# Interoperability Challenges in Internet of Things Systems: a Service-Oriented Computing Approach

Zingaro Stefano

Alma Mater Studiorum · Università di Bologna

**Computer Science and Engineering**
Ciclo XXXII

**Settore Concorsuale:** 01/B1
**Settore Scientifico Disciplinare:** INF/01

# Interoperability Challenges in Internet of Things Systems: a Service-Oriented Computing Approach

**Presentata da:**
Stefano Pio Zingaro

**Supervisore:**
Maurizio Gabbrielli

**Coordinatore Dottorato:**
Davide Sangiorgi

**Cosupervisore:**
Ivan Lanese

**Esame finale anno 2020**

# Abstract

Internet of Things systems are pervasive systems evolved from cyber-physical to large-scale systems. Due to the number of technologies involved, software development involves several integration challenges. Among them, the ones preventing proper integration are those related to the system heterogeneity, and thus addressing interoperability issues. From a software engineering perspective, developers mostly experience the lack of interoperability in the two phases of software development: programming and deployment. On the one hand, modern software tends to be distributed in several components, each adopting its most-appropriate technology stack, pushing programmers to code in a protocol- and data-agnostic way. On the other hand, each software component should run in the most appropriate execution environment and, as a result, system architects strive to automate the deployment in distributed infrastructures.

This dissertation aims to improve the development process by introducing proper tools to handle certain aspects of the system heterogeneity. Our effort focuses on three of these aspects and, for each one of those, we propose a tool addressing the underlying challenge. The first tool aims to handle heterogeneity at the transport and application protocol level, the second to manage different data formats, while the third to obtain optimal deployment.

To realize the tools, we adopted a linguistic approach, i.e. we provided specific linguistic abstractions that help developers to increase the expressive power of the programming language they use, writing better solutions in more straightforward ways. To validate the approach, we implemented use cases to show that the tools can be used in practice and that they help to achieve the expected level of interoperability.

In conclusion, to move a step towards the realization of an integrated Internet of Things ecosystem, we target programmers and architects and propose them to use the presented tools to ease the software development process.

# Contents

# List of Figures

# Listings

# Chapter 1

# Introduction

Among the many visions of people gravitating in the field of Computer Science one still remains a dream, both for the industry and for the academy, that of a software application that seamlessly interacts with the environment. In the last decades, this vision culminated into what the experts call the Internet of Things, that emerged as the logical continuation of its predecessors (distributed, pervasive, and ubiquitous computing). As for the previous, it came with no surprise that the issues raised by the actual implementation of such system is all but trivial. One of the main problems relies on the very composition of the environment, the more complex it is the more complex would be the interaction among its components. Even if we consider the case of a single piece of software interacting with every part of, lets take for example a house, we soon realize that thousands of components are involved and the design of such software becomes a very difficult task. So far, we intentionally avoid to define what is a system component but we can easily say that it is, following our example, a lamp, the fridge, a pet, or any other entity in the house.

In the real world there is no such single piece of software that, alone, interacts with the environment. Instead, several different software applications exchange information with each other and with the environment using either human-to-machine interfaces or machine-to-machine interfaces. Extending our example, suppose that the software is made of several sub-softwares and that any interaction that happened before with the environment, it happens now among the components and all the smaller pieces of software. The depicted scenario is now bigger in terms of number of interactions and the complexity increased. In practice, software developers deal with this complexity every day and a careful handling is needed in different aspects of the development. This happens not just at design time but during coding phase, since communication technologies update quickly, and at run time, because of ever changing physical environments. The question then becomes how to help developers to be more effective in handling the complexity of modern systems, assuming that (almost) nothing can be done to decrease the inner complexity of the system considered.

This question constitutes the very center of this dissertation, that focuses on the provisioning of proper tools that address the open challenges in software interoperability, both at coding and run time.

In this chapter, first, we dig into the scientific context of the investigation (section 1.1) devising the relevant facts in the recent history of Computer Science that would help us to give a definition for Internet of Things systems. Secondly, we briefly present the motivation of the research and the questions that lie behind the matter of discussion in section 1.2. Then, in section 1.3, we describe the essence of the approach trying to answer the research questions or, better, trying to address the challenges that arose from the problem analysis. Finally, we outline the dissertation in section 1.4 to give to the reader a functional index of the text.

## 1.1 Background

Since the 1960s, several paradigms determined the progress of computing architectures spanning from Mainframes to modern systems. In this process, it is possible to observe a shift in human interaction with programs — i.e. the objects of computation. At the very beginning of this process, the setup for a Mainframe architecture required many experts, while, some after, this relation became one-to-one when Personal Computers (PCs) brought, for the first time, computation in everyday life [30]. Networking and communication technologies pushed this interaction even further, allowing one user to control many devices.

In the early 1980s, the ability to take control over dislocated systems helped the growth of a global network — i.e. the World Wide Web (WWW) [65], employed by researchers in distributed computing [121] as a playground for distributed applications testing.

In the 1990s, cellular technologies allowed portable devices to connect to available networks while moving in the physical space, thus shifting reference architectures topology from being almost fixed to be very dynamic and heterogeneous.

In the 2000s, the explosion of the smartphone market pushed the adoption of a new computing model that took into account the communication capabilities of the devices as it was never done before. Both new pervasive systems (small and constrained devices) and old ones became part of a connected network exchanging information about the environment, e.g. user analytics, quality of service. In this ubiquitous scenario [94][1] each vendor tended to

[1] To disambiguate on the meaning of “pervasive” against “ubiquitous”, we point the reader

develop its own market, along with the communication technology stack needed to interconnect its devices. In the following years pervasive and ubiquitous computing fixed in the literature [110], developers begin to discuss heterogeneity handling problems, and a new notion of the modern system arises concerning invisible and immersive interactions between humans, machines and the environment [133] maturing the so-called *Cyber-Physical System* (CPS).

CPS are large-scale systems whose development focuses on the integration of computation with physical processes — i.e. they allow mapping the physical world with the digital one [64]. The full realization of CPS advocates for the creation of an integrated and self-regulating System of Systems (SoS) that behaves as a "connected domain". In modern architectures, the physical entities that compose the domain map with its cyber/digital version — the Thing — and the interconnection between Things relies on the set of communication technologies proper of the Internet [93].

In the following, we refer to such architectural style and the set of technologies that it underlies with the term "*Internet of Things*" (IoT) [135].

## 1.2 Research Problem

IoT emerged in the ubiquitous technologies scenario, as the next logical step to pervasive systems. Its paradigm states that the connectivity and communication technologies should be available "for anything, anywhere, and at any time" [135]. The IoT vision is that of a global network of interacting objects that accommodate to the environmental context without any human involvement. Such perspective advocates for a system capable of (i) safely connect any conventional electronic device to the global network (*anything*), (ii) embed information and communication in the environment invisibly (*anywhere*), and (iii) process relevant information in real-time and continuously feedback the system (*any time*).

So far, we described the IoT as an integrated and self-regulated system behaving as a connected domain where safely interacting entities process data in real-time in an infinite feedback loop with the environment. Unfortunately, IoT systems are far to meet the vision yet described.

IoT systems show a high degree of interaction among their components as well as extremely high heterogeneity. Furthermore, they present a wide variety of media protocols, standards, data representation formats, and possible deployment platforms.

to [98].

In this babel of technologies, IoT developers hardly manage to program interoperable applications due to the lack of proper tools supporting system integration [48]. Here, we consider "integrated" an application for IoT systems free from communication defects in which all the relevant information reach the correct place in a way that the recipient can understand what it receives [135].

Overall, we state that this dissertation aims to improve the effectiveness of the software development procedure of integrated applications for IoT systems.

To effectively handle such heterogeneity and reach interoperability among system components, the approach to IoT applications software development needs to be enriched both in the programming phase and in the deployment-related practices [72].

On the one hand, programmers need support to enable protocol and data format -agnostic programming. Agnosticism would allow them to abstract from the communication technologies, focusing on the application logic rather than on its deployment. Consequently, developers and IoT architects would benefit from tools addressing deployment automation. Since IoT network topologies change rapidly during the application life-cycle, developers need to check if the application requirements reflect the architecture profile continually.

Summarizing what we depicted so far, we identify the following three challenges as the leading problems of our research.

- The first (challenge $C1$), concerns the conceptualization of a solution that guarantees programmers to code natively applications exploiting media protocols interoperability.

- The second (challenge $C2$), relates to providing programmers with a solution for efficient data handling in IoT applications, tackling the data format interoperability.

- The third (challenge $C3$), aims to provide architects with a tool that automates the deployability check of an IoT application into the existing architecture, thus enabling continuous integration of software in the IoT context.

Our claim is that by addressing these challenges in the IoT context and answering the interoperability related issues would give to the developers a way to effectively handling at least the three aspects of system heterogeneity in our focus.

## 1.3 Approach

In general, since the research challenges relates to the effectiveness of the development procedure, it is straightforward to see them as software engineering problems. To improve the effectiveness of software development we propose ready-to-use practical tools that can be used to build both prototypes and mature solutions and allow the integration of these solutions into test or production pipelines. The problem of integration is not new in the distributed computing scenario and has been successfully addressed in the field of Service-Oriented Architectures (SOAs) since the very birth of the pervasive system paradigm [32]. SOA proved to be one of the most effective techniques to integrate components in highly heterogeneous systems — e.g. web services and telecommunication services [32] — and, in recent history, even in the IoT context [38].

Leveraging the work done in the area of SOAs, we address challenges $C1$, $C2$, and $C3$ by adopting a Service-Oriented approach applying the concepts on which SOA relies to programming languages. The linguistic approach to SOA is known as Service-Oriented Computing (SOC) that aims to use SOA principle to develop proper language constructs to define service-centric applications. Concretely, we address $C1$ by integrating two of the most adopted protocols for IoT communications into an existing programming language, Jolie. We adopt a similar approach for $C2$, extending the Jolie language to include a query framework for handling modern data formats. Finally, we propose a tool for $C3$ that allows to find the optimal deployment plan of an IoT application, by taking advantage of a declarative specification language to describe both the application and the infrastructure and a state-of-the-art configuration optimizer to minimize the total running cost of the application.

The approach that we propose addressing $C1$, $C2$, and $C3$, i.e. the extension of a programming language, we leverage the work done in SOC [96, 127, 63] and use linguistic abstractions to reason on IoT systems. Our claim is that without proper abstractions, guaranteeing interoperability among different technology stacks is highly complex. Furthermore, the problem intensifies when one has to modify the technology stack used for some specific interaction.

The thesis of this dissertation asserts that Service-Oriented Architectures approaches can be successfully extended to support the design, development and deployment of distributed software application for IoT systems. In conclusion, the following constitutes the thesis statement: Service-Oriented Computing can be used to program integrated IoT systems.

## 1.4 Outline of the Dissertation

The reminder of this dissertation is structured in three main chapters in which we introduce the reasons and issues that motivate our investigations and report our results. All results have been produced during the course of the PhD studies and are here presented in an extended form.

- In **Chapter 2**: we present high-level concepts that are valuable both for the general implementation of interoperable systems and for the development of linguistic solutions. The matter of Chapter 2 is to present a feasible solution to the issue related to challenge $C1$.
- In **Chapter 3**: we discuss a solution for effective data handling in IoT applications, addressing data formats interoperability and performance stability (challenge $C2$), following a linguistic approach to service-oriented computing.
- In **Chapter 4**: we propose a tool that allows one to find the optimal deployment plan by using a declarative specification language and a state-of-the-art configuration optimizer, thus addressing the challenge $C3$.
- In **Chapter 5**: we summarise the contributions of this dissertation and relate them to similar existing work. We also discuss some interesting directions of future investigation.

# Chapter 2

# Protocols Interoperability in IoT Systems

IoT systems show a high degree of interaction among components as well as extremely high heterogeneity. Programming an IoT application involves a careful selection of the technology stack to be used since a choice not taking into account interoperability issues, would prevent the integration in the existing IoT ecosystem.

This work aims to code IoT systems effectively and to accomplish both cross-layer and cross-platform seamless integration, we tackle the challenge of IoT transport and application protocols interoperability, following a linguistic approach to microservice-oriented computing.

Leveraging the work done in the *Service-Oriented Architecture* (SOA) area, we extend the syntax of an existing programming language, Jolie, increasing its expressive power, and build the JIoT interpreter. We implemented and integrated into the Jolie interpreter two of the most adopted protocols in IoT communication: CoAP and MQTT.

On one hand, JIoT supports the leading technologies both from Service-Oriented Computing and IoT — i.e. *TCP/IP*, *Bluetooth*, *RMI*, and UDP at the transport level, and *HTTP*, *SOAP*, CoAP, and MQTT at the application level. On the other hand, it provides uniform linguistic abstractions to exploit heterogeneous communication stacks, allowing the programmer to specify in a declarative way the desired technologies, and to change them (even at runtime) easily. To validate our methodology and present the tool features, we design and implement a smart building automation case study, using the proposed technology.

In conclusion, in this work we present high-level concepts that are valuable both for the general implementation of interoperable systems and for the development of other linguistic solutions.

## 2.1 Introduction

IoT advocates for multi-layered software platforms, each adopting its media protocols and data formats [48, 93, 36].

The problem of integrating layers of the same IoT platform, as well as different IoT vertical solutions, involves many levels of the communication stack, spanning from link-layer communication technologies, such as BLE, ZigBee and WiFi, to application-layer protocols like HTTP, CoAP [9, 115], and MQTT [7, 92], reaching the top-most layers of data-format integration [81].

Technology-wise, developers and IT staff, in charge of the design of IoT platforms interaction, can choose between two approaches at odds. The first approach favours optimal in-layer communications — i.e. selecting media protocols and data formats best suited for the interactions happening among homogeneous elements, such as edge devices (connectionless protocols and binary data formats [36]), mid-tier controllers (gateways and aggregators on the RESTful stack [61]), or Cloud nodes (scalable publish-subscribe message queues [42]). Following the approach that favours the adoption of fixed standards is optimal for in-layer communication. However, at the cross-layer level, the heterogeneity and possible incompatibility of the chosen standards make enforcing integrity within the IoT system complex and the resulting integration fragile. The second architectural approach favours cross-layer consistency, enforcing a unique communication stack over a single IoT platform. Here cross-layer integration is more straightforward thanks to the adoption of a single medium and data format. However such enforced uniformity is the leading cause of the phenomenon known as "IoT island" [119, 45], where IoT platforms take the shape of vertical solutions that provide little support for collaboration and integration with each other. How to overcome this limitation is currently a hot topic, also tackled by ongoing EU projects — e.g. symbiote [45] and biotope [8].

In this chapter, we tackle the problem of IoT integration (both cross-layer and cross-platform) following a language-based approach focused on integration at both the transport (TCP or UDP) and the application layer. To reach our goal we do not start from scratch, but we leverage the work done in the area of Service-Oriented Architectures (SOAs) [32], and we build on the Jolie programming language [86, 88, 89, 60]. In particular, we rely on those abstractions provided by Jolie that (i) let different communication protocols seamlessly coexist and interoperate within the same program and (ii) let programmers dynamically choose which communication stack is the most suited for any given communication.

Concretely, we fork the Jolie interpreter — written in Java — into a prototype called JIoT [39], standing for "Jolie for IoT".

JIoT supports all the protocols already supported by the Jolie interpreter. These protocols consists of TCP at the transport level, and SOAP, RMI and HTTP at the application level. In addition, JIoT presents support to the application-level protocols for IoT, namely CoAP (and, as a consequence, UDP at the transport level) and MQTT.

Notably, when the application protocol supports different representation formats, such as XML or JSON, of the message payload, as in the case of HTTP and CoAP, JIoT, like Jolie, can automatically marshal and un-marshal data as required.

JIoT is available at [39], and released under the GNU GPL v2.1 license. The code snippets reported in this chapter are based on version 1.2 of JIoT.

We structure the presentation of this work as follows. Section 2.2 describes the current solutions addressing interoperability issues in the IoT context, and focusing on approaches closer to ours. In section 2.3, we overview our approach and summarize our contribution in subsection 2.3.1. In subsection 2.3.3, we discuss the main challenges we faced in our development and we present how a programmer can use CoAP/UDP and MQTT in JIoT, detailing our implementation in subsection 2.3.4. In section 2.4, we describe a smart building automation scenario where a JIoT architecture coordinates the IoT application. Finally, we position our contribution with respect to related work and we draw final remarks in section 2.5.

## 2.2 Related Work

In the literature, there are many proposals for platforms, middleware, smart gateways, and general systems, all aimed at solving the interoperability problem arising from the current "babel" of IoT technologies (protocols, formats, and languages). Without any claim of being complete, here we mention a few notable examples which are related to our approach.

Recently the W3C started the Web of Things (WoT) Working Group [131]. WoT aims to define a standard stack of layered technologies, as well as software architectural styles and programming patterns, to uniform and simplify the creation of IoT applications. In this context, the W3C is working on a WoT Architecture [132]. The central concept of the architecture is the notion of "servient", a virtual entity that represents a physical IoT device. Servients provide technology-independent, standard APIs that developers can use to operate in heterogeneous environments transparently. Remarkably, both the

WoT proposal and ours concern high-level abstractions for low-level access to devices provided via — e.g. HTTP, CoAP, and MQTT.

More in general, there are many proposals for the integration of WoT and IoT. For example, [51] and [22] define global platforms covering different layers of IoT, including an accessibility layer which integrates concepts like smart gateways and proxies to facilitate the connection of (smart) Things into the Internet infrastructure, using architectural principles based on REST.

Smart gateways and proxies are used in several industrial proposals to facilitate the development of applications. The common denominator of some of these proposals — e.g. [117, 109, 107] — is the abstraction of low-level functionalities provided by embedded devices — e.g. connectivity and communication over low-level protocols like ZigBee, Z-Wave, Wi/IP/UPnP. Smart gateways are used also to translate (or integrate) CoAP into HTTP [120, 70, 82] and to integrate both CoAP and MQTT by means of specific middleware [123].

Eclipse IoT [124] is an IoT integration framework proposed by the Eclipse IoT Working Group. Aim of Eclipse IoT is to build an open IoT stack for Java, including the support for device-to-device and device-to-server protocols, as well as the provision of protocols, frameworks, and services for device management. There exist several European projects, notably INTER-IoT [41] and symbIoTe [45], that address the issue of interoperability in IoT and have produced several concrete proposals. Finally, a work close to ours is [130], where a middleware converts IoT heterogeneous networks into a single homogeneous network.

To conclude our revision, we narrow our focus on language-based integration solutions for IoT. The work mostly related to ours is SensorML [125]. SensorML, the abbreviation of Sensor Model Language, is a modelling language for the description of sensors and, more in general, of measurement processes. Some features modelled by the language are discovery and geolocalization of sensors, processing of sensor observations, and functionalities to program sensors and to subscribe to sensor events.

## 2.3 Approach

Without proper language abstractions, guaranteeing interoperability among protocols belonging to different technology stacks is highly complex. Furthermore, the problem intensifies when one has to modify the technology stack used for some specific interaction. The replacement may be either static — e.g. because of the deployment of new, heterogeneous devices in a pre-existing

system — or dynamic — e.g. to support a changing topology of different mobile devices. Contrarily, with JIoT most of the complexity of guaranteeing interoperability is managed by the language interpreter and hidden from the programmer.

As an illustrative example of the proposed approach, let us consider a scenario where we want to integrate two islands of IoT devices, both collecting temperature data, but relying on different communication stacks, namely HTTP over TCP and CoAP over UDP.

The end goal is to program a collector which receives and aggregates temperature measurements from both islands.

Following the structure of Jolie programs, the collector programmed in JIoT is composed of two parts: (i) a *behavior*, specifying the logic of the elaboration, and (ii) a *deployment*, describing in a declarative way how communication happens. This separation of concerns is fundamental to let programmers easily change which communication stack to use, preserving the same logic for the elaboration.

As an example of program behaviour, let us consider the code in listing 2.1, where `main` is the entry point of execution of Jolie programs.

```
1 main
2 {
3     //    ...
4     receiveTemperature( data )
5     //    ...
6 }
```

**Listing 2.1:** Example of a Jolie input behaviour.

Above, line 5 contains a reception statement. Receptions in Jolie indicate a point where the program waits to receive a message.

In this case, the collector waits to receive a temperature measurement on *operation* `receiveTemperature` (an operation in Jolie is an abstraction for technology-specific concepts such as channels, resources, URLs, ...). Upon reception, it stores the retrieved value in variable `data`.

Besides the logic of computation of the collector, we also need to specify the deployment — i.e. on which technologies the communication happens. In the example above, it concerns how the collector receives messages from other devices. In Jolie this information is defined within *ports*. For example, the port to receive (denoted with keyword `inputPort`) HTTP measurements can be defined as in listing 2.2.

```
1 interface TemperatureInterface {
```

```
2     OneWay:     receiveTemperature( string )
3 }
4
5 inputPort CollectorPort1 {
6     Location:   "socket://localhost:8000"
7     Protocol:   http
8     Interfaces: TemperatureInterface
9 }
```

**Listing 2.2:** Example of interface and input port in Jolie.

Port `CollectorPort1` specifies that the collector expects inbound communications via `Protocol` `http` using a TCP/IP socket receiving at URL `"localhost"` on TCP port `8000`. A port exposes a set of operations, collected within a set of `Interfaces`. In the example, the input port `CollectorPort1` declares to expose interface `TemperatureInterface`, which is defined at lines 1–3 of listing 2.2. The interface declares the operation `receiveTemperature`, including the type of expected data (`string`), as a `OneWay` operation, namely an asynchronous communication that does not require any reply from the collector (except the acknowledgement automatically provided by the TCP implementation).

Thanks to port `CollectorPort1`, the collector can receive data from the HTTP island. To integrate the second island, we need to define an additional port, similar to `CollectorPort1`, except for using UDP/IP datagrams at the transport layer and CoAP [115, 9] at the application layer. Hence, the whole code of the collector becomes:

```
1 interface TemperatureInterface {
2     OneWay:     receiveTemperature( string )
3 }
4
5 inputPort CollectorPort1 {
6     Location:   "socket://localhost:8000"
7     Protocol:   http
8     Interfaces: TemperatureInterface
9 }
10
11 inputPort CollectorPort2 {
12     Location:   "datagram://localhost:5683"
13     Protocol:   coap
14     Interfaces: TemperatureInterface
15 }
16
17 main {
18     // ...
19     receiveTemperature( data )
20     // ...
```

```
21 }
```

**Listing 2.3:** Code of the Collector Example.

The example above highlights how, using the proposed language abstractions, the programmer can write a unique behaviour and exploit it to receive data sent over heterogeneous technology stacks.

Indeed, the `receiveTemperature` operation takes measurements from both the `inputPort`s. For instance, if communication over `CollectorPort2` fails, port `CollectorPort1` can still receive data.

Programmers can also specify elaborations that depend on the used technologies by using different operations in different ports. Jolie supports both inbound and outbound communications, the latter declared with `outputPort`s, whose structure follows that of `inputPort`s. Furthermore, the `Location` and `Protocol` of `outputPort`s can be changed at runtime, enabling the dynamic selection of the appropriate technologies for each context.

As mentioned, Jolie enforces a strict separation of concerns between behaviour, describing the logic of the application, and deployment, describing the communication capabilities. The behaviour is defined using the typical constructs of structured, sequential programming, communication primitives, and operators to deal with concurrency (parallel composition and input choices [89]). Jolie communication primitives comprise two modalities of interaction. Outbound `OneWay` communications send a message asynchronously, while `RequestResponse` communications send a message and wait for a reply (they capture the well-known pattern of request-response interactions [129]). Dually, inbound `OneWay` communications wait to receive a message, without sending a reply, while inbound `RequestResponse`s wait for a message and send back a reply.

Jolie supports many communication media (via keyword `Location`) and data protocols (via keyword `Protocol`) in a simple, uniform way. One of the main features of the Jolie language and the reason why we base our approach on it is because of the large number of technologies it already supports.

Each communication port declares the socket and data protocol used to communicate, hence, to switch to a different technology stack, one needs to change the declaration of `Location` and `Protocol` of a given port.

As expected, the behaviour— i.e. the actual logic of computation — of any Jolie program is unaffected by any change to its ports. Hence, a Jolie program can provide the same service — i.e. the same behaviour— through different media and protocols just by specifying different deployments. Being born in the field of SOAs, Jolie supports the main technologies from that

area: (i) communication media like TCP/IP sockets, Bluetooth L2CAP, Java RMI, and Unix local sockets; and (ii) data protocols like HTTP, JSON-RPC, XML-RPC, SOAP and their respective SSL versions.

### 2.3.1 Contribution

To substantiate the effectiveness of our language-based approach to IoT integration, we added to Jolie support for the primary communication stacks used in the IoT setting. Concretely, the added contribution of JIoT with respect to Jolie is the integration of two application protocols relevant in IoT scenarios, namely CoAP [115, 9] and MQTT [7, 92]. Notably, in JIoT the usage of such protocols is supported by the same linguistic abstractions that Jolie uses for SOA protocols such as HTTP and SOAP.

Even if Jolie provides support for the integration of new protocols, when set in the context of IoT technology, the task is non-trivial. Indeed, all the protocols previously supported by Jolie exploit the same internal interface, based on two assumptions: (i) the usage of underlying technologies that ensure reliable communications and (ii) a point-to-point communication pattern.

However, those assumptions do not hold when considering the two IoT technologies we integrated:

- CoAP communications can be unreliable in case of the adoption of UDP connectionless datagrams (as stated in the reference guide [115]). CoAP provides options for reliable communications, however, in the IoT settings, developers usually disable these features since it is crucial to preserve battery and bandwidth;

- MQTT communications rely on the publish-subscribe paradigm, which contrasts with the point-to-point paradigm underlying the Jolie communication primitives. Hence, we need to define a mapping to express publish-subscribe operations in terms of Jolie communication abstractions. In doing so, we need to balance two factors: (i) preserving the simplicity of use of the point-to-point communication style and (ii) capturing the typical publish-subscribe flow of communications. Such a mapping is particularly challenging in the case of request-response communications. Remarkably, the mapping that we present in this work is general and could also serve other contexts.

