University of Pisa

Department of Computer Science

Bachelor's Degree in Computer Science (L-31)

# Goal-driven Management of IoT Indoor Environments

Supervisors:
Prof. Antonio Brogi
Dr. Stefano Forti

Candidate:
Giuseppe Bisicchia

**A.Y. 2019/2020**

## Abstract

This thesis aims at designing and prototyping a goal-oriented system for managing domotics IoT devices by suitably reconciling possibly conflicting goals set by different stakeholders. The prototype exploits *Micro:bit* devices for sensing and actuating, extends the *Web of Things* standard with REST interfaces, and employs the *Prolog* language for reasoning.

# Contents

# Chapter 1

# Introduction

## 1.1 Context

The number of devices connected to the Internet is growing continuously [1]. In 2019, over 20 billion Internet of Things (IoT) devices were connected and this number is expected to increase rapidly [2]. The IoT is now part of our lives [3] and becomes more and more tied to it so as to even influence our everyday happiness [4, 5]. Undoubtedly, one of the most interesting IoT fields is that of smart environments, empowering rooms or environments with sensors and actuators to automatically manage them [6, 7]. The development of smart environments is certainly interesting as intelligent systems for the management of environments can greatly improve people's lives, becoming a fundamental support to help them perform simple and complex daily tasks. A very interesting and challenging problem in this area is that of goal-driven management of environments in order to maximise energy savings and user satisfaction.

To this end, many techniques might be used to reconcile possibly contrasting goals (e.g. two users in the same room who want different temperatures) set by users or system admins, e.g. via fuzzy logic [8], multi-agent systems [9, 10] or neural networks [11]. However, most commercial solutions such as IFTTT [12] or Google Home [13] and Alexa [14] only allow to set simple individual goals to be met by the IoT systems they manage and do not account for the possibility of mediating among contrasting objectives [15].

Overall, the goal-oriented approach in the IoT in general, and in the smart environments in particular, is interesting and worth being studied because it is a very simple way of formalising an IoT system: users and devices are distinct entities, each with its own objectives and all must coexist in the same environment. Good conflict management can, therefore, lead to an effective management of complex IoT systems: representing each device at stake as an agent with its objectives and creating a system that allows finding the best compromise taking into account also the needs and goals of the users [15].

## 1.2 Considered Problem

To support the plethora of new IoT-enabled verticals and to promptly process the data they produce, the exploitation of pervasive computing capabilities along the Cloud-IoT continuum has been proposed by industrial consortia and academia [16, 17]. Particularly, *Fog computing* is a novel highly distributed paradigm in which heterogeneous compute, storage, and networking capabilities cooperate to support IoT application deployments all through the Cloud-IoT continuum [18].

In this context, the Department of Computer Science of the University of Pisa, Italy, launched the project "Giò: a Fog computing testbed for research and education" [19]. The project centers around the idea of designing and realising Smart Ambient Systems and applications enabled by IoT sensors and actuators and by a Fog computing infrastructure. One of the objectives of the project is the development of goal-oriented systems for automatically managing indoor lighting via IoT window shutters and the A/C system by negotiating both individual goals (e.g. temperature, brightness) and global goals (e.g. energy consumption). Two different types of conflict might arise. First, different users can set different preferences regarding their desired state of the room, but for peaceful coexistence, it is necessary to find an agreement that is acceptable to everyone. Second, in addition to the preferences of each user, there are also global objectives set by the Department administration that must be met.

This thesis proposes a solution on how to solve the problem of satisfactorily managing conflicts that might arise among the different stakeholders setting goals over shared Smart Ambient IoT systems in the Giò project. This work aims at answering questions such as:

- *Which is the best temperature to be set and maintained to satisfy most of the users' preferences while complying policies set by the Department administration?*

- *What is the best temperature to set in an air conditioning system shared between multiple rooms in order to satisfy all users?*

- *How to set the brightness of the room in order to both satisfy users and to maximise energy savings?*

## 1.3 Objectives of the Thesis

The ultimate goal of this thesis can be stated as follows:

> *Design and implement an IoT goal-driven system capable of monitoring and automatically managing interior natural lighting (via IoT-enabled roller shutters) and temperature (via A/C thermostats) by mediating possibly conflicting goals set by users and system administrators.*

In designing the system, that we called *GiòEnv*, we decided to use a bottom-up approach: we started from the physical infrastructure, then we worked on the software infrastructure and finally we modeled the mediation process. The first point we considered was how to monitor environments. It was also necessary to design the infrastructure for enacting the changes to the environment decided by the system. To do this we took inspiration from the Computer Science Department of the University of Pisa, considering the infrastructure already present and starting from that to create the system. When we started our work, the Department already featured the *GiòRooms* system composed by intelligent roller shutters that are used to change the internal brightness of the rooms, and an air conditioning system regulating the internal temperature. Therefore, in the system design, we took brightness and temperature as the only environmental parameters, also considering that an easy extension to many other parameters will also be possible.

*Micro:bit* [20], as prototypes, were used for monitoring and achieving the desired states. The software infrastructure is a microservice [21] system, as we believe that IoT systems lend themselves very well to being managed with an architecture of that type [22, 23, 24]. For the representation and effective management of the digital twins of IoT devices, we took advantage of a new standard for IoT systems (*Web of Things* [25, 26]). To make the *Web of Things Server* communicate with the physical system in the best possible way we evaluated two alternatives: a new reliable protocol based on UDP, called *ReTRo*, and a possible extension of the *Web of Things*.

As for individual and global objectives, we proposed a though and expressive way to express preferences regarding the desired state of a particular environment. Finally, we modelled and designed a system for the mediation of conflicts, which we believe can have general value for other systems that need conflict management regarding the use of shared resources.

## 1.4 Outline of the Thesis

The rest of the manuscript is organised as follows.

**Chapter 2** Some technologies used in this work are illustrated.

**Chapter 3** The architecture of the system (*GiòEnv*) and its implementation are illustrated.

**Chapter 4** The testbed environment used and some experiments are presented.

**Chapter 5** The contributions of this thesis are summarised and some possible future work is discussed.

# Chapter 2

# Background

## 2.1 *Micro:bit*

A *Micro:bit* [20] is an open-source embedded system designed in the UK by the BBC to teach computer science [27]. The *Micro:bit* is a very small device, half the size of a credit card, but with a high computing power [28]: it mounts a 16MHz 32-bit ARM-Cortex-M0 processor with 256KB Flash ROM and 16KB RAM. It is equipped with a micro USB port, a 5x5 LED matrix, two programmable buttons, 19 pin GPIO, and 3 crocodile clip compatible for rapid prototyping. It also has an accelerometer, a magnetometer, a temperature sensor, and a brightness sensor. It can also communicate via Bluetooth Low Energy [29] or with the proprietary Nordic Gazell radio protocol [30]. Moreover, it can be powered by either USB or an external battery pack.

The radio protocol implements a broadcast communication but allows the partition of the band into 101 channels and it is possible to choose between 8 power levels.

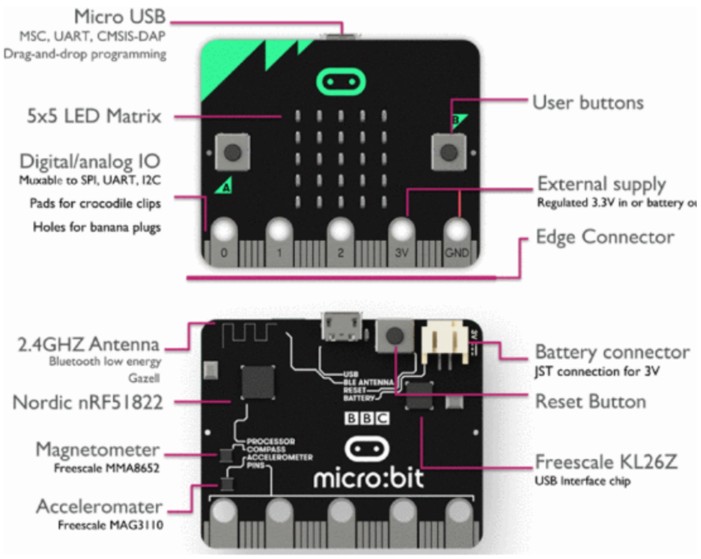


Figure 2.1: A *Micro:bit* scheme.

It is possible to program *Micro:bit* through MicroPython or through the Make-Code platform [31]. This allows programming the device through a simple and intuitive drag&drop interface (like Scratch) or in JavaScript. A simulator is offered too.

Simplicity in programming is given by a powerful runtime software [32], developed by Lancaster University, which provides an easy to use environment.

These features make the *Micro:bit* also an excellent tool for rapid prototyping [27, 33], and thanks to the pins its functionality is easily extendable. The proprietary protocol also allows for easy *Micro:bit*-to-*Micro:bit* communication.

