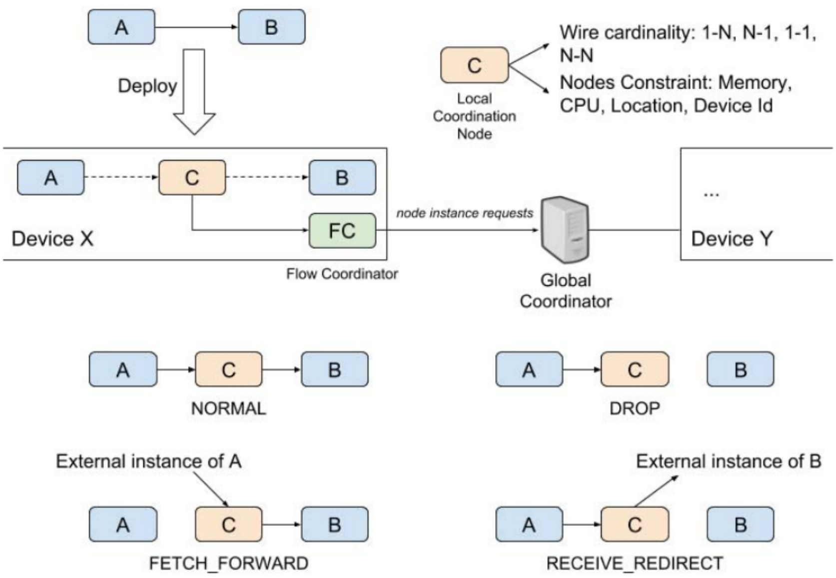

Figure 3.2: Coordination between nodes in D-NR [32]

3.2.2 *FogFlow & uFlow*

Another approach was made in the publication by Szydlo et al. [62], where they focused on the transformation and decomposition of data flow. Parts of the flow can be translated into the executable parts, such as Lua code. Their contribution includes the concepts of data flow transformation, a new run-time environment called *uFlow* that can be executed on a variety of resource-constrained embedded devices and the integration with the Node-RED platform.

The solution found consisted of the transformation of a given data flow, where the developer chooses the computing operations that will be run on the devices. The operations that will run directly on the devices are implemented in the form of embedded software, using the developed framework *uFlow*, which allows for parts of the flow to be run on heterogeneous devices. All this is integrated with Node-RED. The communication between the devices is made only through the cloud, with no support for peer-to-device communication. The results were promising, with a decrease in the number of measurements made by the sensors. However, there was room for improvement, with the automation of the decomposition and partitioning of the initial flow, the detection bottlenecks which will move computations accordingly from the cloud to the fog. Figure 3.3 represents a situation of partitioning and assignment of tasks. There are two IoT devices and a Node-RED instance running in the Cloud. The system's goal is to measure soil humidity and ambient light. If a button is pressed or fertilizer is needed, an e-mail is sent to the gardener. The partition of computation is made in the assumption that the closer a selected process it to the source of data, the higher the amount of data transmitted between computing operations. After parts of the flow are assigned to specific devices, they are altered in order to be executed by *uFlow* and Node-RED. It is possible to observe in Figure 3.3 the results of the transformation process, where the parts of the flow surrounded by color are executed in the device with the respective color.

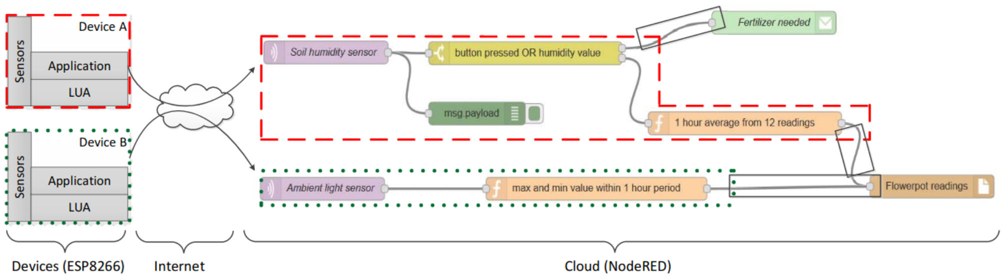

Figure 3.3: Partition and assignment of parts of the flow [62]

In 2019, they continued their work with the publication [55], where they built the model and engine *FogFlow*, which enables the design of applications able to be decomposed onto heterogeneous IoT environments according to a chosen decomposition schema. To achieve a level of decentralization and heterogeneity, they abstract out the application definition from its architecture and rely on graph representation to provide an unambiguous, well-defined model of computations. The application definition should be infrastructure-independent and contain only data processing logic, and its execution should be possible on different sets of devices with different capabilities. Several algorithms for flow decomposition were mentioned [45] [36], but none were specified in terms of results. Figure 3.4 represents the *FogFlow* architecture, which is composed by three modules: (1) the *FogFlow* API, which enables the creation of the application definition, (2) the Graph Module, responsible for processing and transforming the application definition into a data flow graph and finally the (3) Execution Model, which translates the graph and generates executables ready to be run on the assigned devices.

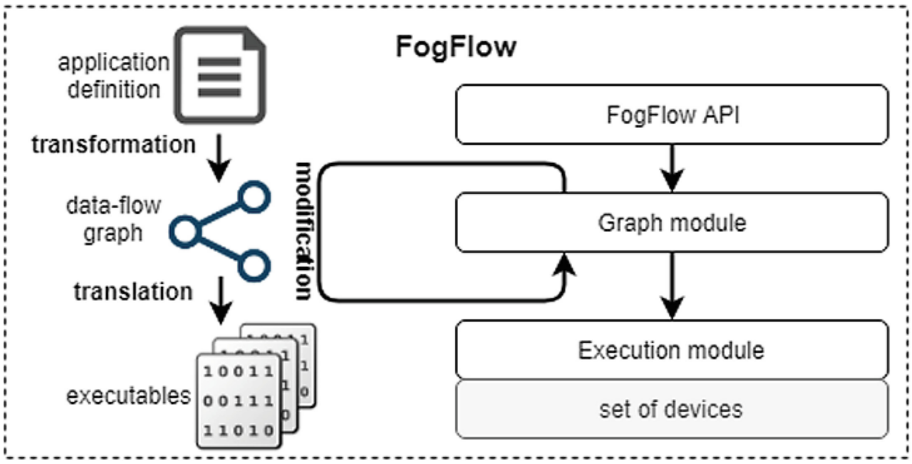

Figure 3.4: *FogFlow* architecture [55]

3.2.3 *FogFlow*

There is another tool with the same name *FogFlow* but created by Cheng et al. [60]. In the first publication related to this tool [20], the contributions made were the implementation of a standards-based programming model for Fog Computing and scalable context management. The first contribution consists in extending the dataflow programming model with hints to facilitate the development of fog applications. The scalable context management introduces a distributed approach, which allows overcoming the limits in a centralized context, achieving much better performance in terms of throughput, response time and scalability. The *FogFlow* framework focuses in a Smart City Platform use case, separated in three areas: (1) Service Management, normally hosted in the cloud, (2) Data Processing, present in cloud and edge devices and (3) Context Management, which is separated in an device discovery unit hosted in the cloud and IoT brokers scattered in edge and cloud.

In more recent work [19], *FogFlow* was improved to provide infrastructure providers with an environment that allows them to build decentralized IoT systems faster, with increased stability and scalability. The architecture can be seen in Figure 3.5, where the IoT system is represented by dynamic data flows that are orchestrated between sensors (Producers) and actuators (Consumers). The application is first designed using the *FogFlow* Task Designer, a hybrid text and visual programming environment, which results in an abstraction called Service Template. This abstraction contains specifics about the resources needed for each part of the system. Once the Service Template is submitted, the framework will determine how to instantiate it using the context data available. Each task is associated with an operator, which consists of a Docker image, and its assignment is based on how many resources are available on each edge node, the location of data sources and the prediction of workload. Edge nodes are autonomous since they can make their own decisions based on their local context, without relying on the central cloud.

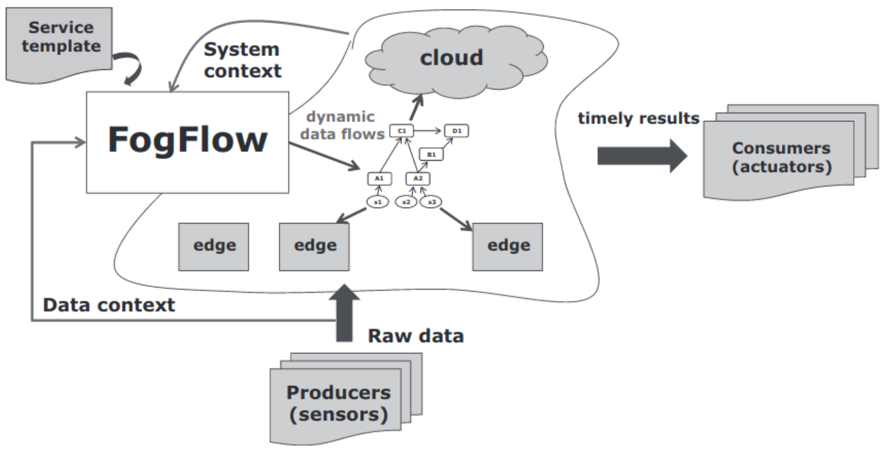

Figure 3.5: *FogFlow* high level model [19]

3.2.4 DDFlow

DDFlow [47], first mentioned in Section 3.1.2, presents another distributed approach by extending Node-RED with a system run-time that supports dynamic scaling and adaption of application deployments. The coordinator of the distributed system maintains the state and assigns tasks to available devices, minimizing end-to-end latency. Dataflow notions of *node* and *wire* are expanded, with a *node* in DDFlow representing an instantiation of a task that is deployed in a device, receiving inputs and generating outputs. *Nodes* can be constrained in their assignment by optional parameters, *Device*, and *Region*, inserted by the developer. A *wire* connects two or more nodes and can have three types: *Stream* (one-to-one), *Broadcast* (one-to-many) and *Unite* (many-to-one).

In a DDFlow system, each device has a set of capabilities and a list of services that correspond to an implementation of a *Node*, as can be seen in Figure 3.6. The devices communicate this information through their Device Manager or a proxy, if it is a constrained device. The Coordinator is a web server responsible for managing the DDFlow applications and is composed of three parts, which can be seen in Figure 3.6: (1) a visual programming environment were DDFlow application are built, (2) a Deployment Manager that communicates with the Device Managers of the devices and (3) a Placement Solver, responsible for decomposing and assigning tasks to the available devices. When an application is deployed, a network topology graph and a task graph are constructed based on the real-time information retrieved from the devices. The Coordinator proceeds with mapping tasks to devices by minimizing the task graph's end-to-end latency of the longest path. Dynamic adaptation is supported by monitoring the system and adapting to changes. If changes in the network are detected, such as the failure or disconnection of a device, adjustments in the assignment of tasks are made. In addition to this, the Coordinator can be replicated onto many devices to improve the reliability and fault-tolerance of the system.

In the evaluation made to DDFlow, the system is able to recover from network degradation or device overload, whereas in a centralized system this would cause its total failure.