In the remainder of this section, we organize the presentation of our contributions as follows. In subsection 2.3.2 we provide the boundaries of our research

and investigation on linguistic abstractions for the IoT context. In subsection 2.3.3 we discuss the relative challenges encountered in the design of the solutions proposed. Finally, we detail on our implementation in subsection 2.3.4 including:

- a general account on how media and protocols work separately from the Jolie interpreter and how developers could implement them as independent modules;
- extensive details on the implementation of UDP, CoAP, and MQTT protocols.

### 2.3.2 Limitations

Here, we briefly discuss the current limitations of JIoT related to its usage in the programming of low-level Edge devices — like Arduino and other microcontrollers. JIoT supports dynamic scenarios where the nodes in the network can switch among many technology stacks according to internal or environmental conditions, such as available energy or communication link quality. From preliminary discussions with collaborators and IoT practitioners, we collected positive opinions on the idea of using JIoT for low-level programming Edge devices. Given these remarks, we investigated the feasibility of running JIoT programs over Edge devices, possibly including additional language abstractions to provide low-level access to on-board sensors and actuators. However, our survey revealed a market of devices fragmented over incompatible hardware architectures and characterized by strong constraints over both computational power and energy consumption. Considering these limitations, we concluded that supporting the execution of JIoT-like programs over Edge devices would require a substantial engineering effort.

While this research direction is promising, we deem it non-urgent, since currently, developers tend to program elementary behaviours for Edge devices [36], which usually capture some data — e.g. through one of their sensors — and then send them to mid-to-top-tier devices. The latter usually process and coordinate the flow of data: they have powerful hardware, they communicate over reliable channels, and they have fewer (if any) constraints concerning battery/energy consumption.

Considered the discussion above, in this chapter we omit the low-level programming of Edge devices, and we focus on mid-to-top-tier ones, which can host the JIoT runtime and which, given their topological context, directly benefit from the flexibility of the approach.

### 2.3.3 The JIoT Programming Language

Jolie currently supports some of the leading technologies used in SOAs — e.g. HTTP, SOAP. However, only a limited amount of IoT devices uses the media and protocols already supported by Jolie. Indeed, protocols such as CoAP and MQTT, which are widely used in IoT scenarios, are not implemented in Jolie. Integrating these protocols, as we have done, is essential to allow Jolie programs to directly interact with the majority of IoT devices. We note that emerging frameworks for interoperability, such as the Web of Things [51], rely on the same protocols we mentioned for IoT. Thus JIoT is also compliant with them.

However, there are some challenges linked to the integration of these technologies within Jolie:

- *lossless vs lossy protocols* — In SOAs, machine-to-machine communication relies on lossless protocols: there are no strict constraints on energy consumption or bandwidth, and it is not critical how many transport-layer messages are needed to ensure reliable delivery. That is not true in IoT networks, where communication is constrained by energy consumption, which defines what technology stack is the best suited. Indeed, many IoT communication technologies, among which the most renowned CoAP application protocol, rely on the UDP transport protocol — a connectionless protocol that gives no guarantee on the delivery of messages but allows one to limit message exchanges and, by extension, energy and bandwidth consumption. Since Jolie assumes lossless communications, the inclusion of connectionless protocols in the language requires careful handling to prevent bad behaviours;

- *point-to-point vs publish-subscribe* — The premise of the Jolie language is to provide communication constructs that do not depend on a specific technology. To be able to accomplish the goal, the underlying language assumes a point-to-point communication abstraction, which is common to many protocols like HTTP and CoAP. However, to integrate the MQTT protocol in Jolie, we need to model Jolie point-to-point semantics as MQTT publish-subscribe operations. Indeed, Jolie already provides language constructs usable with many communication protocols. Hence, the less disruptive approach would take advantage of the same constructs, which ideally suit in point-to-point settings, and thus also for MQTT. In the depicted context, we require to find, for each point-to-point construct, a corresponding effect on the publish-subscribe paradigm. The final result

is that the execution of a given Jolie behaviour is similar under both point-to-point and publish-subscribe technologies.

### Supporting Constrained Application Protocol in Jolie

The *Constrained Application Protocol* (CoAP) is a specialized web transfer protocol for constrained scenarios where nodes have low power and networks are lossy. The goal of CoAP is to import the widely adopted model of REST architectures [34] into the IoT context, that is, to optimize machine-to-machine interaction and ease the integration with REST-like protocols, such as HTTP. In particular, like HTTP, CoAP makes use of GET, PUT, POST, and DELETE methods.

Following the RFC [115], CoAP is implemented on top of the UDP transport protocol [102], with optional reliability. Indeed, CoAP provides two communication modalities: a reliable one, obtained by marking the message type as Confirmable (CON), and an unreliable one, obtained by marking the message type as Non-confirmable (NON).

As an example, we consider a scenario with a controller, programmed in JIoT, that communicates with one of many thermostats in a home automation scenario. Thermostats are accessible at the generic address `"coap://localhost/##"` where `"##"` is a two-digit number representing the identifier of a specific device. Each thermostat accepts two kinds of interactions: a GET request on URI `"coap://localhost/##/getTemperature"`, that returns the current temperature, and a POST request on URI `"coap://localhost/##/setTemperature"`, that sets the temperature of the HVAC (heating, ventilation, and air conditioning) system.

We comment below listing 2.4, where we report the code of a possible JIoT controller that interacts with a specific thermostat.

```
1 type getTmpType: void { .id: string }
2 type setTmpType: int { .id: string }
3
4 interface ThermostatInterface {
5     RequestResponse: getTmp( getTmpType )( int )
6     OneWay: setTmp( setTmpType )
7 }
8
9 outputPort Thermostat {
10     Location: "datagram://localhost:5683"
11     Protocol: coap {
12         .osc.getTmp << {
13             .messageCode = "GET",
14             .contentFormat = "text/plain",
15             .messageType = "CON", // or "NON"
```

```
16              .alias = "/%!{id}/getTemperature"
17          };
18          .osc.setTmp << {
19              .messageCode = "POST",
20              .messageType = "CON", // or "NON"
21              .alias = "/%!{id}/setTemperature"
22          }
23      }
24      Interfaces: ThermostatInterface
25  }
26
27  main {
28      getTmp@Thermostat( { .id = "42" } )( temp );
29      if ( temp > 27 ) {
30          setTmp@Thermostat( 24 { .id = "42" } )
31      } else if ( temp < 15 ) {
32          setTmp@Thermostat( 22 { .id = "42" } )
33      }
34  }
```

**Listing 2.4:** JIoT controller communicating over CoAP/UDP.

Our scenario includes two CoAP resources, referred to as `"/getTemperature"` and `"/setTemperature"`.

We model them in JIoT at lines 4–7 of listing 2.4, by defining the `interface ThermostatInterface`, which includes a `RequestResponse` operation `getTmp`, representing resource `"/getTemperature"`, and a `OneWay` operation `setTmp`, representing resource `"/setTemperature"`.

By default, we map operation names to resource names, hence in our example we would need resources named `"/getTmp"` and `"/setTmp"`, respectively. However, one can override this default by defining the coupling of resource names and operations as desired. The freedom that the language guarantees in terms of names and operations coupling allows programmers to use interfaces as high-level abstractions for interactions, while the grounding to the specific case stays a deployment matter.

Here we purposefully choose to use operation names that differ from resource names to underline that the two concepts are related but loosely coupled.

On the one hand, the coupling between the name of the resource and that of the operation is a way of quickly binding actions exposed by the CoAP server with operations. On the other hand, decoupling resource names and operations permits to handle more complex deployments where, for instance, a single operation responds for different resources.

At lines 9–25 we define an `outputPort` to interact with the `Thermostat`. At line 10, we specify the `Location` of the thermostat. Recalling that the scheme of

the resources of the thermostats is `"coap://localhost/##/\ldots"`, we define the `Location` of the port using the UDP `"datagram://"` protocol, followed by the first part of the resource schema `"localhost"` and the UDP port on which it accepts requests. Here we assume thermostats to use CoAP standard UDP port, which is `"5683"`. Note that, in the `Location`, we do not define the address of a specific thermostat — e.g. `"datagram://localhost:5683/42"`. On the contrary, we specify the generic address to access thermostats in the system, while the specific binding will be done at runtime, thanks to the `.alias` parameter of the `coap` protocol described later on.

At line 11 we define `coap` to be the protocol used by the `outputPort`. At lines 12–22 we specify some parameters of the `coap` protocol — this matches the standard way in which Jolie defines parameters for `Protocol`s in ports.

Here, we follow the methodology presented in [86] for the implementation of the HTTP protocol in Jolie — indeed CoAP adopts HTTP naming schema and resource interaction methods. In particular, we draw from [86] the parameter prefix `".osc"`, whose name is the acronym of "operation-specific configuration" and which developers generally use for configuration parameters related to a specific operation.

In the example, we define `.osc` parameters for both operations `getTmp` and `setTmp`. At line 13 we specify that the CoAP verb used for operation `getTmp` is `"GET"`. At line 14 we define, using the `.contentFormat` parameter that the encoding of the payload of the message is in text format. Other accepted values for the `.contentFormat` parameter are `"json"` and `"xml"`. Marshalling and unmarshalling are automatic and transparent to the programmer. The structure of Jolie variables enables this feature, which is always tree-shaped. Hence they can be easily translated into representations based on that shape. At line 15 we set the `.messageType` parameter to `"CON"`, that stands for Confirmable. Accepted values for the `.messageType` parameter are Confirmable and Non-confirmable (`"NON"`), the latter being the default value. In the first case, the sender will receive an acknowledgement message from the receiver, in the second case, it will not. At line 16, following the practice introduced in [86], we specify that `getTmp` is an `alias` for a resource whose path concatenates a static part, given by the `Location`, and the instantiation of the template `"/%!{id}/getTemperature"` provided by protocol parameter `.alias`.

The template is instantiated using values from the parameter of the operation invocation in the behavior, e.g. value `42` at line 28[1]. Hence, the interpretation of the declaration at line 16 is that when invoking operation `getTmp` at runtime,

[1] In Jolie the dot `.` defines path traversals inside trees. Hence, the notation `.id = 42` indicates a tree with an empty root and a sub-node called `id`, whose value is `42`.

the element `id` of the invocation will be removed from the payload and used to form the address of the requested resource. The aliasing for operation `setTmp` (line 21) is similar to that of `getTmp`, while the operation uses verb `POST`. Since here the `.contentFormat` parameter is omitted, the default `"text/plain"` is used.

To conclude, we briefly comment on the runtime execution of the example, described in the behaviour at lines 28–33. At line 28 the controller invokes operation `getTmp`. When in the presence of outgoing `RequestResponse`, the invocation defines on which port to perform the request (`Thermostat`) and presents two pairs of round brackets: the first contains the data for the request, the second points to the variable that will store the received response. Recalling the aliasing defined at line 16, at line 28 we define the value of element `id = 42`, thus the URI of the resource invoked at runtime is `"coap://localhost/42/getTemperature"`. Notably, in the example we hard-coded the `id` of the device, however in a more realistic setting the value of `id` would be retrieved dynamically — e.g. as an execution parameter, from a configuration file or a database. Once received, the response from thermostat `42` is assigned to variable `temp`. The example concludes with a conditional in which, if the temperature is above `27` degrees (line 29), the software sets the thermostat to a lower room temperature (`24` degrees), while, if the temperature lies below `15` degrees, it sets the thermostat to `22` degrees.

Dually to `outputPort`s, `inputPort`s allow the programmer to specify inbound communications. The parameters described above are valid also for `inputPort`s, with the only difference that `messageType` works only for `RequestResponse`s, and specifies whether the communication of the reply is reliable or not.

Note that, concerning the `.alias` parameter, the template is instantiated using the address of the incoming communication, and the values take place among the elements of the payload.

### Supporting Message Queue Telemetry Transport in Jolie

*Message Queue Telemetry Transport* (MQTT) is a publish/subscribe messaging application protocol built on top of the TCP transport protocol.

A typical publish/subscribe interaction pattern can be diagrammatically represented as in figure 2.1 where:

1. a Subscriber subscribes to the topic (a) at some Broker;
2. a Publisher publishes a message to the topic (a) at the same Broker;
3. the Broker forwards the message to the topic (a) to the Subscriber.

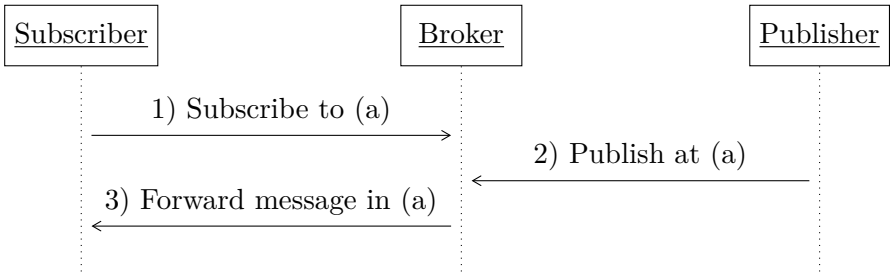


**Figure 2.1:** Typical publish/subscribe interaction pattern.

More generally, the Broker forwards the messages published on a topic to all current subscribers.

On top of the underlying mechanism of publishing/subscribing, MQTT defines three levels of quality of service (QoS) for the delivery of each message published by a publisher. QoS levels determine whether messages can be lost or duplicated.

Concretely, QoS levels are as follows:

- *At most once* — the message can be lost, no duplication can occur.
- *At least once* — delivery of the message is guaranteed, but duplication may occur.
- *Exactly once* — delivery of the message is guaranteed and duplication cannot occur.

To present how we model the MQTT protocol in JIoT, we first detail the simpler case of `OneWay` communications in subsection 2.3.3. Then, we address the more complex case of `RequestResponse` communications in listing 2.7. Notably, our modelling of end-to-end communications over a publish/subscribe channel is independent of JIoT — i.e. it is a general reference on how to implement one-way and request-response communications on top of any publish/subscribe channel.

*One-Way Communications in MQTT.* We first consider the case of inbound communications and then the case of outbound communications.

We exemplify `OneWay` inbound communications using the example in listing 2.5, which is a revision of the example in listing 2.3 by omitting the ports `CollectorPort1` and `CollectorPort2` and by adding an MQTT `inputPort` named `CollectorPort3`.

```
1 interface TemperatureInterface {
2     OneWay: receiveTemperature( string )
3 }
4
5 inputPort CollectorPort3 {
6     Location: "socket://localhost:8050"
7     Protocol: mqtt {
8         .broker = "socket://localhost:1883"
9     }
10    Interfaces: TemperatureInterface
11 }
12
13 main {
14    // ...
15    receiveTemperature( data )
16    // ...
17 }
```

**Listing 2.5:** Code of the Collector Example, revised for MQTT.

As expected, the program behaviour and the structure of the `inputPort` are unchanged. Main novelties are:

- the used `Location` (line 6) has the prefix `"socket://"` (as seen in the HTTP port) since MQTT relies on TCP transport protocol;
- the used `Protocol` (line 7) is `mqtt`;
- the `.broker` protocol parameter (line 8), which is compulsory when the `mqtt` protocol is used in `inputPort`s, specifies the address of the Broker.

From the developers perspective, the syntax and the effects of the communication primitives are the same as listing 2.3. However, we actually exchange several messages to capture that effect in MQTT, as shown in figure 2.2.

Beyond defining such message exchanges, we also need to decide how to identify the topic on which the two endpoints perform the message exchange.

Regarding the message exchanges, an inbound `OneWay` communication receives a datum from the communication partner. To obtain the same effect using the publish/subscribe paradigm, one has first to subscribe at the Broker to the chosen topic and then wait to receive a message on that topic, forwarded by the Broker. How topics are selected will be detailed later on. The execution of reception on a `OneWay` operation comprises two actual communications: a subscription from the program to the Broker and message delivery in the opposite direction. However, subscription to topics and the execution of a message reception are logically separated and happen at different moments.

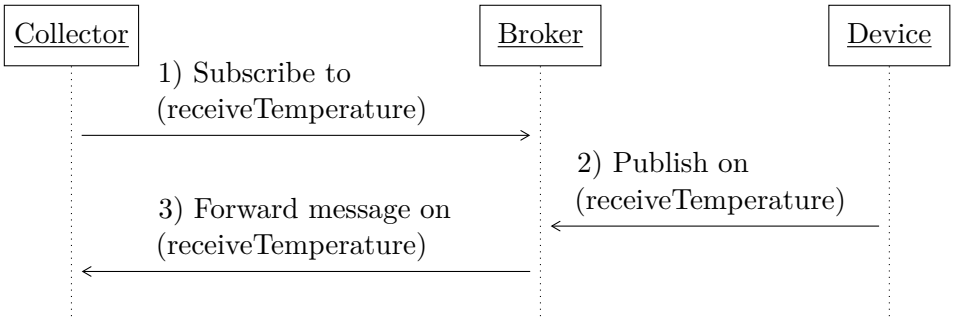


**Figure 2.2:** Representation of the example in listing 2.5.

Indeed, the subscription is performed when the JIoT program is launched for all operations present in MQTT `inputPort`s. This choice is more in line with the expected behaviour of Jolie programs — and of Service-Oriented programs in general. The service stores the messages that belong to operations whose reception statements are not yet enabled, until the actual execution of the reception. If the service would perform the subscription along with the execution of the `OneWay` operation, previous messages could be no more available. In JIoT, the compulsory parameter `.broker` is needed precisely to know the address at which the subscription occurs. The address for the delivery of the actual message is the usual `Location` of the `inputPort`.

Regarding the selection of topics, similar to what done for CoAP resources, in MQTT by default we map JIoT operations to topics. Otherwise, we use the `.osc` parameter `.alias` to detach the coupling between operations and topics. We remark that `.alias` parameters in `inputPort`s have a different behavior in MQTT with respect to HTTP and CoAP. In CoAP the name of the resource extracted from the received message is used to derive the instantiation of the `.alias` template. The interpreter inserts the values resulting from the match among the elements of the payload before storing it in the target variable `data`. Instead, in MQTT, the `.alias` parameter is used to identify the topic for subscription. For example, in listing 2.5, one could add the `Protocol` parameter `.osc.receiveTemperature.alias = "temperature"` to specify that the selected topic for operation `receiveTemperature` is `"temperature"`. Note that, since there is no outgoing data, templates in MQTT `inputPort`s, such as `"temperature"` in the example, are constants (we require all such constants defined within the same `inputPort` to be distinct). Having only constant aliases is not a relevant limitation in the context of IoT, where topics are mostly statically fixed. Addressing this limitation without disrupting the uniformity of the Jolie

programming model is not trivial, ans we leave the discussion concerning the implementation of a new feature as further investigation.

```
1 type TmpType: int { .id: string }
2
3 interface ThermostatInterface {
4     OneWay: setTmp( TmpType )
5 }
6
7 outputPort Broker {
8     Location: "socket://localhost:1883"
9     Protocol: mqtt {
10        .osc.setTmp << {
11            .format = "raw",
12            .QoS = 2, // exactly once QoS
13            .alias = "%!{id}/setTemperature"
14        }
15    }
16    Interfaces: ThermostatInterface
17 }
18
19 main {
20    // ...
21    setTmp@Broker( 24 { .id = "42" } )
22    // ...
23 }
```

**Listing 2.6:** Example of outgoing MQTT `OneWay` communication.

To conclude the mapping of `OneWay` operations in MQTT, we consider here the case of outbound operations, exemplified in listing 2.6.

Outgoing `OneWay` operations cause the publication of the value passed as the parameter of the invocation (line 21) at the Broker. The address of the Broker is defined by the `Location` (line 8) of the `outputPort Broker`. The topic map with the name of the `operation` and the parameter of the invocation, using protocol parameter `.alias` as usual. Being an MQTT publication, we specify the `.QoS` protocol parameter (line 12), which selects the QoS level "Exactly once" for the operation `setTmp`. Similarly to what we have done in CoAP with the `contentFormat` protocol parameter, we define in `.format` the encoding of the message payload, in this case, a "raw" stream of bytes.

*Request-Response Communications in MQTT.* Let us consider the example in listing 2.4 to discuss about `RequestResponse` communications, revising the code in listing 2.5 by replacing the CoAP protocol with MQTT. We omit `OneWay` communications and concentrate on the outbound operation that interact with

a `RequestResponse` operation. Afterwards, we will also discuss the dual inbound `RequestResponse`.

```
1 type TmpType: void { .id: string }
2
3 interface ThermostatInterface {
4     RequestResponse: getTmp( TmpType )( int )
5 }
6
7 outputPort Broker {
8     Location: "socket://localhost:1883"
9     Protocol: mqtt {
10        .osc.getTmp << {
11            .format = "raw",
12            .QoS = 2, // exactly once QoS
13            .alias = "%!{id}/getTemperature",
14            .aliasResponse = "%!{id}/getTempReply"
15        }
16    }
17    Interfaces: ThermostatInterface
18 }
19
20 main {
21    // ...
22    getTmp@Broker( { .id = "42" } )( temp )
23    // ...
24 }
```

**Listing 2.7:** JIoT controller communicating over MQTT.

Syntactically, the main novelty of listing 2.7 with respect to the `outputPort` in listing 2.6 is the addition of `Protocol` parameter `.aliasResponse` at line 14. This parameter specifies the name of the topic where the receiver will publish its response.

For the developers, a `RequestResponse`, when outbound, is composed of an outgoing communication followed by an inbound reply. The outgoing communication is implemented using the approach already seen for `OneWay` communications — i.e. using the `.alias` `Protocol` parameter to identify the topic. Then, one has the issue of relating the outgoing request with its reply. Many standard point-to-point communication technologies, such as HTTP/TCP and the already discussed CoAP/UDP, support request-response communications by defining means to link a given outgoing request to its reply. MQTT, relying on the Publish/Subscribe paradigm, does not provide dedicated means to do such a linking. Thus we specify topics for both the request and the response. To secure the communication, keeping track of the similar topics used between

the client and the server remains a developers responsibility. The programmer can, for instance, send the topic for the response inside the payload of the request message.

We identify the topic for the reply with the `.aliasResponse` `Protocol` parameter. Like for `.alias` parameters, the template of the `.aliasResponse` is instantiated using the content of the message sent in the behaviour. For example, in listing 2.7, we use `.id` at line 22 to obtain `"42/getTemperature"` and `"42/getTempReply"`, respectively the publication and reply topics.

We can now describe the pattern of interactions that we use to implement the outgoing `RequestResponse` communication at line 22 in listing 2.7. As a reference, the pattern of interactions is depicted in the left part of figure 2.3. We will describe the right part later on, after having introduced inbound request-response communications.

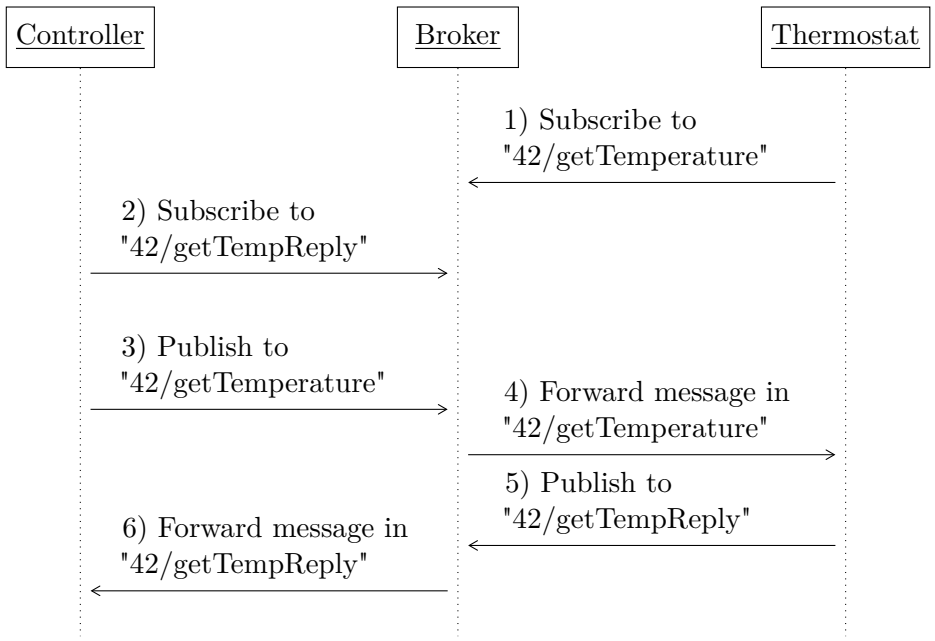


**Figure 2.3:** Interaction in the temperature automation example in MQTT.

First, the controller subscribes to the reply topic `"42/getTempReply"` at the Broker. Then, the controller sends to the Broker the request message on topic `"42/getTemperature"`. The execution of the `RequestResponse` terminates when the Broker forwards the reply received on topic `"42/getTempReply"` to the controller.

Differently from inbound `OneWay` communications, here we do not subscribe

to the reply topic at bootstrapping time. Indeed, it would be useless since no relevant message can arrive on this topic before the controller sends its message to the Broker, and anticipating the subscription would complicate the usage of runtime information in templates.

To exemplify inbound `RequestResponse` communications, we assume that the thermostat in our example runs the JIoT interpreter. We report its code in listing 2.8.

```
1  type TmpType: int { .id: string }
2
3  interface ThermostatInterface {
4      RequestResponse: getTmp( TmpType )( TmpType )
5  }
6
7  inputPort Thermostat {
8      Location: "socket://localhost:9000"
9      Protocol: mqtt {
10         .broker = "socket://localhost:1883";
11         .osc.getTmp << {
12             .format = "raw",
13             .alias = "42/getTemperature",
14             .aliasResponse = "42/getTempReply"
15         }
16     }
17     Interfaces: ThermostatInterface
18 }
19
20 main {
21     // receive and store the temperature
22     getTmp( temp )( temp ){
23         // update and send back the temp content
24     }
25 }
```

**Listing 2.8:** JIoT thermostat communicating over MQTT.

At line 13 in listing 2.8, the `.alias` parameter `"42/getTemperature"` must be defined statically, as required for `inputPort`s. When the thermostat controller starts, it subscribes to topic `"42/getTemperature"`. When a message on this topic arrives, the interpreter passes the payload (empty in this case) to the behaviour. The body of the `RequestResponse` (lines 22–25) is executed to compute the return value.