## 2.2 *Web of Things*

The *Web of Things* [25, 26] is a set of W3C standards [34, 35, 36, 37, 38] to create a uniform way to interact with devices and applications of the Internet of Things: its purpose is to enable interoperability across IoT platforms and application domains. The basic idea is not to create anything new but to use already existing protocols and standards. More specifically, the idea is to use the technologies of the World Wide Web to interact and connect physical IoT devices ensuring interoperability. The main concept is to represent real-world resources through a standard representation and make them available through the Web.

The *Web of Things* Building Blocks[1] are:

- Architecture [34]: defines the abstract architecture and its terminology and conceptual framework;
- Thing Description [35]: defines the format in which Things and their possible interactions are represented;
- Scripting API [36]: defines a common JavaScript API, for interacting with Things, similar to the Web browser APIs;
- Binding Templates [37]: provides informational guidelines on how to define network-facing interfaces in Things for particular protocols and IoT ecosystems;
- Security and Privacy Guidelines [38]: provides guidelines for the secure implementation and configuration of Things.

[1]The software stack that implements the *Web of Things* Building Blocks is called *Servient*. A *Servient* can host and expose Things and/or process *Thing Descriptions* and interact with Things [34]. So, a *Servient* can play both the role of server and of client.

The central component of the *Web of Things* is the *Thing Description*: all the Things are indeed represented in the same way thus offering a common interface to interact. *Thing Description* describes the data and metadata of a Thing: a physical or virtual entity. Also, descriptions of possible interactions are provided. *Thing Description*, by default, are encoded in JSON [39], a format readable by both humans and machines. Each Thing is identified by a URI and it is possible to interact with it through a REST [40] interface [41, 42]. In addition to HTTP, other protocols can be used to interact with the Thing.

A Thing is described in terms of properties, actions, and events: the properties define the state of the Thing and can be readable and/or writable, the actions are the functionalities that can be requested of a Thing and the events are "notifications" that the Thing can launch. A *Thing Description* can be directly exposed by the Thing or this task can be delegated (e.g. when a device has limited hardware).

This standard, therefore, offers a uniform way of representing and interacting with Things, all without having to create new technologies and protocols. The *Web of Things* is a substrate on which applications that use IoT can rest.

## 2.3 *Prolog*

*Prolog* [43] is a programming language, developed and implemented in 1972, that realizes the logic programming paradigm. *Prolog* uses a subset of first-order logic, the Horn's clauses: first-order literal disjunctions universally quantified with at most one positive literal. A program in *Prolog* consists of a set of facts, rules, and goals: the facts specify what is true[2], the rules establish relationships between

[2] *Prolog* uses the "Closed World Assumption": everything that is not known true is considered false.

the facts and the goals are questions, over these relations, to be answered.

The intrinsic feature of the *Prolog* is that, unlike procedural programming languages, in *Prolog* we describe what we know about the problem by means of facts and relationships between them and then ask questions and we do not have to worry about how to find the answers. *Prolog* programs are finite sets of rules of the form:

```
a :- b1, b2, ..., bn.
```

where `a`, `b1`, .., `bn` are atomic literals. A rule can be read as: `a` is true if `b1` **and** `b1` **and** ... **and** `bn` are true. `a` is also called *Head* while `b1`, .., `bn` are called *Body*. A rule without a *Body* is also called a *fact*, because it is always true. With the semicolon we can express the disjunctions:

```
a :- b1; b2; ..., bn.
```

a rule with this form meas: `a` is true if `b1` **or** `b1` **or** ... **or** `bn` are true. It is possible to express the negation through the symbol \\+:

```
a :- \+ b1.
```

which means: `a` is true if `b1` is false. In *Prolog* variables are denoted by a string consisting of letters, numbers and underscore characters, and beginning with an upper-case letter or underscore. Prolog also admits the use of the lists indicated with the usual notation through ”[” and ”]”.

Finally, a query is a rule without a head:

```
:- b1, b2, ..., bn.
```

A *Prolog* computation, initiated by a query, consists of a proof by contradiction by means of the inference rule called resolution (Robinson, 1965): for this reason it is also called, in this case, resolution by refutation. In more detail in *Prolog* a resolution technique called SLD resolution [44] is used.

A *Prolog* program can be interpreted in two different ways: declaratively or procedurally. One way to easily clarify this duality is to compare the semantics of the rules in the two cases. A rule has the form *Head :- Body.* The procedural meaning of a rule is: *to solve Head, solve Body*: this corresponds to the concept of function call. Instead, declaratively a rule means: *Head is true if Body is true.*

*Prolog* gives its best when problems need to be solved through complex symbolic computations: for this reason, it is widely used in the field of artificial intelligence [45], expert systems, theorem proving, and deductive databases.

Recently, some authors have proposed to realise Logic Programming as a Service (*LPaaS*) [46, 47]. The idea that drove its development is to offer an inference engine in the form of service. The aim is to offer distributed situated intelligence [48] in pervasive systems, like smart environments.

# Chapter 3

# *GiòEnv*: Design & Implementation

In this chapter, we introduce the architecture of our prototype system *GiòEnv* and its implementation. We also analyse its individual components and how they interact with each other.

## 3.1 *GiòEnv* Overview

### 3.1.1 Big Picture

As aforementioned, *GiòRooms* features actuators to open, close, and adjust blind window shutters so to regulate natural indoor lighting, as well as digital thermostats regulating the A/C system of groups of rooms.

*GiòEnv* extends the functionalities of *GiòRooms* so to monitor the status of the rooms with respect to brightness and temperature and, if needed, to suitably act upon their IoT actuators to reach the desired status. *GiòEnv* implements an IoT-based monitoring of the rooms and a goal-oriented system to determine the target room status by suitably reconciling possibly conflicting goals set by different users for what concerns the temperature and the brightness of a room.

Fig. 3.1 sketches a black-box view of the *GiòEnv* system. Within the system, we identify two stakeholders: the room user and the Department administrator. User's goals, written in the form of *if-then* rules, express how a user wants the system to change the state of the room based on the current one. The administrator instead defines policies, in the form of *Prolog* rules, that express what is the state to be implemented given the preferences of users.

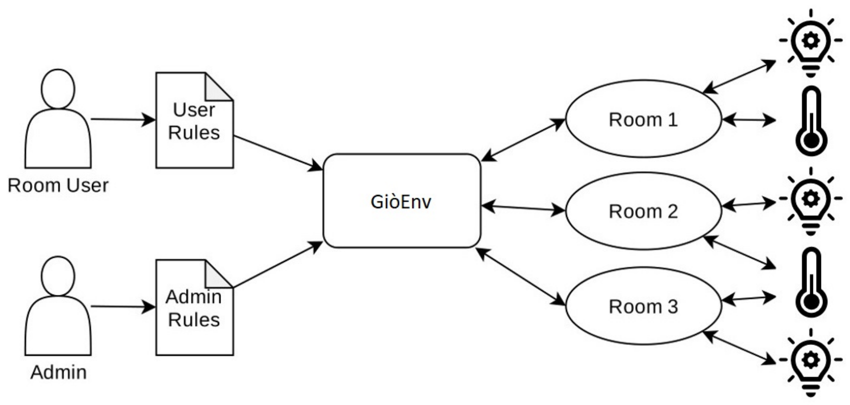


Figure 3.1: Blackbox view of *GiòEnv*.

As for the rooms, these have various sensors inside to perceive their status and actuators that can modify it. It is important to note that the actuators, as in the case of air conditioning, can be shared between multiple rooms while lighting actuators are not shared between multiple rooms. This defines a more complex environment model, as the goals of users in different rooms must be correlated when those rooms share some actuators. In our prototype, we have considered only two types of actuators: air conditioning and automatic shutters.

The task of *GiòEnv* is to wait for changes in the status of one or more rooms (e.g. a user enters/leaves a room, a parameter varies) and initiates the mediation process. This is done not considering users room by room separately but correlating all users inside rooms that share the same actuator.

### 3.1.2 Architecture Bird's Eye-View

*GiòEnv* is organised into a microservice-based architecture composed of a set of independently deployable services, which interact via REST APIs thus enabling low coupling and high reusability [21, 49].

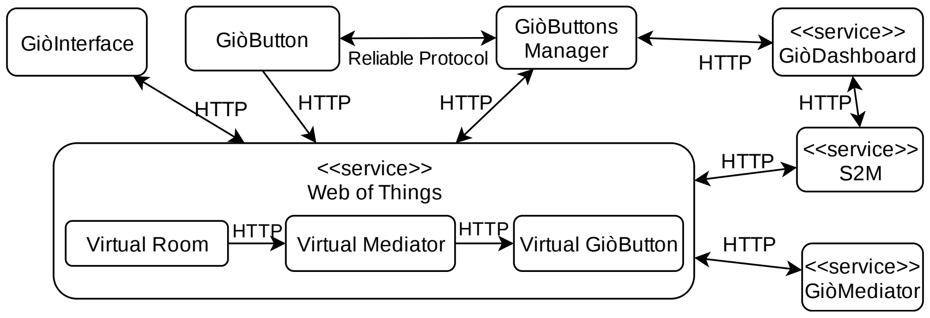


Figure 3.2: A bird's-eye view of *GiòEnv*.