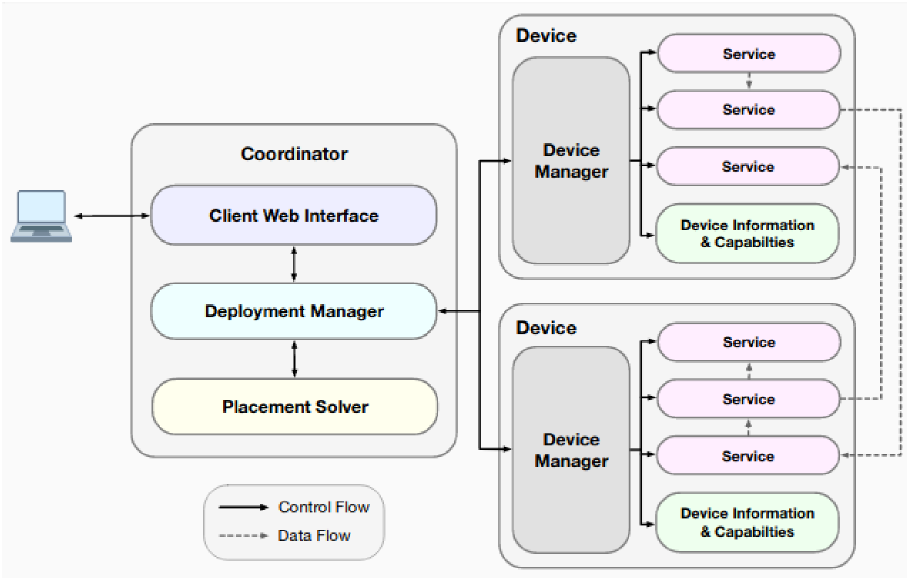

Figure 3.6: DDFlow architecture [47]

3.2.5 Analysis and Conclusions

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

Leverage devices A decentralized architecture takes advantage of the computational power of the devices in the network, assigning them tasks. However, some tools can have limitations on the type of devices, making constraints or only focusing on the devices of the Fog tier and not Edge.

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

Open-source The license of software or tool is essential in terms of its usability. Open-source allows access to the code, making it possible for its analysis, improvement, and reuse.

Computation decomposition To implement a decentralized architecture, it is important to decompose the computation of the system into independent and logical tasks that can be assigned to devices. This is made using algorithms, which can be specified or mentioned.

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

Tool	Leverage devices	Capabilities communication	Open-source	Computation decomposition	Run-time adaptation
DDF [34] [10] [32] [33]	Limited[1]	•	•	Limited[2]	•
FogFlow & *uFlow* [62] [55]	•	Limited[3]		Limited[2]	Limited[3]
FogFlow [60] [20] [19]	•	-	•	Limited[2]	•
DDFlow [47]	Limited[4]	•		Limited[2]	•

Table 3.6: Small circles (•) mean *yes*, hyphens (-) means *no information available*, empty means *no* and asterisk (*) means more than one.

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

From the analysis of the analyzed and the characteristics Table 3.6, it can be concluded that the state of the art in decentralized architectures in visual programming tools applied to IoT is incomplete. All the tools leverage the devices in the network but in different ways. DFF [34] assumes that all devices run Node-RED, which limits the type of devices that can be leveraged since it needs to have minimum resources to run it. *FogFlow* and *uFlow* [55] [62] is the only tool that specifies how it truly leverages constrained devices, with the transformation of sub-flows into Lua code, with DDFlow [47] assuming that all devices have a list of specific services they can provide, that should match the node assigned to them.

Regarding the method used to decompose and assign computations to the available devices, DDFlow describes the process with the use of a longest path algorithm focused on reducing end-to-end latency between devices. *FogFlow* and *uFlow* [55] [62] mention several algorithms that could be used, but don't specify which one was implemented. Both DDF [34] and *FogFlow* [20] [19] don't specify the algorithm used besides some constraints but are the only tools with their source code accessible and with an open-source license. All the tools claim to have support for run-time adaptation to changes in the system, such as device failures.

3.3 Summary

Section 3.1 presents a Systematic Literature Review of visual programming tools applied to the Internet-of-Things. Each tool derived from the research is summarized and characterized to understand the state of the art regarding this topic of interest. Section 3.2 describes visual programming tools for building IoT systems that employ a decentralized architecture, pointing out their advantages but also their shortcomings.

Chapter 4

Problem Statement

This chapter describes the problem, as can be seen in Section 4.1. In Section 4.2 it is presented the wanted features for the proposed solution and in Section 4.3 the scope of the project is defined. Section 4.4 contains the hypothesis this dissertation presents. The experimental methodology is outlined in Section 4.5. Finally, this chapter is summarized in Section 4.6 with an overview of the topics mentioned before.

4.1 Current Issues

Chapter 3 contains several solutions that provide a decentralized architecture in visual programming tools applied to the Internet-of-Things paradigm. However, some of these tools solve specific problems or make assumptions regarding the scale of the system and the constraints of the devices. We can define the problem in these issues:

1. **Leveraging devices in the network**: since most tools use a centralized architecture, including Node-RED, they do not leverage the devices in the network. Fog Computing introduces a decentralized solution, one that can be applied to Node-RED by distributing the computational tasks across the edge devices.

2. **Communication of computational capabilities**: some of the current tools require the developer to manually introduce the resources of each device in the network, which is not a scalable solution. Others have a specific list of services, manually inserted, that the devices can provide. Information about the computational capabilities of the devices in the network is vital for the successful distribution of computation across the devices.

3. **Detecting non-availability**: when a device fails or becomes unavailable, it is important for the system to automatically realize and adapt. The majority of current solutions do not possess this feature, which is vital if a system aims to dynamically adapt to changes in the environment.

4. **Code generation of sub-flows**: to truly leverage constrained devices, it is important to convert sub-flows or "tasks" into runnable code. Devices that support simple firmware capable of executing code can be used to execute blocks of code, despite their limited capabilities.

5. **Provide self-adaption of the system**: devices can fail, as well as the connection between them or even the network. It is important for the system to discover and identify these changes and adapt to them at run-time, in order to keep functioning.

4.2 Desiderata

Chapter 3 contains several solutions that provide a decentralized architecture in visual programming tools applied to the Internet-of-Things paradigm. However, some of these tools solve specific problems or make assumptions regarding the scale of the system and the constraints of the devices.

Desiderata is a Latin word that translates to "*things wanted*". In the context of this document, this section contains requirements wanted in a solution that aims to solve all the issues identified in Section 4.1. The requirements are the following:

D1: Communicate computational capabilities of devices connected so that this information can be sent to an orchestrator that will decompose the total computation workload based on this data.

D2: Decomposition and partition of computation so that the total computational requested can be distributed through all the devices in the network, using information about the computational capabilities and availability of the devices in the network.

D3: Convert computational tasks into runnable code so that each computational task can be executed in edge and fog devices, which contain limited resources.

D4: Provide self-adaptation of the system so that it can adapt to the non-availability of resources or even appearances of new devices.

4.3 Scope

The focus of this dissertation is the development of a prototype that allows for a decentralized orchestration of an IoT system. Despite security being a critical feature, it is considered a secondary goal, allowing the dissertation to focus on its primary goals. The scope of the project is a home automation system, where its devices have firmware capable of running MicroPython code and accepting custom code. They also need to be able to communicate their capabilities to an orchestrator.

4.4 Main Hypothesis

This dissertation is built around the following hypothesis:

> *"Given an IoT system with several heterogeneous devices connected, capable of running custom code, a decentralized architecture is more resilient, efficient and scalable than a centralized one."*

The attributes presented in the hypothesis will be measure against a system using the current development branch of Node-RED. These attributes consist of:

- **Resilience** means the system's capability to adapt to failures and changes. It will be measured by injecting failures and measuring the recovery patterns.
- **Efficiency** how fast the system can execute the logic of the system and communicate between nodes. The efficiency of a system is measured by the latency when reacting to system events.
- **Elasticity** specifies how a system can grow and shrink. This attribute will be tested by increasingly adding or removing devices in different scenarios and assessing the overall system's behavior.

Review this, not sure if this makes sense now

4.5 Experimental Methodology

In the interest of validating whether or not the solution implemented achieves the *desiderata* and solves the current issues, we will develop test scenarios that use simulations. Each one of the test scenarios will be verified against the attributes mentioned in Section 4.4 to prove the proposed hypothesis.

Review this, not sure if this makes sense now

4.6 Summary

Section 4.1 starts by presenting issues and lack of features not fully present in the current tools presented in the State of the Art Chapter 3. Section 4.2 presents a *desiderata* that aims to fix the issues presented in Section 4.1. The hypothesis of this dissertation is presented in Section 4.4, as well as an experimental methodology to prove it, in Section 4.5.

Chapter 5

Solution

This chapter describes how the problem presented in Chapter 4 was solved by stating the solution implemented and the reasons for the choices taken. **End with the sections' descriptions**

5.1 Overview

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

Node-RED was modified to meet the distributed computation communication demands by replacing the built-in communication by an MQTT-based one. Two main components, as *nodes*, were introduced to the Node-RED palette: (1) the *Registry node* which maintains a list of available devices and their capabilities and, (2) the *Orchestrator node* which partitions and assigns computation tasks to the available devices. Additionally, support was added to Node-RED to generate MicroPython-compatible code from custom *nodes*, *i.e.*, code-to-code transformation.

Additionally, a MicroPython-based firmware was developed that is able to receive and run arbitrary Python code scripts generated by Node-RED, and communicate with other devices or Node-RED itself using MQTT.

An high-level overview of the system can be seen in Fig. 5.1. Each module will be analyzed in detail in the following sections.

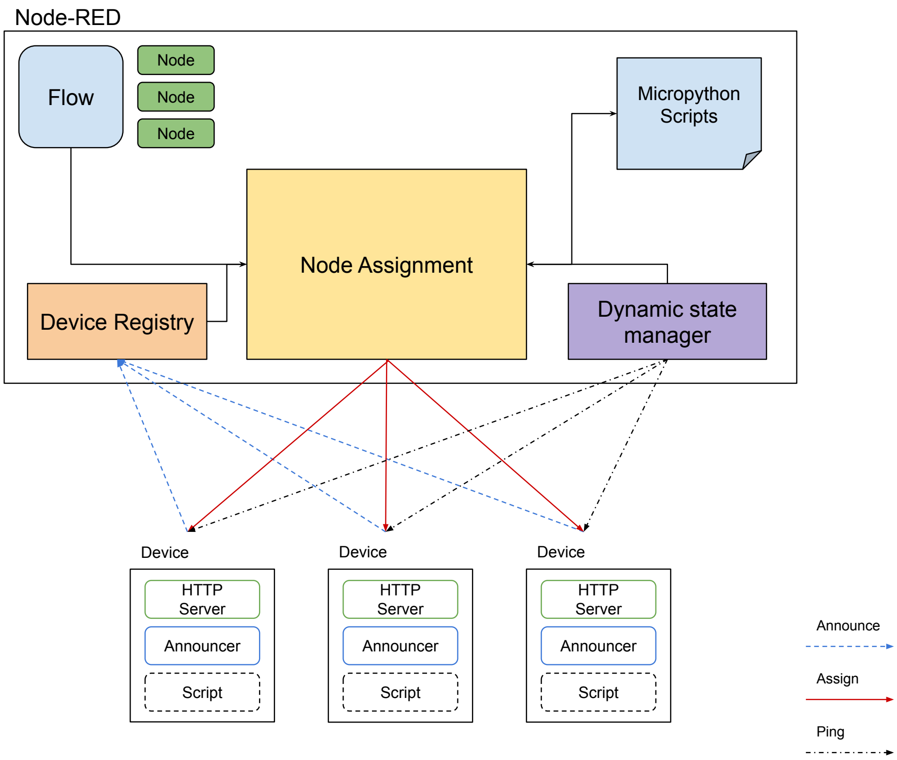

Figure 5.1: Solution's overview, presenting three devices as orchestration *targets*.

Explain and improve graphic

5.2 Implementation Details

The implemented solution consists of two co-dependent modules that are necessary for the functionally of the system. The first module consists of the changes implemented to Node-RED runtime to allow for its decentralization, detailed in Section 5.2.2. The second module consists of the solution found to take advantage of external devices with limited capabilities, explained in Section 5.2.1.

5.2.1 Devices Setup for Decentralization Support

For the purposes of this work we considered constrained devices that are capable of running custom code. Amongst the available hardware solutions, taking into consideration both costs and features, we picked two IoT development devices based on the Espressif Systems ESP32 and

Table 5.1: Comparison between the Espressif Systems ESP32 and ESP8266 systems on chip.

	ESP8266	**ESP32**
MCU	Single-core 32bit	Dual-Core 32bit
Frequency	80 MHz	160 MHz
SRAM	160kBytes	512kBytes
Flash	SPI, 16MBytes	SPI, 16MBytes
802.11 b/g/n WiFi	Yes	Yes
Bluetooth	No	Yes
Programming Lang.	Lua, Python and C	

ESP8266 systems on chip (SoC) [26, 25]. An overview on these devices hardware capabilities is given on Table 5.1.