Finally, the interpreter publishes the content of the variable retrieved on the reply topic `"42/getTempReply"`, as specified by `osc` parameter `.aliasResponse`. While in this example the parameter `.aliasResponse` is statically defined,

our implementation supports the definition of dynamic `.aliasResponse`s as in `outputPort`s — e.g. as seen in listing 2.5.

We summarize the exchange between the controller and the thermostat (figure 2.3) in the following:

1. when the thermostat is started, it subscribes to topic `"42/getTemperature"` at the Broker;
2. when the outgoing `RequestResponse` is executed, the controller subscribes to topic `"42/getTempReply"` at the Broker;
3. the controller publishes the request message to topic `"42/getTemperature"`;
4. the Broker forwards the message in topic `"42/getTemperature"` to the thermostat;
5. the thermostat publishes the response, after computing the value, on the topic `"42/getTempReply"`;
6. the Broker forwards the message on topic `"42/getTempReply"` to the controller.

We remark that `RequestResponse` operations in Jolie are meant to be end-to-end communications. To ensure this in a publish/subscribe setting while using the approach above, one has to assume that other participants could subscribe — and thus access pertinent information – to the selected topics, which essentially act as namespaces.

### 2.3.4 Implementation

To illustrate the structure of our implementation, we discuss how media and protocols are separated from the Jolie interpreter and available as independent libraries. In the remainder, we proceed describing the highlights of the implementation of UDP and CoAP and of MQTT.

#### Programming a Jolie Extension

In Jolie, the implementations of the supported application and transport protocols are independent. Independency of application and transport protocols enables the composition of any of the first with any of the latter. Concretely, the Jolie language is written in Java and provides proper abstract classes that represent application and transport protocols. Each protocol results from an implementation of the corresponding abstract classes. Each implementation

comes as a separate library, that the interpreter loads, just in the case that the service use it. The compositional nature of the interpreter expedites the integration of new protocols in the language. Concerning the implementation strategy, a second aspect that is important to take into account is that we developed the JIoT communication core, strongly relying on the Netty framework [73]. Indeed, we believe that taking advantage of the Netty provided solutions has been crucial to meet the requirements of system communication, high concurrency, and real-time interaction proper of the IoT context. Netty is based on Non-blocking I/O (*NIO*), which provides developers with asynchronous, event driven abstractions. The adoption of the Netty technology has dramatically optimized the handling of stability and scalability of both the interpreter communication core and the protocols related libraries.

To better illustrate this structure, we report in figure 2.4 a conceptual representation of the call flow that originates from the execution logic of the language and interacts with the external libraries present in a given installation. The flow starts from the Execution Engine, which interprets Jolie commands, and which is the originator of the communication flows, represented by arrow ⓪ . From there, the call reaches the Communication Core, which implements the generic logic of channel creation, in turn relying on the pairing of a medium and a protocol. In the interpreter, this division is generalized with abstract factories for media and protocols. At runtime, the Communication Core proceeds (arrows ① ) to load the medium factory requested in the call from the Execution Engine — in the figure we assume this is Socket — and, from that, it obtains an implementation of the actual logic of TCP/IP channels, split between a channel class, to handle outbound communications, and a listener class, for inbound communications. Finally, the Communication Core associates (arrows ② ) a protocol to the obtained medium. The flow is similar to that of media: the Communication Core loads the protocol factory requested in the call from the Execution Engine — in the figure we assume this is HTTP — and, from that, it obtains an object that implements the logic of the HTTP protocol.

*Implementation of CoAP/UDP in Jolie.* Since by specification the CoAP protocol relies on the UDP medium protocol, in order to integrate CoAP in Jolie we also had to integrate the UDP medium. As discussed in subsection 2.3.3, this entailed the creation of two new libraries for the Jolie interpreter: a medium library for UDP and a protocol library for CoAP.

We remark that since UDP and CoAP are independent libraries, our implementation of UDP can also be used to support other protocols relying on UDP,

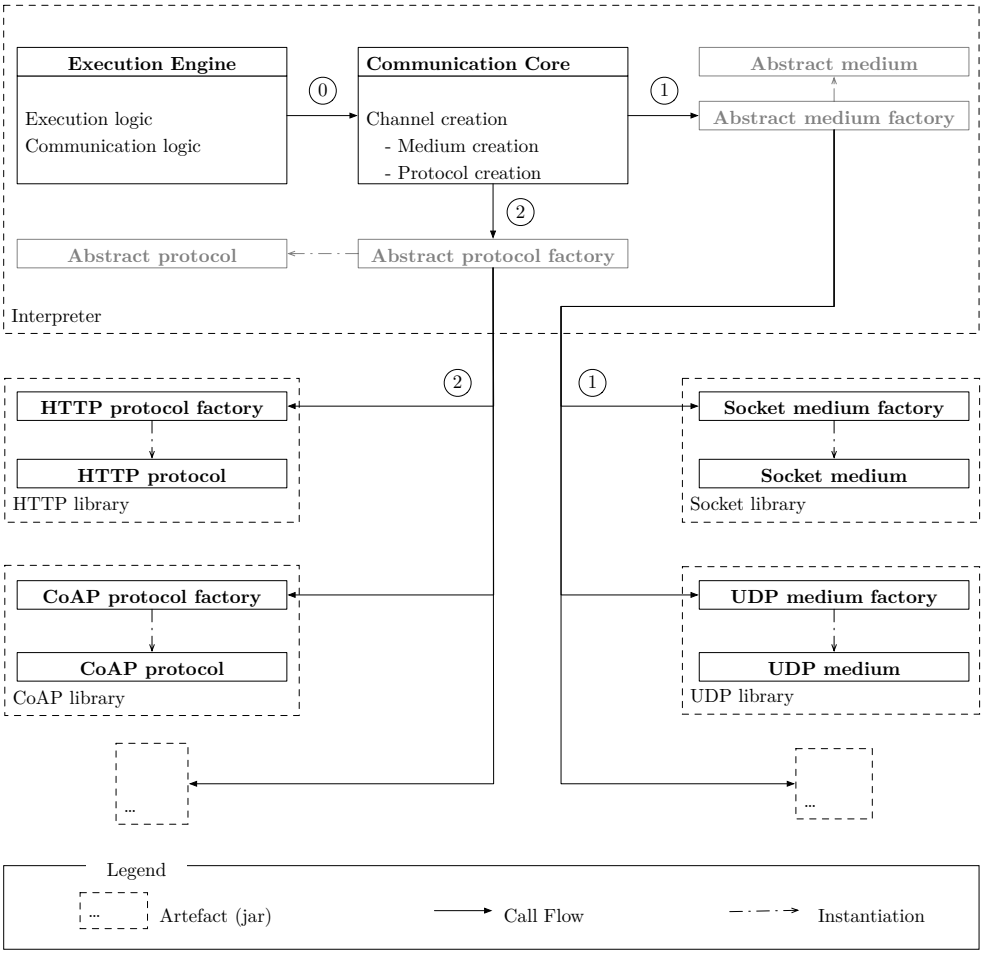


**Figure 2.4:** Conceptual representation of the call flow among the Jolie interpreter, augmented with JIoT protocols, and its communication libraries.

such as MQTT-SN [55]. The implementation of UDP consists of a listener and a channel class, both based on the Netty framework [74]. Since the structure expected by Jolie and the one provided by Netty are similar, the integration of UDP is smooth. One interesting point is that the interpreter captures exceptions raised by Netty and transforms them into Jolie exceptions. The application protocol receives a notification for these exceptions, and, eventually, manage them or raise them at the level of the Jolie program behaviour.

The implementation of the CoAP library consists of a class taking care of encoding and decoding the message abstraction of Jolie, namely the Commu-

nication Message, into a CoAP formatted one. A second class, handling the encoding and decoding of a CoAP message into a buffer of bytes, is based on the work done in nCoAP [95], an open-source project providing a CoAP implementation for Java, based itself on Netty. CoAP supports request-response communications and, in particular, CoAP messages include the following fields [115]. (i) The *Uri-Host*, *Uri-Port*, *Uri-Path*, and *Uri-Query* options specify the target resource of a request — i.e. the address where the sender expects the reply. (ii) The *Message ID*, to detect message duplication and to match acknowledgement or reset messages to messages of type Confirmable or Non-confirmable. (iii) The *Token* — to match a response with a request. Hence, the implementation of the Jolie `RequestResponse` communications abstraction in CoAP is sound — i.e. valid, consistent — also with a transport protocol which is not connection-oriented, such as UDP. The not connected-oriented nature of UDP would be a problem for protocols that do not provide this facility, such as HTTP, which is indeed not commonly used over UDP.

Notably, Jolie comes with a formal semantics (in terms of a process calculus) [50], which enables to reason on the behaviour of Jolie programs rigorously. Formalism has been instrumental in the evolution of the language — e.g. to specify and prove properties on the fault handling mechanisms of the language [49] or to correctly implement sessions [87] based on correlation mechanisms [99]. The semantics in [50] only considers reliable communications and needs to be also extended to cover the unreliable case.

*Implementation of MQTT in Jolie.* By specification, MQTT relies on the TCP/IP protocol, already implemented in Jolie. Support TCP/IP means that, theoretically, the implementation of MQTT would have only entailed the creation of a dedicated MQTT protocol library. However, Jolie assumes an end-to-end communication pattern where the caller initiates the creation of a communication channel with a server, which in turn expects such inbound requests. For this reason, given a certain medium, `inputPort`s and `outputPort`s use a medium-specific implementation of, respectively, a listener class and a channel class.

This pattern, separating listeners from channels, does not apply to publish/subscribe protocols, where both the subscriber and the publisher need to establish a connection with the broker. In our implementation, we mediated between the two approaches with a Publish-Subscribe medium, which is a wrapper implementing the logic of Publish-Subscribe message handling on any other point-to-point medium available (TCP socket in the case of MQTT) to the interpreter. Although we strove to separate the concerns between the

Jolie interpreter and this new Public-Subscribe channel, we had to introduce a minimal update into the Jolie Communication Core so that it could choose between the standard end-to-end media and the new wrapper.

The MQTT protocol class both encodes and decodes messages and implements the QoS policies of the MQTT standard. Concretely, as for CoAP, we based the implementation of MQTT on Netty [74].

The main difficulty in the implementation of the protocol is the definition of the message patterns needed to implement `OneWay` and `RequestResponse` communications. Beyond being invoked at (operation) execution time, to perform port initialization properly, the interpreter needs to invoke the MQTT class also at bootstrapping time.

## 2.4 Case Study

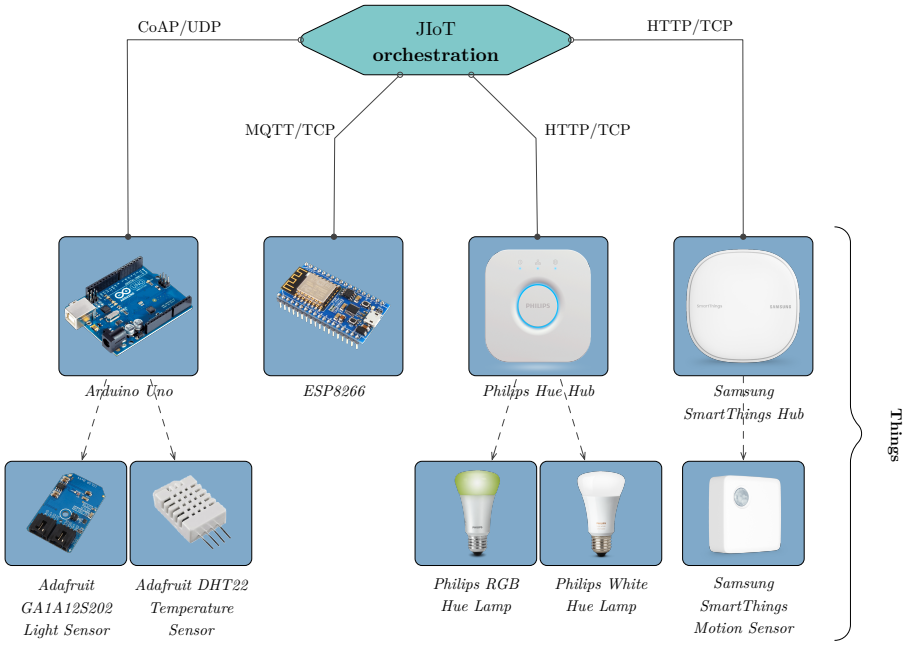


**Figure 2.5:** Conceptual overview of the home automation case study.

In this section, we detail the programming of a home automation case study with JIoT where we considered:

- local, cross-layer communication among things and mid-tier controllers (Edge devices and fog nodes);

- remote, cross-layer interactions among Cloud nodes and mid-tier controllers.

We remark that the techniques presented in this case study are not specific to home automation. Any heterogeneous IoT scenario, where different technology stacks show a high level of interaction, can benefit, in principle, with the proposed style. The case study is peculiar as a new Thing can be included in the system at runtime. We released the source code of the system under the GNU GPL v.3.0 license, making it available at [39]. We report in figure 2.5 a schematic overview of the case study, where
cloud nodes and mid-tier controllers (represented by the element labeled "JIoT orchestration" in figure 2.5) are programmed in JIoT and orchestrate the behavior of a number of heterogeneous Things, whose low-level programming is omitted here:

- *Philips Hue Hub* — a hub to control the Philips Hue smart home devices;
- Two *Philips Hue Lamps* — connected to the hub above;
- *Samsung SmartThings Hub* — a hub to control devices following the SmartThings specification [117];
- *Samsung SmartThings Motion Sensor* — connected to the hub above and used as a presence sensor;
- *Arduino Uno* — a general-purpose microcontroller;
- *Adafruit GA1A12S202 Analog Light Sensor* — connected to the Arduino above;
- *Adafruit DHT22 Temperature Sensor* — also connected to the Arduino above;
- *ESP8266* — a Wi-Fi enabled microcontroller to manage a pre-existing thermostat.

The case study combines commercial solutions — e.g. the Philips Hue Hub and the Hue Lamps system where the Hub controls the Lamps— with custom ones — spanning from sensors directly connected to a board, as it happens for the Adafruit DHT22 temperature sensor, to solutions that integrate a pre-existing hardware, like the ESP8266 that manages a pre-existing thermostat. As illustrated in figure 2.5, this heterogeneity of devices provides for a whole scenario where we need JIoT programs that use a different application and

transport protocols. In particular, in the depicted scenario, Philips and Samsung Hubs communicate with the orchestrator over HTTP/TCP, the Arduino over MQTT/TCP, and the ESP8266 over CoAP/UDP.

In the case study we build a simple logic providing two functionalities: lighting and temperature system control. The lighting system turns on the lights when the motion sensor detects someone at home and the outdoor luminosity is below some threshold. The temperature control checks the temperature and turns on the heating system when the temperature is below some threshold. The latter considered threshold has different values depending on whether someone is at home or not.

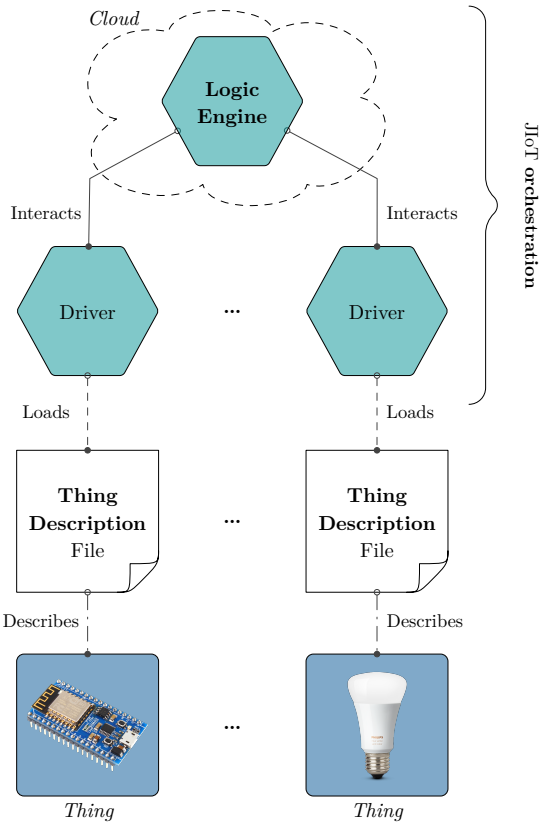


**Figure 2.6:** Scheme of the orchestration in the case study.

### 2.4.1 Structure of the orchestration

We now describe the structure of the orchestration in the case study, as illustrated in figure 2.6. The orchestration is composed of multiple JIoT programs. From top to bottom of figure 2.6, the `LogicEngine` contains the general logic of system control — i.e. the one that collects the data from sensors and coordinates the execution of the actuators in the system. Since the `LogicEngine`, interacts with a multitude of mid-tier devices, its natural deployment would be in the Cloud, where it is possible to automatically scale the service — both vertically and horizontally — according to the number of managed devices and the computation load. At the mid-tier level we have JIoT `Drivers`. Each `Driver` interacts with a specific Thing and it is deployed in a mid-tier machine in the proximity of the controlled Thing.

### 2.4.2 Thing Descriptions

In the case study, the `Drivers` are statically configured to manage a single fixed device using a JSON-LD 1.1 (JSON Linkage Data) — a lightweight Linked Data format for configuration files [21]. The choice of JSON-LD is not mandatory, but it has the benefit of following the official W3C Web of Things [131] definition of Thing Description (TD). By using TD in the JSON-LD format, we aim at making our `Drivers` already compliant with other WoT frameworks, simplifying future integrations with other WoT systems.

While discussing the full structure of TD is out of the scope of this chapter, we chapter in listings 2.9 and 2.10 case studys of TDs used in our case study. In listing 2.9 we report the TD for the DHT22 temperature sensor. For each device the JSON-LD file specifies whether it is a sensor or an actuator (key `"type"`) and provides a textual description (key `"description"`) and its name (key `"name"`). Each TD provides a list of properties (key `"properties"`) that can be read. Each property is described by the property identifier, `temperature` in our case study. The property identifier has various sub-elements describing it. In our case study we use just key `label` to describe the unit of measure.

```
{
  "type": "sensor",
  "description": "Thing using JSON-LD 1.1 serialization",
  "name": "Adafruit DHT22 Temperature Sensor",
  "properties": [
    {
      "temperature": { "label": "Celsius" }
    }
  ]
```

```
10 }
```

**Listing 2.9:** Adafruit DHT22 TD.

JSON-LD configuration files for MQTT and HTTP devices are similar to the ones depicted in listing 2.9. Also, configuration files for sensors and actuators are similar between each other. As an case study, we reported in listing 2.10 the configuration file for Philips Hue Lamps.

```
1 {
2   "type": "actuator",
3   "description": "Thing using JSON-LD 1.1 serialization",
4   "name": "Philips Hue Lamp",
5   "actions": {
6     "toggleLight": {
7       "description": "Turn on or off the lamp."
8     }
9   }
10 }
```

**Listing 2.10:** Philips Hue Lamp TD.

The main differences with respect to the previous TD (listing 2.9) are (i) the `type` is now `"actuator"`; (ii) the key `actions` replaces the key `properties`; (iii) the key `description` is used also to describe the single action.

In principle, a TD can describe multiple properties belonging to a group of one or more Things controlled by the same `Driver`. For simplicity, here we have one TD for each Thing and, correspondingly, one `Driver` that controls one Thing. We also assume that each sensor provides one property.

### 2.4.3 System Deployment

Deployment-wise, JIoT provides a vast choice regarding the technology stack to use between the `LogicEngine` and the `Drivers`. Moreover, since we developed both programs in JIoT, it is easy to change their deployment, switching to the technology stack that best suits a given scenario — e.g. HTTP, to exploit caching, or binary formats like SODEP [89], to limit bandwidth usage. Here, we choose to use the HTTP/TCP stack to make our system compatible with the majority of existing third-party solutions [61]. However, different technology stacks fit different purposes. The benefit of JIoT is that programmers can re-use the same software components adapting their deployment to the desired communication stacks. For case study, if our goal was to be natively compatible with other JavaScript IoT frameworks, we could have used the JSON-RPC

binary protocol; if we wanted to deploy our system as part of a Service-Oriented Architecture [32], we could have used the SOAP protocol.

While JIoT-to-JIoT deployment is flexible, the technology supported by the Things define the deployment towards them. Concretely, in our case study each `Driver` communicates with its Thing using (one of) the protocol(s) supported by the latter.

```
interface driverInterface {
    RequestResponse: engineRequest
}

outputPort Driver {
    Protocol: http
    Interfaces: driverInterface
}

define getTemperature {
    sum = 0 ;
    n = 0 ;
    for ( device in devices ) {
        if( device.type == "sensor" && is_defined( device.
            properties.temperature ) ) {
            Driver.location = device.driverLocation ;
            request.operationName = "getTemperature" ;
            engineRequest@Driver( request )( response ) ;
            sum = sum + response.deviceResponse ;
            n++
        }
    } ;
    if( n!=0 ) {
        temperature = sum / n
    }
}
```

**Listing 2.11:** `LogicEngine` `Driver` `outputPort` and `getTemperature` procedure.

### Components Behavior

When started, a `Driver` service loads the TD configuration file of its Thing. Then, it registers itself to the `LogicEngine`. In the registration, it sends the information retrieved from the TD, enriched with two additional pieces of information: the address where it is possible to contact the Thing — i.e. the `Driver` location — and the identifier of the user to which the Thing belongs. Once registered, the `Driver` acts as a forwarder between the `LogicEngine` and the Thing.

The `LogicEngine` runs, hypothetically, on the Cloud and manages several sensors and actuators. More precisely, the `LogicEngine` has one running session for each user (distinguished according to the user identifier), managing all her sensors and actuators. Each session is associated with an array of devices that can be scanned to find the location of `devices` with specific properties and interact with them — e.g. at lines 10–25 of listing 2.11 the procedure `getTemperature` of the `LogicEngine`, computing the average temperature recorded by the sensors of one user.

Briefly, procedure `getTemperature`, (i) scans the `devices` structure (line 13) containing all registered `Drivers`; (ii) selects those whose `type` is `"sensor"` and have a property (under the sub-structure `properties`) named exactly `temperature`. Note how Jolie tree-shaped variables ease the exploration of structured data; in this case the one sent by the `Drivers` at registration time (and read from their associated JSON-LD file); (iii) it dynamically sets (line 15) the location of `outputPort Driver` (lines 5–8) to contact the selected `Driver`; (iv) it sets the request operation to `getTemperature` (line 16); (v) it retrieves the temperature sensed by the Thing controlled by the selected `Driver`, invoking it through operation `engineRequest`; (vi) it aggregates the sensed temperature in variable `sum` and keeps track of the number of requests in variable `n` (lines 18–19); and (vii) finally, it computes the mean `temperature` (lines 22–24).

```
27 define setTemperature {
28     for ( device in devices ) {
29         if( device.type == "actuator" && is_defined( device.
               properties.temperature ) ) {
30             Driver.location = device.location ;
31             request.operationName = "setTemperature" ;
32             request.deviceRequest = comfortTemperature ;
33             engineRequest@Driver( request )( response )
34         }
35     }
36 }
```

**Listing 2.12:** `LogicEngine` setTemperature procedure in the orchestrator.

The procedures that calculate the mean of the sensed external luminosity and the one to check the presence of people at home are similar to the one in listing 2.11, except that the searched properties are `light` in the first case, and `motion` in the second.

We report in listing 2.12 one of the procedures managing the actuators, specifically the one used to set the temperature. The main difference with respect to the logic in listing 2.11 is that procedure `setTemperature`, (i) selects the `devices` whose `type` is `"actuator"` (line 29); (ii) sets the request operation to

"setTemperature" (line 31) and passes the value in variable comfortTemperature as parameter (lines 32–33).

Note that the operation called on the Driver is engineRequest both in listing 2.11 and listing 2.12. This support the extension of the LogicEngine with new procedure definitions that implement a given goal without requiring to change the interface between the LogicEngine and the Drivers. In turn, a request with the same operationName — e.g. "setTemperature" — triggers different behaviors in different Drivers, as each implements the specific interaction with its associated Thing.

Any parties would register their devices one-by-one using a suitable security scheme — e.g. basic, token, API key. One can specify the security scheme in the JSON-LD TD. Every time a device for a new user is registered, the service spawns a new session of the LogicEngine managing the devices owned by the user. Thus, new users and new devices can enter the system at any time.

### Cloud Deployment

We conclude this section with the description of the deployment of the LogicEngine, naturally devoted to being in the Cloud. We chose to take advantage of the state-of-the-art deployment tool Docker [60], principally, to deal with the packaging of the service along with all of its dependencies. Via proper configuration, Docker Swarm can automatically deploy the resulting container (of almost 105MB) to the Amazon Web Service (AWS) EC2 instances two managed via Docker Swarm [90]. Since the LogicEngine microservice runs on the worker node, the manager node can balance the load of requests to the worker. We reported in listing 2.13 the content of the Dockerfile — the entry point for Docker image configuration — used to deploy the LogicEngine image.

We conclude this section with the description of the deployment of the LogicEngine, naturally devoted to being in the Cloud. We chose to take advantage of the state-of-the-art deployment tool Docker [79], principally, to deal with the packaging of the service along with all of its dependencies. Via proper configuration, Docker Swarm can automatically deploy the resulting container (of almost 105MB) to the Amazon Web Service (AWS) EC2 instances[2] managed via *Docker Swarm* [118]. Since the LogicEngine microservice runs on the worker node, the manager node can balance the load of requests to the worker. We reported in listing 2.13 the content of the *Dockerfile* — the entry point for Docker image configuration — used to deploy the LogicEngine image.

```
1 FROM openjdk:8-jdk-alpine3.9
```

[2] A mini-cluster composed of two free-tier instances *tiny* flavoured.

```
2
3 RUN java -jar jiot.jar -jh /usr/lib/jolie/ -jl /usr/bin/
4 ENV JOLIE_HOME /usr/local/lib/jolie
5
6 ADD logic_engine.ol /home/.
7 WORKDIR /home
8 RUN jolie logic_engine.ol
```

**Listing 2.13:** The Dockerfile used to deploy the `LogicEngine`.

At line 1 we declared the starting image for the container, which was tested using the lightweight *Linux Alpine* distribution (version 3.9) for *amd64* architectures, that comes with the OpenJDK Runtime Environment (version 1.8.0 update 212) pre-installed. At lines, 3–4 we install the JIoT forked interpreter — copied in the container at runtime — and we set the environmental variable `JOLIE_HOME` to point to the location of the installed interpreter. At lines 6–7 we add the source code of the `LogicEngine` in the home directory of the image. Finally, at line 8, we bootstrap the execution of the `LogicEngine`.