As shown in Fig. 3.2, *GiòEnv* includes one IoT component and seven microservices:

***GiòButton*** Is the component of an IoT device that takes care of monitoring and triggering the actions required by *GiòEnv*.

***GiòInterface*** Displays an interactive map of the environment showing the current environmental parameters and the links relating to the rooms.

***GiòButtons Manager*** Acts as a driver for the *GiòButton* prototype for data collection and subsequent sending to the *Web of Things Server*.

***GiòDashboard*** Enables visualising data about the current status of rooms and IoT devices.

***S2M*** Stores the system's data.

***GiòMediator*** Performs the goal mediation process.

***Web of Things Server*** Maintains the digital twins[3] of the rooms and of the IoT devices and orchestrates the goal mediation process.

To mediate among possibly contrasting goals set by the users and by the Department administration (as mentioned in Sect. 3.1), *GiòEnv* employs first-order logic to model the system and the goals of the users and the administrator, and it then exploits an inference engine to carry out the decision-making process and obtain the new states of the rooms.

We believe that the *User Interface* divided into two parts offers an interesting solution: the user can use one interface to monitor the entire environment (*GiòInterface*) and a second one to check the details of the rooms (*GiòDashboard*). In our opinion, this dualism leads to a clear division of tasks, which can only benefit the way users interact with the system.

When a change in a room is perceived by a *GiòButton*, the notification is sent to the *GiòButtons Manager* who is responsible for propagating the message to the *GiòDashboard* and to the *Web of Things Server*. Here the status of the *Virtual Room* , digital twin of the room to which the notification refers, is updated. At this point, the *Virtual Room* notifies the *Virtual Mediator* of the change, which triggers *GiòMediator*, in charge of the mediation and conflict resolution process. Once the mediation decisions have been made, the *Virtual Mediator* routes them

[3]Digital replicas of physical entities.

to the various *Virtual GiòButton*s, digital twins of the *GiòButton*s. These take care of notifying the *GiòButtons Manager* of the actions to be performed, which are then sent to the appropriate *GiòButton*s. In addition, user preferences, mediation policies, and information necessary for *GiòDashboard* are stored on *S2M*. *GiòInterface*, on the other hand, can read the *Virtual Room*s status and display their data.

In the next section, after describing the simple language that *GiòEnv* exploits to allow users to express goals, each service composing *GiòEnv* is described in detail along with its functioning, following the workflow summarised above.

## 3.2 *GiòEnv* Components

### 3.2.1 *Simple Storage Microservice*

The *Simple Storage Microservice* (S2M) has been implemented to offer a simple and uniform way to store data sensed by the deployed *GiòButton*s as well as goals set within *GiòEnv* . S2M design was inspired by Amazon S3 bucketing system and permits storing heterogeneous object data in JSON format [39]. The conceptual model is the same as S3. A user can dynamically create objects and for each of them create properties at will. The user can then request the creation and deletion of new objects, owned within them, and the addition or modification of the data. Everything is offered through a REST [40] interface described in Table. 3.1.

| | GET | POST | PUT | PATCH | DELETE |
|---|---|---|---|---|---|
| / | Return the list of all buckets | | | | |
| /<bucket> | Return the metadata of a specific bucket | Create a new empty bucket | | | Delete a specific bucket and all its objects |
| /<bucket>/<object> | Return the data and metadata of a specific object | Create a new object | Replace the data of a specific object | Add new data to a specific object | Delete a specific object |
| /<bucket>/<object>/metadata | Return the metadata of a specific object | | | | |

Table 3.1: S2M REST API.

### 3.2.2 *GiòButton*

*GiòButton* is the prototype IoT device that allows *GiòEnv* to interact with the real world. *GiòButton* performs two main tasks:

- *Sense*, which can collect environment data through the sensors offered by the hardware, as well as changes in the state of the room, and send them to *GiòEnv*,
- *Actuate*, which can receive commands from the system, so to trigger suitable IoT actuators that permit to change the state of the room to reach a newly established target state.

The current implementation of *GiòButton* exploits an intermediary *GiòButtons Manager* to communicate with its digital twin in the *Web of Things Server*[4]. To ensure independence from the underlying hardware and therefore to maximise interoperability, we have established a simple communication protocol between the *GiòButton*s and the *GiòButtons Manager*.

[4]If it is possible to connect *GiòButton* to the network, it can communicate directly with the *Web of Things Server*. If the device does not have a connection, the presence of an intermediary (the *GiòButtons Manager*), is required.

The protocol only requires that messages exchanged have the form:

`KEY$VALUE$RECEIVERNAME`

where `RECEIVERNAME` is the identifier of the receiver, while `KEY` and `VALUE` are used for the actual exchange of information represented by key-value pairs. The protocol, thanks to the $ symbol, does not establish any limitation on the type of communication channel used on the number of characters that can be used for each field. Indeed, the devices can use any type of communication channel and length of messages they want, as long as they respect the structure just indicated.

**Implementation**

The current prototype implementation of the *GiòButton* relies upon the *Micro:bit* hardware (Sect. 2.1). This choice was made because *Micro:bit* is an excellent prototyping platform and it features built-in [28] a temperature and a brightness sensors, a LED screen, some buttons, and a radio antenna. This allowed us to implement a basic version of *GiòButton* using only one type of hardware, perfectly suiting our system needs. Indeed, we needed to monitor the temperature and brightness in the rooms. Besides, the LED screen and the buttons of the *Micro:bit* allow easy and intuitive interaction with users (see Table. 3.2) and the presence of a radio antenna enabled simple communication with the *GiòButtons Manager*.

The current version of *GiòButton* relies upon some important battery-saving expedients. Particularly, room status updates are sent only when changes of parameters exceed a certain (configurable[5]) threshold and, anyhow, after a certain

[5]By default the temperature must increase by at least one degree Celsius to send an update, instead as regards the brightness this must vary by at least 5 points (*Micro:bit* measures the brightness in a range of 0-255).

interval of time[6] no changes are sent. As for the *Micro:bit* LED screen, this is used only in the initial phase, that of pairing (more information below). Otherwise, the screen remains off[7] until a function is invoked (see Table. 3.2).

| | Pairing-Phase | Working-Phase |
|---|---|---|
| Button A | Nothing | Show the actual temperature |
| Button B | Nothing | Show the actual brightness |
| Buttons A+B | Stop the pairing phase | Show the device's name |
| Shake | Nothing | Start a new pairing-phase |

Table 3.2: Possible interactions with a *GiòButton Micro:bit*.

From a communication point of view, *GiòButton* relies upon the radio *Micro:bit*-to-*Micro:bit* communication protocol [29, 30]. We chose to use the radio protocol instead of the Bluetooth Low Energy (BLE) featured by the *Micro:bit* as it permits greater ease of use and flexibility.

To enable communication of the *Micro:bit*s with the *GiòButtons Manager* we have chosen to use some *Micro:bit*s as intermediaries. These particular *Micro:bit*s, called *Radio Servers*, receive the status notifications from all the other *Micro:bits* and send them via serial port to the device on which the *GiòButtons Manager* runs. There can be many *Radio Servers*, deployed withing the managed cyber-physical environment, connected to one or more *GiòButtons Managers*. Fig. 3.3 sketches how the *GiòButton*s communicate with the *GiòButtons Manager* via a *Micro:bit Radio Server*.

To ensure reliability we have chosen to design a protocol over the *Micro:bit* radio. First, we decided that the messages exchanged should always contain, besides the sender ID, also the receiver ID. Each *Micro:bit* is in fact equipped with

[6]By default, set to one hour.
[7]In case of debugging the screen can still show some information.

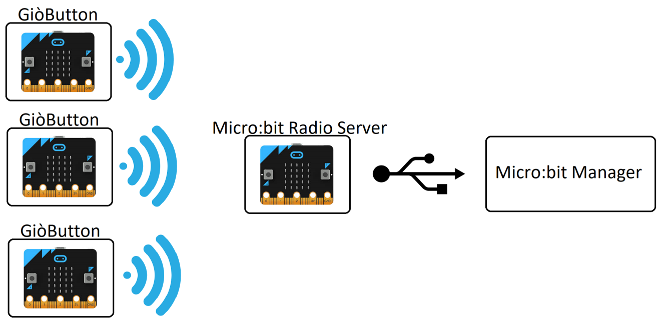


Figure 3.3: Example of the *Micro:bit* communication.

a unique serial number, but it was not possible to directly use the serial number of the receiver. Indeed, while the sender's number can be sent by enabling a specific setting[8], this is not possible for the receiver as the *Micro:bit* protocol is a broadcast protocol [30]. By including the name of the receiver in the message, it is possible to implement a simple ack exchange mechanism to notify the acknowledgment of receipt of the message. In this way, when a *Micro:bit* receives a message it checks the receiver's name and if the name coincides with it sends back to the sender, recognizable by the ID[9], a confirmation ack. If the sender does not receive the ack within a certain configurable time interval (5 seconds by default), then it re-sends the message for a limited number of times (5 by default). Besides, it is also possible to configure each *Micro:bit* by choosing whether to enable it for broadcast reception or not. In this way, when a *GiòButton* has to send an update it does not need to know the name of a *Micro:bit server* but can simply send the message in broadcast, indeed in *GiòEnv* only *Micro:bit servers* are

[8]See: https://makecode.microbit.org/reference/radio/set-transmit-serial-number

[9]Since the size of the string to be sent is limited to 19 characters (see https://makecode.microbit.org/reference/radio/send-string), we have decided to use what is called in *Micro:bit* a *friendly name*: a hash function of the serial number that returns a string of 5 characters that can be easily remembered. It is up to the administrator to verify that *Micro:bits* with the same name do not appear in the same network.

broadcast-enabled, so only *servers* will handle, and confirm with an ack, update messages[10].