The first challenge was to find a way to take advantage of the constrained devices, by making them run arbitrary scripts of code and communicate with other devices. Since both selected devices are able to run MicroPython firmware, Python was the selected programming language. Further, MicroPython already packs a small-footprint HTTP server and packages are available to implement asynchronous operations — *i.e.*, `uasyncio` — and MQTT pub-sub communication — *i.e.*, `MicroPython-mqtt`.

firmware component diagram?

As the devices must be able to receive arbitrary Python scripts (sent by Node-RED) and run them, the HTTP server was used to receive the Python payloads, which are then saved in the device SPI Flash and can be later executed (loaded to the SRAM). Further, the same HTTP server was used to implement an endpoint that returns the state of the device, as well as an announcing mechanism (*cf.* Section 5.2.2.4). These features built as integral part of the firmware that runs on the devices.

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

However, this solution is not without some limitations from which we highlight: (1) the developed firmware only supports MQTT QoS levels 0 and 1 due to `MicroPython-mqtt` limitations, and (2) ESP8266 severe memory limitations prohibit the use of the FAIL-SAFE mechanism if the given script is too big.

5.2.2 Decentralized Node-RED Computation

Node-RED is a centralized tool by design, which takes advantage of events to allow communication between *nodes* in a *flow*. To implement a decentralized architecture, some changes were made to the Node-RED runtime. These changes consisted mainly in (1) implementing a new way of communication between *nodes*, (2) add code-to-code generation features (*i.e.*, JavaScript to MicroPython), (3) . These are described in the following paragraphs.

5.2.2.1 Node-RED Node-to-Node Communication

Node-RED *nodes* communicate using events — node.js `EventEmitter`, where a *node* only communicates with *nodes* it is wired to. The communication is one-way only (forward message passing), with the *node* only sending data to the *nodes* it is connected to by output. These output wires are used to access the *nodes* the message must be sent to, and their `receive()` method is called. This method triggers the event `emit()` which will the caught by a specific method of each *node*, implementing its own logic.

This implementation is local and JavaScript specific, making it impossible to be used in a decentralized architecture where *nodes* will be executed outside of Node-RED. It was necessary to implement a way of communicating between *nodes* external to Node-RED that could be supported by low capacity devices. The solution found was MQTT, which fits as a good solution by its low-footprint and high-popularity amongst IoT solutions [61].

Thus, the Node-RED `Node` class was modified to use MQTT pub-sub communication instead of the in-place communication. Each *node* publish messages to an unique and addressable topic generated at the flow start, to which the next *node* in the *flow* subscribes to. This happens for every *node* with the exception of *producer nodes* that only publish messages and *consumers* that only subscribe to topics.

Since the modifications were made at the class (from which every *node* derives from) all the existent *nodes* and sub-*flows* are compatible with this modification without any changes in their code. However, as it will be described next, if we want a node to be orchestrable, the code of the *nodes* themselves needs to be changed.

Insert code block with example? Or an image explaining?

Insert here image that shows a flow with wires and the respective flow with examples of topics

5.2.2.2 Code Generation

To orchestrate Node-RED *nodes* computation amongst devices there is the need to generate MicroPython-compatible code from the existent JavaScript. Additionally, it is necessary to support multiple *nodes* in one script (*i.e.*, condensate), thus a generalized strategy was defined that could fit any type of *node*.

This was accomplished by adding specific code generation methods to each orchestrable *node*, which provide (1) their functionality, and (2) input/output capabilities. Since every *flow* communication is now MQTT-based, the only input and output a *node* can have is in its topics. A exception to this is in *nodes* that are producers, meaning that they generate input and don't receive it.

The code generation happens after the *Orchestrator node* defines an assignment between *nodes* and devices. This generation creates a device-specific Python scripts that follows the result of the task assignment procedure (script which can correspond to several *nodes*), adding wrapping code that ties up the script. This wrapping code is responsible for subscribing to all the input topics of all the nodes, stopping the script's processes and forwarding the MQTT messages to the respective node's code.

Insert Node-RED flow example and the respective MicroPython script

However, there are limitations to this solution. If two consecutive *nodes* are assigned to the same device, the communication between them is still dependent on MQTT. This is not an ideal solution, since one *node* could call the other through code, passing its output as arguments.

5.2.2.3 Custom Nodes

As previously mentioned, all the existent *nodes* are compatible with the modified Node-RED. Nonetheless, for a *node* to be orchestrable, it must be modified to comply with the code generation needs. As proof-of-concept, the following *nodes* were modified or created to be orchestrable:

`IF` ***node*** which receives an input and verifies if it complies with all the given rules, returning true or false;

`AND` ***node*** which receives a given number of inputs and verifies if all of them are true or false, returning the corresponding boolean;

`TEMP-HUM` ***node*** that read the temperature and humidity from a DHT sensor present in a specific pin;

`FAIL` ***node*** that raises a MemoryError exception;

`NOP` ***node*** that simply redirects the received message in its input to its output;

`MQTT IN` **and** **`MQTT OUT`** ***nodes*** that subscribe and publish MQTT topics, respectively.

Additionally, each of these *nodes* has two accessible properties: Predicates and Priorities. Similar to the Kubernetes logic of assigning containers to machines [14], the predicates dictate constraints that cannot be violated, and priorities are requests that are advisable and recommended but can be violated if impossible to comply.

These *nodes* provide enough functionality to wire simple *smart home* scenarios and validate our approach.

5.2.2.4 Device Registry

IoT systems are typically built on top of heterogeneous parts, with different capabilities and resources and their network can be highly-dynamic (devices appearing and disappearing according to factors such as battery levels, hardware/software failures and communication interference). In order to maintain a *list* of the available devices in the network along with their capabilities there is a need for a Device Registry [51].

When a device becomes available it sends information about itself to a MQTT topic. This information contains the device's IP address, their capabilities and their status — if the device has failed before. In its turn, Node-RED contains a *Registry node* that listens to the announcements MQTT topics and saves the devices information. If this *node* is connected to an *Orchestrator node*, each new device is communicated so that the orchestration can be updated.

When a device has an OUT-OF-MEMORY error, it triggers a FAIL-SAFE, where it reboots the HTTP server, stops running any script and restarts all communications. After this action, the device announces itself again but with a flag that indicates that it has failed. This way, the *Orchestrator node* knows that a device is active but not running any code, and that it is possible that it failed due to too much work. In that case, it can dynamically adapt and assign less *nodes* to the device, reducing the chances of causing another OUT-OF-MEMORY error.

5.2.2.5 Computation Decentralization

Node-RED must be capable of distribute computation amongst available resources (*i.e.*, devices), thus, given a set of tasks, it must assign them to available devices, ensuring that they will be performed.

The requirements to achieve this are two-fold: first, a *node* should act as coordinator, which when provided with an available devices list, along with their respective capabilities (*cf.* Section 5.2.2.4), should decide which device should execute specific computation *nodes* — *Orchestrator node* — and, second, the orchestrable *nodes* should provide both Predicates and Priorities that must be meet to assure their correct execution (*cf.* Section 5.2.2.3).

update pseudocode

The assigning algorithm uses the devices capabilities and each node's Predicates and Priorities to assign *nodes* to devices. With a greedy approach, the algorithm filters the devices that comply with each node's predicates and assigns the one with a higher value of a heuristic, *cf.* Algorithm 1. This heuristic takes into account the number of priorities the device can provide, as well as the number of already assigned *nodes* the device has. The goal is to assign each *node* to the best possible device, spreading the tasks through all the available devices. An example of a possible assignment can be seen in Figure 5.2, where the assignment matches the nodes' priorities with the devices' tags while spreading the nodes over the available devices.

After assigning all *nodes* to a specific device, a code script is generated for each device (*cf.* Section 5.2.2.2). Due to the constrained memory of the devices, the quantity of *nodes* assigned to a device may exceed their resources. In that case, the device will FAIL-SAFE and return an error to the assignment request. The orchestrator will receive this information and repeat the process, assigning less *nodes* to the devices that returned an OUT-OF-MEMORY error. If a device does not

Algorithm 1: Greedy algorithm for *node* assignment.

Input : deviceList
Output : bestDevice

init
  node: *node*
  bestIndex: 0
  bestDevice: *null*

onInput
  for *device* ***in*** *deviceList* **do**
    if ***not all*** *node.predicates* ***in*** *device.tags* **then**
      return
    $intersectionIndex \leftarrow \sum node.priorities \in$ device.tags / $\sum priorities \in$ node
    $nodesPerDevice \leftarrow \sum nodes \in$ device
    $nodesPrioritiesPerDeviceRatio \leftarrow \sum node.priorities \in$ device.tags / $\sum device \in$ tags
    $matchIndex \leftarrow intersectionIndex * 0.5 + (1/nodesPerDevice + 1) * 0.4 + nodesPrioritiesPerDeviceRatio * 0.1$
    if $matchIndex > bestIndex$ **then**
      $bestIndex \leftarrow matchIndex$ *// Device is best for node*
      $bestDevice \leftarrow device$
  return *bestDevice*

return any response, the orchestrator will assume that the device is unavailable and not assign any *node* to it.

Add an example of an assignment (JSON file) or/and a visual alternative with blocks and labels. With devices with capabilities and the assigned *nodes* predicates and priorities.

The *Orchestrator node* can be triggered — proceeding to a system (re)orchestration — by the following events: (1) start of the system, when there is already a defined flow in the configuration, the assignment start after a period of 3 seconds, to give time for the devices to be registered by the registry node; (2) deployment of the entire flow using the Node-RED editor or API; (3) appearance of a new device detected by the *Registry node*; (4) failure or recovery of a device, which, working as a complement to the *Registry node*, is detected using PING/ECHO pattern [54] which periodically *pings* the devices in the system to assert their operational status.

Maybe add an image of the process, with the possible steps that can trigger an announcement, or the flow of the process => Sequence diagram?

There are, however, some limitations in the assignment process, mostly due to the algorithm used. There are cases when the resulting assignment is not the best possible solution, and sometimes the orchestration can fail completely since it can not comply with the constraints imposed by nodes. As an example, given a scenario where the number of devices is small for the quantity of nodes, it is possible that the devices are kept at the limit of their resources, *i.e.*, memory. If there is a *node* which constraints can only be complied by one device, but that one device already has the maximum number of *nodes* it can handle, the assignment is not possible.

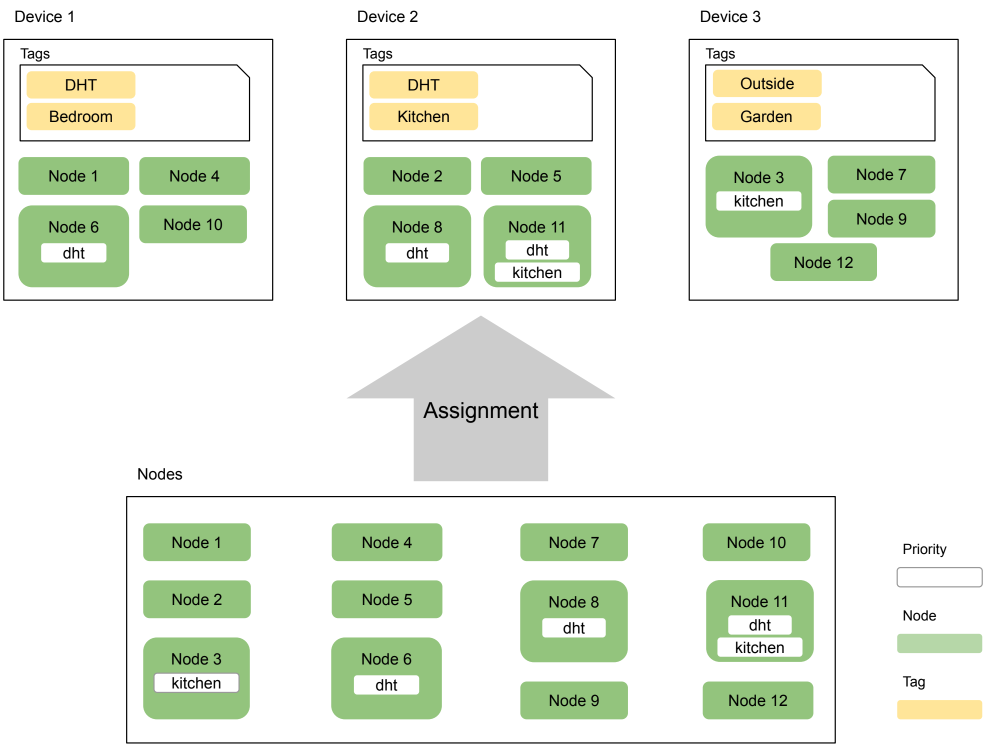

Figure 5.2: Node assignment example

In addition to this, the assignment algorithm does not take into account the connections between *nodes*. As mentioned before, sequential nodes in the same device communicate via MQTT topics instead of calling themselves through code. However, if that was not the case, it would be advantageous if sequential *nodes* were assigned to the same device. This would allow better performance, less communication load and less dependency on an external MQTT client.