## 2.5 Discussion

IoT advocates for multi-layered platforms, going from Cloud nodes to Edge devices, where each layer adopts its communication standards. While this freedom is optimal for in-layer interaction, it puzzles cross-layer integration due to incompatibilities among protocols. Enforcing a unique communication stack within the same IoT platform does not provide a feasible solution, as it leads to the "IoT islands" phenomenon, where disparate platforms hardly interact with each other.

In this chapter, we proposed a language-based approach for the integration of disparate IoT platforms. We built our treatment on the Jolie microservice-oriented programming language. This first result is an initial step towards a more comprehensive solution for IoT ecosystem integration and management. Concretely, we developed the JIoT interpreter, a Jolie fork, and implemented the support for two of the most widely used IoT protocols. The inclusion enables Jolie programmers to interact with the majority of present IoT devices. Summarizing our results: (i) we included in Jolie the CoAP application protocol, also extending the Jolie language to support the UDP transport protocol, (ii) we added the support for the MQTT protocol and, in doing so, (iii) , we tackled the challenging problem of mapping the renowned pattern of request-responses (typical of HTTP and other widely used protocols) into the publish/subscribe message pattern of MQTT. The mapping abstracts from

peculiarities of MQTT and applies to others publish/subscribe protocols.

A specific language, with proper abstractions, offers a single linguistic domain to integrate disparate low-level IoT devices and intermediate nodes (collectors, aggregators, gateways) seamlessly. Moreover, Jolie is already successfully used for building Cloud-based, microservice solutions [38, 78]. The Jolie microservices-oriented approach makes the language useful also for assembling advanced architectures for IoT — e.g. to handle real-time streaming and processing of data from many devices. We claim that, while we propose a dedicated language, the majority of the comparable approaches provide API specifications. Although related to our aim, the proposals in the literature tackle the problem of IoT integration from a framework perspective: they provide chains of tools, each addressing a specific level of the integration stack. Differently, we extend a language tailored explicitly for system integration and advanced flow manipulation, Jolie, to support integration of IoT devices. The benefit, here, is that, while solutions based on frameworks require dedicated proficiencies on each of the included tools, Jolie programmers can directly work at any level of the IoT stack, without the need to acquire specific knowledge on the tools in a given framework. Lastly, in section 2.2, we pointed to SensorML as the closest to our approach but, while some traits of SensorML are common to our proposal, the scopes of the two languages sensibly differ. Indeed, while Jolie is a high-level language for generic programming architectures, SensorML only models the lowest layer of the IoT system — implementing Thing discovery, and processing of sensor observations.

Currently, one of the significant limitations in the proposed approach is the lack of a light-weight version of the JIoT language, to be used on low-power IoT devices. Indeed, in this chapter, we assumed that these devices are programmed with low-level languages since they can support only a very constrained execution environment. Letting developers implement all the components of an IoT network in the same language would ease not only its implementation but also testability, deployment, and maintenance. However, achieving such a result would require a very challenging engineering endeavour. Despite of this effort, the industrial interest around IoT led to the spread of board prototypes (Intel Galileo, Raspberry Pi) and consequently, attracted big software technology providers, such as Oracle, that started to support the development of Java software on these low-mid-tier boards with a customized version of the JDK, namely *Java Micro Edition* (ME). This technology could provide a valid compromise between the need of solutions such as JIoT to program lower-level boards, that would be possible since JIoT interpreter runs over JDK, and the high engineering effort to rebuild the solution from scratch.

In the future, it would be interesting to investigate the integration of more

IoT transport and application protocols [36], in order to extend the usability of the language in the IoT setting. Another exciting direction is studying how JIoT can support the testing of IoT technologies — e.g. to test how different protocol stacks perform over a given IoT topology. Thanks to the simplicity of changing the combination of the used protocols, experimenters can quickly test many configurations, also enjoying a more reliable platform to compare them. Indeed, usually even changing one of the protocols in the configured stack would require an almost complete rewrite of the logic of network components. Contrarily, this change requires an update of the deployment part of programs, leaving the logic unaffected. Furthermore, one could even implement such an update programmatically, making the practice of repeated experimenting on IoT networks more accessible and more standardized.

# Chapter 3

# Data Handling in IoT Systems

IoT systems present, along with great transport and application protocols heterogeneity, an incredible variety of possible data representation formats. A common practice in programming IoT applications is to encode data in tree-shaped formats, such as XML and JSON, that usually need to be processed in real-time and, in some restrictive scenarios, data must not persist in the system above a certain threshold. While developers prefer to use a query language to express complex data manipulations, typical execution engines are external from the main application memory, increasing communication overhead and application response time.

The aim of this work is to provide a solution for effective data handling in IoT applications, addressing data formats interoperability and performance stability, following a linguistic approach to microservice-oriented computing.

Using a query framework integrated with the application language, represents a better option for ephemeral data handling. We build on our solution, described in chapter 2, extending the syntax with query operators to manipulate tree-shaped, document-oriented data structures. To accomplish a correct implementation, we first formalise the TQuery framework, that is an instantiation of *MQuery*, a sound variant of the *MongoDB Aggregation Framework*. Then, we integrate the framework operators into the JIoT interpreter.

TQuery benefit from existing development support tools — the Jolie syntax and type checker — and execute queries within the application memory. Furthermore, since JIoT natively supports tree data structures and automatic management of heterogeneous encodings, it provides a uniform way to use the framework operators on any data format supported by the language. To validate our approach and describe the new features of the tool, we extend the case study presented in section 2.4 with an algorithm requesting real-time handling and ephemerality. Using the very same example, we evaluate the performance of our implementation, showing interesting low-response times when compared with state-of-the-art solutions.

In conclusion, in this work we move a step towards a comprehensive solution

that addresses both protocols interoperability and data handling challenges for IoT applications programming.

## 3.1 Introduction

IoT applications, as all modern software systems, need to address two basic requirements concerning data management. The first requirement is *velocity*, intended here as the need for reliable performance, while the second is *variety*, that relates to the concept of interoperability between different data manipulation technologies [76]. Velocity concerns managing high throughput and real-time processing of data [122]. Variety focuses on the data representation aspects and deals with the problem of formats heterogeneity, that non-trivially complicates some usual task related to data handling — e.g. aggregation, query, and storage.

Recently, in addition to velocity and variety, systems pervasiveness highlighted the importance of *ephemerality*, primarily due to the introduction of the latest international regulations [114]. It is the case of IoT application, that involves the manipulation of data originating from, for instance, eHealth scenarios [24] or, more in general, edge computing contexts [116]. Due to regulations restrictions, developers have to process data in real-time without relying on persistency.

To disambiguate the different interpretations concerning "ephemerality handling" given in the IoT literature, we provide here a functional description[1]. We denote the term ephemerality, in the context of data handling in IoT applications, as the transient manipulation of data structures, whose life cycle never reaches the writing on a disk, nor it involves sending the structure to a process not controlled by the application that requested the data.

As previously stated, interest in ephemeral handling increased with the rise of new applicative scenarios, where data persistence represents an issue more than a profit. In the everyday practice, IoT developers deal with resource-constrained and heterogeneous systems and, with external regulation requirements that limit their decisional power on data management design, worsening the development process effectiveness. Recently, the General Data Protection Regulation (GDPR) [126] imposed restrictions about data provision and persistence in IT systems. These restrictions often apply to pervasive scenarios — e.g. smart healthcare [6] and privacy data protection [90] — challenging developers to implement ephemeral data manipulation.

[1] Definitions of ephemeral data structures, tend to consider "ephemeral" just as the negation of "persistent".

To correctly implement data manipulation in IoT context, with a general-purpose language, in a way that the algorithm respects the intended protocol, often reveals to be a time consuming and error-prone practice [106, 71]. Developers usually take advantage of proper query languages, paired with a (sub-system) engine that executes the queries [18]. To the best of our knowledge, developers chose where to execute the engine between two approaches:

1. **DBMS approach.** — Using a DataBase Management System (DBMS) that executes queries outside of the application memory [76].

2. **In-memory approach.** — Including a library that executes queries within the application memory.

In practice, the DBMS approach is the most common. Since the early days of Web [21], developers integrated back-end programming languages with relational "Structured Query Language"s (SQL) for data manipulation and persistence [134]. In modern software development, developers strongly rely on the pattern of application language, joined with SQL DBMS. The latest trend[2] see Relational SQL DBMSs (RDBMS) share the stage with Non-relational NoSQL [76] DBMSs. MongoDB [85] and Apache CouchDB [4] are representative examples of document-oriented DBMSs. Document-oriented databases natively support tree-like nested data structures, such as XML and JSON. JSON data format is extensively supported by the most relevant protocols in the web service and IoT contexts — e.g. HTTP, SOAP, CoAP, and MQTT.

The management of tree-shaped data formats avoids error-prone encoding and decoding procedures with table-based structures, as it usually happens with RDBMSs. However, when considering ephemerality in the IoT context, the challenges related to the DBMS approach (1) overcome its benefits, even if we consider the NoSQL variant. In the following list, we summarized those challenges, which we use as the background knowledge to describe our solution.

(I) **Dependency**. An external DBMS is an additional standalone component that needs to be installed, deployed, and maintained. To interact with the DBMS, the developer needs to import in the application of specific drivers (libraries, RESTful outlets). As with any software dependency, this exposes the applications to challenges of version incompatibility [58].

(II) **Security**. The companion DBMS is subject to weak security configurations [12] and query injections, increasing the attack surface of the application.

[2]Up to 2019.

(III) **Consistency**. Queries to the external DBMS are typically black-box entities — e.g. encoded as plain strings — making them opaque to analysis tools available for the application language — e.g. type checkers [18].

(IV) **Performance**. Integration bottlenecks and overheads degrade the system performance. Bottlenecks derive from resource constraints and slow application-DB interactions — e.g. typical database connection pools [128] represent a potential bottleneck in the context of high data-throughput. Also, data must be inserted in the database and eventually deleted to ensure ephemerality. Overheads also come in the form of data format conversions (see item (V)).

(V) **Heterogeneity**. The DBMS typically requires a specific data format for communication, forcing the programmer to develop ad-hoc data transformations to encode and decode data in transit — i.e. to insert incoming data and returning or forwarding the result of queries. We claim that implementing these procedures (marshalling and un-marshalling) is cumbersome and error-prone.

On the other side, IoT developers explored less the in-memory approach (2) that is, using a query engine running within the application memory. The lack of interest in in-memory data management is, yet, interpretable as a historical bond between query languages and persistent data storage[3].

To conclude, we chose to address all the challenges listed above, leveraging in-memory approach (2), and thus adopting a language-based solution. Specifically, the proposed methodology addresses challenges (I) and (II) by design. It aims at sensibly reduce the issue of challenge (III), since developers can implement both queries and data in the application language. It tackles by design issue related with challenge (IV), since, in our setting, there are less resource-dependent bottlenecks and no overhead due to data insertions or deletions, since the data disappears from the system as the handling process terminates. It also addresses by design data heterogeneity (V).

Examples of in-memory approaches are LINQ [77, 18] and CQEngine [97]. While LINQ and CQEngine grant good performance — addressing the performance challenge — interoperability could still be a problem. Those proposals either assume an SQL or rely on a table-like format, which entails continu-

[3]In the writer opinion, the spread of in-memory approaches is, in a way, prevented, by the high abundance of relatively static, monolithic, and highly-dependable applications still "out there in the IT jungle".

ous, error-prone conversions between their underlying data model and the heterogeneous formats of the incoming and outgoing data.

In conclusion, our solution proposes the adoption of JIoT, a specific programming language tailored for the IoT context, that already supports standard document-oriented formats, and thus it can effectively address interoperability issues and relieve developers from the burden of variety.

## 3.2 Related Work

We differentiate among the related work with the same criterion used in section 3.1. When it comes to select the data handling strategy, developers can either choose a DBMS approach or an in-memory one.

Concerning the first, we review only those works that target document-based data structures, as we do, but, in general, these NoSQL systems, can either target documents, key/value stores, and graphs. As already stated, the DBMS approach closest to ours is certainly the MongoDB Aggregation Framework [57], that we extensively describe in out treatment. Similarly to MongoDB approach is the *CouchDB* [4] query language — which manipulate JSON documents through a JavaScript interface — in an external engine.

Among the others comparable DBMS approaches, we cite *ArangoDB* [5], *Google Big Table* [17], and *Apache HBase* [43]. ArangoDB is a native multi-model engine for nested structures that comes with its own query language (the ArangoDB Query Language). Google Big Table and Apache HBase are external engines explicitly tailored for in big data scenarios, employing distributed computing to address scalability issues — they usually pair with clustered operative systems such as Apache Hadoop [35].

Aside from the external engines solutions, we considered in our review six works that, following the in-memory approach, are directly comparable with ours. In general, these work aim to integrate in the application language the data manipulation abstractions and enable data querying within the application memory. The six frameworks are: the *Redis* [111] store system, the *Object-relation Mapping* (ORM) [37], the *Opaleye* [31] *Haskell* library, *LevelDB* [56], *LINQ* [77], and *CQEngine* [97].

Redis is an in-memory store system that supports string, hashes, lists, and sets. The Object-Relation Mapping (ORM) framework, actually rely on DBMS, as they map objects used in the application to entities in the DBMS, to provide persistence. Similarly to ORM, the Opaleye library provides a *Domain Specific Language* (DSL) that generates *PostgreSQL* code — a famous RDBMS. LevelDB, inspired by Big Table, provides both on-disk and in-memory

library for data manipulation in C++, Python, and Javascript. As we cited in section 3.1, *Microsoft* developed the LINQ solution, which provides query operators targeting both SQL tables and XML nested structures using *.NET* query operators. Similarly, CQEngine provides a library for querying Java collections with SQL-like operators.

## 3.3 Approach

Inspired by the in-memory approaches devised in section 3.1, we implemented a framework for ephemeral data handling in microservices. The purpose of the implementation is practical, since we need to model data manipulation and manage data formats interoperability in IoT systems.

Our framework includes a query language and an execution engine, that can integrate document-oriented queries into the JIoT chapter 2 interpreter — that inherits from the Jolie programming language [89, 59]. The interpreter for the JIoT programming language [39] and the implementation of our framework [44] are open-source projects.

Notably, JIoT comes with a runtime environment that automatically translates incoming and outgoing data — e.g. XML, JSON, and raw — into the native, tree-shaped data values of Jolie — the interpreter always represents values for variables as tree data structures. By using JIoT, IoT applications developers do not need to handle data conversion themselves, since the interpreter efficiently manages the runtime, that belongs to the same language they use to program the service application logic. Essentially, by being integrated in JIoT, our framework addresses challenge (V) by supporting *data formats interoperability by construction.*

In this section, we report the formal model, called TQuery, to guide the implementation of our JIoT framework. TQuery is inspired by MQuery [10], a sound variant of the MongoDB Aggregation Framework [84]; the most popular query language for NoSQL data handling.

On the one hand, we abstract away implementation details and reason for the overall semantics of our model. We favoured a "theory-to-practice" strategy to avoid inconsistent or counter-intuitive query behaviours. Being this one of the significant drawbacks of the MongoDB Aggregation Framework implementation — that, actually, lacks of a proper formalization — as highlighted in [10]. On the other hand, we believe that formalization provides a general reference for implementors. We kept an eye on technical development needs while focusing on the formalism in order to balance our design choices. For instance, in the formalization of our TQuery we chose to adopt a tree semantic rather than a

set one (as chosen by MQuery authors).

### 3.3.1 The TQuery Framework

In this section, we report the formal syntax and semantics of the TQuery operators. We start denoting trees with the letter $t$, as data structures containing two elements. First, trees contains a *root* value that we denote with $b$, and $b$ assumes `string`, `integer`, ..., or `null` values (denoted with $v$). Second, associated with every root, trees contains a set of one-dimensional arrays[4] filled with sub-trees. Labels identify arrays, and we usually denote a label with letter $k$. Formally:

$$t := b\{k_i : a_i\}_i \quad \text{with} \quad a := [t_1, \cdots, t_n]$$

We indicate with $k(t)$ the extraction of the array pointed by label $k$ in $t$: if $k$ is present in $t$ we retrieve the related array, otherwise we return the `null` array $\alpha$ (different from the empty array, instead denoted with $[\,]$). Formally:

$$k(b\{k_i : a_i\}_i) = \begin{cases} a & \text{if } (k : a) \in \{k_i : a_i\}_i \\ \alpha & \text{otherwise} \end{cases}$$

We assume the range of arrays to run from the minimum index (one) to the maximum, that corresponds to the cardinality of $a$, denoted with $\#a$, which we also use to represent the size of the array. We indicate the extraction of the tree $t$ at index $i$ in array $a$ with the index notation $a[i]$ (in this case $a[i] = t$). In case $a$ contains an element at index $i$ we retrieve it, otherwise we retrieve the `null` tree, denoted with $\tau$. Formally:

$$a[i] = \begin{cases} t_i & \text{if } a = [t_1, \cdots, t_n] \ \wedge \ 1 \leq i \leq n \\ \tau & \text{otherwise} \end{cases}$$

We denote a path $p$ to express tree traversal. Paths are concatenations of expressions, indicated with $e$, or the sequence termination $\epsilon$[5]. Each expressions $e$ evaluates to its label $k$. Formally:

$$p := e.p \ \mid \ \epsilon$$

The application of a path $p$ to a tree $t$, written $[\![\, p \,]\!]^t$, retrieves an array that contains the sub-trees reached traversing $t$ following $p$. We indicate with

[4] Vectors, in the mathematical notation.

[5] In the remainder, we omit to indicate sequence termination $\epsilon$ when referring to paths.

$e \vdash k$ (we read $e$ evaluates to $k$) that the evaluation of expression $e$ in a path $p$ results in the label $k$. Paths neglect array indexes: for a given path $e.p$, such that $e \vdash k$, we always apply the sub-path $p$ to all trees pointed by $k$ in $t$. We report the array concatenation operator, denoted with ::, such that: $[t_1, \cdots, t_n] = [t_1] :: \cdots :: [t_n]$ and the associated properties. Given two arrays $a'$ and $a''$, the concatenation operation :: retrieves an array $a$ of size $\#a = \#a' + \#a''$ where elements $a[1], \cdots, a[\#a']$ correspond point-wise to elements $a'[1], \cdots, a'[\#a']$ and elements $a[\#a' + 1], \cdots, a[\#a' + \#a'']$ correspond point-wise to elements $a''[1], \cdots, a''[\#a'']$.

Finally, we can denote $[\![ p ]\!]^t$, which either retrieves an array $a$ of trees $t_1, \cdots, t_n$ or the `null` array $\alpha$ ,in case the path is not applicable. Formally:

$$[\![ p ]\!]^t = \begin{cases} [\![ p' ]\!]^{t_1} :: \cdots :: [\![ p' ]\!]^{t_n} & \text{if } p = e.p' \ \wedge \ e \vdash k \ \wedge \ k(t) = [t_1, \cdots, t_n] \\ [t] & \text{if } p = \epsilon \\ \alpha & \text{otherwise} \end{cases}$$

For completeness, we also report the structural equivalences on arrays, that we will use in the remainder of the formalisation.

$$\begin{aligned} &\alpha :: \alpha \equiv \alpha \\ &\alpha :: [] \equiv [] :: \alpha \equiv [] :: [] \equiv [] \\ &\alpha :: a \equiv a :: \alpha \equiv [] :: a \equiv a :: [] \equiv a \end{aligned}$$

In the following, we first present the general TQuery syntax and, then, we dedicate the remaining of the section to the semantics of each operator.

### The TQuery Syntax

A query in TQuery syntax corresponds to a sequence of stages, each denoted with the letter $s$, applied on an array $a$. We use the staging operator, denoted with $\rhd$ to evaluate the left expression and pass the result as input to the right expression. Formally:

$$a \rhd s_1 \rhd \cdots \rhd s_n$$

We report in figure 3.1 the syntax of TQuery stages — *match* ($\mu$), *unwind* ($\omega$), *project* ($\pi$), *group* ($\gamma$), and *lookup* ($\lambda$). Besides, we complete the definitions of the stages with the other four syntactic rules. In particular, rule two completes the match operator, rules three and four complete the project stage, and rule five complete the group stage. For those that remain, we already provided a definition above (the only difference is that $p$, $q$ and $r$ denote paths too).

In particular, we list the five possible stages, and we briefly discuss their semantic in the following.

$$
\begin{aligned}
s &:= \mu_{\varphi} \mid \omega_{p} \mid \pi_{\Pi} \mid \gamma_{\Gamma:\Gamma'} \mid \lambda_{q=a.r \rangle p} \\
\varphi &:= \texttt{true} \mid p = a \mid p_1 = p_2 \mid \exists p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \\
\Pi &:= p \mid d \rangle p \mid p, \Pi \mid d \rangle p, \Pi \\
d &:= b \mid p \mid [d_1, \cdots, d_n] \mid \varphi \mid \varphi ? d_1 : d_2 \\
\Gamma &:= p \rangle p' \mid p \rangle p', \Gamma
\end{aligned}
$$

**Figure 3.1:** Syntax of TQuery.

$\mu_{\varphi}$ The purpose pf the *match* operator is to select trees according to the criterion $\varphi$. $\varphi$ criterion is either (i) the boolean truth `true`, (ii) a condition expressing the equality of the application of path $p$ and the array $a$, (iii) a condition expressing the equality of the application of path $p_1$ and the application of a second path $p_2$, (iv) the existence of a path $\exists p$, or (v) the standard logic connectives negation $\neg$, conjunction $\wedge$, and disjunction $\mid$ .

$\omega_{p}$ The purpose of the *unwind* operator is to flatten an array reached through a path $p$ and outputs a tree for each element of the array.

$\pi_{\Pi}$ The purpose of the *project* operator is to modify trees by projecting away paths, renaming paths, or introducing new paths, as described in the sequence of elements in $\Pi$, which are either a path $p$ or a *value definition* $d$ inserted into a path $p$. Value definitions can be: (i) a boolean value $b$ (`true` or `false`), (ii) the application of a path $p$, (iii) an array of value definitions, (iv) a criterion $\varphi$, or (v) the ternary expression which, depending the satisfiability of criterion $\varphi$ selects either value definition $d_1$ or $d_2$.

$\gamma_{\Gamma,\Gamma'}$ The purpose of the *group* operator is to group trees according to a grouping condition $\Gamma$ and aggregates values of interest according to $\Gamma'$. Both $\Gamma$ and $\Gamma'$ are sequences of elements of the form $p \rangle p'$ where $p$ is a path in the input trees, and $p'$ a path in the output trees.

$\lambda_{q=a.r \rangle p}$ The purpose of the *lookup* operator is to join input trees with trees in an external array $a$. When we apply $\lambda$ on an array $a$, we look for those applications of paths $q$ and $r$ that are equal both in $a$ and $a'$ trees, respectively.

### The Match Operator $\mu$

When applied to an array $a$, match $\mu_\varphi$ returns those elements in $a$ that satisfy $\varphi$. If there is no element in $a$ that satisfies $\varphi$, $\mu_\varphi$ it retrieves the empty array $[\,]$ (different from the `null` array $\alpha$). Below, we mark $t \models \varphi$ (we read $t$ satisfies $\varphi$) the satisfiability of criterion $\varphi$ by a tree $t$.

The application of the two paths satisfies the criterion $\varphi = (p_1 = p_2)$ both when the application to the input tree $t$ retrieves the same array $a$ or when both paths do not exist in $t$ — i.e. their application coincide on $\alpha$.

The main differences with [10] regard the matching over the equality of an array — i.e. when $\varphi = (p = a)$. In [10], Botoeva et al. considered the equation $p = v$ where $v$ can either be a literal, an array or a (JSON) object. In our setting, instead, the equality $p = a$ means the array equality — i.e. each tree in the array found under path p in $t$ must be point-wise equal to the elements in $a$. Formally:

$$[t] :: a \rhd \mu_\varphi = \begin{cases} [t] :: (a \rhd \mu_\varphi) & \text{if } t \models \varphi \\ a \rhd \mu_\varphi & \text{if } \#a > 0 \\ [\,] & \text{otherwise} \end{cases}$$

$$t \models \varphi \text{ holds iff } \begin{cases} \varphi & = \texttt{true} \\ \varphi & = (\exists p) \ \wedge \ [\![\, p \,]\!]^t \neq \alpha \\ \varphi & = (p = a) \ \wedge \ [\![\, p \,]\!]^t = a \\ \varphi & = (p_1 = p_2) \ \wedge \ t \models \big((p_1 = a) \ \wedge \ (p_2 = a)\big) \\ \varphi & = (\neg\varphi') \ \wedge \ t \not\models \varphi' \\ \varphi & = (\varphi_1 \ \wedge \ \varphi_2) \ \wedge \ (t \models \varphi_1 \ \wedge \ t \models \varphi_2) \\ \varphi & = (\varphi_1 \ \vee \ \varphi_2) \ \wedge \ (t \models \varphi_1 \ \vee \ t \models \varphi_2) \end{cases}$$

### The Unwind Operator $\omega$

To report the semantics of the unwind operator $\omega$, we introduce the *unwind expansion operator* $\mathbf{E}\,(t, a)^k$ (read "unwind $t$ on $a$ under $k$"). Informally $\mathbf{E}\,(t, a)^k$ retrieves an array of trees with cardinality $\#a$ where each element has the shape of $t$ but, element $a_i$ is the only element under label $k$ in the $i$-th tree of the resulting array. Formally, given a tree $t$, an array $a$, and a key $k$ we denote $\mathbf{E}\,(t, a)^k$.

$$\mathbf{E}\,(t, a)^k = \begin{cases} \Big[b\Big(\big(\{k_i : a_i\}_i \setminus \{k : k(t)\}\big) \cup \{k : [t']\}\Big)\Big] :: \mathbf{E}\,(t, a')^k & \text{if } {}^{6} \\ [\,] & \text{otherwise} \end{cases}$$

We can now report on the unwind operator, over both $a$ and $p$. The induction over $a$ results in the application of the unwind expansion operator **E** over all elements of $a$. The induction over $p$ splits $p$ in the current key $k$ and the continuation $p'$. We use key $k$ to retrieve the array in the current element of $a$ with label $k$ that is, $[\![\, k.\epsilon \,]\!]^t$ (we omit $\epsilon$ in the formalism below since it does not add any valuable information). We apply $\omega'_p$ on this element to continue the unwind application until we reach the termination with $p = \epsilon$. Formally, we report $a \rhd \omega_p$:

$$a \rhd \omega_p \begin{cases} \mathbf{E}\,(t, [\![\, k \,]\!]^t \rhd \omega_{p'})^k :: a' \rhd \omega_p & \text{if } p = e.p' \wedge e \vdash k \wedge a = [t] :: a' \\ a & \text{if } p = \epsilon \\ [\,] & \text{otherwise} \end{cases}$$

### The Project Operator $\pi$

We start by defining the auxiliary operators we used in the definition of the project. Auxiliary operators $\pi_p(a)$ and $\pi_p(t)$ formalise the application of a branch-selection over a path $p$. Then, the auxiliary operator $\texttt{eval}(d, t)$ returns the array resulting from the evaluation of a definition $d$ over a tree $t$. Finally, we report the projection of a value (definition) $d$ into a path $p$ over a tree $t$ that is, $\pi_{d \rangle p}(t)$.