Finally, for what concerns security, we implemented a simple pairing mechanism for the devices. When a new *GiòButton* connects to the network, its messages are not processed until acceptance by the administrator takes place. To do this once the *Micro:bit* is started, it starts sending a sequence of 6 characters, randomly generated, and this is also displayed on its screen, and until the administrator enters that sequence read on the screen in the system, which instead keeps the version sent via radio, the update messages are not managed. It is the *GiòButtons Manager*'s job to manage such a pairing mechanism.

### 3.2.3 *GiòButtons Manager*

The *GiòButtons Manager*'s job is to interface *GiòEnv* with his physical devices: the *GiòButtons*. *GiòButtons Manager* acts as an intermediary who receives the messages from the *Micro:bit* and forwards it to the system.

Besides the fact that *Micro:bit*s cannot connect to the Internet, we relied upon an intermediary to enforce the Single Responsibility Principle [50]. The *GiòButtons Manager* maintains a copy of the mapping between rooms and IoT devices, this allows associating the information received from the devices to the rooms in which they are located. Mapping is necessary because the environment is dynamic and associations can vary over time. This choice also guarantees the maximum degree of decoupling between the various parts of the system.

[10]The format of the messages respects the protocol established for the *GiòButtons* (`KEY$VALUE$RECEIVERNAME`), with some other limitations: `RECEIVERNAME` has a maximum length of 5 characters while `KEY` and `VALUE` have both a maximum length of 6 characters. For broadcast messages `RECEIVERNAME` is “broad” and for ack messages `KEY` is “ack” and `VALUE` is empty.

### Implementation

As already anticipated (see Figure 3.3 at page 23), the operation of the *GiòButtons Manager* is strictly linked to the presence of one or more *Micro:bit servers*. Indeed, *servers* are the way in which the *GiòButtons Manager* can communicate with the *Micro:bits*.

From the *Micro:bit*s point of view, the role of the *GiòButtons Manager* is to listen on a serial port, waiting for notifications from the *Micro:bit*s and forwarding these to the *Web of Things Server*. Also it is waiting for notifications of status change by the *Virtual Room*s from the *Web of Things Server* (see Figure 3.2 at page 17) and sends these, again via serial, to the *Micro:bit server* to which it is connected. In principle, there is no reason why a single *GiòButtons Manager* should be associated with a single *Micro:bit Radio Server*, but we made this choice to keep the system architecture simple and to be able to monitor messages exchange more effectively. So, in our implementation a *GiòButtons Manager* is associated with only one *Micro:bit Radio Server* and vice versa.

Instead, from the point of view of the *Web of Things Server*, the *GiòButtons Manager* sends[11] the *Micro:bit*s update notifications to their digital twins and also to the rooms that are associated with them. In addition, it remains listening to receive commands to change the state of the rooms and forwards these requests to the associated *Micro:bit*s.

For the mapping between *Micro:bit*s and rooms we have chosen to use a simple JSON [39] file[12], which associates each *Micro:bit* with a series of resources that must manage (as input or output), and for each specific resource (the rooms

[11]We have experimented two ways to send notifications to the *Web of Things Server*: the details on section 3.2.4.

[12]It is up to the administrator to create and modify the file in order to respect reality.

that have that resource). In our prototype we used two input resources: temperature (`temp`) and brightness (`light`); in addition, two output resources were used: air conditioning (`ac`) and automated shutters (`windows`). In Figure 3.4 it is possible to see how a resource of a *Micro:bit* (identified by its friendly name) can be shared between multiple rooms, in addition, different *Micro:bit*s can handle different resources in the same room.

```
{
    "tuvov": {
        "temp": [279,284],
        "light":[279],
        "ac":[42,279,284],
        "windows":[279]
    },
    "tetoz":{
        "temp":[42],
        "light":[42],
        "ac":[],
        "windows":[42]
    },
}
```

Figure 3.4: Example of JSON object mapping the associations between *Micro:bit*s (tuvov and tetoz) and rooms (42,279,284): tetoz senses temperature changes in room 42, while tuvov who controls the air conditioning.

As anticipated, the *GiòButtons Manager* is also responsible for pairing the devices: when a new device enters the network, it sends a random code, the *GiòButtons Manager* associates the ID of that device with that code, and until the administrator enters that code, which he will read from the device screen, any other message from that device will be ignored.

### 3.2.4 *Web of Things Server*

The *Web of Things Server* is the central component of the whole system: it maintains the digital twins of *GiòButton*s and rooms and orchestrates the mediation of conflicts by interacting with *GiòMediator*. We have chosen to use the *Web of Things* [25, 26] for the construction of our server as it offers a simple and uniform interface for interacting with IoT devices [25].

Within *GiòEnv* the task of the *Web of Things Server* is to receive notifications of state changes from the *GiòButtons Manager* and trigger the mediation process in order to obtain the new state of the rooms affected by the change. Then, the new status is sent to the *GiòButton*s that will implement the request.

It is worth noting that the *GiòButton*s do not receive a command to be executed but the desired new state. This allows for a high degree of decoupling between the server and the physical devices, this also makes the decision-making process independent of the effective implementation of the goals. In a nutshell: the server decides the *what* to do it and the devices the *how*.

#### Mediation

The mediation process takes place through some *Web Things*[13] [34] that interact with each other and with the other microservices allow the intelligent management of the environment. We use three types[14] of *Web Things*: *Virtual Room*, *Virtual GiòButton* and *Virtual Mediator*. The first two correspond to the digital twins of, respectively rooms and *Micro:bit*s of the system. The *Virtual Mediator* is instead a purely virtual entity that has the task of triggering the goal mediation

[13]An abstraction of a physical or a virtual entity.
[14]More details in Appendix B.

process, by interacting with *GiòMediator*. There is only one *Virtual Mediator* but as many *Virtual GiòButton*s and *Virtual Room*s as there are physical counterparts.

Whenever a change of state takes place in a room, this is notified by the *Virtual Room* to the *Virtual Mediator* which stores the most recent status of all the rooms in addition to the preferences of all users and the administrator. Once the notification has been received and the status updated, all this information is sent, together with the specific conflict resolution rules, to *GiòMediator* which will carry out the mediation process and return the changes to be made in the rooms. At that point, the *Virtual Mediator* will notify to the various *Virtual GiòButton* the new states they must implement.

### Implementation

The implementation of the *Web of Things Server* has been long and complex, with some important changes of direction. We decided to choose the *Web of Things* even before it officially became a standard. We chose not to implement the server from scratch, believing that the existing implementations [36] were entirely suitable for our purposes.

Our initial choice was to use the Mozilla proposal [51] and the first implementation of our *Web of Things Server*, which also includes a protocol (*ReTRo*) to enable communication of the server with the *GiòButtons Manager* , is described in Appendix A. Meanwhile, the standardisation work continued, to culminate on April 9, 2020 making the Architecture [34] and *Thing Description* [35] W3C recommendations. This important step, combined with the fact that the first implementation was starting to show its limitations, led us to reconsider the initial choice and re-analysing the proposals, so we chose to switch from Mozilla's *Web* *of Things Server* to Eclipse's *ThingWeb* [52]. Contrary to the REST API of Mozilla, this implementation is more complex, offering various additional features, allowing communication through various protocols and not least a much more active community. Furthermore, it fully respects the scripting API [36] proposed[15] by W3C.

In this new version, we decided to abandon the *ReTRo* protocol used in the first version and decided to extend the *Web of Things*'s REST API. We will now analise in detail this last version.

**Eclipse's ThingWeb** More than a server, ThingWeb is an entire ecosystem of features and services related to the *Web of Things*. Many communication protocols are implemented and many others are continuously implemented: the community is indeed very active. Unlike Mozilla, ThingWeb fully adheres to the (non-standard) proposal of API [36] of the W3C and this allows for greater uniformity and interactivity.

The problem of communication between *GiòButtons Manager* and *Web of Things Server* was there to be solved. Our idea was to transform the *Web of Things Server* into a dynamic service: using APIs not only for interacting with users but also to create Things, eliminate them, modify them, and notify changes of state. This last point requires more attention: among the APIs, some allow updating the properties of a Thing, but these require that property to be writable, while the properties that reflect the values of a sensor, such as temperature, are read-only and there is no standard way to update these properties.