Add an example of an assignment (JSON file) or/and a visual alternative with blocks and labels. With devices with capabilities and the assigned nodes predicates and priorities.

Maybe add an image of the process, with the possible steps that can trigger an announcement, or the flow of the process

5.2.2.6 Dynamic State Management

delete this section

After the assignment process, each device is doing their part to allow the system to work as expected. However, given the simplicity of the devices used, such as ESP8266s and ESP32s, they are prone to failures, with possible causes ranging form power loss to faulty hardware, among others.

Due to this limitations, the orchestrator periodically pings the devices in the system, registering any change in their state. If a state is noticed, the orchestrator will repeat the process of assignment taking into account the changes in the device's availability. This detects not only non-availability but also if a device becomes active again after a failure.

Don't know what to add more

5.2.2.7 Limitations

- Number of nodes that support MicroPython code generation is small
- Duplicate messages when redeploying the totality of the flow (maybe fixable later)
- Re-orchestration only supported when deploying the entire instance (all flows)
- Nodes do not stop working when the Node-RED instance is stopped
- Script generated does not take into account nodes that communicate directly, forcing all communications through MQTT instead of a node calling the method of another with the output as argument.
- Assignment algorithm does not take into account the assignment of sequential nodes in the same device.

Not sure where to put this. What is there more to add?

Chapter 6

Evaluation

This section evaluates how the solution developed proves the hypothesis proposed in Chapter 4.

Complete...

6.1 Scenarios and Experiments

The testing of the proposed solution diverged in two different scenarios. The first simulates physical devices with the use of Docker containers running the Unix port of MicroPython, allowing the construction of scalable scenarios with minimal costs. The second scenario uses physical devices, such as ESP8266 and ESP32, connected to the same Wi-Fi.

1. A room has 3 sensors that give temperature and humidity readings every minute. There's a virtual sensor that compares the results (of both temperature and humidity) and triggers depending on some configured thresholds. An AC uses those readings to decide (a) if it switches on/off, (b) its operating mode: cool, heat, and dehumidify. The Minimal Working System (MWS) consists in (a) one temperature sensor, (b) one humidity sensor, (c) one node capable of making the decision, and (d) working communication channels amongst them.

2. 20 devices, where each device redirects its input to its output. improve

The first scenario aims to test the features of the developed solution with a moderately simple Node-RED flow, taking advantage of the nodes developed for MicroPython code generation support. The second scenario allows the comparison of the developed solution to the already existing solutions.

Refactor this enumeration, with some explaining of some terms

Scenario 1:

1. **Sanity check.** All tasks are simple readings and forwarding, no compensation or other fault-tolerance strategy. Each sensor does its own thing. Orchestration is centralized. We expect all roundtrips to take less than the smallest part that can be resolved (measurement capability, which we estimate to be <1s). This will be executed using both Docker containers and physical devices.

2. **Re-orchestration.**

   - **Experiment A.** MWS is achieved via multiple possible configurations by selective (provoked) device failure (fail-stop);
   - **Experiment B.** Inconsistent device behaviour, e.g. appear and disappear in shorter intervals lower that the time needed for orchestrating convergence (OCT), that leads to activity impacting the MWS;
   - **Experiment C.** With 4 devices, each one with different processing capabilities. During orchestration, some devices will develop an out-of-memory error because they can't process all the processing tasks assigned to them, specifically the size of the script given. The orchestrator decides to send less tasks to these devices. The system will converge in a working solution. *This scenario will be implemented with a modified device script. When devices receive a script, it will generate a memory error if the length of the script passes a certain threshold. This simulates the memory constraints of devices when receiving a file to big.*
   - **Experiment D.** With 4 devices, some of them have a memory leak with an unknown cause. After random time Random(t0,t1), these problematic devices stop working with an out-of-memory error. The orchestrator thinks that the devices can't handle the quantity of processing tasks assigned to them, so in the re-orchestration it will assign fewer tasks. Since these devices will always break, the orchestrator will eventually not consider these devices in the assignment of nodes. *This scenario will be implemented with a modified device script that will trigger an out-of-memory error after a random period after executing the given tasks.*
   - **Experiment E.** With 4 devices, there is a device that is sensitive to a particular node, which causes the device to give out an out-of-memory error. The orchestrator will potentially assign this node to the specific device. When the device gives out the out-of-memory error, the orchestrator will eventually converge in a solution where the node is not assigned to the particular device, and the system will converge. *These out-of-memory errors will be simulated with the use of a failure node that forces an* `MemoryException` *in the device.*
   - **Experiment F.** With 50 devices, each second the device has a probability to fail. This failure can go from 0 to 10 seconds, randomly chosen. The orchestrator must deal with

the random failure of the devices and re-orchestrate the system. This experiment is considered a stress test, causing repeated failures and forcing constant re-orchestration.

Verifies that:

(a) **Restrictions (predicates) are enforced.** Check that possible configurations lead to solutions that enforce defined predicates;

  i. Temperature and humidity might coexist in the same, or in dedicated, devices;

(b) **Priorities are honored.** Check that all specified priorities were taken into account, and only violated if necessary;

  i. Priority is given to edge devices, but fog and cloud can be used;

  ii. Priority is given to the maximum level of decentralization, but some centralization can be used.

Scenario 2: With the use of 20 physical devices, both ESP8266 and ESP32, implement a line topology, where a message is sent to a starting device, that will propagate it to its output. All the devices implement this propagation logic, which results in the initial message reaching the end of the line. The propagation time is measured, starting when the message is sent and ending when the message reaches the last node. This experience is implemented in several environments:

1. **Node-RED original**. Runs the experiment in the original Node-RED, using the default option (events) as the communication channel between nodes.

2. **Node-RED + MQTT**. Run the experiment in Node-RED, using MQTT as the communication channel between nodes.

3. **Node-RED modified + Dockers (same host)** Runs the experiment in the modified version of Node-RED, with each node assigned to a different virtual device, running in a Docker container. The Docker containers are running the firmware created in a Unix MicroPython port. The virtual devices are in the same host machine running the MQTT server instance.

4. **Node-RED modified + Dockers (different host)** Runs the experiment in the modified version of Node-RED, with each node assigned to a different virtual device, running in a Docker container. The Docker containers are running the firmware created in a Unix MicroPython port. The MQTT server instance is in a different host machine from the one running the virtual devices. All machines are connected through Wi-Fi.

5. **Physical + MQTT** Runs the experiment in physical devices without the developed firmware. Each device runs a simple MicroPython script that executes the wanted behaviour, communicating by MQTT. No Node-RED is used.

6. **Node-RED modified + MQTT + Physical + Firmware** Runs the experiment with the developed solution, with each node assigned to a different physical device running the developed MicroPython firmware. All communicating is made using MQTT.

6.2 Discussion

Complete...

6.2.1 Scenario 1

As mentioned previously, the first scenario consists of a system that controls an A/C. This system takes into account readings of 3 temperature and humidity sensors to define if the room's temperature is too hot, cold of humid and sends commands to the A/C with the respective actions. The Node-RED implementation of this system can be seen in Figure 6.1.

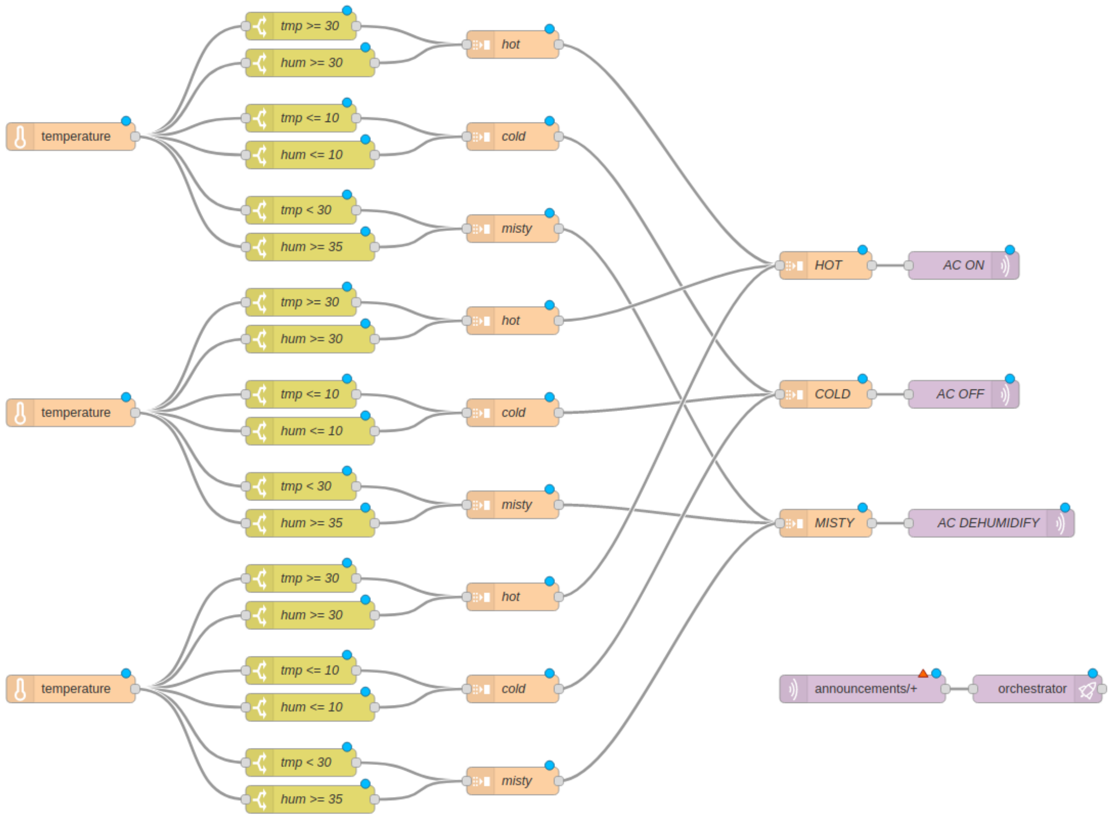

Figure 6.1: Node-RED implementation of scenario 1

The sanity check experiments will prove that the devices can satisfy the nodes, meaning that the system works as its meant to once the assignment is completed. With this premise, this will not verified in the other experiments.

6.2.1.1 Sanity Check

The first experiment made to the developed solution tested the overall functionality of the tool and its efficiency. Using 4 Docker containers with the MicroPython Unix port with the developed firmware, the scenario presented in Figure 6.1 was partitioned and assigned evenly to them. The assignment can be observed in Figure 6.2, where it was allocated 9 *nodes* for each device.

Figure 6.2: Sanity check experience measurements

The usage of RAM was significant, varying from 60Kb to 200Kb, as it can be observed in Figure 6.2. The flash size only decreases around 150000 bytes, when the device receives a script to execute. This quantity matches the overall payload received by the devices.