The projection $\pi$ for a path $p$ over an array $a$ results in an array $\pi_p(a)$ whose elements are the projection of the elements of $a$:

$$\pi_p(a) = \pi_p([t_1, \cdots, t_n]) = [\pi_p(t_1), \cdots, \pi_p(t_n)]$$

The projection for a path $p$ over a tree $t$ implements the actual semantics of branch-selection, where, given a path $e.p'$, $e \vdash k$, we remove all the branches $k_i$ in $t = b\{k_i : a_i\}$, keeping only $k$ (if $k \in \{k_i\}$) and continue to apply the projection for the continuation $p'$ over the (array of) sub-trees under $k$ in $t$ that is, $[\![\, k.\epsilon \,]\!]^t$. Formally:

$$\pi_p(t) = \begin{cases} v\{k : \pi_{p'}([\![\, k.\epsilon \,]\!]^t)\} & \text{if } [7] \\ t & \text{if } p = \epsilon \\ \tau & \text{otherwise} \end{cases}$$

[6] $a = [t'] :: a' \wedge t = b\{k_i : a_i\}_i$
[7] $[\![\, p \,]\!]^t \neq \alpha \wedge p = e.p' \wedge t = b\{k_i : a_i\}_i \wedge e \vdash k$

The operator $\texttt{eval}(d,t)$ evaluates the value definition $d$ over the tree $t$ and returns an array containing the result of the evaluation. Formally:

$$\texttt{eval}(d,t) = \begin{cases} [d\{\ \}] & \text{if } d \in V \\ [t \models \varphi\{\ \}] & \text{if } d \in \varphi \\ [\![\, d \,]\!]^t & \text{if } d \in P \\ \texttt{eval}(d,t) :: \texttt{eval}(d',t) & \text{if } d = [d] :: d' \\ \texttt{eval}(d',t) & \text{if } d = \varphi ? d_{\texttt{true}} : d_{\texttt{false}} \ \wedge\ d' = d_{t \models \varphi} \\ \alpha & \text{otherwise} \end{cases}$$

The projection of a value definition $d$ on a path $p$ retrieves a tree where the projection inserted, under path $p$, is the $d$ evaluation over $t$.

$$\pi_{d \,\rangle\, p}(t) = \begin{cases} v\{k : [\pi_{d \,\rangle\, p'}(t)]\} & \text{if } p = e.p' \ \wedge\ e \vdash k \ \wedge\ \texttt{eval}(d,t) \neq \alpha \\ v\{k : \texttt{eval}(d,t)\} & \text{if } p = e.\epsilon \ \wedge\ e \vdash k \ \wedge\ \texttt{eval}(d,t) \neq \alpha \\ \tau & \text{otherwise} \end{cases}$$

Before formalizing the projection, we report the auxiliary operator to merge trees, denoted with $\oplus$, used to merge the result of a sequence of projections $\Pi$. Formally:

$$\begin{aligned} &([t] :: a) \ \oplus\ ([t'] :: a') = [t \ \oplus\ t'] :: a \ \oplus\ a' \\ &a \ \oplus\ [\,] = [\,] \ \oplus\ a = a \ \oplus\ \alpha = \alpha \ \oplus\ a = a \\ &b\ \{k_i : a_i\}_i \ \oplus\ b'\ \{k_j : a_j\}_j = \tau \quad \text{if } b \neq b' \\ &t \ \oplus\ \tau = t \\ &t \ \oplus\ t' = b\{k_h : k_h(t) \ \oplus\ k_h(t')\}_{h \in I \cup J} \quad \text{if } {}^{8} \end{aligned}$$

To conclude, we first report the application of the projection to a tree $t$ ($t \rhd \pi_\Pi$), which merges the results of projections $\Pi$ over $t$ into a single tree. Second, we report the projection operator $\pi_\Pi$ to an array $a$ ($a \rhd \pi_\Pi$), which corresponds to the projection to all elements of $a$. Respectively, we formally write:

$$\pi_\Pi(t) = \begin{cases} \pi_p(t) \ \oplus\ (t \rhd \pi_{\Pi'}) & \text{if } \Pi = p, \Pi' \\ \pi_{d \,\rangle\, p}(t) \ \oplus\ (t \rhd \pi_{\Pi'}) & \text{if } \Pi = d \,\rangle\, p, \Pi' \\ \pi_p(t) & \text{if } \Pi = p \\ \pi_{d \,\rangle\, p}(t) & \text{if } \Pi = d \,\rangle\, p \end{cases}$$

[8] $t = b\{k_i : a_i\}_{i \in I} \ \wedge\ t' = b\{k_j : a_j\}_{j \in J}$

and:

$$a \rhd \pi_{\Pi} = [t_1, \cdots, t_n] \rhd \pi_{\Pi} = [\pi_{\Pi}(t_1), \cdots, \pi_{\Pi}(t_n)]$$

**The Group Operator $\gamma$**

The group operator, applied to an array $a$, is parametric for two sequences of paths, we write $q_1 \rangle\, p_1, \cdots, q_n \rangle\, p_n : s_1 \rangle\, r_1, \cdots, s_m \rangle\, r_m$ We call the first sequence of paths $(q \rangle\, p)$, ranged $[1, n]$, the *aggregation set*, while we call the second sequence $(s \rangle\, r)$, ranged $[1, m]$, the *grouping set*.

Intuitively, the $\gamma$ operator first groups together the trees in $a$ which have the maximal number of paths $s_1, \cdots, s_m$ in the grouping set whose values coincide. Then, it projects the values in $s_1, \cdots, s_m$ in the corresponding paths $r_1, \cdots, r_m$. Once the operation succeeds and the trees grouped, the $\gamma$ operator aggregates all the different values, avoiding duplicates, found in paths $q_1, \cdots, q_n$ from the aggregation set, projecting them into the corresponding paths $p_1, \cdots, p_n$.

We start the definition of the $\gamma$ operator by expanding its application to an array $a$. Note that, in the expansion below, on the right, we use the series-concatenation operator $\vdots$ and the set $H$, the element of the power set $2^{[1,m]}$, to range over all possible combinations of paths in the grouping set. Namely, the expansion corresponds to the concatenation of the resulting arrays from the group operator, on a subset of paths in the grouping set. Formally:

$$\gamma_{q_1 \rangle\, p_1, \cdots, q_n \rangle\, p_n : s_1 \rangle\, r_1, \cdots, s_m \rangle\, r_m} \rhd a = \vdots_{\forall H \in 2^{[1,m]}} \gamma^{H}_{q_1 \rangle\, p_1, \cdots, q_n \rangle\, p_n : s_1 \rangle\, r_1, \cdots, s_m \rangle\, r_m}(a)$$

In the following definition of the expansion, we will mark $\{\!\{\, a \,\}\!\}$ the casting of an array $a$ to a set, allowing us to keep only unique elements in $a$ and lose their relative order. Each $\gamma^{H}_{q_1 \rangle\, p_1, \cdots, q_n \rangle\, p_n : s_1 \rangle\, r_1, \cdots, s_m \rangle\, r_m}(a)$ returns an array that contains those trees in $a$ that correspond to the grouping illustrated above. When applied over a set $H$ with $h \in H$, $\gamma$ considers all the combinations of values identified by paths $s_h$ in the trees present in $a$. In the formalization's condition, we use the array $a'$ to refer to those combinations of values. In the definition, we impose that, for each element in $a'$ in a position $h$, there must be at least one tree in $a$ that has a non-null array $(\neq \alpha)$ under path $s_h$. Hence, for each combination $a'$ of values in $a$, $\gamma$ builds a tree that contains under paths $r_h$ the value $a'[h]$ (as encoded in the projection query $\chi$ and from the definition of the operator $a'[h] \rangle\, H, a$. It also contains under paths $p_i$, $1 \leq i \leq n$, the array containing all the values found under the correspondent path $q_i$ in all trees in $a$ that match the same combination element-path in $a'$ (as encoded in $\theta_i$).

The grouping is valid (as encoded in $\psi_i$) only if we can match trees in $a$ where either we have a non-empty value for $q_i$, there are no paths $s_j$ that

are excluded in $H$, or for all paths considered in $H$, the value found under path $s_h$ corresponds to the value in the considered combination $a'[h]$. If the three previous conditions do not hold, $\gamma$ returns the empty array $[\,]$. We now formalize the expansion writing:

$$\gamma^{H}_{q_1 \rangle\, p_1,\cdots,q_n \rangle\, p_n:s_1 \rangle\, r_1,\cdots,s_m \rangle\, r_m}(a) = \begin{cases} \overset{\bullet\bullet}{\underset{\forall a'}{\bullet\bullet}} \left[ \bigoplus_{i=1}^{n} \pi_{\chi,\theta_i}(\tau) \right] & \text{if } [9] \\ [\,] & \text{otherwise} \end{cases}$$

We conclude defining the operator $a'[h] \rangle\, H, a$, that we used to unfold the set of aggregation paths and the related values contained in $H$, Formally:

$$a'[h] \rangle\, H, a = \begin{cases} a'[j] \rangle\, a[j], (a'[h] \rangle\, (H \setminus \{j\}), a) & \text{if } |H| > 1 \ \wedge\ j \in H \\ a'[j] \rangle\, a[j] & \text{if } |H| = 1 \ \wedge\ j \in H \\ \epsilon & \text{otherwise} \end{cases}$$

Note that, for case where $\gamma^{\varnothing}_{\cdots}$ that is, for $H = \varnothing$, $a'[h] \rangle\, H, a$ retrieves the empty path $\epsilon$, which has no effect — i.e. it projects the input tree — in the projection $\pi_\chi$. Hence, the resulting tree from grouping over $\varnothing$ will just include (and project over $p_1, \cdots, p_n$) those trees in $a$ that do not include any value reachable by paths $s_1, \cdots, s_m$.

Similarly to MongoDB implementation of the aggregation framework, we chose to allow one to omit paths $p_1, \cdots, p_n$ and $r_1, \cdots, r_n$ in $\Gamma : \Gamma'$. We intend this omission as an indication of the fact that the user wants to preserve the structure of $q_i$, provided that the following structural equivalence holds[10]:

$$\gamma_{q_1,\cdots,q_n:s_1,\cdots,s_m} \equiv \gamma_{q_1 \rangle\, q_1,\cdots,q_n \rangle\, q_n:s_1 \rangle\, s_1,\cdots,s_m \rangle\, s_m}$$

9

$$\begin{aligned} h \in H \ \wedge\ & a'[h] \in \left\{ [\![\, s_h \,]\!]^t \ \middle|\ t \in \{\!\{ a \}\!\} \ \wedge\ [\![\, s_h \,]\!]^t \neq \alpha \right\} \\ & \wedge\ \chi = (a'[h] \rangle\, H, [r_1, \cdots, r_n]) \\ & \wedge\ \theta_i = \overset{\bullet\bullet}{\underset{\forall t_i}{\bullet\bullet}} [\![\, q_i \,]\!]^{t_i} \rangle\, p_i \ \wedge\ t_i \in \{\!\{ a \rhd \mu_{\psi_i} \}\!\} \supset \varnothing \\ & \wedge\ \psi_i = \exists q_i \ \wedge\ \neg \bigvee_{j \notin H} \exists s_j \ \wedge\ \bigwedge_h \Big( (s_h = a'[h]) \ \wedge\ \exists s_h \Big) \end{aligned}$$

[10] In the implementation, we store the values obtained from $q_i$s with missing $p_i$s within a default path `_id`.

**The Lookup Operator $\lambda$**

Informally, the lookup operator joins two arrays, a source $a$ and an adjunct $a'$, respect to the two source paths $q$ and $r$, and to the destination path $p$. When applied to an array of trees, the lookup retrieves an array where each of its elements has, under path $p$, an array of trees obtained from the match ($\mu_{r=a'}$) in expression $\beta_i$. For each element $a[i]$ ($1 \leq i \leq n$), $\beta_i$ matches those trees in $a'$ for which either (i) there is a path $r$ and the array reached under $r$ equals the array found under $[\![ q ]\!]^{a[i]}$ or (ii) there exist no path $r$ that it, its application returns the `null` array $\alpha$, and also $q$ does not exist in $t_i$ ( $[\![ q ]\!]^{a[i]} = \alpha$). Formally:

$$a \rhd \lambda_{q=a'.r \rangle p} = [\pi_{\epsilon,\beta_1}(a[1])] :: \cdots :: [\pi_{\epsilon,\beta_n}(a[n])]$$

s.t.

$$\beta_i = (a' \rhd \mu_{r=a''}) \rangle p \ \wedge \ a'' = [\![ q ]\!]^{a[i]} \ \wedge \ 1 \leq i \leq n$$

## 3.4 Case Study

In this section, we present a non-trivial case study to overview the TQuery implementation and its operators, by means of their JIoT programming interfaces. Remarkably, the following case study constitutes the first concrete evaluation of the MQuery framework [10]. Besides the framework formalisation, that we illustrated in section 3.3, we now describe its implementation details by means of the case study, that will help us showing the semantics of TQuery.

Our case study leverages the work did in [69], where the authors delineate a *smart thermostat* algorithm. This algorithm take advantage of the behavioural patterns of the building inhabitants to save energy, by automatically turning off and on the HVAC system. The data handling strategy follows the principle of "data never leave the building" in compliance with the GDPR privacy policy [108].

The setting of the case study is the same one we used in section 2.4 that is, an IoT application taken from a smart building automation scenario, where a set of heterogeneous devices performs a simple distributed application logic deployed among Cloud instances, mid-tier controllers, edge and low-level devices. Although in section 2.4, we omitted to specify a storage system[11], it

[11] We dit not consider a persistence strategy neither we implemented a dashboard-related service, since we wanted it to be as general as possible, thus allowing one to easily integrate its own data model.

is common to provide, along with the logic engine, a persistence strategy, that allows one to store the sensed data (modulo document-based conversion).

Concerning the case study — to illustrate the formal semantics of TQuery in an ephemeral data handling scenario — we will focus on the mid-tier, edge, and low-level entities of the network. Hence, from the framework perspective, we do not show here the output of TQuery operators, which are reported in their relative subsections in section 3.3. Concerning the network entities, we consider the HVAC controller system of figure 2.5 composed by the Arduino Uno equipped with the temperature and humidity sensors, the SmartThings motion sensor, and the ESP8266 acting as a gateway for the external connected and remotely configurable thermostat.

Briefly, the algorithm described in [69] could be summarized in the following. The smart thermostat uses motion sensors records to infer when occupants are away from or in specific rooms and turn the HVAC system on or off without sacrificing occupant comfort. We report in listing 3.1, in a JSON-like format, code snippets exemplifying the two kinds of data structures. Both structures are arrays, marked `[ ]`, containing tree-like elements, marked `{ }`. At line 1, we have a snippet presenting, for each `date`, an array of detected temperatures (`t`) and humidities (`h`). At line 2 we show a snippet of the presence logs [69], where, to each year (`y`) corresponds an array of monthly (`M`) measures, to a month (`m`), an array of daily (`D`) logs, and to a day (`d`), an array of logs (`L`), each representing a presence survey with its absolute UTC time (`at`).

```
1 [ { date: "20190829", t: ["32",...], h: ["94",...] }, { date:
    "20190830", t: ["32",...], h: ["83",...] }, ... ]
2 [ { y: "2019", M: [ ..., { m: "08", D: [ { d: "29", L: [ { at:
     "0801", p: "in" }, { at: "0936", p: "out" }, ... ] }, { d:
     "30", L: [ { at: "0833", "out" }, ... ] }, ... ] }, ... ]
   }, ... ]
```

**Listing 3.1:** Environmental and presence logs data structures.

On the data structures in listing 3.1, we define a JIoT microservice, from where we reported the `main` routine in listing 3.2, which describes the handling of the data and the workflow of the algorithm, using our implementation of TQuery.

```
1 getuserPseudoID@SmartBuilding( userData )( pseudoID );
2 credentials
3 |> getHumidityAndTemperature@Arduino
4 |> match { date == "20190829" || date == "20190830" }
5 |> project { t in temperatures, pseudoID in user_id }
6 |> temps ;
```

```
7 detectComfort@SmartBuilding( temps )( isComfort );
8 if( !isComfort ) {
9   credentials
10  |> getPresencePatterns@SmartThingsHub
11  |> unwind  { M.D.L }
12  |> project { y in year, M.m in month, M.D.d in day, M.D.L.p
        in presence }
13  |> match { date == "20190829" && date == "20190830" }
14  |> group { presence by day, month , year }
15  |> project { presence, pseudoID in user_id }
16  |> lookup { user_id == temps.user_id in temps }
17  |> detectState@SmartBuilding
18 }
```

**Listing 3.2:** Microservice implementing `DetectState` algorithm.

The example is detailed enough to let us illustrate all the operators in TQuery: `match`, `unwind`, `project`, `group`, and `lookup`. Note that, while in listing 3.2 we hard-codes some data — e.g. strings representing dates, such as `20190829`) — for presentation purposes, we would normally use parametrised variables.

In listing 3.2, line 1 defines an outbound request (a solicit response) to an external microservice, provided by the **`SmartBuilding`**. The service offers functionality `getUserPseudoID` which, given some identifying `userData` (acquired earlier), provides a pseudo-anonymized identifier — needed to treat data — saved in variable `pseudoID`.

At line 2, we evaluate the content of variable `credentials`, which holds the certificates that allows the service provider to access the sensors measurements for a given user. In the routine, `credentials` is passed by the chaining operator at line 3 as the input of the external call to functionality `getHumidityAndTemperature`. `getHumidityAndTemperature` operation retrieves the environmental data, as shown in listing 3.1 at line 1 from the **`Arduino`** of the user. At lines 3–6 (and later at lines 10–17) we use the chaining operator `|>` to define a sequence of calls, either to external services, marked by the `@` operator, or to the internal TQuery library. The `|>` operator takes the result of the left expression and passes it to the right expression.

While the default syntax of operation call in JIoT is the one with the double pair of parenthesis — e.g. at line 1 — thanks to the chaining operator `|>` we can omit to specify the input of `getHumidityAndTemperature` (passed by the `|>` at line 3) and its output (the environmental data exemplified at listing 3.1) passed to the `|>` at line 4.

At line 4 we use the TQuery operator `match` to filter all the entries of the environmental data, keeping only those collected in the last two days — i.e. since `20190830`. The result of the `match` is then passed to the `project` operator

at line 5, which removes all nodes but the temperatures, found under `t` and renamed `in temperatures` (this is required by the interface of functionality `detectComfort`, explained below). The `project` also includes in its result the `pseudoID` of the user, `in` node `user_id`. We finally store (line 6) the prepared data in variable `temps` (since it will be used both at line 7 and 16).

At line 7, we call the external operation `detectComfort` to analyze the temperatures and check if the environment conditions, storing the result in variable `isComfort`. After the analysis on the temperatures, `if` the temperature `isComfort` retrieves `true` (line 8) and we stop testing, otherwise we continue for detecting the environment state — smart thermostat algorithm in [69] identifies three possible states for a given area, these are *Active*, *Away*, and *Sleep*. To be able to assess the state, at lines 9–10, we follow the same strategy described for lines 2–3 to pass the `credentials` to functionality `getPresencePatterns`, used to collect the presence logs of the user from the **`SmartThingsHub`**. Since the presence logs is a nested structure, having years, months, and days super-levels, to filter the logs relative to the last two days, we first flatten the structure through the `unwind` operator applied on nodes `M.D.L` (line 11). For each nested node, separated by the dot `.`, the `unwind` generates a new data structure for each element in the array reached by that node. Concretely, the array returned by the `unwind` operator at line 11 contains all the presence logs in the shape shown in listing 3.3.

```
1{ [ { year: "2019", M: [ { m: "08", D: [ { d: "29", L: [ { at:
     "0801", p: "in" } ] } ] } ] }, { year: "2019", M: [ { m: "
    08", D: [ { d: "29", L: [ { at: "0936", p: "out" } ] } ] }
    ] } ] }
```

**Listing 3.3:** Data structure after the `unwind` application.

In listing 3.3 there are as many elements as there are presence logs and the arrays under `M`, `D`. After that, `L` contain only one presence log. Once flattened, at line 12 we modify the data-structure with the `project` operator to simplify the subsequent chained commands by renaming the node `y in year`, we move and rename the node `M.m in month` (bringing it at the same nesting level of `year`); similarly, we move `M.D.d`, renaming it `day`, and we move `M.D.L.p`, renaming it `presence` — `M.D.L.at` not included in the `project`, are discarded. On the obtained structure, we filter the presence logs relative to the last two days with the `match` operator at line 13. At line 14 we use the `group` operator to aggregate the sessions recorded in the same day — i.e. grouping them `by day`, `month`, and `year`. Finally, at line 15 we select, through the `project`, only the aggregated values of `presence` (getting rid of `day`, `month`, and `year`) and we include under node `user_id` the `pseudoID` of the user. That value is used at

line 16 to join, with the `lookup` operator, the obtained presence logs with the previous values of temperatures (`temps`). The resulting, merged data-structure is finally passed to the **`SmartBuilding`** services by calling the functionality `detectState` (line 17). In conclusion, to correctly set the thermostat on and off, it would be sufficient to call the relative operation on the ESP8266 `outputPort` (properly configured), that we omitted in the case study not being part of this framework and already discussed in section 2.4.

### 3.4.1 Benchmark

As a preliminary[12] result, we benchmarked the query at lines 4–5 of listing 3.2 against a comparable architecture based on MongoDB. We draw our experiment, adapting test performed on MongoDB in an IoT setting from the literature [136]. We programmed two microservices: `TQueryService` that contains the implementation at lines 4–5 of listing 3.2; `MongoDBService` implements the same logic (lines 4–5) in terms of MongoDB queries. The `MongoDBService` implements this simple behaviour: (i) ensures data are in JSON format, (ii) inserts the data in the database, (iii) sends the query (match and project) as one instruction to the database, and (iv) deletes the inserted data to ensure ephemerality.

To run our tests, we use five instances of the JSON data structure at line 1 of listing 3.2. Each instance cover one year of recordings, and instances include synthetically-derived recordings at increasing sampling rate — i.e. the first instance contains one sampling per minute (1440 samplings per day), the second contains two samplings per minute, ....

We simulate bursts of requests in four subsequent batches, each with ten concurrent calls, forty requests in total. A third microservice loads the data and sends the ten separate requests to `TQueryService` and, respectively, `MongoDBService`, at a time, using different available data formats (XML, JSON, and raw). We draw our benchmarks in the figure below, reporting the average time over the forty requests for each sampling. In `TQueryService`, we start the timer before executing the first query instruction (`match`), and we stop it after we obtain the result of the last (`project`). In `MongoDBService`, we start the timer before executing the insertion in the database and stop it after we queried and deleted the data. We run our benchmarks on a machine equipped with a 1,7 GHz Intel Core i7 dual-core processor and 16GB RAM, running macOS 10.15.1, Java 13, JIoT 1.2-beta, and MongoDB Community Server 4.2.0-build.3.

[12] We compared and drawn our benchmarks on the solely project operator against a comparable frameworks, since the remaining tests are still an ongoing chapter.

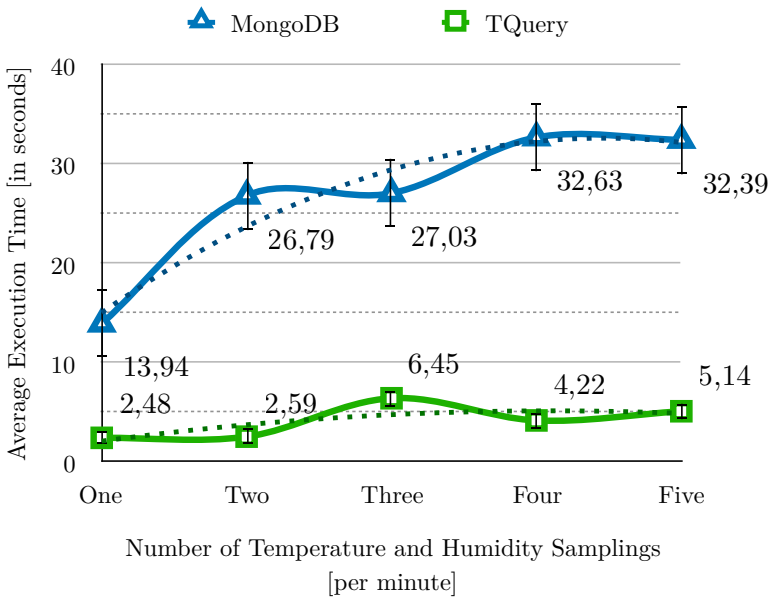


**Figure 3.2:** Benchmark results for TQuery and MongoDB.

In figure 3.2 we illustrate the benchmark results, plotting the average execution times of the query ($y$ axis, time indicated in seconds) for both TQuery (green squares) and MongoDB (blue triangles), over the different requests ($x$ axis, labeled from one to five). In all cases, TQuery performs significantly better than its MongoDB alternative. It is possible to indicate several factors that justify the behaviour depicted in figure 3.2, among the many, we selected the ones that, in our opinion, contributed the most.