Our solution extends the protocol also considering this possibility. This result

[15]It is still a W3C Working Draft.

was achieved by adding two possible attributes to the properties, namely `input` or `output`. A property marked as input allows the remote updates (through the PATCH method). A property marked as output cannot be updated in this way, but (if allowed) through the classic writing. This also allows the possibility to specify different security rules based on the possible ways to perform an update[16]. This option allows the updating of the digital twins of the devices from the devices themselves and without the need to implement other protocols or mechanisms.

We specified that our idea was to transform the *Web of Things Server* into a *dynamic* service because we decided to introduce, also, some APIs that allow the creation of things remotely something which is not allowed by the standard. To do this it was necessary to define a method to be able to exchange not only the *Thing Description*s but also the scripts related to these. Indeed, it is possible to specify certain procedures to be performed[17] when an action is invoked or when a property is written or read. These procedures must be sent, together with the *Thing Description*, to the server when the Thing is created. We decided to extend the *Thing Description* by admitting the possibility of specifying procedures to be performed.

The new *Thing Description* has the following attributes:

- `thing`: contains the standard *Thing Description*,
- `initialScript`: the code to be executed before the actual creation of the Thing,

[16]This is not currently allowed by ThingWeb but guaranteed by the standard. Here is the issue that we have opened on the official ThingWeb repository to discuss the problem: https://github.com/eclipse/thingweb.node-wot/issues/211

[17]The security of the operation is delegated to ThingWeb which allows the execution of external scripts.

- `endScript`: the code to be executed after the creation of the Thing,
- `handlers`: the procedures to be performed associated with actions and properties,
  - `actions`: it is possible to specify for each action some code to execute when this is invoked,
  - `properties`: it is possible to specify for each properties some code to execute when when this is read and/or written.

With these modifications, the *Web of Things Server* allows any type of interaction without the need to rely on external technologies.

In this way, when the *GiòButtons Manager* receives an update from a *Micro:bit* , all it has to do is retrieve the specific *Thing Description* and send the update via PATCH. Furthermore, when a new *Micro:bit* is added to the system, the *GiòButtons Manager* automatically requests the creation of the associated *Thing Description.*

### 3.2.5 *GiòMediator*

In *GiòEnv*, *GiòMediator* is the service that performs the mediation process and therefore decides the commands to be executed. To do this, it interacts closely with the *Web of Things Server.* It receives from the Server the data for mediation (rooms status, users' preferences, administrator's policies) and returns the decisions which will then be sent to the *GiòButton*s. The information necessary for the process are the data of the rooms (i.e. environmental parameters, associated actuators, users inside), the preferences of the users, and the administrator's policies that permit resolving possible conflicts. *GiòMediator* will examine all the preferences expressed by users regarding the status of the desired room and mediate them by applying set policies. Preferences are expressed in the form of if-then rules and allow the user to define the desired state of the room starting from the current one[18].

Recall that the environmental parameters taken into consideration are the brightness and temperature of the rooms and *GiòEnv* can modify them thanks to the air conditioning and automatic shutters.

**Logic Programming-as-a-Service** *GiòMediator* is implemented as an *LPaaS*, which means that what is offered is a full-fledged inference engine, usable in any way the service programmers need. We recall that an *LPaaS* offers an inference engine as a service [46, 47]. This features a high expressiveness and flexibility in the formalisation of the problem, being able to use the power of the first-order logic. Both user preferences and mediation policies are expressed with logic programming. Furthermore, the representation of the environment also becomes much more intuitive and simple thanks to the possibility of expressing it via a simple set of predicates.

### Implementation

Our implementation, written in Python, is based on the Flask framework. The microservice is stateless to facilitate simplicity of development and scalability and a minimal REST [40] interface is offered. The information is exchanged through JSON [39] objects. The microservice expects to receive an object con-

[18]More details in the section B.1.

taining the attributes `facts`, `policies` and `goals` and returns an object containing the `decisions` attribute, that contains everything that was generated by the inference engine. As engine we have chosen to use ProbLog [53] and in particular its library in Python, which allows fast and easy integration with Flask.

The choice to offer a stateless service was dictated by the Single Responsibility Principle [50]. The system architecture was designed following the rule that each component of the system should have only one responsibility. So in the case of *GiòMediator* the need to carry out the mediation process, this regardless of the type of application that requires the service. It also guarantees greater ease in scaling the service. It is, however, possible to design a stateful version of *GiòMediator*, as proposed in [46, 47]. This would, however, lead to a greater quantity and frequency in the interactions with the service, having to continually update the data regarding the status of the rooms stored in.

Below we will illustrate how we modelled the problem of goal mediation.

### Knowledge Representation of Goal Mediation

Three main entities need to be represented in to solve the goal mediation problem we are considering in *GiòEnv*: rooms (with their sensors and actuators), user preferences, and administrator mediation policies.

**Representing Rooms** First, in *GiòMediator* , rooms can be declared as facts of the form:

```
room(Room).
```

where `Room` is a literal value denoting the unique room identifier. Sensed brightness and temperature values related to a room can be declared as in:

```
light(ValueL, Room).
```

and

```
temperature(ValueT, Room).
```

where `ValueL` and `ValueT` are literal values[19] indicating the status of the relative parameter.

Similarly, outdoor light and temperature are represented as:

```
outdoor_light(ValueOL, Room).
outdoor_temperature(ValueOT, Room).
```

where `ValueOL` and `ValueOT` are literal values ranging in the same set of values of `light` and `temperature`.

Besides, other data, such as time, is denoted as:

```
daytime(Time).
```

where `Time` is a literal value showing the part of the day[20].

As for the actuators, these are defined in the following way:

```
actuator(Actuator, Type, Room).
```

where `Actuator` is the unique actuator identifier, `Type` defines an actuator's capability to control a certain environmental parameter (e.g. temperature, light) and

[19]For `temperature` the range is `(very_low, low, medium, high, very_high)` and for `light` the range is `(low, medium, high)`.
[20]The possible values are: `morning,afternoon,evening,night`.

`Room` specifies a room with which it is associated. It is important to remember that actuators can be shared between multiple rooms.

Finally to describe the presence of a user inside a room we use:

```
inRoom(User, Room).
```

where `User` is a unique user identifier.

**Example.** A room identified as `room1` associated with an actuator `a1` with type `temperature` and `light` and with `user1` and `user2` in, can be specified as:

```
room(room1).
actuator(a1, temperature, room1).
actuator(a1, light, room1).
inRoom(user1, room1).
inRoom(user2, room1).
```

Similarly, a room `room2` with two actuators `a1`, shared with `room1`, with type `temperature` and `a2` with type `light` and with `user3` in, can be represented as:

```
room(room2).
actuator(a1, temperature, room2).
actuator(a2, light, room2).
inRoom(user3, room2).
```

Finally, the following data for the two rooms is declared as in:

```
daytime(morning).

% room1
light(low, room1).
outdoor_light(high, room1).
```

```
temperature(high, room1).
outdoor_temperature(high, room1).

% room2
light(medium, room2).
outdoor_light(high, room2).
temperature(medium, room2).
outdoor_temperature(high, room2).
```

**Representing User Preferences** A user can be declared as:

```
user(User).
```

where `User` is the unique user identifier. Each user can express her preferences regarding the desired status. These are expressed in the form:

```
set(User, Room, Type, Value) :- Preconditions.
```

where `Type` defines an environmental parameter the user `User` wants to change (e.g. temperature, light), `Room` specifies in which room the user wants the parameter to be changed, `Value` indicates the new value that the user wants that parameter to assume and `Preconditions` can be any condition on environmental data. The rule then reads: *if the* `Preconditions` *are true then the user* `User` *wished the parameter* `Type` *to have value* `Value` *in the room* `Room`.

**Example.** Suppose that `user1` always wishes a very warm and bright room, this can be formalised as:

```
user(user1).
set(user1, R, temperature, very_high) :- inRoom(user1, R).
set(user1, R, light, high) :- inRoom(user1, R).
```

user2 instead wants a very cool room if it is hot inside and is ok with low light if it is bright outside:

```
user(user2).
set(user2, R, temperature, very_low) :- inRoom(user2, R),
                                    (temperature(high, R)
                                    ;
                                    temperature(very_high, R)).
set(user2, R, light, low) :- inRoom(user2, R), outdoor_light(high, R).
```

Finally, user3 wishes a cool room if it is very hot both inside and outside and wishes a lot of light only if she is in room1, otherwise she wishes a medium temperature if it is not very hot inside and outside:

```
user(user3).
set(user3, R, temperature, low) :- inRoom(user3, R),
                                    temperature(very_high, R),
                                    outdoor_temperature(very_high, R).
set(user3, R, temperature, medium) :- inRoom(user3, R),
                                    \+ temperature(very_high, R),
                                    \+ outdoor_temperature(very_high, R).

set(user3, room1, light, high) :- inRoom(user3, room1).
```

**Representing Admin Policies** The great flexibility of our system model is seen precisely in the way the users and the administrator express their policies: there is no limit to the procedures that they can express, but those of the languages used (Prolog). This allows the administrator to define various policies sequentially as well. The only rule to be respected is the one that defines the structure of the rule that initiates the mediation process, which must have the following form:

```
mediate(Actuator, Type, PrefList, Action).
```

where `Actuator` identifies a certain actuator and type `Type` specifies the actuator's type considered. `PrefList` is a list of pairs `(User, Value)` where the users are associated with their preferences. Finally, `Action` can be of any form, but those that correspond to a command to be sent to the actuator must have the following:

```
todo(Actuator, Type, Value).
```

to set the parameter `Type` of actuator `Actuator` to the value `Value`.