As mentioned before, once the orchestrator defines the *nodes* assignment, a script is built and send to the devices. The confirmation of this delivery is necessary for the system to conclude the assignment phase and start the constant verification of the state of the system. The time it takes to deliver the script was timed and it can be consulted in Figure 6.2. The usage of virtual devices running in the same host as the Node-RED instance allows for smaller times, which are measured in milliseconds.

Once the devices execute the script given to them, each *node* allocated to a device start to communicate with each other, publishing and consuming MQTT messages. To verify that the system works, the messages of all communicating topics were captured. This messages prove that the system works, since all *nodes* are receiving their inputs and producing outputs. The number of communications can be consulted in Figure 6.2. As it can be observed, the number of communications in Device 1 is bigger than any other. This is due to the fact that in this device two temperature-humidity *nodes* were allocated, which publish 3 messages, a number communication events bigger than any other node. The Device 2 contains the other temperature-humidity node.

It can be concluded from this sanity check experiment that the solution developed works, not only by spreading the computation throughout the available devices, but it also results in a functional system.

Include *node* assignment of nodes? JSON? Or modify the node-red flow and say the device each *node* was assigned to?

Is it necessary to add the table/graph with the times it took to send the scripts to the devices?

6.2.1.2 Sanity Check with Physical Devices

The sanity check experiment was repeated using physical devices, more specifically 4 ESP32. Similar to the virtual devices, the assignment of *nodes* to devices spread the number of *nodes* equally, with each device having 9 *nodes* to their responsibility. This assignment result can be seen in Figure 6.3.

The usage of RAM in physical devices is smaller than the one used by virtual devices. This can be explained with the possible optimization differences in the MicroPython ports, as well as the changes implemented in libraries to support the Unix port.

The flash size of the physical devices is smaller than the physical ones, as expected. It can be observed in Figure 6.3 that the device with the biggest payload, Device 1, ends up having the smaller flash size, which is logical. The overall size of the payloads are very similar to the ones in the previous experiment.

The script delivery time for physical devices is bigger than the virtual devices. Since they are not running in the same machine, the WiFi stack as well as the devices' hardware hinder the speed of communication. The uptime is similar to the previous experiment, since no device failed.

6.2.1.3 Experiment A

This experiment aims to test the system's ability to re-orchestrate when a device fails or becomes available. During the occurrence of the experiment, devices were turned off one by one until only one was left running. The expected behaviour is that the system detects that a device has become unavailable and re-orchestrates, assigning *nodes* to the available devices. In the end, only one device is running and all the *nodes* are assigned to it.

It can be observed in Figure 6.4 that the uptime of the devices stops increasing one by one, identifying the moment the device fails. Once a failure happens, the system (re)orchestrates, assigning the nodes of the device to the other available devices, increasing their number. This can

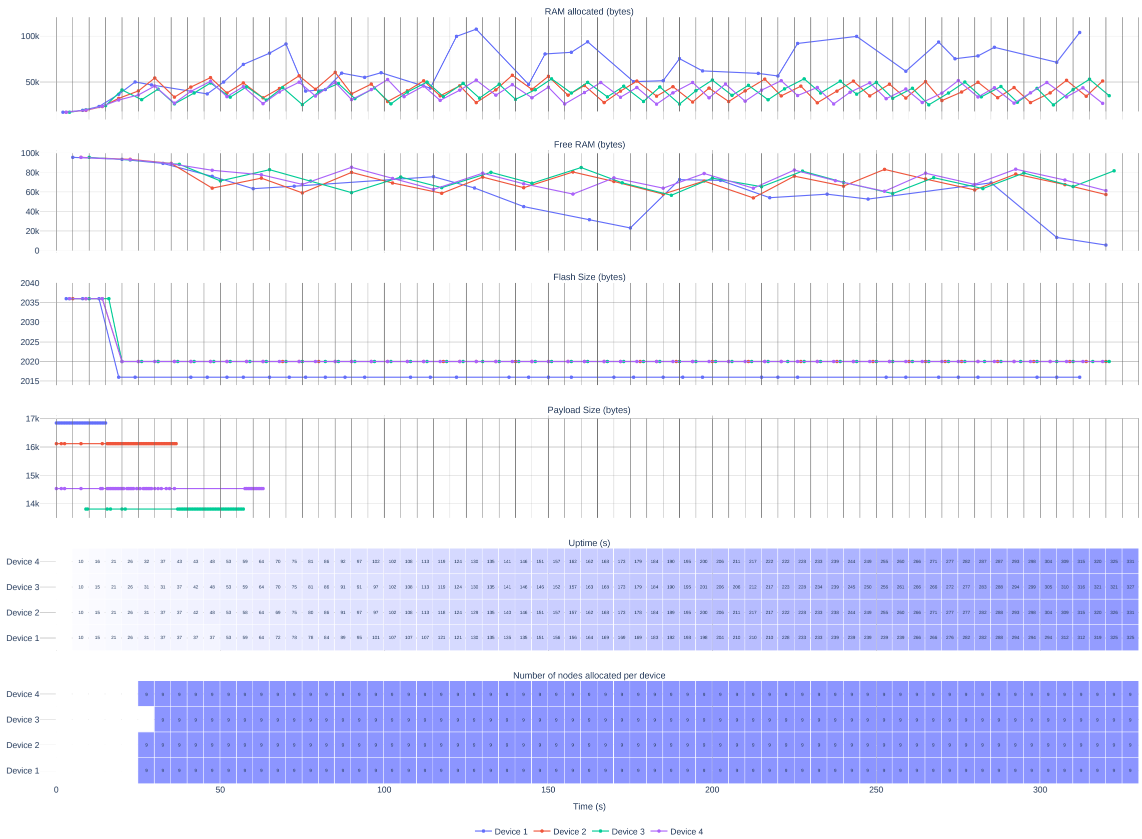

Figure 6.3: Sanity check with physical devices experience measurements

also be observed in Figure 6.4. This increase in the number of nodes assigned to the available devices can also be observed in the payload size. When all devices fail except one, the one remaining is the only one that receives the payload, which is higher than any other previously received.

The information regarding the number of nodes is not updated to 0 once the device fails, since it is no longer active to send the updated metric.

This experiment proves that the system identifies the failure of devices and takes actions to rectify it. This includes repeating the assignment process, taking into account the available devices.

6.2.1.4 Experiment A with physical devices

This experiment is very similar to the previous one, the only difference is in the use of physical devices instead of virtual ones. The payloads and number of nodes assigned through the experiment is very similar to the ones achieved with virtual devices. The only thing to note is the fact that Device 2, the last remaining active device, fails when receiving the bigger payload. This payload contains the code for all the nodes of the system, since no other device is available.

This exposes one of the limitations of using physical devices, the memory. The device does not have enough memory to handle that big of a script, so it FAIL-SAFEs, informing the system

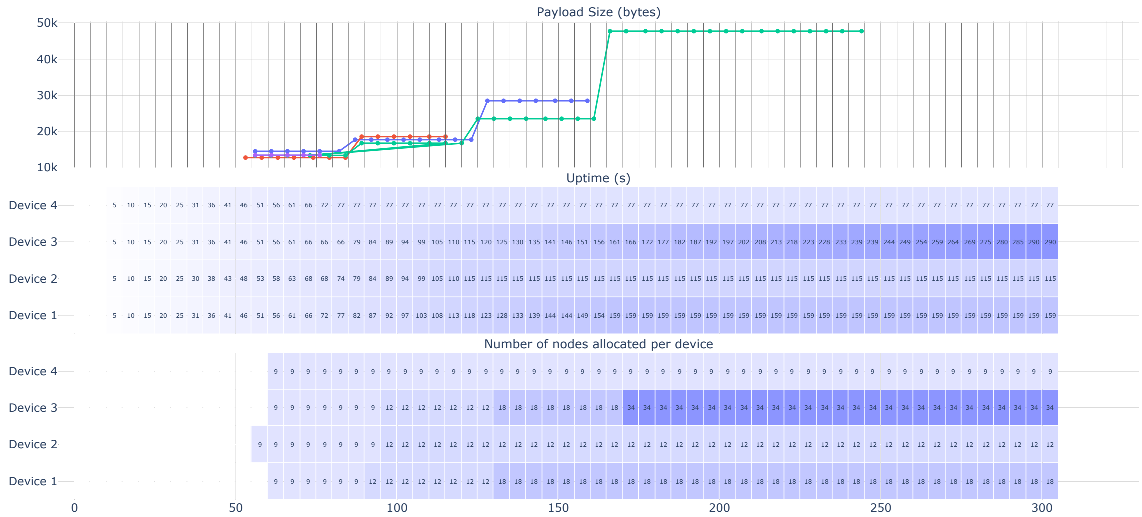

Figure 6.4: Experiment A measurements

that there was an *Out-of-Memory* error. In its turn, the system assigns less nodes to the device.

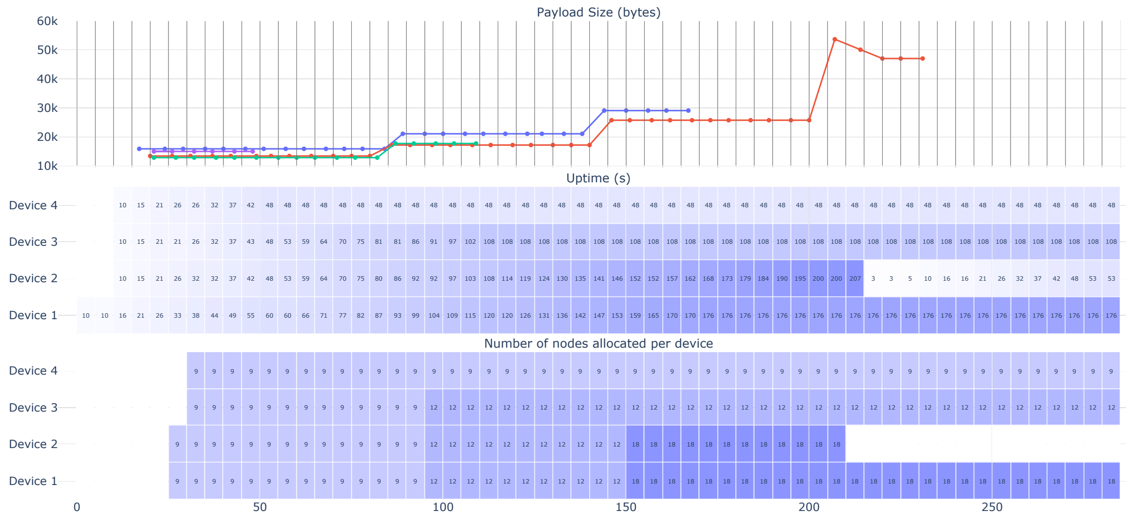

Figure 6.5: Experiment A with physical devices measurements

6.2.1.5 Experiment B

Similar to the two previous experiments, this one takes it a step further, not only testing the system's ability to recover when devices fail, but also when they recover. In Figure 6.6, Device 3 and 4 fail early on and the system recovers, spreading the nodes assigned to them to other devices. Device 4 recovers around the 100s, fails again and then recovers. This change was not caught by the system, since it was very fast, and the system only re(orchestrates) the second time Device 4 recovers. During the course of the experiment, Device 3 and 4 continue to fail and recover, and the system always adapts.

This ability for the system to re(orchestrate) when a device recovers can be taxing to the functionality of the system. If a device is constantly failing and recovering, the system will always adapts itself, halting its functionality to orchestrate itself.