- **Velocity.** The TQuery microservice, being implemented as an in-memory application, presents a lower data transmissions than the antagonist.
- **Variety.** Before to send the requests to the external MongoDB engine, the data need to be converted in JSON format, for every request except the one that receives data directly in JSON. This pattern increases the response time of the MongoDB microservice. On the contrary, the TQuery microservice avoids the last conversion step, working with the internal tree representation.
- **Ephemerality.** TQuery microservice does not rely on any persistence

scheme; therefore it does not involve any disk writing I/O operation. Moreover, the external database engine connection in MongoDB, not present in the Tquery microservice, generates overheads for each of the subsequent requests.

Notably, both the framework presents an asymptotic tendency[13] — the dotted polynomial regression lines in figure 3.2. In a test scenario that, ideally, continues *ad infinitum*, we would observe a stationary behaviour of both TQuery and MongoDB frameworks. Still, we empirically proved that the two performance do not converge and, for a reason above described, our approach is strongly faster than the MongoDB framework, when tested in IoT application context.

## 3.5 Discussion

IoT systems present a specific challenge regarding data handling in resource-constrained and restrictively regulated applications. In this chapter, we focus on ephemeral data handling and contrast DBMS-based solutions compared to integrated query engines within application memory. We indicate the issues that make DBMS solutions unfit for ephemeral scenarios and propose a formal model, called TQuery, to express document-based queries over tree-shaped data structures, such as XML or JSON. TQuery instantiates MQuery [10], a sound variant of the MongoDB Aggregation Framework [84], one of the leading NoSQL DBMSes for document-oriented queries.

We adopt a linguistic approach, following microservice-oriented computing principles, implementing TQuery in JIoT, the language presented in chapter 2. JIoT offers variety-by-construction — i.e. the language runtime automatically and efficiently handles data conversion, and all JIoT variables are trees-shaped data structures. These factors allowed to separate input and output data-formats from the data handling logic, hence providing programmers with a single, consistent interface to use TQuery on any data-format supported by JIoT. Furthermore, to evaluate both the TQuery operators formalization and the resulting implementation, we present a non-trivial case study.

Regarding the MongoDB Aggregation Framework, we conduct a specific comparison study, showing that our solution provides a substantial increase in performance. When compared with other DBMS frameworks, the main difference with our solution is the adoption of an external engine to execute

[13] The approximated second grade function, e.g $-1.4881 * x^2 + 13.203 * x - 3.3177$ for MongoDB, has the actual shape of a concave curve.

queries. Indeed, we believe that this limitation of DBMS unfits for their usage in IoT ephemeral scenarios. The same hold for ORMs and Opaleye in-memory frameworks; since the underlying models rely on DBMSs engines, they present the same issues of those systems. Redis and LevelDB lack support for tree-shaped data structures, a crucial feature in the IoT context. LINQ and CQEngine solutions do not provide automatic data-format conversion — e.g. from JSON to tree object — as we do. The lack of conversion automatisms require developers to include external library dependencies implementing such behaviour or to deal with the conversion internally by adopting ad-hoc data structures.

The main limitation of this work is the lack of an extensive comparison with other DBMS solutions and, most of all, with more comparable in-memory proposals — e.g. LINQ and CQEngine. Moreover, a careful evaluation of the performance (time profiling) of the TQuery implementation would allow us to get more useful insight concerning the framework's speed discrepancy.

Furthermore, one can implement the support for new data formats in JIoT, which makes them automatically available to our TQuery, or, by expanding the set of available operators in TQuery, it would make possible to express more complex queries.

# Chapter 4

# Deployments Integration in IoT Systems

An IoT system is a distributed system where the deployment of software components includes power-constrained controllers, middleware (communication) devices, and computational nodes with a specific location in space. To properly deploy an IoT application, each software component should run in the most appropriate execution environment, and satisfying different constraints — e.g. network bandwidth, location, CPU cores, and memory. This work presents a tool, Foehn, that aims to compute the optimal configuration for an IoT application on a given infrastructure, allowing flexibility in the definition of the objective function. Foehn exploits the FogTorch specification language to define both the IoT application and the infrastructure, and the ZephyrusDeclarative Requirement Language to specify the QoS function to optimise and, if needed, further constraints on the desired deployment. Foehn can be executed both from the command line and as a service providing its functionality via an API. The latter execution modality is particularly suited to integrate Foehn in DevOps pipelines. This aims at automating the transition from package to release phases, thus improving the effectiveness of software development process for IoT applications.

## 4.1 Introduction

Modern applications are highly heterogeneous, not just technology-wise — e.g. multi-component, polyglot, and multiprotocol — but also for being deployable in different locations.

Finding the best location where to deploy a specific software component is not a trivial task and developers cooperate with infrastructure management staff to provide solutions satisfying the given requirements. These requirements are usually related to the cost-effectiveness of the final deployment plan, i.e. they require the total cost to be as low as possible. However, they can also be

related to the quality of the offered service, e.g. requiring it to be above or below a certain threshold. Finding optimal deployment plans is a combinatorial task [112], hence the resulting complexity may be very high, in particular when the number of locations with different features increases, as in the case of modern IoT systems, which include components deployed in the Cloud and in the Fog. Hence, such a task cannot be performed manually and developers and IT staff strive for tools able to exploit configuration selection automatically. Our solution is based on state-of-the-art SMT (Satisfiability Modulo Theories) solvers that make the problem tractable in many practical cases [68]. In particular, our tool, Foehn, allows one to specify the architecture of a software to be deployed on a hybrid Cloud-Fog/Edge-Thing infrastructure as well as the facilities and the constraints of the infrastructure itself. It also allows one to fix a sequence of QoS measures to optimise (with a selected priority) and tries to find an optimal deployment plan that satisfies all the constraints. More precisely, since the problem is in general computationally infeasible [26], there are three possible outcomes: an *optimal* solution is produced, the non-existence of the solution is proved, or the SMT solver timeouts.

In order to ensure interoperability with other tools in the area, the input and output syntaxes of Foehn are compatible with pre-existing tools. In particular, the description of the architecture of the IoT application as well as the one of the infrastructure are defined using the FogTorch specification language [13] (we remark here that the deployment of IoT systems specified in FogTorch has been also considered in [13, 14], but their approach does not aim at finding optimal solutions). Similarly, the description of the requirements and of the QoS measures to optimize as well as the optimal configuration produced by Foehn follow the syntax provided by Zephyrus2 [26] (we remark that Zephyrus2 produces optimal deployment for Cloud systems, but not for IoT systems).

Foehn is written in Python 3 and it is available at [40], released under the GNU GPL v3.0 license. The code snippets reported in this chapter are based on version 1.0 of Foehn.

Foehn can be invoked both from the command line and as a service exposing a REST API. This second execution modality aims at simplifying the integration of Foehn in DevOps pipelines. Thus, Foehn contributes to a fully automated pipeline from package to release phases, as required in modern continuous deployment [19] and continuous delivery [54] software engineering practices, as in the DevOps paradigm [53].

More precisely, automatic deployability planning may fit in the three distinct phases of the software development procedure:

1. *at design time*, performing what-if analysis to assess application resiliency

under changes of the infrastructure or to compare different application configurations [46],

2. *at coding time*, selecting the most appropriate technology considering the deployability of its dependencies, and

3. *at run-time*, to find possible reconfigurations to answer changes in the infrastructure (e.g., link failures) or in the required QoS.

In the last case, Foehn can support the interaction between the developers and IT staff of DevOps teams. Indeed, on the one hand, developers continuously change QoS requirements, without the guarantee that the infrastructure satisfies the new constraints. On the other hand, IT staff needs to update the infrastructure and, consequently, its QoS profile, having limited information about the possible effects on the application. Our tool enforces separation of concerns: developers can update the description of the IoT application, IT staff manages the description of the infrastructure, and Foehn ensures that the two are compatible, that is deployment is possible, also indicating a target configuration for the application, which may then be reconfigured. Dynamic reconfiguration is however not considered in the present chapter.

The remainder of this work has the following structure. In Section 4.3 we describe our approach and our contribution. We then put our tool at work on a case study which extends the FogTorch one [13] in Section 4.4. We review related work on the topic of deployment planning in the IoT context in Section 4.2. Finally, we draw conclusions and discuss future directions in Section 4.5.

## 4.2 Related Work

Nowadays, Infrastructure as a Service (*IaaS*) and Platform as a Service (*PaaS*) solutions usually accomplish the goal of application deployment in modern infrastructures. The first provides a set of low-level resources forming a minimal computing environment — e.g. to cite some IaaS commercial solution, Red Hat CloudForms [104] with the support of the OpenStack Platform [105], Amazon AWS [23], DigitalOcean [29], and Microsoft Azure [80]. With IaaS, developers pack the whole software stack into VMs along with its dependencies and execute them on provider's locations. Exploiting the IaaS allows great flexibility but also requires significant expertise and knowledge of both the cloud infrastructure and the application components involved in the process [38]. The most common solutions for the deployment of the application in the Cloud is to rely on pre-configured VMs — e.g. Bento Boxes, Cloud Blueprints,

and AWS CloudFormation — or to exploit configuration management tools such as Puppet [103] or Chef [100].

PaaS level, on the other hand, provides a full development environment: developers code in a programming language supported by the framework offered by the provider, and then automatically deployed to the Cloud. Application in PaaSes are usually natively scalable and can exploit the elasticity of the Cloud to accommodate more requests. However, we are not aware of PaaSes that can guarantee the QoS-aware optimal and automatic planning for the allocation of services in distributed infrastructures. Recently, the emerging serverless paradigm, better known as Function as a Service (*FaaS*), resembles the main concepts and benefit of PaaS without the effort of defining the running environment. The service provider takes care of selecting the most appropriate deployment architecture, selecting it with internal policies. However, the "higher" level of automation that FaaS provides to customers comes at the price of flexibility: in such scenario, developers have nor control on the underlying architecture neither the available technologies — such as the supported programming languages or storage engines.

In the attempt to enumerate the different existing approaches that propose a strategy to select the infrastructure configuration for IoT applications we realized that Fog and Edge computing paradigms both present a "fuzzy" definition from the infrastructural point of view.

In the literature, Fog and Edge represent two computational models. The first was designated as the support model for Cloud computing, bringing computation closer to data sources. The latter is used as a generic term indicating where computation should happen in the network — i.e. at the endpoints of the network.

Nevertheless, to the best of our knowledge, there exists little work concerning the infrastructural differences between the two, that is the distinction from Edge deployment locations and Fog nodes ones. Concretely, the majority of the presented approaches all refers to Fog infrastructures as the reference architecture for the deployment of IoT applications. This infrastructures comprise of both Fog nodes and Edge devices.

In our work, we draw inspiration from FogTorch[14], a software prototype to automatically infer a set of eligible deployment plans for IoT applications. FogTorch relies on the model defined in [13] to provide the input specifications that feed the ad-hoc QoS-aware configuration optimizer.

Similar approaches, and thus comparable with our work, also implement prototypes to deal with IoT applications deployment in Fog infrastructures. In the first, Saurez et al. [112] proposed a `C++` programming framework for the Fog that provides a public interface (*API*) for resource discovery and QoS-

aware incremental deployment via containerization. In the other, Gupta et al. [52] prototyped a simulator to evaluate resource management and scheduling policies applicable to Fog environments concerning their impact on latency, energy consumption and operational cost.

Several projects proposed planning solutions to deploy multi-component applications to different Cloud IaaS or PaaS. For instance, SeaClouds [15] and Aeolus [26] provide deployment models in Cloud infrastructure for software components with specific functional and non-functional requirements. Li et al. [66] proposed to use *OASIS TOSCA* [20] to model IoT applications in hybrid IoT/Cloud scenarios.

Recently, [62] have linked services and networks QoS by proposing a QoS- and connection-aware Cloud service composition approach to satisfy QoS requirements in the Cloud. Nevertheless, the emerging Fog paradigm, differently from the Cloud [62], introduces new issues, mainly due to the heterogeneous nature of its components and the high degree of interaction with the IoT.

To conclude, we report in [83], they aims at modelling QoS profiles for IoT hybrid infrastructures — Cloud + Edge, with Fog support. These approaches do not implement any deployment-related tool to select the most appropriate architecture configuration but serve the purpose of delivering a suitable model to reason on software component requirements and infrastructure profiles. In one of such works, Misra and Sarkar [83] aimed at evaluating Fog nodes latency and energy consumption in IoT scenarios, as compared to traditional Cloud scenarios.

Concerning models at networking level, several approaches proposed to include latency and bandwidth QoS in the infrastructure profile to achieve connectivity and coverage optimization [137, 3], improved resource exploitation of wireless sensors networks (WSN) [25], and to estimate reliability and cost [67].

## 4.3 Approach

IoT systems are deployed in hybrid infrastructures, where Cloud instances, federated Fog nodes and user-managed Edge devices coexist. Without proper abstractions to define system specifications, guaranteeing application deployability in highly heterogeneous architectures is a very complex task. Furthermore, finding a feasible deployment plan for a multi-component application is an NP-hard problem [75] and, in the worst case of looking for the optimal deployment plan it is undecidable [26]. Contrarily, Foehn automatically deals with the exploration of the solutions space hiding the search complexity to the

user.

To illustrate our approach with a running example, let us consider a selected portion of the case study presented in subsection 2.4.3, where we want to integrate target deployment requirements with actual deployment infrastructure, disposing of a `LogicEngine` and a `Driver` microservices.

The goal is to check deployability of a temperature collector system and retrieve the optimal plan (if it exists).

Following the FogTorch [13, 14] specification language, the system description in Foehn would contain: (i) the *target application* description along with the application desiderata, and (ii) the *existing infrastructure* description with the architecture profile. Indeed, we believe that this separation of concerns, helps the differentiation among developers and IT staff responsibility in deployment design.

As an example of target application description let us consider the code in listing 4.1. To increase readability and comply with the real Foehn specification input, we will use JSON as the data format for the code snippets.

```
1 {
2   "components": [{
3     "name": "LogicEngine",
4     "hardware": { "ram": 2 }
5   }, {
6     "name": "Driver",
7     "hardware": { "ram": 1 },
8     "things": [{ "type": "TemperatureSensor" }]
9   }],
10  "links": [{
11    "from": "Driver",
12    "to": "LogicEngine",
13    "latency": 5
14  }]
15 }
```

**Listing 4.1:** Example of Foehn `Collector` components description.

Listing 4.1 contains, at lines 2–9, the specification for the software components, and at lines 10–14 the communication links. The two microservices connects via a communication link, defined in one direction only that is, from the `Driver` to the `LogicEngine`. The intended behaviour includes a middleware collector which sends temperature measurements (from the controlled Thing) to an external service for further manipulation. Notably, the only link `"qos"` requirement is a requested `"latency"` of five (milliseconds)[1].

[1] In listing 4.1 we deliberately omitted units — e.g. `"ram"` (GB) and `"latency"` (ms) since

In addition, the system description also accounts for the actual infrastructure. As an example of such existing architecture, we report the description in listing 4.2.

```
{
  "infrastructure": {
    "nodes": {
      "cloud": [{
        "name": "SmallCloud",
        "hardware": { "ram": 2, "ram_cost": 2 }
      }],
      "fog": [
        { "name": "SmallEdge",
          "hardware": { "ram": 2, "ram_cost": 2 } },
        { "name": "TinyEdge",
          "hardware": { "ram": 1, "ram_cost": 2 } }
      ]
    },
    "links": [{
        "from": "SmallEdge",
        "to": "SmallCloud",
        "upload": [{ "qos": { "latency": 5 } }]
      },
      {
        "from": "TinyEdge",
        "to": "SmallCloud",
        "upload": [{ "qos": { "latency": 60 } }]
      }
    ],
    "things": [{
        "name": "SmallTemperatureSensor",
        "type": "TemperatureSensor",
        "fog_node": "SmallEdge"
      },
      {
        "name": "TinyTemperatureSensor",
        "type": "TemperatureSensor",
        "fog_node": "TinyEdge"
      }
    ]
  }
}
```

**Listing 4.2:** Example of Foehn `Collector` infrastructure description.

Listing 4.2 presents three main specification sections:

we neglect to check them in the Foehn model, assuming that developers are responsible for unit representation consistency.

1. the *available nodes*, stating the possible deployment locations in the underlying infrastructure (lines 3–13) They are denoted with the term `"nodes"`, and they can be either of type `"cloud"` or of type `"fog"`;

2. the *existing connections*, which describes the communication `"links"` (lines 15–25), and inform about their `"QoS"` profile (lines 18 and 23);

3. the *deployed Things*, that retrieves the information concerning the user-managed devices, the `"things"` (lines 26–36) that are already deployed in the IoT system.

The depicted situation is almost self-explaining: the existing infrastructure includes, apart from the single `"cloud"` instance, two possible `"fog"` deployment locations, with different `"QoS"` profiles, and two link connections. Both of `"fog"` locations provide two `"things"` of type `"TemperatureSensor"` (lines 28 and 33). Remarkably, `"things"` are collected via their `" type"`, allowing the decoupling of the Thing from the service that controls it. This encoding is particularly useful in case one disposes of several Things exploiting the same pattern of interaction, and want to find the best suited among them in the application context, acting as a *Domain Name Service* (DNS), as advocated, for instance, by *Sensing as a Service* [101] or WoT store [113] paradigms.

In the assumption that the resulting deployment plan would strive to minimize the total IoT application `cost`, and that it does not take into account the QoS constraints. Here, we take advantage of the Zephyrus notation [2], derived from the Aeolus model [26], to indicate the objective functions to minimize. The final configuration for our example would consider to deploy the `"LogicEngine"` service into the `"SmallCloud"` location, and the `"Driver"` service into the `"TinyEdge"` location.

Contrarily, in a QoS-aware scenario, it is clear that the `"TinyEdge"` location does not satisfy the communication links QoS requirements, namely the `" latency"`, and thus resulting in the deployment of the `"Driver"` service into the `"SmallEdge"` location. Keeping in mind that the objective function to minimize remains the same ($\mathcal{M} = C$), we would indicate the QoS-aware constraints satisfiability as:

$$\mathcal{S} \models \mathcal{P}_{(SmallEdge,SmallCloud)} \leq \mathcal{R}_{(Driver,LogicEngine)}$$

where, $\mathcal{P}$ and $\mathcal{R}$ denotes, respectively, the QoS profile of the infrastructure and the QoS constraints of the application (in both cases it corresponds with the upload latency). Notably, the previous condition does not hold ($\mathcal{S} \not\models \mathcal{P} \leq \mathcal{R}$). Formally:

$$\mathcal{P}_{(TinyEdge,SmallCloud)} \not\leq \mathcal{R}_{(Driver,LogicEngine)}$$

Similarly to objective functions notation, we take inspiration from the Zephyrus specification language to define the deployment constraints (the full grammar is available in [1]).

In conclusion, the example depicted above highlights how, using proper linguistic abstractions, DevOps can express, in a declarative way, the application and the infrastructure descriptions, and non-trivial associated QoS constraints. Moreover, we show that having a model for reasoning about metrics minimization and deployment constraints satisfiability helps to exploit optimal configuration planning.

### 4.3.1 Contribution

To provide DevOps with an effective method for the integration of different IoT application deployments that guarantees components deployability, we used Zephyrus [68] to support the deployment planning in the IoT context. Concretely, Foehn integrates a specification language for software components and deployment locations that are relevant in the IoT scenario. Such specifications are communication links components, with their relative QoS requirements, and Fog deployment locations. Remarkably, Foehn supports the new specifications with the same linguistic abstractions provided by FogTorch [13] — a state-of-the-art tool for IoT application deployment planning.

Zephyrus does not provide a mechanism for the internal integration of extensions but, following a microservice-oriented approach, provides an easy-to-use REST interface. We then built a prototype, called Foehn[2], that embeds the Zephyrus functionalities and, in turn, provides a REST interface that exposes its service.

### 4.3.2 The Foehn Tool

In figure 4.1 we illustrate a general overview of the approach, introducing a refined software development procedure in the IoT context[3]. Each rectangle relates to a different phase of the procedure. In principle, our approach propose a refinement focusing at packaging time (at the center of the figure) for the DevOps loop. This refinement consists of introducing a deployability check-in between the packaging and the release phases. Foehn performs the check, which results in deployment feedback. If the deployment plan exists, the feedback is positive and it is optimal, and the response contains the plan itself. On the

[2]Foehn, Föhn, or Favonio is the Latin name given to the Zéphyros wind by Greeks.

[3]We refer to the software development procedure as the set of practices implementing the *Agile* lean methodology, namely *DevOps*, introduced in section 4.1

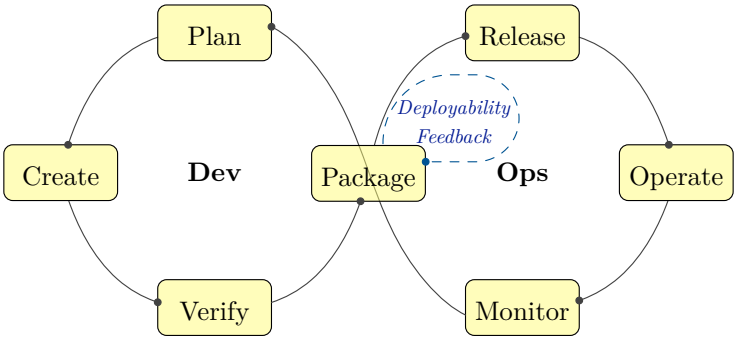


**Figure 4.1:** Refined version of the DevOps "infinite loop".

contrary, the feedback is negative for the given specification if no deployment plan exists. In this setting, the packaging has two possible outcomes. The first is to obtain a deployment configuration and, in case it is positive, move forward to the release phase; the second is to suspend the ongoing procedure for further refinement of the software components and infrastructure definition (the blue dotted arrow in between Package and Release rectangles). As shown

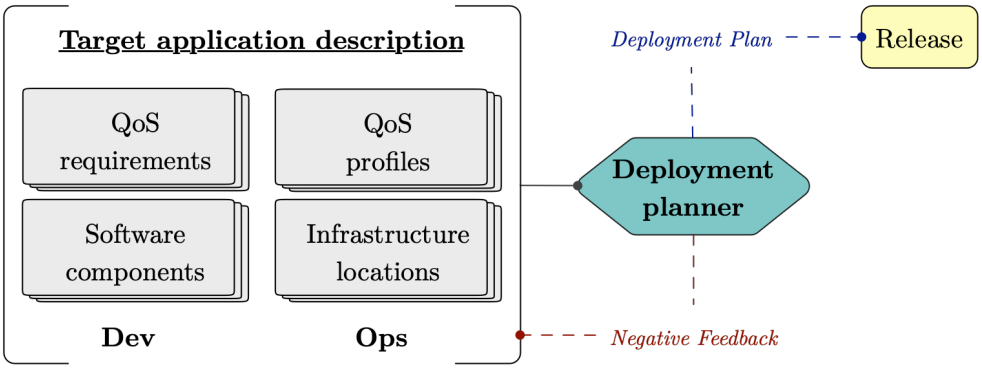


**Figure 4.2:** Focus of the deployability check phase.

in figure 4.2, to get a deployment feedback, Foehn needs a set of specifications for the IoT application and infrastructure.

Now, we describe the Foehn specification language, following the example at the beginning of this section. In the remainder of this section we use the *emphasized* notation to indicate terms that are part of the Foehn model. In contrast with the `"code"` notation to refer to the input specification language (in JSON format).

### The Application Description

Developers encode the verified software "components" ready to be released as microservices, along with the communication "links" desiderata. We assume a microservice-oriented architecture in this setting for secure deployment independence and smooth integration of functional requirements [68].

*Components* — "components"

First, we report that a unique "name" identifies a component, which contains in "softwares" the set of software dependency identifiers, in "hardware" the set of hardware requirements (in the form $\{harware_1, \cdots, hardware_n\}$), and in "things" the set of Thing identifiers which are controlled by this component. "hardware" contains an arbitrary number of consumable or non-consumable resources, each in the form "name":value.

*Links* — "links"

Similarly, we report that the target application communication link has the "from" and the "to" endpoint component identifiers and a list of link requirements. Every link requirement has the form $requirement_1, \cdots, requirement_n$, and each entry is identified by its unique name associated with its value — e.g. the definition "latency":5 has name "latency" associated with the value 5.

### The Infrastructure Description

IT staffs encode the application "infrastructure", that includes information about available Cloud instances, Fog "nodes", existing communication "links", and deployed "things".

*Nodes* — "nodes"

Both Cloud and Fog deployment locations share the same structure, are identified by a unique "name" and contain a set of "hardware" resources where the only difference to its application component counterpart is that each resource, has an associated cost, denoted following the scheme "<resource_name>_cost".

*Links* — "links"

Also an existing communication link is identified by the "from" and "to" endpoint location identifiers. Upload and download QoS profiles for this link are in the "download" and the "upload", respectively. "code" profile specification includes the "percentage" and the set of profiles in the form $\{profile_1, \cdots, profile_n\}$. The "qos" is identified by a "name" and has a *value* associated — e.g. "latency":5.

The *percentage* associated with each *qos* allows one to define a reliability model for the given connection — e.g. a communication link could provide 99% of the times the profile with latency 5, and 1% of the times the profile with infinite latency, we would write: `{ "qos": { "latency":5 }, "percentage":99 },{ "qos": { "latency":"INF"}, "percentage":1 }`.

*Things* — `"things"`

The description of each deployed Thing is identified by its unique `"name"`. It contains the `"type"` of this Thing and the identifier for the controller node `"fog_node"`.