It is interesting to note that mediation takes place not considering the rooms but the actuators. This solves the problem of shared resources, indeed, mediation takes place by considering together all users who express a preference for an environmental parameter controlled by the same actuator.

To guarantee a "fine-grained" control as regards the implementation of the commands, the administrator defines a correspondence between literal and numerical values. Such correspondences have the form:

```
value(Type,Literal,Numerical).
```

For example:

```
value(temperature,very_high,24).
value(temperature,high,22).
value(temperature,medium,20).
value(temperature,low,18).
value(temperature,very_low,18).


value(light,high,250).
value(light,medium,180).
value(light,low,100).
```

where we can see how, for example for legal reasons, the temperature can never drop below 18 degrees Celsius or rise above 24. The brightness is instead expressed in a range of 0-255.

**Example.** We now show some simple administrator policies. The first we want to propose is that of the director: if the director has a preference, that has to be chosen.

```
mediate(A,Type,UVs,todo(A,Type,W)) :- member((U,V),UVs),
                                      headOfDpt(U),
                                      value(Type,V,W).
```

Figure 3.5: The Director's Policy.

Another policy that we can take as an example is that of the chilly: if someone wants a very high temperature that is implemented.

```
mediate(A,temperature,UVs,todo(A,temperature,W)) :-
                                        member((_,very_high),UVs),
                                        value(temperature,very_high,W).
```

Figure 3.6: The Chilly's Policy.

The last policy that we take as an example is that of the majority: the most voted preference is implemented.

```
mediate(A,Type,[X|Xs],todo(A,Type,W)) :-
        getStats([X|Xs],Stats),
        getMax(Stats,(V,_)),
        value(Type,V,W).

getStats([],[]).
getStats([(_,T)|L], SS) :- getStats(L,S), add(T,S,SS).

add(T,[],[(T,1)]).
add(T,[(T,N)|L], [(T,NewN)|L]) :- NewN is N+1.
add(T,[(T1,N1)|L], [(T1,N1)|NewL]) :- T \== T1, add(T,L,NewL).

getMax([(V,N)|L],M) :- myGetMax((V,N),L,M).
myGetMax((V,N),[],(V,N)).
myGetMax((V,N),[(_,N1)|L],M) :- N>=N1, myGetMax((V,N),L,M).
myGetMax((_,N),[(V1,N1)|L],M) :- N<N1, myGetMax((V1,N1),L,M).
```

Figure 3.7: The Policy of the Majority.

We can use multiple policies by ordering them: we test one after the other and the first one that returns a value is applied. For example, if the director is there, we choose his preferences, otherwise, if someone wishes a very high temperature, that is applied and if not, the majority vote value is chosen.

### Conflict resolution process

The mediation process takes as input the descriptions of all the rooms, the user goals and the goals of the administrator and for each couple `(actuator, type)` finds the preferences expressed by the users for that couple and applies the policies of the administrator in order.

The conflict resolution process is defined by three predicates: `go/0`, `decideActions/1` and `decideAction/3`.

The predicate `go/0`, as shown in Fig. 3.8, finds all the pairs `(actuator,type)` (line 2) and puts them into a list, removes all the duplicates (line 3) and passes the list to `decideActions/1` (line 4).

```
1 go :-
2         findall((A,Type),actuator(A,Type,_),As),
3         sort(As,SAs),
4         decideActions(SAs).
```

Figure 3.8: The `go/0` predicate.

Predicate `decideActions/1`, Fig. 3.9, for each pair `(actuator, type)` finds all the preferences express by users about that pair, thanks to the association with the rooms, and puts them into a list (lines 3-5), then passes the list to `decideAction/3` (line 6).

```
decideActions([]).
decideActions([(A,Type)|As]) :-
        findall((U,V),
                (user(U),inRoom(U,R),actuator(A,Type,R),set(U,R,Type,V)),
                UVs),
        decideAction(A,Type,UVs),
        decideActions(As).
```

Figure 3.9: The `decideActions/1` predicate.

As shown in Fig. 3.10, the predicate `decideAction/3` calls the admin's mediation policies (line 2) and if an action is returned, saves it in its knowledge base to fetch it at the end of computation (line 3).

```
1 decideAction(A,Type,Xs) :-
2         mediate(A,Type,Xs,Action),
3         assert(Action).
4 decideAction(_,_,[]).
```

Figure 3.10: The `decideAction/3` predicate.

**Example.** Applying the process just described to the examples of rooms, users and policies previously exposed we will obtain as a result:

```
todo(a1, light, 100).
todo(a1, temperature, 24).
```

Indeed, for the pair `(a1, light)` we have two preferences `(user1, high)` and `(user2, low)` and since neither `user1` nor `user2` is the is director (Fig. 3.5) and there are no preferences regarding temperature, the Policy of the Majority is applied (Fig. 3.7). Since we have a tie, the first value generated is chosen, in this case `low` that correspond to 100. For the couple `(a1, temperature)` the preferences expressed are `(user1, very_high)`, `(user2, very_low)` and `(user3, medium)`. In this case *GiòMediator* applies the Chilly's Policy (Fig. 3.6), so the value is `very_high` that correspond to 24 degree Celsius.

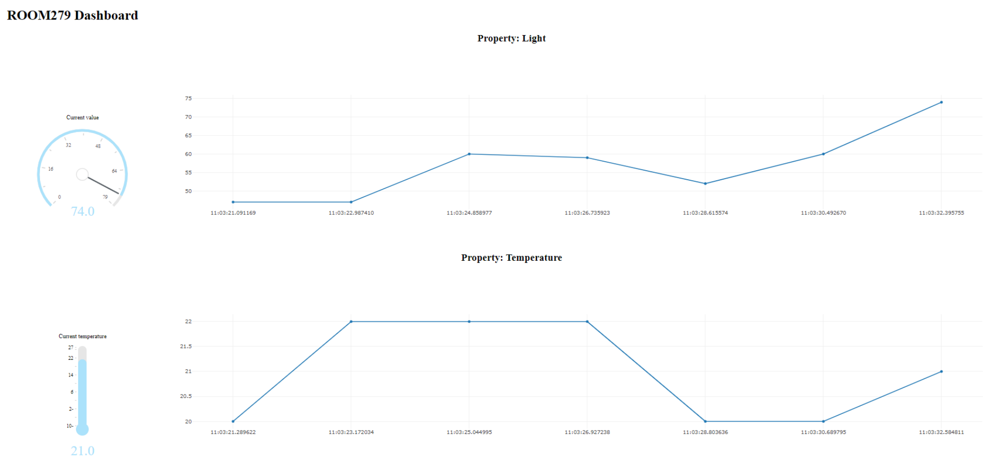
### 3.2.6 *GiòDashboard*

*GiòDashboard* is one of the two microservices that implement the system's User Interface. This component is designed to offer an easy way to monitor the status of *Micro:bit*s and rooms (see Figure 3.11). It allows users to visualise various dashboards regarding various parts of the building.

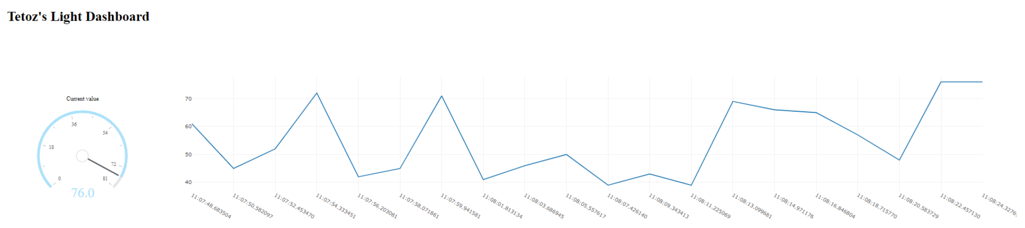
The microservice is designed to be dynamic: it is not necessary to specify at startup which dashboards to display regarding which components. Despite being used, in our implementation, to show data regarding *Micro:bit*s and rooms, the service is actually completely data agnostic: it can show data inherent to any component, for this reason, we could call it a *Dashboards-as-a-Service*.

The microservice uses an object-property representation of data: various objects can be created (each one identified by a unique name) and each of them can have many properties. One dashboard is built for each of these.

The dynamicity of the service is given by the fact that it is possible to create new objects and new properties at runtime or add existing ones. When an update of the status of a property of an object is sent, this is created if not yet existing. The user does not have to do any preparatory work, the only thing she has to do is send the updates to the service specifying only the object, the property, and



(a) A room Dashboard



(b) A *Micro:bit* light dashboard

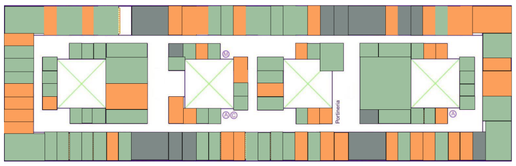
Figure 3.11: The two views of the *GiòDashboard*.

the current value. The microservice will take care of the rest.