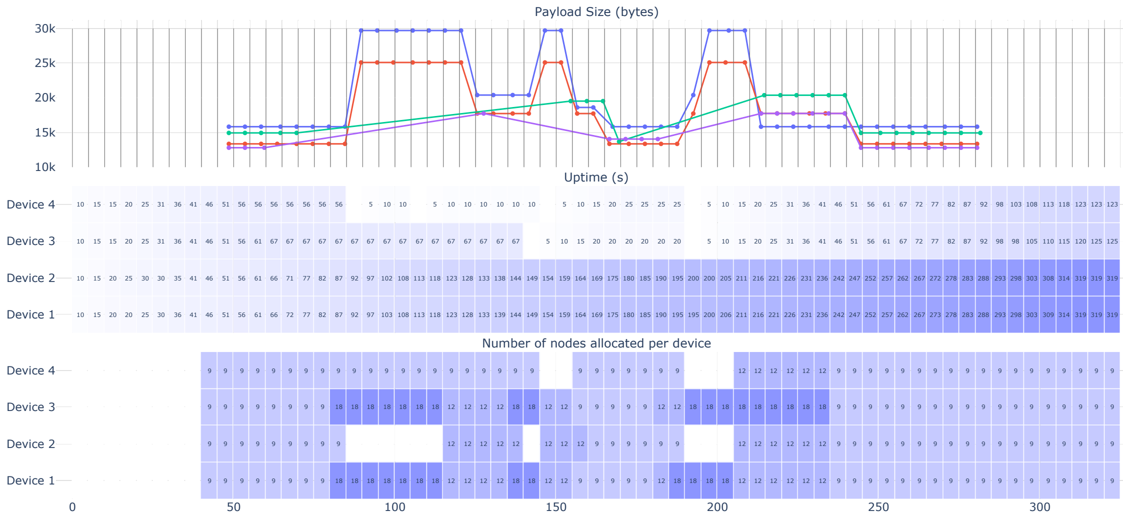

Figure 6.6: Experiment B measurements

6.2.1.6 Experiment C

This experiment aims to test the system's ability to recover and adapt to the devices' memory constraints. More specifically, memory errors that can arise when writing the received script into the device SPI.

In the Figure 6.7, both the Device 2 and 4 are memory constrained. When the first assignment is made, around the 50 seconds, both these devices FAIL-SAFE due to *Out-of-Memory* errors. The number of nodes presented in Figure 6.7 are the ones assigned after devices 2 and 4 communicate to the orchestrator their limitations.

To assess if the system saves information about the limitations of the devices, one of them was turned of and later turned on. This event is identified in the Figure 6.7. As it can be observed, Device 2 uptime stops increasing around the time of the event and its nodes are distributed by the other devices, with the exception of Device 4, which is memory constrained. After the recovery of Device 2, the system (re)orchestrates and the same number of nodes is assigned to the devices. However, Device 4 failed when Device 2 recovered. This implies that the system repeated the process assignment process again, ignoring the previously known information about memory constraints.

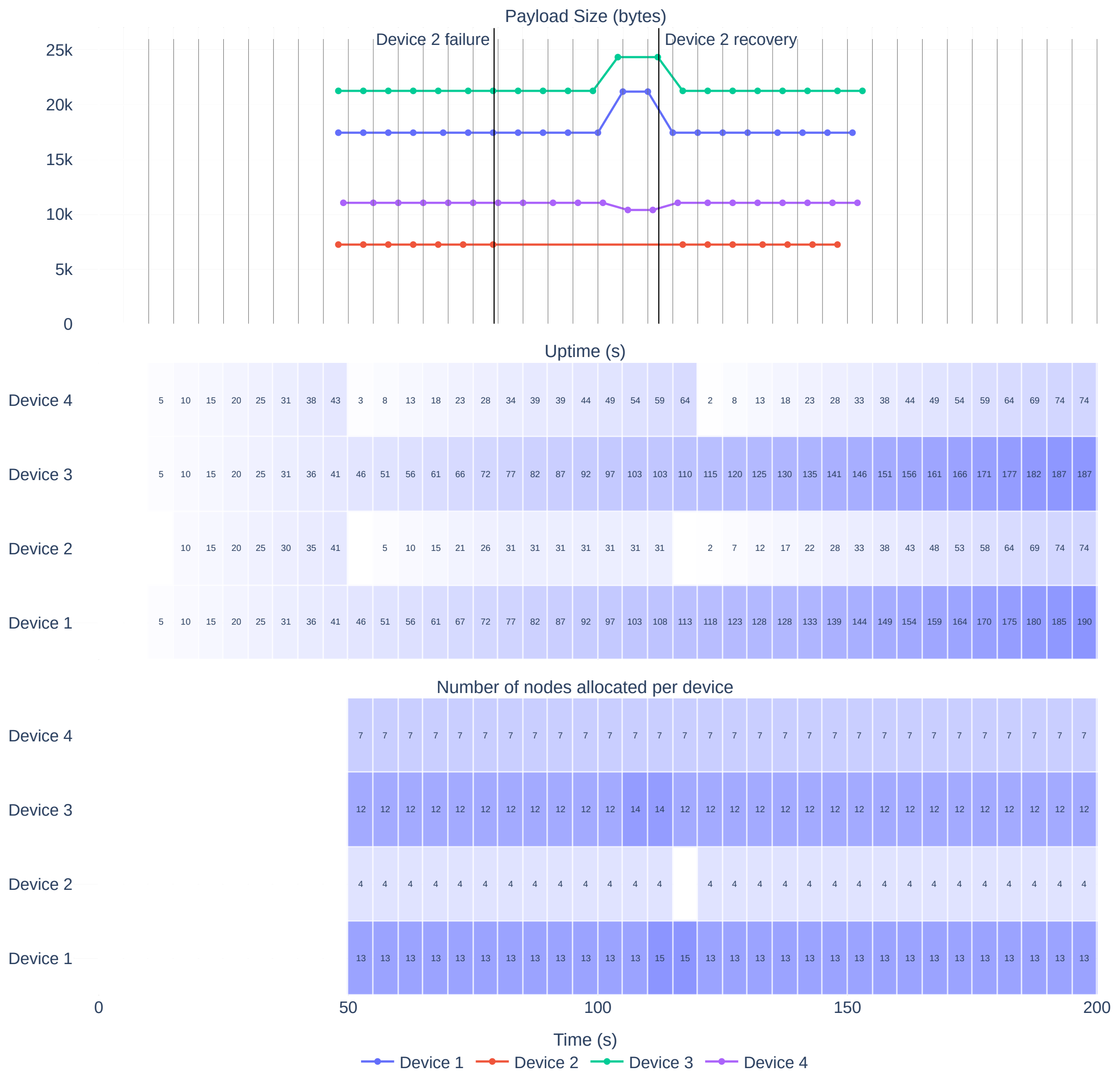

Figure 6.7: Experiment C measurements

6.2.1.7 Experiment D

In this experiment the goal is to check into the system's ability to handle a damaged device that has a memory leak. This device, which in this situation is Device 2, will always generate an *Out-of-Memory* error after a random period of time. The system should be able to exclude this device during the course of the assignment process.

As it can be observed in Figure 6.8, Device 2 is consistently failing after the first assignment of nodes, around 75 seconds. The number of nodes assigned decreases, until no node is assigned and the device is excluded from consideration. This is an iterative process, in which the system will decrease the number of nodes it assigns to a device if the device communicates an *Out-of-Memory* to the orchestrator. Eventually, if the device does not handle any node, the minimum number of nodes the device can handle is 0, excluding the device from the assignment process.

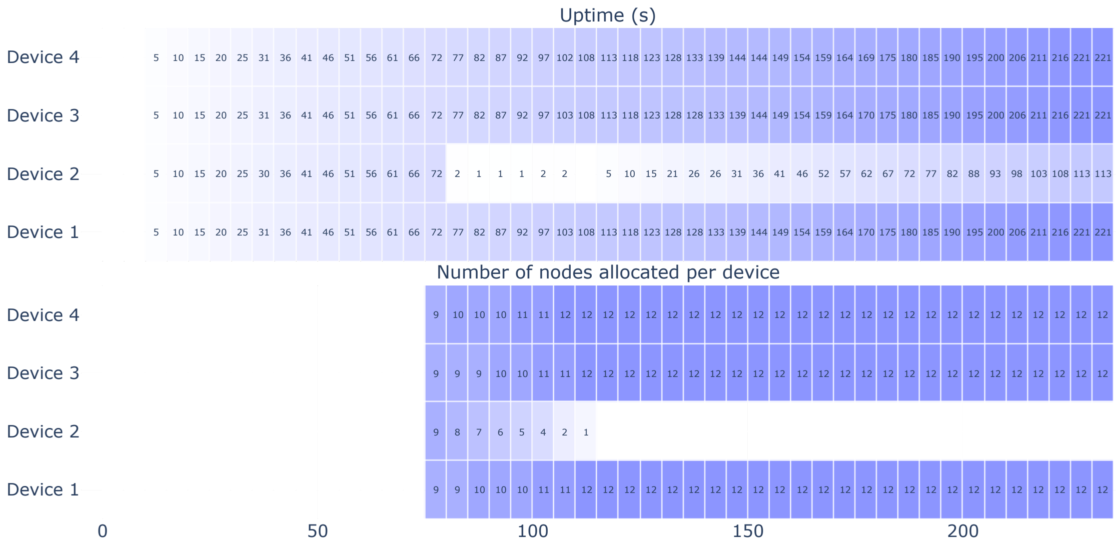

Figure 6.8: Experiment D measurements

6.2.1.8 Experiment E

TODO

This experiment purpose is inject a node that causes *Out-of-Memory* errors in specific devices. With this, the system should (re)orchestrate and converge in a solution where the specific nodes are assigned to devices not affected by them. In their turn, the devices affected by these nodes should have less nodes assigned to them. The system and devices do not know that a specific node is creating the *Out-of-Memory* errors and interpret the error as a device problem.

Since the first assignment can already be correct by default, meaning that these faulty nodes are assigned to devices not affected by them, some changes were made to force the system to (re)orchestrate. The devices were all turned off and on in in different order, this repeated 3 times. These events can be observed in Figure 6.9 around the 125, 200 and 275 second timestamps.

Acho que este gráfico não dá bem para perceber o que se está a passar. Já agora, esta experiência em si é uma repetição das duas outras. Isto é, testa o que as duas anteriores já testaram.

Maybe remove this experiment. Graph can't convey what happens

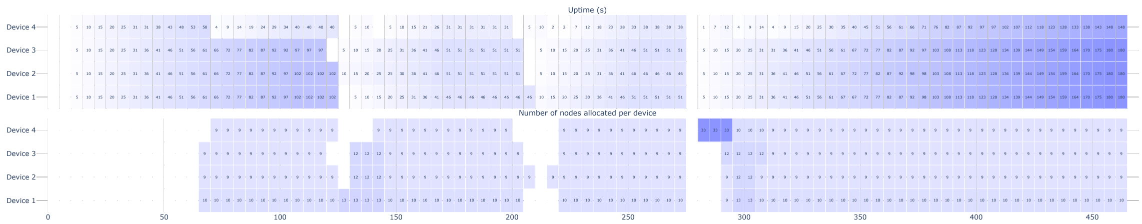

Figure 6.9: Experiment E measurements

6.2.1.9 Experiment F

TODO

Possibly will have to repeat this experiment, since I can’t find a pattern to note...its all a blob of events

Figure 6.10: Nodes assignment distribution

6.2.2 Scenario 2

As mentioned previously, several experiences were made to compare the developed solution to existing ones. To this end goal, a simple experiment of passing a message through several devices was implemented and the time the message takes to pass through all the devices was measured. The implementation of the scenario in the Node-RED tool is shown in Figure 6.12. The *nothing* nodes execution consists of only redirecting their input to their output. The message consisting of the current timestamp is inserted into the system by the *inject* node with user input, and the same message is showcased by the *debug* node, the green one.

This same setup was replicated in several environments, as mentioned before. Each experiment was replicated 10 times, resulting in the data seen in the Appendix A tables. These tables were analysed to construct the Table 6.1 and its visually representation in Figure 6.13.

The Figure 6.13 demonstrates that the developed solution is considerably less efficient in communicating message between nodes. However, given the other experiences, it is possible to conclude that this lack of efficiency is caused not by the firmware created but because of the stack of communication the message as to go through, as well as the nature of MicroPython.