### Optimization and Satisfiability Specification

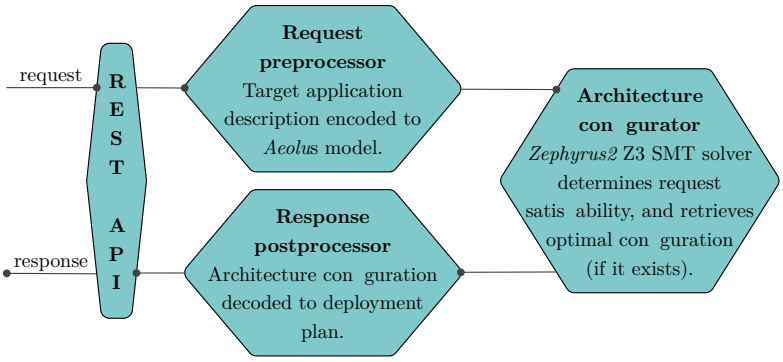


**Figure 4.3:** Pipeline overview of the Foehn deployment planner.

Once DevOps encoded both the application and the infrastructure description into the Foehn specification language, they can decide either to move to the deployment plan phase without specifying any additional information or to define a custom objective function. In the first case of no explicit objective function provided, Foehn will assume total IoT application cost minimization. In the second case, Foehn allows them to express a list of objective functions to minimize in the given order. The specification language for the arithmetic expression that expresses the function follows the specification grammar of Zephyrus, available in [1].

Finally, as summarized in figure 4.3 Foehn invokes Zephyrus solver instance [2], specifically with the Z3 [91] satisfiability-modulo-theories (SMT) solver option. To let Zephyrus interprets the input, Foehn encodes the application description and eventually user-defined objective functions into the Zephyrus Problem Specification Language. In case Zephyrus finds a feasible and optimal

solution — i.e. it exists a deployment plan that satisfies the QoS constraints and minimize the objective function — the deployment plan is collected. Since Foehn adds artificial specifications in the encoding process, the response from Zephyrus needs to be further processed to produce the final deployment plan. In particular, fictitious deployment locations were produced to handle communication links deployment, and we need to exclude them from the output architecture configuration.

At this stage, the positive feedback is ready to be delivered to the final user. Otherwise, the tool retrieves a negative response, meaning that it does not exist a feasible deployment plan for the IoT application in the infrastructure with the specified QoS profile.

## 4.4 Case Study

To exemplify our deployability check method, we consider a smart buildings IoT application where interaction among components happens both for environmental monitoring purpose and to trigger some actuation on the system.

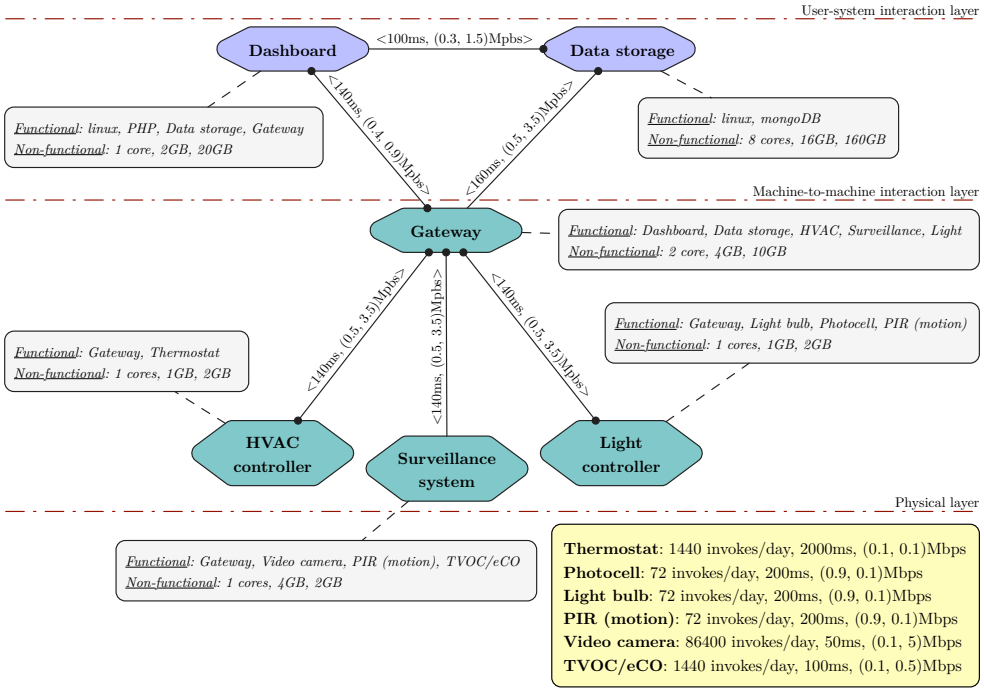


**Figure 4.4:** Overview of the smart building use case.

Notably, the proposed smart building application is an extension of the case study provided and discussed in section 2.4 and permits to take control over specific subsystems, monitor environmental variables — such as temperature, air quality, or luminosity and implement policies for managing potentially dangerous situations. Main differences with the case study in section 2.4 are:

- The JIoT orchestration in figure 2.6 spreads over multiple components depicted in figure 4.4: the `LogicEngine` service of figure 2.5 resides in the `Gateway` component. The `Driver`, being it an abstraction for a subsystem controller, it takes place in components *HVAC*, *Surveillance*, and *Light* of figure 4.4.
- The deployment of the system is no longer fixed, since that is the purpose of the current example — i.e. the `LogicEngine` service could be deployed both in a Cloud or Fog node.
- The things with names *ESP8266*, *Adafruit GA1A12S202 Analog Light Sensor*, *Hue Lamp*, and *SmartThings Motion Sensor* in described in section 2.4 map respectively with *Thermostat*, *Photocell*, *Light bulb*, and *PIR (motion)* of figure 4.4.
- Two more things compose the IoT system depicted in figure 4.4: a *video camera* for security footage recording and a *TVOC/eCO* gas sensor.

Since the purpose of the example presented in this work is related but different from the one devised in section 2.4, we illustrate the case study in a schematic representation in figure 4.4 and describe the IoT application in the following.

### 4.4.1 The Application

The case study consists of a smart building control system through a web-based dashboard that allows to monitor and act on three subsystems controllers: (i) the first being a heating, ventilation, and air conditioning (*HVAC*) controller, (ii) the second being a smart lighting controller hub, and (iii) the third being a surveillance monitoring subsystem.

The logic of the application spreads over the microservices (depicted as hexagons) and the low-level devices equipped with sensors or actuators. In figure 4.4 we identify three layers of interactions (from top to bottom): the *user-system*, the *machine-to-machine*, and the *physical*. For each layer, we present a snippet that uses the specification language to describe the IoT application and the infrastructure. We use the JSON format notation in compliance with the input format of Foehn.

### The User-System Interaction Layer

The user-system interaction layer consists of two independent microservices, the web-based dashboard service and the data storage service. The first allows infrastructure monitoring, data visualization through web interface running on a web server, e.g. *Apache* or *Nginx*, while the latter implements operations to enable data persistence through a database engine, e.g. *MySQL*, *MongoDB* [85], or *Redis* [111].

```
1 {
2   "name": "dashboard",
3   "software": [ "linux", "php" ],
4   "hardware": { "vm": "small" }
5 }
```

**Listing 4.3:** `Dashboard` component specification.

In listing 4.3 we report the dashboard software component, where the first element in the JSON is the identifier (line 2), the second is a list of software (line 3) prerequisites, and the third is the specification of the hardware non-functional requirements (line 4) of this component. Concerning the hardware virtual machine, we indicate the name of identifiers for Cloud-provided VMs, taken from the literature [14] and listing them in table 4.1. Note that, as we illustrate in figure 4.4, one can specify hardware capabilities as the list of resources of the VM, e.g. for the `small` flavor, `{"cores":1, "memory":2, "storage":20}`. The specification for the `DataStorage` microservice is similar.

| VM Identifier | Cores | Memory | Storage |
|---|---|---|---|
| `tiny` | 1 | 1 | 8 |
| `small` | 1 | 2 | 20 |
| `medium` | 2 | 4 | 40 |
| `large` | 4 | 8 | 80 |
| `xlarge` | 8 | 16 | 160 |

**Table 4.1:** VMs flavors adapted from OpenStack Platform [105].

### The Machine-to-Machine Interaction Layer

At this machine-to-machine middleware layer, it is possible to find a wide range of different devices types, from gateways and brokers to data centres and embedded servers, reaching the edge most level of unplugged personal devices

and microcomputers. The general goal for this entity is to (i) perform local processing of data when in a proximity network with sensors or actuators — e.g. measurements aggregation or filtering, and (ii) enable cross- and inter-layer communication to ensure interoperability among components.

The need for geographically exploiting data processing pushed computation closer to sensors and actuators, raising the interest in emerging Fog and Edge computing paradigms. Recently, Ferrández Pastor et al. [33] proposed to use the Edge and Fog deployed in IoT technologies with two main aims:

1. To ease the integration of interoperable services in automated and non-automated buildings (providing *integration*).

2. To allow the distribution of smart services between all of the building's subsystems (enabling *interoperability*).

Similarly to the dashboard component, the definition of a middleware contains an identifier, a list of software prerequisites, and a specification of hardware requirements. Moreover, we introduce a new field indicating the list of connected low-level devices — sensors and actuators — that interact with this component. Coherently, in listing 4.4 we show the HVAC component specification and the list of `things` identifiers (at line 5).

```
1 {
2   "name": "hvac_controller",
3   "hardware": { "cores": 1, "ram": 1, "storage": 10 },
4   "things": ["thermostat"]
5 }
```

**Listing 4.4:** Application HVAC component specification.

### The Physical Layer

Finally, the physical layer consists of software components statically deployed on power-constrained devices with communication capabilities — i.e. micro-controllers equipped with sensors and actuators able to connect to a proximity network.

In our case study, we selected six different things exploiting both sensing and actuation functionalities. We present them below, grouped in three blocks, that represent the subsystem they belong (as shown in figure 4.4).

1. The thermostat, as many commercial solutions for home automation, e.g. the *Google Nest Thermostat* or *Netatmo*, consists of a thermostat equipped with temperature and humidity sensors and an actuator to control the HVAC system.

2. The light monitoring and control system consists of a photocell sensor, a light bulb, and a motion sensor (PIR), such as the one used by *Philips Hue* or *Samsung SmartThing* commercial solutions where ad-hoc configured hubs control IP-less devices by using *ZigBee* or Bluetooth Low Energy (*BLE*) technologies.

3. Last, the surveillance monitoring system, deployed for threats detection and air-quality assessment, consists of a wireless connected security camera — e.g. *Nest Cam* or *Arlo Ultra* — and a gas detector group of sensors — e.g. *Nest Protect Smoke & CO Alarm* or *Adafruit Air Quality* deployed on *Arduino* micro-controller.

Note that we selected all the commercial solutions described above according to their ability to provide public interfaces (in the majority of cases REST-like *API*) to access device state. Moreover, state-of-the-art solutions exist to integrate this API in pre-existing heterogeneous systems and thus permitting to build interoperable cross-platform applications [51, 72], thus allowing this system to be easily deployable in a real-world scenario.

```
1 {
2   "name": "thermostat",
3   "qos": {
4     "to": { "latency": 2000, "bandwidth": 0.1 },
5     "from": { "latency": 2000, "bandwidth": 0.1 }
6   },
7   "invokes": 1440
8 }
```

**Listing 4.5:** Application Thermostat thing specification.

We report the thing component, the first element in the JSON is an identifier, the second the QoS requirements for the communication link with the controller, and the third a value representing the invocation rate for this component. In listing 4.5 we show the JSON specification for the thermostat component that, on the one hand, senses and actuates on temperature and humidity variables and, on the other hand, it communicates with the HVAC controller. Between lines 3 and 6, we specify the QoS, in this example, represented by latency and bandwidth for the incoming communications (line 4), and the outgoing communications (line 5). Last, we define the invocation rate per-day (line 7) that is, the maximum number of times that the controller will contact the thing, 1440 in one day or one time for every minute.

```
1 {
```

```
2   "from": "smart_gateway",
3   "to": "dashboard",
4   "latency": 140,
5   "up": 0.9,
6   "down": 0.4
7 }
```

**Listing 4.6:** Application communication link specification.

In listing 4.6 we report the link definition between the two endpoints `"smart_gateway"` and `"dashboard"`, defined at lines 2 and 3. QoS requirements are specified at lines 4–6 and comprise the solely latency, upload, and download bandwidth.

### 4.4.2 The Deployment Infrastructure

```
1 {
2   "name": "cloud_node",
3   "hardware": {
4     "cores": 2,
5     "cores_cost": 2,
6     "ram": 4,
7     "ram_cost": 3,
8     "storage": 10,
9     "storage_cost": 1
10  },
11  "software": [["linux", 0], ["php", 0], ["mongoDB", 45]],
12  "x": 52.195097,
13  "y": 3.0364791
14 }
```

**Listing 4.7:** Infrastructure Cloud node deployment location specification.

In listing 4.7 we report the node definition for a Cloud instance. Lines 3–10 indicates `"hardware"` capabilities for this node and relative cost models. `"software"` is the list of yet provided software, and `"x"` and `"y"` indicates the geo-spatial position of the Cloud provider.

```
1 {
2   "name": "fog_node",
3   "hardware": {
4     "cores": 2,
5     "cores_cost": 4,
6     "ram": 4,
7     "ram_cost": 5,
8     "storage": 32,
```

```
9       "storage_cost": 3
10    },
11    "software": [["linux", 0], ["php", 0], ["mysql", 15]],
12    "x": 44.497106,
13    "y": 11.355990
14  }
```

**Listing 4.8:** Infrastructure Fog node deployment location specification.

In listing 4.8 we report the Fog node location that, similarly to the `"cloud_node"` contains information concerning supported `"hardware"`. `"software"`, and spatial position.

```
1  {
2    "from": "fog_node",
3    "to": "cloud_node",
4    "download": [
5      {
6        "qos": { "latency": 40, "bandwidth": 5.25 },
7        "percentage": 0.98
8      },
9      {
10       "qos": { "latency": "INF", "bandwidth": 0 },
11       "percentage": 0.02
12     }
13   ],
14   "upload": [
15     {
16       "qos": { "latency": 40, "bandwidth": 2.25 },
17       "percentage": 0.98
18     },
19     {
20       "qos": { "latency": "INF", "bandwidth": 0 },
21       "percentage": 0.02
22     }
23   ]
24 }
```

**Listing 4.9:** Infrastructure communication link specification.

In listing 4.9 we report the link specification and QoS profiling of the existing connections. Lines 4–13 and 14–23 contains respectively, the definition `"download"` and `"upload"` QoS profiles. We highlight the practical usage of percentage in reliability modelling (lines 7 and 11 first, lines 17 and 21 then). In the Foehn model `"percentage"` must sum to 1, string value that can not be cast to a digit counts as the infinite value (maximum representable number depending on the architecture).

```
1 {
2   "name": "thermostat",
3   "type": "temperature_actuator",
4   "fog_node": "fog_node"
5 }
```

**Listing 4.10:** Infrastructure Thermostat thing specification.

In listing 4.10 we report the deployed thing specification for the `"thermostat"`. At line 3 it is denoted the `"type"` for the current thing, and at line 4 the linked fog node specify the unique `"fog_node"` in charge of interacting with this thing.

### 4.4.3 The Optimal Deployment Plan

```
1 {
2   "locations": {
3     "edge_node_2": {
4       "0": {
5         "surveillance_system": 1,
6         "videocamera": 1,
7         "tvoc": 1
8       }
9     },
10    "edge_node_1": {
11      "0": {
12        "light_controller": 1,
13        "motion": 1,
14        "light_bulb": 1,
15        "photocell": 1
16      }
17    },
18    "fog_node": {
19      "0": { "hvac_controller": 1, "thermostat": 1 }
20    },
21    "cloud_node": {
22      "2": { "smart_gateway": 1 },
23      "1": { "dashboard": 1, "linux": 1, "php": 1 },
24      "0": { "data storage": 1, "mongodb": 1 }
25    }
26  }
27 }
```

**Listing 4.11:** Optimal deployment plan for smart building IoT application.

In listing 4.11 we report the final optimal system configuration for the IoT application. Foehn retrieves a JSON formatted list of deployment locations, where selected nodes contain, with a progressive identifier, the services chosen

for deployment. We note that the `"gateway"` is planned for deployment in one of the cloud virtual machines, similarly to the `"dashboard"` and `"data_storage"`.

## 4.5 Discussion

In DevOps practice, one the main challenges is to provide both developers and IT staff of tools capable of automatizing the software development procedure. The stress on automation is particularly important when it comes to the release phase, immediately after the application has been packaged. Developers continuously change QoS requirements during development, without the guarantee that the infrastructure satisfies the new constraints. On the other hand, IT staff need to update the infrastructure and, consequently, its QoS profile, having no clue of the possible drawbacks on the application.

To move to the actual deployment phase, both groups need to check on the compatibility of their partial specifications. In principle, one can do that by hand, that is, given the infrastructure QoS profile, to manually check the software components deployability. Although this is feasible for small-size projects, in IoT systems, our area of interest, where many components with different functional and non-functional requirements interact among each other exposing both teams to a time-consuming, error-prone and costly practice.

We adopt a linguistic approach to define a specification language and use a state-of-the-art optimizer to solve the Constraint Optimization Problems (COP). We prototype our solution into a tool for QoS-aware optimal deployment planning of IoT applications. We called it Foehn, freely available at [40] under the GNU GPL v3.0 license. Foehn exploits the Microservices-Oriented Computing paradigm to provide a light-weight and easy-to-integrate tool that exposes a REST interface.

We validate our approach and the Foehn tool trough a use case from a smart building automation scenario. Using the proposed specification language to describe the IoT application we can produce an optimal deployment plan.

We believe that the work most similar to ours, from which we draw inspiration, is FogTorch[13]. Nevertheless, differently from our tool, FogTorch does not guarantee the optimality of the deployment plan found and considers only the complete application running cost as the objective function for the optimizer. Since our tool receives input containing rules expressed in the Aeolus specification language, we can generalize the optimization process to more complex tasks. Moreover, in case we do not retrieve any deployment configuration, our approach guarantees that the solution is not deployable for the given infrastructure.

Other approaches such as [112] or [52] do not model network desiderata and

low-level devices requests. Also, they assume tree-like infrastructure topologies are not taking into account shared resources among edge devices and Fog nodes. Furthermore, [52] does not consider QoS requirements among the parameters defining the set of eligible deployments.

Our approach leverages the work done in [68] for software components deployment using the Zephyrus tool.

Zephyrus has been already used in different deployment tools [28, 47, 38, 11] all addressing deployability in the Cloud — i.e. without considering IoT or Fog locations — and, differently from our approach, without modelling network links or QoS constraints between final locations.

Since Zephyrus is originated from the Aeolus model [28, 26, 27], a general component-based system that relies on finite-state automata [16] to model Cloud applications, the encoding of Foehn specification language into the Zephyrus one has been straightforward.

The implementation consisted of two *Python* classes, one in charge of decode/encode from the Foehn language to the Zephyrus one and the second acting as a proxy gateway for incoming and outgoing requests.

Foehn does not integrate with Zephyrus optimizer but rather wraps it to be able to use a customized specification language. In this sense, it would be more natural to provide those abstractions as part of the specification language. At the moment, tested configuration for medium-sized IoT applications took more then thirty seconds to retrieve the architecture configuration (if existing). This make Foehn unfits for real-time decision-making high scenarios.

Explore new solving paradigms, such as the combination of symbolic and learning-based artificial intelligence, would raise a new exciting perspective to both fields of research. A possible solution to the latter mentioned future direction is the exploitation of local search algorithms to quickly retrieve a feasible (yet non-optimal) solution.

# Chapter 5

# Conclusion

IoT systems are large-scale modern systems that advocates for multi-layered, distributed software platforms, each adopting its protocols stack and data formats. The development of IoT application suffers the same problem of every large-scale distributed application. The lack of holistic approaches to software development makes it difficult to integrate the solutions tailored explicitly for the sub-systems. Contrarily, adopting a holistic approach, permits to deal with solutions covering the entire development life-cycle.

Among the many integration challenges, our focus narrows to the most IoT-related interoperability issues of systems heterogeneity handling. We distinguish between two different phases of the software development procedure that most suffer from the absence of tools addressing integration problems. On the one hand, there is the need to support developers with technologies that enable protocol- (challenge $C1$) and data- (challenge $C2$) agnostic programming. On the other hand, there exists the urgency to provide IT staff with proper tools to automate the deployability check of IoT applications in the existing architecture and to test the different QoS profiles deployability (challenge $C3$).

Programming IoT applications involves the integration of many layers of the communication stack, from transport to application protocols, reaching data-format representations. Deploying IoT applications involves the strategic selection of suitable architecture configurations in distributed, heterogeneous infrastructures. Once again, we stressed on the necessity of a holistic approach dealing with $C1$, $C2$, and $C3$ in contrast with solutions that advocates for the adoption of tailored standards.

We addressed $C1$, $C2$, and $C3$ by adopting a linguistic approach based on Service-Oriented Computing, leveraging the work done in SOAs. In particular, to accomplish $C1$, we extended the syntax of an existing programming language, Jolie, increasing its expressive power, and build the JIoT interpreter (chapter 2). Then, after the formalization of an appropriate variant of the MongoDB Aggregation Framework, named TQuery, we integrated the operators into the JIoT interpreter to front $C2$ (chapter 3). Finally, addressing $C3$, we proposed

a tool, named Foehn, that allows one to find the optimal deployment plan by using a state-of-the-art configuration optimizer and thus satisfy the QoS constraints of the IoT application while minimizing user-defined metrics, such as the total running cost or the network inter-nodes distance (chapter 4).

In chapter 2, we presented high-level concepts for the general implementation of interoperable IoT systems using linguistic solutions. We integrated the CoAP and MQTT application protocols and the UDP transport protocol into the Service-Oriented Jolie language. Doing so, we also faced the mapping of the publish/subscribe message pattern of MQTT into the request-response original pattern of Jolie.

In chapter 3, we implemented TQuery in JIoT, offering automatic data format heterogeneity conversion by-construction and performance-oriented in-memory data manipulation. These factors allow to separate input and output data formats from the data handling logic and manage data structure within the application memory. Hence, we provided programmers with a single, consistent interface for data manipulation through the TQuery operators (match, unwind, project, group, and lookup), on any data-format supported by JIoT (XML, JSON, raw).

In chapter 4, we implemented Foehn, a tool that allows one to find the optimal deployment plan of IoT distributed applications. When it comes to IoT systems deployment, proper specification language abstractions are needed to describe both the application and the infrastructure. Only thanks to this abstractions both developers and IT staff can cooperate and produce an appropriate releasing plan, that match QoS constraints with actual QoS profiles. Hence, we included in our model communication links QoS constraints specifications, geospatial information for Cloud, Fog, and Things, as well as user-defined objective function description. We implemented all of the above in a tool that parses the specifications given in JSON format and instantiates a state-of-the-art SMT-based optimizer to find the final optimal QoS-aware deployment plan of the IoT application.

In chapter 2 and chapter 3, we tackle $C1$ and $C2$ and build a specific language, with proper abstractions, to offer a single linguistic domain to integrate Cloud, Edge and Things computing seamlessly. While we proposed a dedicated language, comparable approaches provide API specifications [131], or they faced the problem from a framework perspective, thus providing chains of tools, each addressing a specific sub-system technology [70]. Differently, we extended a language (its expressiveness power) with the main benefit of letting programmers directly work at any level of the IoT application stack, instead of developing specific knowledge on the usage of a given framework. As an added value, JIoT is a high-level, declarative language, not intended for modelling

a specific layer of the system but built to reason on the (service-Oriented) architecture as a whole. We gave a notable example of such expressiveness power for the language-based solution, in the IoT applications ephemeral data handling scenario, where we conducted a comparison study that shows how our solution provided a substantial increase in performance (averagely five times faster).

In chapter 4, we tackled C3 and build a tool to provide DevOps with a declarative IoT application description language, and an automatic tool that can be easily integrated into existing DevOps pipelines.

In conclusion, in this dissertation, we moved a step towards a comprehensive, holistic solution that addressed software development challenges in the IoT context by adopting a linguistic approach.

From work done in this dissertation, it is possible to draw at least two directions in future scenarios. First, explore the consolidation of the tools proposed in this dissertation, moving them to an advanced Technology Readiness Level (TLR) stage, by prototyping the solutions in industrial settings, and thus fostering the technological transfer. Finally, the last direction concerns the augmentation of the expressive power of the proposed language-based solutions. For instance, in JIoT, to investigate the integration of more transport, application protocols, and data formats in order to extend the usability of the language. Similarly, one could explore the further generalization of the Foehn model, to include more complex deployment scenarios.

Without any claim to be complete in any of the above discussed matters, we believe that the adoption of linguistic approaches, in particular, their Service-Oriented fashion, can foster the advancement of the Internet of Things to a new stage of a fully integrated ecosystem.

# Bibliography


[1] Erika Ábrahám, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer, and Jacopo Mauro. *Zephyrus2*. 2016. URL: https://bitbucket.org/jacopomauro/zephyrus2 (visited on August 31, 2019) (cit. on pp. 75, 78).

[2] Erika Ábrahám, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer, and Jacopo Mauro. "Zephyrus2: On the Fly Deployment Optimization Using SMT and CP Technologies". In: *Dependable Software Engineering: Theories, Tools, and Applications*. Springer International Publishing, 2016, pp. 229–245 (cit. on pp. 74, 78).

[3] Ahmed B. Altamimi and Rabie A. Ramadan. "Towards internet of things modeling: a gateway approach". In: *Complex Adaptive Systems Modeling* (2016) (cit. on p. 71).

[4] Apache Software Foundation. *CouchDB*. 2018. URL: https://couchdb.apache.org/ (visited on August 31, 2019) (cit. on pp. 47, 49).

[5] ArangoDB. *ArangoDB*. 2014. URL: https://www.arangodb.com (visited on August 31, 2019) (cit. on p. 49).

[6] Stephanie B. Baker, Wei Xiang, and Ian Atkinson. "Internet of Things for Smart Healthcare: Technologies, Challenges, and Opportunities". In: *IEEE Access* 5 (2017), pp. 26521–26544 (cit. on p. 46).

[7] Andrew Banks and Rahul Gupta. *MQTT Version 3.1.1*. Oasis standard. Oasis, 2014. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/ (cit. on pp. 10, 16).

[8] *The bIoTope Project*. 2017. URL: http://www.biotope-project.eu/ (visited on August 31, 2019) (cit. on p. 10).