In addition, dashboards are generated on demand and remain stored within the system, the user can then retrieve the history of all generated dashboards.

## Implementation

The microservice exploits the storage offered by S2M: for this reason, it offers an object-property representation of data. The service allows viewing dashboards covering an entire object or a single property. The microservice offers a REST [40] interface: with the PATCH method, it is possible to send an update of a specific


(a) A heat map showing the current temperature in the rooms.

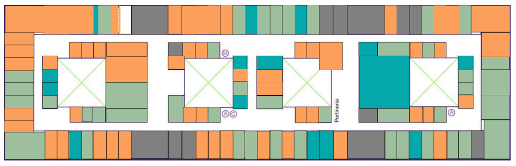

(b) A heat map showing the current light in the rooms.

Figure 3.12: Two screens from *GiòInterface* showing the the temperature (a) and light (b) heat maps.

property of a particular object. The microservice will receive the update and mark it with a timestamp before storing it in the history of that property.

As anticipated, it is possible to request the display of the dashboard relating to a specific property of a particular object or the entire object. In the first case, the current value of that property will be displayed with a graph showing its evolution over time, in the second case, for each property the data indicated above will be displayed. The user can also request the history of all generated dashboards (each identified by a particular URL) and can, therefore, view them. The user can also request the status of a particular object and the history of its properties.

### 3.2.7 *GiòInterface*

*GiòInterface*, the second part of the User Interface of *GiòEnv*, shows the map of the environment (in our case the Computer Science Department of the University of Pisa) and allows navigation from one point to another, as well as showing information regarding each office.

This component was an existing service of the *Giò project* [19]: we have only extended the interface to adapt it to the possibility of displaying information regarding the current environmental parameters.

Our contribution was to add the possibility of displaying heat maps (see Figure 3.12), concerning the various environmental parameters. It is thus possible to visualise the global situation of the whole environment, seeing the information necessary for each office and how it is combined with that of others. The administrator, therefore, can have a full view of the entire environment and in case, she can also obtain detailed information on a specific office.

In addition to the heat maps, which give general information on the environment, there are also the office files and in these, we have added the information that concerning the current environmental parameters of that room and a link to the associated dashboard, so that can have the history and additional information, about that room, in few clicks.

# Chapter 4

# Testing and Use Cases

## 4.1 Testing

### 4.1.1 Testbed

The testbed environment we set up was the following:

- 4 *Micro:bits*;
- 1 Laptop Dell with Windows 10 and Intel i7 x64, 16 GB of RAM where the following applications were run:
  - *GiòButtons Manager*;
  - *GiòInterface* and its backend.
- 1 Virtual Machine over Unipi with 2vCpu, 4GB of RAM and Ubuntu in which the following applications were run:
  - *Web of Things Server*;

  - *GiòMediator*;
  - *GiòDashboard*;
  - *S2M*.

### 4.1.2 Experiments

Various experiments have been carried out to test the *GiòEnv* system, checking the functionality both individually and in relation to the rest of the system.

As for the physical infrastructure, this was tested by checking various cases in the communication between the *Micro:bit*s and also simulating sudden interruptions of some devices. In particular, the *Micro:bit servers* have been tested by simulating peaks in the quantity and frequency of messages in communications. The *Micro:bit*s have been tested in two configurations: 1 *server* and 3 *GiòButton*s and 2 *server* and 2 *GiòButton*s

The various microservices were instead tested using *PostMan*[21] to send various types of requests and to verify the responses of the services. This testing methodology was carried out both for individual microservices and to simulate entire scenarios. Moreover, for *S2M* we also implemented a simple CLI to create small testing scripts to interact easily and intuitively.

To test the entire system we also created simple Python scripts that simulated certain scenarios, creating various *Virtual Room*s and *Virtual GiòButton*s and simulating the notification of changes in the state of the rooms. This is to have some "standard" situations with which to test the system and the changes made. In this way, certain situations could be easily and uniformly tested without having to reproduce them physically.

[21] *PostMan* (https://www.postman.com/) is a platform for API development.

We also carried out numerous tests, manually simulating changes in environmental parameters and user entry and exit. The simulated scenario was that of three rooms, one of which independent and two that shared the air conditioning, and five users. The simulations were carried out by acting on the sensors in order to reproduce a given situation and after having received the commands from *GiòEnv*, the sensors were still operated to implement the required status. In the meantime, user entrances and exits were simulated via *PostMan*.

## 4.2 Use Cases

We will now discuss some possible lifelike *GiòEnv* use cases to illustrate its potential. The examples will be treated considering likely situations within the Computer Science Department of the University of Pisa. We will focus mainly on mediation and conflict resolution policies which will show how *GiòEnv* can adapt to many different situations. We will employ a minimal environment scenario, so as not to make the treatment too complex and long-winded but at the same time be able to present non-trivial situations. The environment considered will, therefore, consist of two rooms that share the actuator for the temperature but have their own actuator for the light. We will also present three user archetypes.

**The Environment** The rooms are so defined:

```
room(room1).
actuator(ac, temperature, room1).
actuator(shutters1, light, room1).
```

```
room(room2).
actuator(ac, temperature, room2).
actuator(shutters2, light, room2).
```

The environmental parameters will vary according to the proposed scenario, in order to trigger certain behaviors to illustrate the potential of the proposed policies.

**The Users** We will call the first user that we are going to introduce `spendthrift user`, this user is, in fact, the archetype of a user who always wishes the maximum brightness and the minimum temperature if inside it is hot, or maximum temperature if inside it is cold.

```
user(spendthrift_user).

set(spendthrift_user, Room, light, high) :- inRoom(spendthrift_user, Room).

set(spendthrift_user, Room, temperature, very_low) :-
                                            inRoom(spendthrift_user, Room),
                                            (temperature(high, Room)
                                            ;
                                            temperature(very_high, Room)).

set(spendthrift_user, Room, temperature, very_high) :-
                                            inRoom(spendthrift_user, Room),
                                            \+ temperature(high, Room),
                                            \+ temperature(very_high, Room).
```

The second user is instead a user attentive to the environment who therefore tries to waste as little energy as possible, always wishing the minimum brightness and adapting to the room temperature.

```
user(eco_user).

set(eco_user, Room, light, low) :- inRoom(eco_user, Room).

set(eco_user, Room, temperature, Value) :- inRoom(eco_user, Room),
                                           temperature(Value, Room).
```

Finally, the last user we will call `the_intern` expresses his preferences only if he is in the room assigned to him: `room1`. In this case, the user who is chilly always wants a high temperature and, also, never wishes a low brightness.

```
user(the_intern).

set(the_intern, room1, light, medium) :- inRoom(the_intern, room1),
                                         light(low, room1).

set(the_intern, room1, temperature, high) :- inRoom(the_intern, room1).
```

Furthermore, in all scenarios we will use the following conversion table between literal and numerical values:

```
value(temperature,very_high,24).
value(temperature,high,22).
value(temperature,medium,20).
value(temperature,low,18).
value(temperature,very_low,18).

value(light,high,250).
value(light,medium,180).
value(light,low,100).
```

### 4.2.1 Remember to Turn Off the Light

Let us start our scenarios by proposing a very simple but useful policy to save energy. The administrator automates the turning off of the lights and the lowering of the temperature when a room is empty. In particular, during the day the temperature is brought to a medium temperature, and at night it is lowered again.

Now we can define the policy for brightness:

```
mediate(A,light,_,todo(A,light,W)) :-
      actuator(A,light,R),
      \+ inRoom(_,R),
      value(light,low,W).
```

and for temperature:

```
mediate(A,temperature,_,todo(A,temperature,W)) :-
      actuator(A,temperature,R),
      \+ inRoom(_,R),
      \+ daytime(evening),
      value(temperature,medium,W).

mediate(A,temperature,_,todo(A,temperature,W)) :-
      actuator(A,temperature,R),
      \+ inRoom(_,R),
      daytime(evening),
      value(temperature,low,W).
```

Finally, the Policy of the Majority is that defined in Fig. 3.7.

Suppose it is afternoon and that `eco_user` is in `room2` and `room1` is empty.

```
daytime(afternoon).
inRoom(eco_user, room2).
```

We also define the following room states:

```
temperature(medium,room1).
outdoor_temperature(high,room1).
light(low,room1).
outdoor_light(high,room1).

temperature(medium,room2).
outdoor_temperature(high,room2).
light(high,room2).
outdoor_light(medium,room2).
```

in this case the result we will have will be:

```
todo(ac, temperature, 20).
todo(shutters1, light, 100).
todo(shutters2, light, 100).
```

Indeed, being `eco_user` the only user present, only his preferences will apply, which concern `room2` and therefore the `ac` and `shutters2` actuators, which according to the rules expressed by the user will be set to `medium` and `low` respectively. `shutters1` is serving an empty room and is therefore set to `low` because it is afternoon.