When the decentralization is applied inside Node-RED, without running any MicroPython, it is possible to see that the introduction of a Mosquitto broker running in the same host causes

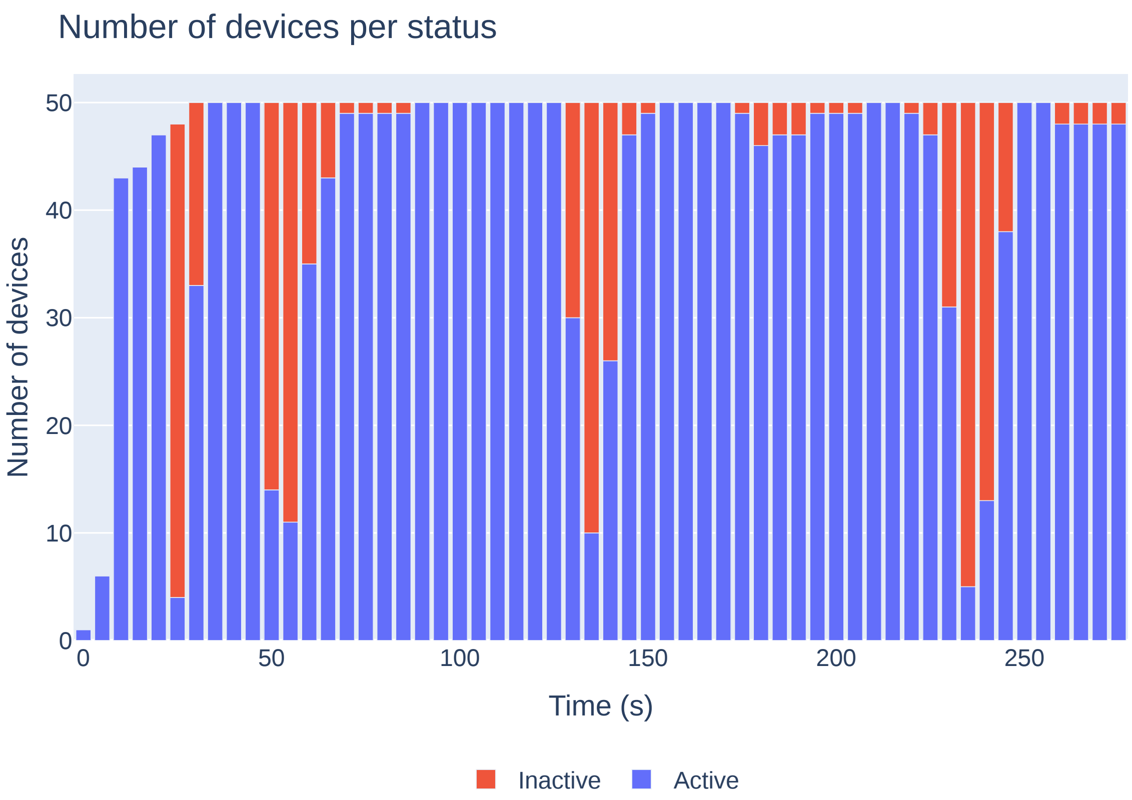

Figure 6.11: Number of devices active and inactive

some latency. The introduction of Dockers running the firmware in the same host as the Node-RED instance and Mosquitto broker causes more latency, making it possible to conclude that MicroPython of the developed firmware also delay de communication. By repeating the same experience as before but with the Mosquitto broker in another machine, it is noticeable that the times are more spread out and the overall latency of the system is bigger. Given the stacks of Wi-Fi that the message as to go through, this result is logical.

Lastly, the experiment was repeated in physical devices, first by running a simple code in he MicroPython flashed devices and insertion of the message using the Mosquitto client, and second by using the whole developed system, with the modified Node-RED and firmware in the devices. The results allows us to conclude that the devices produce the worst time, but firmware developed introduces little latency, visible by the comparison of both their results.

It is possible to conclude that the developed solution's node communication is slower than the original Node-RED mostly due to the nature of Wi-Fi communications and the MicroPython port used.

Add other tables with the timestamps in the annex

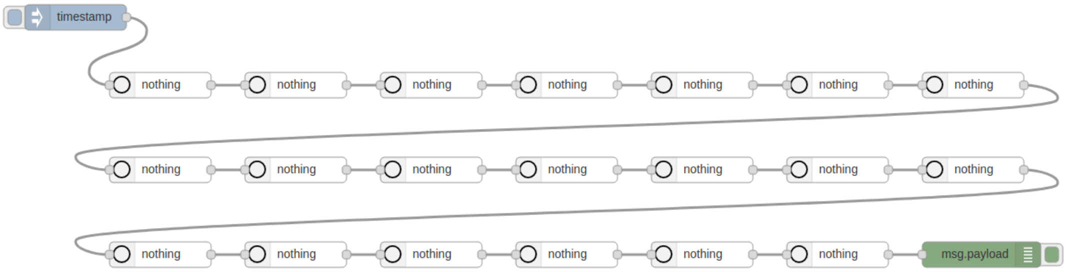

Figure 6.12: Node-RED implementation of scenario 2

6.2.3 Overview

Overview of the evaluation of the system, with conclusions of the evaluation of the system as a whole

6.3 Conclusions

Label	Min	Q2	Q3	Max
Node-RED original	3	10	13.25	15
Node-RED + MQTT	134	430.5	711.25	883
Node-RED modified + Dockers (same host)	1217	1318	1573.75	1665
Node-RED modified + Dockers (different host)	1445	2536	2708	3059
Physical + MQTT	3616	4142	4372	4452
Node-RED modified + MQTT + Physical + Firmware	4168	4569	5087.75	5940

Table 6.1: Scenario 2 results

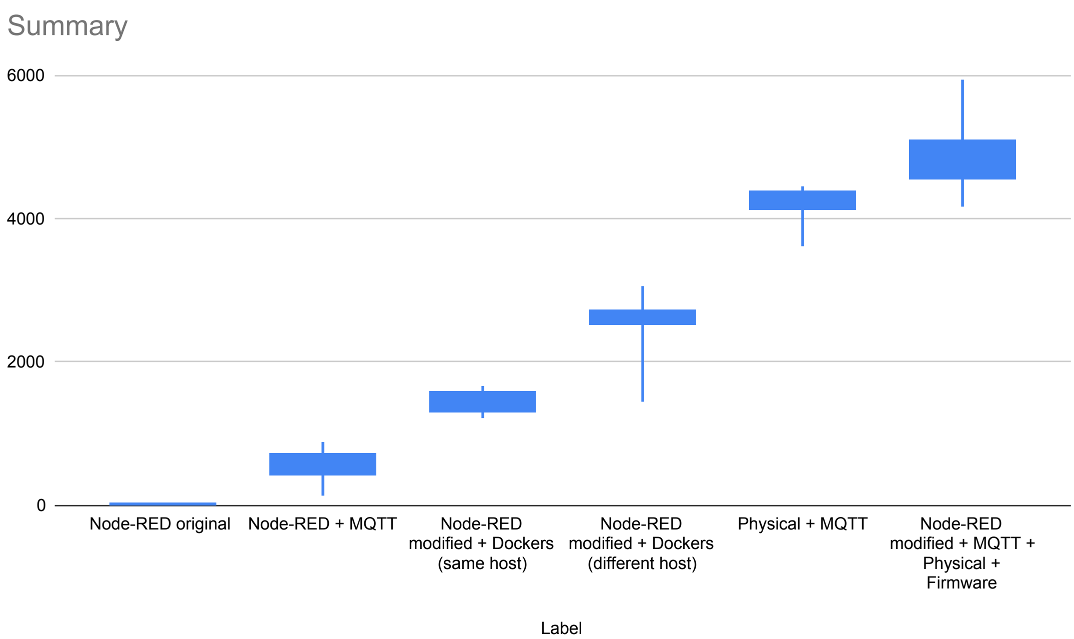

Figure 6.13: Scenario 2 results

Chapter 7

Conclusions

As the number of devices connected to the internet increases, it is important to leverage their capabilities and modify the way systems are built to take advantage of these resources. It is also important to allow end-users with no programming experience to build Internet-of-Things (IoT) systems, with the use of visual programming tools. These tools make the building process easier, reducing the knowledge of programming concepts needed.

Despite the existence of a considering number of visual programming tools applied to IoT, the majority of these tools are centralized. This centralization hinders the resiliency of the system, as the unit responsible for the execution of most or all of the computation is a single point of failure. If this unit or the network fails, the system stops being functional. Another issue of this type of architecture is the lack of usage of the computational capabilities of the rest of the devices in the system.

During the analysis of the state of the art, some issues and missing features were identified, which this dissertation aims to correct. The tools found that possess a decentralized architecture have limiting characteristics such as assumptions about what is a constrained device regarding computational capabilities, lack of open source licenses and simplification of the approach taken to the decomposition and assignment of tasks.

This dissertation aims to solve these issues by expanding an already popular visual programming tool, Node-RED, with a decentralized approach that focuses on leveraging all the devices, even ones that only support the execution of simple blocks of code. The expected result is a decentralized system that can self-adapt to run-time conditions and decomposes the given computations into independent tasks, which are assigned to devices. The assignment's goal is to increase the efficiency of the system, reducing latency and distributing CPU usage.

Complete the conclusion with the conclusions from the evaluation

7.1 Difficulties

Translate this

- Problemas de memória e espaço dos ESPs para execução de scripts de MicroPython com 3+ nós (ESPs lançam erros de alocamento de memória se o script enviado é maior que um certo nº de bytes ou ao fazer redeploy de scripts de médio tamanho). - Alternativas: failsafe (implementado e funcional), pyc (não existe para MicroPython mas existe o .mpy. No entanto, precisa de ser executado para gerar um .mpy a partir de um .pt, o que não se aplica à solução atual), ota (não existem boas soluções para MicroPython)
- Necessidade de implementar novos nós para situações de teste
- Node-red não suporta comunicação de mqtt entre nós de raíz. Node-red teve de ser adaptado para que a comunicação entre nós fosse feita desta maneira.
- Node-red implementation of subflows made it difficult to implement the MQTT communication to them
- Modificar scritps e suporte para o port para Unix do MicroPython - muitas diferenças, limitações e criação de muitos bugs.
- Limitações das bibliotecas usadas de MicroPython, especificamente nas bibliotecas de mqtt e operações assíncronas.

7.2 Challenges

Translate this

- Como reorquestrar um sistema sem forçar o sistema a parar, isto é, garantindo availability (future work)

7.3 Future Work

- Se calhar falhar um simples ping não devia levar a uma reorquestracao imediata, mas sim a mais ou ou outro ping para ter a certeza. Esses "retries" e o respectivo tempo podiam (1) ser configuráveis, ou até (2) serem baseados em cenas de confiança para as quais já há algoritmos bem definidos para isso em sistemas distribuídos.