[9] Carsten Bormann. *CoAP website*. 2016. URL: http://coap.technology/ (visited on August 31, 2019) (cit. on pp. 10, 14, 16).

[10] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, and Guohui Xiao. "Expressivity and Complexity of MongoDB Queries". In: *CEUR Workshop Proceedings*. Vol. 2161. Schloss Dagstuhl - LZI, 2018, 9:1–9:23 (cit. on pp. 50, 54, 59, 65).

[11] Mario Bravetti, Saverio Giallorenzo, Jacopo Mauro, Iacopo Talevi, and Gianluigi Zavattaro. "Optimal and Automated Deployment for Microservices". In: *Lecture Notes in Computer Science*. Vol. 11424 Lncs. 2019, pp. 351–368 (cit. on p. 88).

[12] Brian Krebs. *Extortionists Wipe Thousands of Databases, Victims Who Pay Up Get Stiffed*. 2017. URL: https://krebsonsecurity.com/2017/01/extortionists-wipe-thousands-of-databases-victims-who-pay-up-get-stiffed/ (visited on August 31, 2019) (cit. on p. 47).

[13] Antonio Brogi and Stefano Forti. "QoS-aware deployment of IoT applications through the fog". In: *IEEE Internet of Things Journal* 4.5 (2017), pp. 1–8 (cit. on pp. 68–70, 72, 75, 87).

[14] Antonio Brogi, Stefano Forti, and Ahmad Ibrahim. "Predictive Analysis to Support Fog Application Deployment". In: *Fog and Edge Computing: Principles and Paradigms*. 2018. Chap. 9, pp. 1–40 (cit. on pp. 68, 70, 72, 81).

[15] Antonio Brogi et al. "SeaClouds: A European Project on Seamless Management of Multi-cloud Applications". In: *SIGSOFT Softw. Eng. Notes* (2014) (cit. on p. 71).

[16] Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, eds. *Automata, Languages and Programming*. Vol. 3580. 2. March 2005, p. 110 (cit. on p. 88).

[17] Fay Chang et al. "Bigtable: A distributed storage system for structured data". In: *Tocs* 26.2 (2008), p. 4 (cit. on p. 49).

[18] James Cheney, Sam Lindley, and Philip Wadler. "A practical theory of language-integrated query". In: *ACM SIGPLAN Notices* 48.9 (2013), pp. 403–416 (cit. on pp. 47–48).

[19] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. "On the journey to continuous deployment: Technical and social challenges along the way". In: *Information and Software Technology* 57.1 (January 2015), pp. 21–31 (cit. on p. 68).

[20] OASIS Committee. *Topology and Orchestration Specification for Cloud Applications (TOSCA)–Committee Specification 01*. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.pdf (visited on August 31, 2019) (cit. on p. 71).

[21] World Wide Web Consortium et al. "JSON-LD 1.0: a JSON-based serialization for linked data". In: (2014) (cit. on pp. 37, 47).

[22] Iván Corredor, Eduardo Metola, Ana M. Bernardos, Paula Tarrío, and José R. Casar. “A lightweight web of things open platform to facilitate context data management and personalized healthcare services creation”. In: *International Journal of Environmental Research and Public Health* 11.5 (2014), pp. 4676–4713 (cit. on p. 12).

[23] Ravi Das and Preston de Guise. *Amazon Web Services*. April 2019. (Visited on August 31, 2019) (cit. on p. 69).

[24] David M. Eddy. “Evidence on the Costs and Benefits of Health Information Technology The”. In: *testimony before Congress*. Vol. 24. 1990 (cit. on p. 46).

[25] Hui Deng, Jiguo Yu, Dongxiao Yu, Guangshun Li, and Baogui Huang. “Heuristic Algorithms for One-Slot Link Scheduling in Wireless Sensor Networks under SINR”. In: *International Journal of Distributed Sensor Networks* 11.3 (March 2015), p. 806520 (cit. on p. 71).

[26] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. “Aeolus: A component model for the cloud”. In: *Information and Computation* 239 (December 2014), pp. 100–121 (cit. on pp. 68, 71, 74, 88).

[27] Roberto Di Cosmo et al. “Automatic application deployment in the cloud: From practice to theory and back”. In: *Leibniz International Proceedings in Informatics, LIPIcs* 42 (2015), pp. 1–16 (cit. on p. 88).

[28] Roberto Di Cosmo et al. “Automatic deployment of services in the cloud with aeolus blender”. In: *Lecture Notes in Computer Science*. Vol. 9435. 2015, p. 397 (cit. on p. 88).

[29] LLC DigitalOcean. *Digital Ocean*. 2019. URL: https://digitalocean.com/ (visited on August 31, 2019) (cit. on p. 69).

[30] William H. Dutton, Everett M. Rogers, and Suk Ho Jun. “Diffusion and Social Impacts of Personal Computers”. In: *Communication Research* 14.2 (April 1987), pp. 219–250 (cit. on p. 4).

[31] Tom Ellis. *Opaleye*. 2014. URL: https://github.com/tomjaguarpaw/haskell-opaleye (visited on August 31, 2019) (cit. on p. 49).

[32] Thomas Erl. *SOA: Principles of Service Design*. 2005, pp. 116–119 (cit. on pp. 7, 10, 39).

[33] Francisco Javier Ferrández Pastor, Higinio Mora, Antonio Jimeno Morenilla, and Bruno Volckaert. “Deployment of IoT Edge and Fog Computing Technologies to Develop Smart Building Services”. In: *Sustainability* 10.11 (October 2018), p. 3832 (cit. on p. 82).

[34] Roy T Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000 (cit. on p. 19).

[35] The Apache Software Foundation. *Apache Hadoop*. 2006. URL: https://hadoop.apache.org (visited on August 31, 2019) (cit. on p. 49).

[36] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications". In: *IEEE Communications Surveys and Tutorials* 17.4 (2015), pp. 2347–2376 (cit. on pp. 10, 17, 44).

[37] Mark Fussel. "Foundations of object-relational mapping". In: *ChiMu Corporation* (1997) (cit. on p. 49).

[38] Maurizio Gabbrielli, Saverio Giallorenzo, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. "Self-Reconfiguring microservices". In: *Lecture Notes in Computer Science*. Vol. 9660. Lecture Notes in Computer Science. Springer, 2016, pp. 194–210 (cit. on pp. 7, 43, 69, 88).

[39] Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Stefano Pio Zingaro. *JIoT web site*. 2019. URL: http://www.cs.unibo.it/projects/jolie/jiot.html (visited on August 31, 2019) (cit. on pp. 11, 35, 50).

[40] Maurizio Gabbrielli, Jacopo Mauro, and Stefano Pio Zingaro. *Foehn repository on GitHub*. 2019. URL: https://github.com/spaces-team/foehn (visited on August 31, 2019) (cit. on pp. 68, 87).

[41] Maria Ganzha, Marcin Paprzycki, Wieslaw Pawlowski, Pawel Szmeja, and Katarzyna Wasielewska. "Semantic technologies for the IoT - An Inter-IoT perspective". In: *Proceedings of the IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI* (2016), pp. 271–276 (cit. on p. 12).

[42] Nishant Garg. *Apache Kafka*. Packt Publishing Ltd, 2013 (cit. on p. 10).

[43] Lars George. *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc.", 2011 (cit. on p. 49).

[44] Saverio Giallorenzo, Fabrizio Montesi, Larisa Safina, and Stefano Pio Zingaro. *TQuery GitHub repository*. 2019. URL: https://github.com/jolie/tquery (visited on August 31, 2019) (cit. on p. 50).

[45] Ivan Gojmerac, Peter Reichl, Ivana Podnar Žarko, and Sergios Soursos. "Bridging IoT islands: the symbIoTe project". In: *Elektrotechnik und Informationstechnik* 133.7 (2016), pp. 315–318 (cit. on pp. 10, 12).

[46] Matteo Golfarelli, Stefano Rizzi, and Andrea Proli. “Designing what-if analysis: Towards a methodology”. In: *Proceedings of the ACM International Workshop on Data Warehousing and OLAP, DOLAP*. 2006 (cit. on p. 69).

[47] Stijn de Gouw, Jacopo Mauro, Behrooz Nobakht, and Gianluigi Zavattaro. “Declarative elasticity in ABS”. In: *Lecture Notes in Computer Science*. Vol. 9846. Springer International Publishing, 2016, pp. 118–134 (cit. on p. 88).

[48] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. “Internet of Things (IoT): A vision, architectural elements, and future directions”. In: *Future Generation Computer Systems* 29.7 (September 2013), pp. 1645–1660 (cit. on pp. 6, 10).

[49] Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. “Dynamic Error Handling in Service Oriented Applications”. In: *Fundamenta Informaticae* 95.1 (2009), pp. 73–102 (cit. on p. 33).

[50] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. “SOCK: A Calculus for Service Oriented Computing”. In: *Lecture Notes in Computer Science*. Vol. 4294 Lncs. Springer. Springer International Publishing, 2006, pp. 327–338 (cit. on p. 33).

[51] Dominique Guinard. “A Web of Things Application Architecture - Integrating the Real-World into the Web”. In: *PhD th., ETH Zurich* 19891 (2011), p. 220 (cit. on pp. 12, 18, 83).

[52] 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”. In: *Software - Practice and Experience*. 2017 (cit. on pp. 71, 87–88).

[53] Dave Harrison and Knox Lively. *Achieving DevOps*. Apress, 2019 (cit. on p. 68).

[54] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010, p. 463 (cit. on p. 68).

[55] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. “MQTT-S–A publish/subscribe protocol for Wireless Sensor Networks”. In: *Comsware*. IEEE. 2008, pp. 791–798 (cit. on p. 32).

[56] Google Inc. *LevelDB*. 2017. URL: http://leveldb.org (visited on August 31, 2019) (cit. on p. 49).

[57] MongoDB Inc. *MongoDB*. 2007. URL: https://www.mongodb.com (visited on August 31, 2019) (cit. on p. 49).

[58] Michael Jang. *Linux Annoyances for Geeks: Getting the Most Flexible System in the World Just the Way You Want It*. O'Reilly Media, 2006 (cit. on p. 47).

[59] Jolie Developers Team. *Jolie Website*. 2018. URL: https://www.jolie-lang.org/ (visited on August 31, 2019) (cit. on p. 50).

[60] *Jolie website*. 2019. URL: http://jolie-lang.org (visited on August 31, 2019) (cit. on p. 10).

[61] Josh Juneau. *RESTful Web Services*. ACM Press, 2018, pp. 613–653 (cit. on pp. 10, 38).

[62] Adrian Klein, Fuyuki Ishikawa, and Shinichi Honiden. "Towards network aware service composition in the cloud". In: *Proceedings of the 21st international conference on World Wide Web*. New York, New York, USA: ACM Press, 2012, p. 959 (cit. on p. 71).

[63] Ivan Lanese, Luca Bedogni, and Marco Di Felice. "Internet of things: A process calculus approach". In: *Proceedings of the ACM Symposium on Applied Computing*. ACM Press, 2013, pp. 1339–1346 (cit. on p. 7).

[64] Edward A. Lee. "Cyber Physical Systems: Design Challenges". In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, May 2008, pp. 363–369 (cit. on p. 5).

[65] Barry M Leiner et al. "A Brief History of the Internet". In: *ACM SIGCOMM Computer Communication Review* 39.5 (2009), pp. 22–31 (cit. on p. 4).

[66] Fei Li, Michael Vogler, Markus Claessens, and Schahram Dustdar. "Towards Automated Internet of Things Application Deployment by a Cloud-Based Approach". In: *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications, ICSOCA*. IEEE, December 2013, pp. 61–68 (cit. on p. 71).

[67] Lixing Li, Zhi Jin, Ge Li, Liwei Zheng, and Qiang Wei. "Modeling and Analyzing the Reliability and Cost of Service Composition in the IoT: A Probabilistic Approach". In: *2012 IEEE 19th International Conference on Web Services*. IEEE, June 2012, pp. 584–591 (cit. on p. 71).

[68] Debasmita Lohar, Anudeep Dunaboyina, and Dibyendu Das. *Dependable Software Engineering. Theories, Tools, and Applications*. Vol. 10606. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 315–322 (cit. on pp. 68, 75, 77, 88).

[69] Jiakang Lu et al. "The smart thermostat". In: *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems - SenSys '10*. ACM Press, 2010, p. 211 (cit. on pp. 59–60, 62).

[70] Alessandro Ludovici, Anna Calveras, and Anna Calveras. "A proxy design to leverage the interconnection of CoAP wireless sensor networks with web applications". In: *Sensors (Switzerland)* 15.1 (2015), pp. 1217–1244 (cit. on pp. 12, 90).

[71] Meng Ma, Ping Wang, and Chao-Hsien Chu. "Data management for internet of things: Challenges, approaches and opportunities". In: *2013 IEEE International conference on green computing and communications and IEEE Internet of Things and IEEE cyber, physical and social computing*. IEEE. 2013, pp. 1144–1151 (cit. on p. 47).

[72] Tim A Majchrzak and Tor-morten Grønli Eds. *Towards Integrated Web, Mobile, and IoT Technology*. Vol. 347. Lecture Notes in Business Information Processing. Springer International Publishing, 2019 (cit. on pp. 6, 83).

[73] Norman Maurer. *The Netty project*. 2003. URL: https://netty.io (visited on August 31, 2019) (cit. on p. 31).

[74] Norman Maurer and Marvin Wolfthal. *Netty in Action*. Manning Publications, 2016 (cit. on pp. 32, 34).

[75] Jacopo Mauro and Gianluigi Zavattaro. "On the Complexity of Reconfiguration in Systems with Legacy Components". In: *Lecture Notes in Computer Science*. 2015, pp. 382–393 (cit. on p. 71).

[76] Dinesh P Mehta and Sartaj Sahni. *Handbook of data structures and applications*. Chapman and Hall/CRC, 2004 (cit. on pp. 46–47).

[77] Erik Meijer, Brian Beckman, and Gavin Bierman. "Linq: reconciling object, relations and xml in the. net framework". In: *Sigmod*. Acm. 2006, pp. 706–706 (cit. on pp. 48–49).

[78] Andrea Melis, Marco Prandini, Saverio Giallorenzo, and Franco Callegati. "Insider Threats in Emerging Mobility-as-a-Service Scenarios". In: *Proceedings of the Hawaii International Conference on System Sciences, HICSS*. AIS Electronic Library (AISeL), 2017 (cit. on p. 43).

[79] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux Journal* 2014.239 (2014), p. 2 (cit. on p. 41).

[80] Microsoft. *Microsoft Azure*. 2019. URL: https://azure.microsoft.com/ (visited on August 31, 2019) (cit. on p. 69).

[81] Milan Milenkovic. "A Case for Interoperable IoT Sensor Data and Metadata Formats". In: *Ubiquity* 2015.November (2015), pp. 1–7 (cit. on p. 10).

[82] E. Mingozzi, G. Tanganelli, and C. Vallati. "CoAP proxy virtualization for the web of things". In: *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*. Vol. 2015-Febru. February. IEEE Computer Society, 2015, pp. 577–582 (cit. on p. 12).

[83] Sudip Misra and Subhadeep Sarkar. "Theoretical modelling of fog computing: a green computing paradigm to support IoT applications". In: *IET Networks* 5.2 (2016), pp. 23–29 (cit. on p. 71).

[84] MongoDB Inc. *MongoDB Aggregation Framework*. 2018. URL: https://docs.mongodb.com/manual/aggregation/ (visited on August 31, 2019) (cit. on pp. 50, 65).

[85] MongoDB Inc. *MongoDB Website*. 2018. URL: https://www.mongodb.com/ (visited on August 31, 2019) (cit. on pp. 47, 81).

[86] Fabrizio Montesi. "Process-aware web programming with Jolie". In: *Science of Computer Programming*. Vol. 130. ACM Press, 2016, pp. 69–96 (cit. on pp. 10, 21).

[87] Fabrizio Montesi and Marco Carbone. "Programming services with correlation sets". In: *Lecture Notes in Computer Science*. Vol. 7084 Lncs. Springer, 2011, pp. 125–141 (cit. on p. 33).

[88] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. "JOLIE: a Java Orchestration Language Interpreter Engine". In: *Electronic Notes in Theoretical Computer Science* 181.1 (June 2007), pp. 19–33 (cit. on p. 10).

[89] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. "Service-Oriented Programming with Jolie". In: *Web Services Foundations* (September 2014), pp. 81–107 (cit. on pp. 10, 15, 38, 50).

[90] Menno Mostert, Annelien L. Bredenoord, Monique C.I.H. Biesaart, and Johannes J.M. Van Delden. “Big Data in medical research and EU data protection law: Challenges to the consent or anonymise approach”. In: *European Journal of Human Genetics* 24.7 (2016), pp. 956–960 (cit. on p. 46).

[91] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 4963. 2008, pp. 337–340 (cit. on p. 78).

[92] MQTT community. *MQTT website*. 2014. URL: http://mqtt.org (visited on August 31, 2019) (cit. on pp. 10, 16).

[93] Rouhollah Nabati and Samira Taheri. “the Internet of Things (IoT) a Survey”. In: *Turkish Online Journal of Design, Art and Communication* 6.Jlyspcl (2016), pp. 725–739 (cit. on pp. 5, 10).

[94] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. “Everything Old is New Again: Quoted Domain-specific Languages”. In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. Pepm ’16. Acm, 2016, pp. 25–36 (cit. on p. 4).

[95] *NCoAP A JAVA implementation of CoAP*. 2018. URL: https://github.com/okleine/nCoAP (visited on August 31, 2019) (cit. on p. 33).

[96] Julie L. Newcomb, Satish Chandra, Jean Baptiste Jeannin, Cole Schlesinger, and Manu Sridharan. “IoTa: A calculus for internet of things automation”. In: *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, SIGPLAN*. 2017, pp. 119–133 (cit. on p. 7).

[97] Niall Gallagher. *CQEngine - Java Collection SQL-like Query Engine*. 2018. URL: https://github.com/npgall/cqengine (visited on August 31, 2019) (cit. on pp. 48–49).

[98] Eva Nieuwdorp. “The Pervasive discourse: An analysis”. In: *Computers in Entertainment* 5.2 (April 2007), p. 13 (cit. on p. 5).

[99] Oasis. *Web Services Business Process Execution Language*. URL: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html (visited on August 31, 2019) (cit. on p. 33).

[100] Opscode. *Chef*. 2019. URL: http://www.opscode.com/chef/ (visited on August 31, 2019) (cit. on p. 70).

[101] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. "Sensing as a Service Model for Smart Cities Supported by Internet of Things". In: *Transactions on Emerging Telecommunications Technologies* 25.1 (July 2013), pp. 81–93 (cit. on p. 74).

[102] Jon Postel. *User datagram protocol*. Rfc 768. Ietf, 1980 (cit. on p. 19).

[103] Puppetlabs. *Puppet*. 2019. URL: http://puppetlabs.com/ (visited on August 31, 2019) (cit. on p. 70).

[104] Inc. Red Hat. *Red Hat CloudForms*. 2019. URL: https://www.redhat.com/en/resources/red-hat-cloudforms-unified-management-for-hybrid-environments (visited on August 31, 2019) (cit. on p. 69).

[105] Inc. Red Hat. *Red Hat OpenStack Platform*. 2019. URL: https://www.redhat.com/en/resources/openstack-platform-datasheet (visited on August 31, 2019) (cit. on pp. 69, 81).

[106] Roberto Reda, Filippo Piccinini, and Antonella Carbonaro. "Towards Consistent Data Representation in the IoT Healthcare Landscape". In: *Digital Health Conference*. 2018, pp. 5–10 (cit. on p. 47).

[107] Telefonica Research and Innovation. *Thinking Things*. 2016. URL: http://www.thinkingthings.telefonica.com/ (visited on August 31, 2019) (cit. on p. 12).

[108] Nikolas Rose. "The human brain project: Social and ethical challenges". In: *Neuron* 82.6 (2014), pp. 1212–1215 (cit. on p. 59).

[109] Libelium Comunicaciones Distribuidas S.L. *Meshlium*. 2016. URL: http://www.libelium.com/products/meshlium/ (visited on August 31, 2019) (cit. on p. 12).

[110] Debashis Saha and Amitava Mukherjee. "Pervasive computing: A paradigm for the 21st century". In: *Computer* 36.3 (March 2003), pp. 25–31+4 (cit. on p. 5).

[111] Salvatore Sanfilippo and Pieter Noordhuis. *Redis*. 2018. URL: https://redis.io/ (visited on August 31, 2019) (cit. on pp. 49, 81).

[112] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. "Incremental deployment and migration of geo-distributed situation awareness applications in the fog". In: *Proceedings of the ACM International Conference on Distributed and Event-Based Systems, DEBS*. ACM Press, 2016, pp. 258–269 (cit. on pp. 68, 70, 87).

[113] Luca Sciullo, Cristiano Aguzzi, Marco Di Felice, and Tullio Salmon Cinotti. "WoT Store: Enabling Things and Applications Discovery for the W3C Web of Things". In: *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, January 2019, pp. 1–8 (cit. on p. 74).

[114] Esther Shein. "Ephemeral Data". In: *Communications of the ACM* 56.9 (2013), pp. 20–22 (cit. on p. 46).

[115] Z Shelby, K Hartke, and C Bormann. *The Constrained Application Protocol (CoAP)*. 2014. (Visited on August 31, 2019) (cit. on pp. 10, 14, 16, 19, 33).

[116] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646 (cit. on p. 46).

[117] Inc. SmartThings. *SmartThings*. 2016. URL: http://www.smartthings.com/ (visited on August 31, 2019) (cit. on pp. 12, 35).

[118] Fabrizio Soppelsa and Chanwit Kaewkasi. *Native Docker Clustering with Swarm*. Packt Publishing, 2017, p. 248 (cit. on p. 41).

[119] Sergios Soursos et al. "Towards the cross-domain interoperability of IoT platforms". In: *Proceedings of the European Conference on Networks and Communications, EUCNC*. IEEE, 2016, pp. 398–402 (cit. on p. 10).

[120] Adika Bintang Sulaeman, Fransiskus Astha Ekadiyanto, and Riri Fitri Sari. "Performance evaluation of HTTP-CoAP proxy for wireless sensor and actuator networks". In: *Proceedings of the IEEE Asia Pacific Conference on Wireless and Mobile, APWiMob*. IEEE, 2017, pp. 68–73 (cit. on p. 12).

[121] Andrew S Tanenbaum and Maarten Van Steen. "Distributed Systems: Principles and Paradigms". In: *Angewandte Chemie International Edition* 40.6 (March 2001), p. 9823 (cit. on p. 4).

[122] Omer Tene and Jules Polonetsky. "Big Data for All: Privacy and User Control in the Age of Analytics". In: *Northwestern Journal of Technology and Intellectual Property* 11.5 (2013), p. 239 (cit. on p. 46).

[123] Dinesh Thangavel, Xiaoping Ma, Alvin Valera, Hwee Xian Tan, and Colin Keng Yan Tan. "Performance evaluation of MQTT and CoAP via a common middleware". In: *Proceedings of the IEEE International Conference on Intelligent Sensors, Sensor Networks and Information Processing, ISSNIP*. IEEE, 2014, pp. 1–6 (cit. on p. 12).

[124] *The Eclipse for IoT Project*. 2017. URL: https://iot.eclipse.org/ (visited on August 31, 2019) (cit. on p. 12).

[125] *The SensorML Project*. 2017. URL: http://www.opengeospatial.org (visited on August 31, 2019) (cit. on p. 12).

[126] Brendan Van Alsenoy. "General Data Protection Regulation". In: *Data Protection Law in the EU: Roles, Responsibilities and Liability*. Intersentia, March 2019, pp. 279–324 (cit. on p. 46).

[127] Socrates Varakliotis, Peter T. Kirstein, Antonio Jara, and Antonio Skarmeta. "A process-based Internet of Things". In: *2014 IEEE World Forum on Internet of Things (WF-IoT)*. IEEE, March 2014, pp. 73–78 (cit. on p. 7).

[128] Siva Visveswaran. *Dive into connection pooling with J2EE*. 2000. URL: https://www.javaworld.com/article/2076221/dive-into-connection-pooling-with-j2ee.html (visited on August 31, 2019) (cit. on p. 48).

[129] W3c. *Transport Message Exchange Pattern: Single-Request-Response*. 2001. URL: https://www.w3.org/2000/xp/Group/1/10/11/2001-10-11-SRR-Transport%5C_MEP (visited on August 31, 2019) (cit. on p. 15).

[130] Zhiliang Wang, Yi Yang, Lu Wang, and Wei Wang. "A SOA based IOT communication middleware". In: *Proceedings International Conference on Mechatronic Science, Electric Engineering and Computer, MEC*. IEEE, August 2011, pp. 2555–2558 (cit. on p. 12).

[131] *Web of Things*. 2017. URL: https://www.w3.org/WoT/ (visited on August 31, 2019) (cit. on pp. 11, 37, 90).

[132] *Web of Things Architecture*. 2017. URL: https://w3c.github.io/wot/architecture/wot-architecture.html (visited on August 31, 2019) (cit. on p. 11).

[133] Mark Weiser. "The Computer for the 21st Century". In: *Scientific American* 265.3 (September 1991), pp. 94–104 (cit. on p. 5).

[134] Luke Welling and Laura Thomson. *PHP and MySQL Web development*. Sams Publishing, 2003 (cit. on p. 47).

[135] Lu Yan, Yan Zhang, Laurence T. Yang, and Huansheng Ning. *The Internet of things: from RFID to the next-generation pervasive networked systems*. Taylor & Francis, 2008, pp. 1–318 (cit. on pp. 5–6).

[136] Nazim Yilmaz, Oylum Alatli, Birol Ciloglugil, and Riza Cenk Erdur. "Evaluation of storage and query performance of sensor based Internet of Things data with MongoDB". In: *2018 International Conference on Artificial Intelligence and Data Processing (IDAP)*. IEEE, September 2018, pp. 1–6 (cit. on p. 63).

[137] Jiguo Yu, Ying Chen, and Baogui Huang. "On Connected Target k-Coverage in Heterogeneous Wireless Sensor Networks". In: *Proceedings of the International Conference on Identification, Information, and Knowledge in the Internet of Things, IIKI*. 2016 (cit. on p. 71).