- Reorquestrar o sistema sem forçar o sistema a parar, garantindo availability
- Suporte para outras linguagens e ports no code-generation e devices
- Suporte de code generation para outros nós
- Implementar lógica diferente de brute force para o alocamento de nós a dispositivos (knapsack ou assim)

7.4 Contributions

7.5 Conclusions

Appendix A

Scenario 2 Results

Start	End	Delta
1591876328759	1591876328770	11
1591876329440	1591876329448	8
1591876329991	1591876329994	3
1591876330539	1591876330554	15
1591876331106	1591876331120	14
1591876331658	1591876331667	9
1591876332192	1591876332200	8
1591876332710	1591876332721	11
1591876333222	1591876333237	15
1591876333779	1591876333787	8

Table A.1: Node-RED original results

Start	End	Delta
1591877265187	1591877265346	159
1591877266172	1591877267055	883
1591877267564	1591877267698	134
1591877268318	1591877268955	637
1591877269424	1591877269783	359
1591877270361	1591877271117	756
1591877271635	1591877272012	377
1591877272630	1591877273132	502
1591877273645	1591877273996	351
1591877274541	1591877275277	736

Table A.2: Node-RED + MQTT results

Start	End	Delta
1591877987030	1591877988695	1665
1591877989911	1591877991177	1266
1591877992272	1591877993595	1323
1591877994286	1591877995817	1531
1591877996305	1591877997618	1313
1591877998049	1591877999307	1258
1591877999734	1591878001322	1588
1591878001638	1591878002855	1217
1591878003397	1591878004643	1246
1591878005113	1591878006703	1590

Table A.3: Node-RED modified + Dockers (same host) results

Start	End	Delta
1591908868087	1591908870410	2323
1591908871443	1591908873803	2360
1591908874380	1591908877085	2705
1591908877629	1591908880338	2709
1591908880878	1591908883937	3059
1591908884472	1591908887147	2675
1591908887651	1591908889096	1445
1591908889803	1591908892200	2397
1591908892693	1591908894158	1465
1591908894846	1591908897623	2777

Table A.4: Node-RED modified + Dockers (different host) results

Start	End	Delta
1591904836329	1591904840130	3801
1591904844918	1591904849155	4237
1591904850127	1591904854579	4452
1591904855324	1591904859754	4430
1591904860483	1591904864559	4076
1591904865164	1591904869180	4016
1591904869770	1591904873905	4135
1591904874557	1591904878706	4149
1591904879318	1591904882934	3616
1591904888813	1591904893230	4417

Table A.5: Physical + MQTT results

Start	**End**	**Delta**
1591878582050	1591878587990	5940
1591878589026	1591878593492	4466
1591878594105	1591878598640	4535
1591878599238	1591878603841	4603
1591878604570	1591878608765	4195
1591878609340	1591878614220	4880
1591878615030	1591878620187	5157
1591878620870	1591878625038	4168
1591878625718	1591878630967	5249
1591878631560	1591878635880	4320

Table A.6: Node-RED modified + MQTT + Physical + Firmware

Appendix B

Paper Submitted

This appendix includes the paper submitted...

References

[1] ISO/IEC JTC 1. Internet of things (iot) - preliminary report. *ISO,Tech. Rep.*, 2014.

[2] Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. Visual simple transformations: Empowering end-users to wire internet of things objects. *ACM Transactions on Computer-Human Interaction*, 24(2):10:1—-10:43, apr 2017.

[3] S. A. Al-Qaseemi, H. A. Almulhim, M. F. Almulhim, and S. R. Chaudhry. Iot architecture challenges and issues: Lack of standardization. In *2016 Future Technologies Conference (FTC)*, pages 731–738, Dec 2016.

[4] Tanweer Alam. A reliable communication framework and its use in internet of things (iot). 3, 05 2018.

[5] Fahed Alkhabbas, Romina Spalazzese, and Paul Davidsson. Iot-based systems of systems. *Proceedings of the 2nd edition of Swedish Workshop on the Engineering of Systems of Systems (SWESOS 2016)*, 2016.

[6] AT&T. AT&T Flow Designer. Available: https://flow.att.com, 2020. Last access 2020. [Online].

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

[8] Andreu Belsa, David Sarabia-Jacome, Carlos E. Palau, and Manuel Esteve. Flow-based programming interoperability solution for IoT platform applications. In *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, pages 304–309, 2018.

[9] Adnan Rachmat Anom Besari, Iwan Kurnianto Wobowo, Sritrusta Sukaridhoto, Ricky Setiawan, and Muh Rifqi Rizqullah. Preliminary design of mobile visual programming apps for Internet of Things applications based on Raspberry Pi 3 platform. In *Proceedings - International Electronics Symposium on Knowledge Creation and Intelligent Computing, IES-KCIC 2017*, volume 2017-Janua, pages 50–54, 2017.

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

[11] Michael Blackstock and Rodger Lea. FRED: A hosted data flow platform for the IoT. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs, MOTA 2016*, 2016.

[12] Marat Boshernitsan and Michael Downes. Visual programming languages: A survey. 08 1998.

[13] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee. Scaling up visual programming languages. *Computer*, 28(3):45–54, March 1995.

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

[15] Rajkumar Buyya and Amir Vahid Dastjerdi. *Internet of Things: Principles and Paradigms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2016.

[16] Zenodys B.V. Zenodys. Available: `https://www.zenodys.com/`, 2020. Last access 2020. [Online].

[17] S K Chang. *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, 2002.

[18] S. Chen, H. Xu, D. Liu, B. Hu, and H. Wang. A vision of iot: Applications, challenges, and opportunities with china perspective. *IEEE Internet of Things Journal*, 1(4):349–359, Aug 2014.

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

[20] Bin Cheng, Gurkan Solmaz, Flavio Cirillo, Ernö Kovacs, Kazuyuki Terasawa, and Atsushi Kitazawa. Fogflow: Easy programming of iot services over cloud and edges for smart cities. *IEEE Internet of Things Journal*, PP:1–1, 08 2017.

[21] Gennaro De Luca, Zhongtao Li, Sami Mian, and Yinong Chen. Visual programming language environment for different IoT and robotics platforms in computer science education. *CAAI Transactions on Intelligence Technology*, 3(2):119–130, 2018.

[22] Giuseppe Desolda, Alessio Malizia, and Tommaso Turchi. A tangible-programming technology supporting end-user development of smart-environments. In *Proceedings of the Workshop on Advanced Visual Interfaces AVI*, AVI '18, pages 59:1—-59:3, New York, NY, USA, 2018. ACM.

[23] DGLogik. DGLux5. Available: `http://dglogik.com/products/dglux-for-dsa`, 2020. Last access 2020. [Online].

[24] Barrett Ens, Fraser Anderson, Tovi Grossman, Michelle Annett, Pourang Irani, and George Fitzmaurice. Ivy: Exploring spatially situated visual programming for authoring and understanding intelligent environments. In *Proceedings - Graphics Interface*, pages 156–163, 2017.

[25] Espressif Systems. Esp8266 technical reference manual. Technical report, Espressif Systems, Shanghai, China, 2019.

[26] Espressif Systems. Esp32 technical reference manual. Technical report, Espressif Systems, Shanghai, China, 2020.

[27] Teo Eterovic, Enio Kaljic, Dzenana Donko, Adnan Salihbegovic, and Samir Ribic. An Internet of Things visual domain specific modeling language based on UML. In *2015 25th International Conference on Information, Communication and Automation Technologies, ICAT 2015 - Proceedings*, 2015.

[28] Seongbae Eun, Jinman Jung, Young Sun Yun, Sun Sup So, Junyoung Heo, and Hong Min. An end user development platform based on dataflow approach for IoT devices. *Journal of Intelligent and Fuzzy Systems*, 35(6):6125–6131, 2018.

[29] Mahmoud S. Fayed, Muhammad Al-Qurishi, Atif Alamri, and Ahmad A. Al-Daraiseh. PWCT: Visual language for IoT and cloud computing applications and systems. In *ACM International Conference Proceeding Series*, 2017.

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

[31] Giuseppe Ghiani, Marco Manca, Fabio Paterno, and Carmen Santoro. Personalization of context-dependent applications through trigger-action rules. *ACM Transactions on Computer-Human Interaction*, 24(2):14:1—-14:33, apr 2017.

[32] N. K. Giang, R. Lea, M. Blackstock, and V. C. M. Leung. Fog at the edge: Experiences building an edge computing platform. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 9–16, July 2018.

[33] N. K. Giang, R. Lea, and V. C. M. Leung. Exogenous coordination for building fog-based cyber physical social computing and networking systems. *IEEE Access*, 6:31740–31749, 2018.

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

[35] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29, 07 2012.

[36] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

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

[38] Nikos Kefalakis, John Soldatos, Achilleas Anagnostopoulos, and Panagiotis Dimitropoulos. *A visual paradigm for IoT solutions development*, volume 9001. 2015.

[39] Martin Kleppmann, Adam Wiggins, Peter Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. pages 154–178, 10 2019.

[40] D. S. Linthicum. Connecting fog and cloud computing. *IEEE Cloud Computing*, 4(2):18–20, March 2017.

[41] Wei Liu, Takayuki Nishio, Ryoichi Shinkuma, and Tatsuro Takahashi. Adaptive resource discovery in mobile cloud computing. *Computer Communications*, 50:119 – 129, 2014. Green Networking.

[42] Grasp IO Innovations Pvt. Ltd. GraspIO. Available: https://www.grasp.io/, 2020. Last access 2020. [Online].

[43] C. Martín Fernández, M. Díaz Rodríguez, and B. Rubio Muñoz. An edge computing architecture in the internet of things. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 99–102, May 2018.

[44] Miao Yun and Bu Yuxin. Research on the architecture and key technology of internet of things (iot) applied on smart grid. In *2010 International Conference on Advances in Energy Engineering*, pages 69–72, June 2010.

[45] Mohammed Islam NAAS, Laurent Lemarchand, Jalil Boukhobza, and Philippe Raipin. A graph partitioning-based heuristic for runtime iot data placement strategies in a fog infrastructure. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 767–774, New York, NY, USA, 2018. Association for Computing Machinery.

[46] NooDL. NooDL. Available: https://classic.getnoodl.com/, 2020. Last access 2020. [Online].

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

[48] D. S. Nunes, P. Zhang, and J. Sá Silva. A survey on human-in-the-loop applications towards an internet of all. *IEEE Communications Surveys Tutorials*, 17(2):944–965, Secondquarter 2015.

[49] D. Pathirana, S. Sonnadara, M. Hettiarachchi, H. Siriwardana, and C. Silva. WireMe - IoT development platform for everyone. In *3rd International Moratuwa Engineering Research Conference, MERCon 2017*, pages 93–98, 2017.

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

[51] 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, USA, 2017. The Hillside Group.

[52] Reza Rawassizadeh, Timothy Pierson, Ronald Peterson, and David Kotz. Nocloud: Exploring network disconnection through on-device data analysis. *IEEE Pervasive Computing*, 17, 03 2018.

[53] Partha Pratim Ray. A Survey on Visual Programming Languages in Internet of Things. *Scientific Programming*, 2017, 2017.

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

[55] Joanna Sendorek, Tomasz Szydlo, Mateusz Windak, and Robert Brzoza-Woch. *FogFlow - Computation Organization for Heterogeneous Fog Computing Environments*, pages 634–647. 06 2019.

[56] Ricky Setiawan, Adnan Rachmat Anom Besari, Iwan Kurnianto Wibowo, Muh Rifqi Rizqullah, and Dias Agata. Mobile visual programming apps for internet of things applications based on raspberry Pi 3 platform. In *International Electronics Symposium on Knowledge Creation and Intelligent Computing, IES-KCIC 2018 - Proceedings*, pages 199–204, oct 2019.

[57] W. Shi and S. Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, May 2016.

[58] W. Shi, G. Pallis, and Z. Xu. Edge computing [scanning the issue]. *Proceedings of the IEEE*, 107(8):1474–1481, Aug 2019.

[59] N. C. Shu. Visual programming: Perspectives and approaches. *IBM Syst. J.*, 38(2–3):199–221, June 1999.

[60] SmartFog. FogFlow. Available: https://github.com/smartfog/fogflow, 2020. Last access 2020. [Online].

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

[62] T. Szydlo, R. Brzoza-Woch, J. Sendorek, M. Windak, and C. Gniady. Flow-based programming for iot leveraging fog computing. In *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 74–79, June 2017.

[63] Matúš Tomlein, Sudershan Boovaraghavan, Yuvraj Agarwal, and Anind K. Dey. CharIoT: An end-user programming environment for the IoT. In *ACM International Conference Proceeding Series*, 2017.

[64] Matúš Tomlein and Kaj Grønbæk. A visual programming approach based on domain ontologies for configuring industrial IoT installations. In *ACM International Conference Proceeding Series*, 2017.

[65] NETLab Toolkit. NETLabTK: Tools for Tangible Design. Available: www.netlabtoolkit.org/, 2020. Last access 2020. [Online].

[66] Yannis Valsamakis and Anthony Savidis. Visual end-user programming of personalized AAL in the internet of things. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10217 LNCS, pages 159–174, 2017.

[67] Wyliodrin. Wyliodrin. Available: https://wyliodrin.com/, 2020. Last access 2020. [Online].