The new state of the rooms will then become:

```
temperature(medium,room1).
light(low,room1).

temperature(medium,room2).
light(low,room2).
```

Suppose now that `the_intern` enters in `room1`:

```
inRoom(the_intern, room1).
```

the system will then send the following commands:

```
todo(ac, temperature, 22).
todo(shutters1, light, 180).
todo(shutters2, light, 100).
```

with the state of the rooms which will be updated in the following way:

```
temperature(high,room1).
light(medium,room1).

temperature(high,room2).
light(low,room2).
```

### 4.2.2 Season-Wise

Again with a view to energy savings, an interesting policy could take into account the current season. For example, the administrator could choose between the requests for the temperature, the highest one in spring and summer, and the lowest one in autumn and winter, in order to satisfy at least a part of the users, maximizing energy savings. We must, therefore, add a predicate indicating the season:

```
season(S).
```

where `S` could be `spring, summer, autumn, winter`.

To be able to do the minimum and the maximum we must be able to work with numbers instead of literals, we must therefore implement a function that converts a list of preferences express through literal values into a list of the respective numeral values expressed by the `value` association:

```
convert(Type,Xs,Ys) :- convert(Type,Xs,[],Ys).
convert(_,[],Ls,Ls).
convert(Type,[(U,V)|Xs],Ls,Ys) :-
                        value(Type,V,Y),
                        convert(Type,Xs,[Y|Ls],Ys).
```

Policies can then be written as:

```
mediate(A,temperature,[X|Xs],todo(A,temperature,W)) :-
                                    convert(temperature,[X|Xs],Ls),
                                    (season(summer); season(spring)),
                                    max_list(Ls,W).

mediate(A,temperature,[X|Xs],todo(A,temperature,W)) :-
                                    convert(temperature,[X|Xs],Ls),
                                    (season(winter); season(autumn)),
                                    min_list(Ls,W).
```

Also in this case, we maintain the Policy of the Majority, defined in Fig. 3.7.

Suppose now that it is summer and that `the_intern` and `spendthrift_user` are in `room1` while `eco_user` is in `room2`:

```
season(summer).
inRoom(eco_user, room2).
inRoom(spendthrift_user,room1).
inRoom(the_intern, room1).
```

let the status of the rooms be the following:

```
temperature(high,room1).
light(medium,room1).

temperature(high,room2).
light(low,room2).
```

then the actions that *GiòEnv* will take will be:

```
todo(ac, temperature, 22).
todo(shutters1, light, 250).
todo(shutters2, light, 100).
```

because `eco_user` and `the_inter` prefer a `high` temperature while `spendthrift_user` prefers a `low` temperature but being summer we choose the maximum.

The rooms will then take on the following status:

```
temperature(high,room1).
light(high,room1).

temperature(high,room2).
light(low,room2).
```

### 4.2.3 Virtue Stands in the Middle

The last policy we are going to illustrate allows us to consider everyone's needs without excluding anyone. This is possible by averaging the requests. Each request is transformed into its numerical consideration and is mediated together with the others, the result will be sent to the actuators.

We can easily implement this policy using the `convert` predicate created previously:

```
mediate(A,Type,[X|Xs],todo(A,temperature,Average)) :-
                                        convert(Type,[X|Xs],Ls),
                                        sum_list(Ls, Sum),
                                        length(Ls, Length),
                                        Average is Sum / Length.
```

We also maintain the Policy of the Majority.

Let us suppose this scenario:

```
inRoom(eco_user, room2).
inRoom(spendthrift_user,room1).
inRoom(the_intern, room1).
```

and:

```
temperature(medium,room1).
light(low,room1).

temperature(medium,room2).
light(low,room2).
```

The actions will then be:

```
todo(ac, temperature, 22).
todo(shutters1, temperature, 215).
todo(shutters2, temperature, 100).
```

For the air conditioning `eco_user` wishes a `medium` temperature (20), `the_intern` wishes it `high` (22) and `spendthrift_user` wishes it `very_high` (24) so the average is 22. For `shutters1` `spendthrift_user` wishes the brightness `high` (250) and `the_intern` wishes it `medium` (180) so we get 215 on average. Finally, `eco_user` wishes a `low` brightness ad it is implemented because it is the only preferences for `shutters2`.

# Chapter 5

# Conclusions and Future Work

## 5.1 Summary

In this thesis, after presenting the context, the motivations and the objectives of this work (Chapter 1), and briefly introducing the technologies exploited throughout the thesis (Chapter 2), we have described the design and prototyping implementation of a microservice-based system, *GiòEnv*, for the goal-driven management of IoT-enabled Smart Environments, where users can express their preferences about target temperature and indoor lighting, which are mediated by policies set by the administrator (Chapter 3) through a stateless microservice whose task is to perform the conflict resolution process. Finally, we have illustrated the testbed environment used and the tests carried out on the deployment of the prototype *GiòEnv*. (Chapter 4).

To conclude, in this chapter, we discuss:

- some related work,

- a critical assessment of the proposed contributions,
- possible extensions and some lines for future work.

## 5.2 Related Work

A smart environment is a physical environment in which perception, actuation, and computation capabilities are integrated with the aim of acquiring and exploiting knowledge in order to adapt to users' preferences and requirements [54]. Automated management of the environments to satisfy users' needs and maximise energy savings is one of the crucial points, and many studies have been done on it by investigating numerous approaches. [6, 7].

To this end, various works have focused on fuzzy logic [8] and neural networks [11] or their combinations [55, 56]. Fuzzy logic fits well in applications where input values are not precisely measured and provide a simple description, using linguistic rules, of complex expressions [57] while neural networks can be used to predict energy consumption more reliably than traditional techniques [58, 59]. Differently from those works, *GiòEnv* moves a step towards a declarative approach based on *Prolog* which enables writing concise, easy to understand, modify, and maintain preferences and mediation rules. Besides, by relying on the state-of-the-art resolution, *GiòEnv* decision-making can be *explained* via the proofs obtained by the *LPaaS*.

Other approaches proposed the use of Multi-Agent Systems (MAS) [60] in which the environment is modeled as a set of agents that interact with each other and with users [61]. In [62] the use of a multi-agent approach in the field of smart environments is discussed. The importance of social attitudes and norms in the creation of multi-agent systems for intelligent building control is discussed in [10]. Another example of applying a multi-agent system for controlling smart environments is illustrated in [63], where machine learning techniques, to predict inhabitant movement patterns and typical activities, are proposed. The use of agents, in the context of smart environments, allows a simple system modeling, various types of agents are created with various types of responsibilities from user comfort to energy saving in the rooms. The central point is the process of mediation/negotiation between agents [9, 64]. Soon, IoT will see the development of intelligent objects capable of social interactions (Social IoT) [65]. *Speaking Objects* will be able to take advantage of the use of the speech argumentation. In particular, they will improve the interpretation of the decision-making process and tolerance to uncertainty [65, 66].

In [67], a goal-driven approach based on agents and semantic web is presented, the idea of applying the semantic web to the area of the smart environment would be interesting integrate into *GiòEnv*. Another goal-oriented approach for smart environments can be found in [68], but this work does not take consider the conflict resolution among agents when they have competing goals. The goal-oriented approach is particularly suitable for the management of intelligent environments because users typically know what they want but do not know what to do to achieve their goal [68].

Hierarchical goal management is also studied in [69, 70, 71]. A hierarchical approach consists of dividing the goals into sub-goals which can be solved in parallel or sequence. Using a hierarchical approach in the IoT sector can be very interesting as discussed in [70]. For instance, in [71] this approach is discussed regarding security in smart environments and in particular smart offices due to the high number of potential users, devices and spaces, and the diversity of security roles. However, hierarchical management is also used in other fields, such as many-core resource allocation [69] The management of the goals made by *GiòMediator* is not hierarchical, indeed the various goals expressed by the users are considered together in a single computation. This makes the administration's policy formulation easier for final users of the system.

In [46, 47] *LPaaS* is proposed, a service whose purpose is to provide situated intelligence to pervasive, ubiquitous systems. as illustrated in [72] implementing signal-processing algorithms on resource-limited wireless nodes is extremely complex. There, *LPaaS* offers the possibility of exploiting a simple service for data reasoning on-demand in a light-weight, efficient, and decentralized way. *GiòEnv* exploits, thanks to *GiòMediator*, the potential offered by *LPaaS* to perform the mediation process and the resolution of conflicts [73].

## 5.3 Assessment of Results

With this thesis, we moved some first steps in the field of goal-driven management of smart environments following a declarative approach, enabled by *LPaaS*, and prototyped into a microservice-based IoT application for managing A/C and natural lighting of an office environment.

Our work has focussed on designing and implementing a system for managing emerging conflicts between users' goals and between users' goals and the administrator's goals. We have therefore designed a prototype system to manage IoT devices (*GiòButton*s & *GiòButtons Manager*) that monitors the status of the room and triggers the required commands on the involved cyber-physical devices. The digital twins of exploited IoT devices are hosted on a *Web of Things Server*. To enable communication between the IoT and the computing layer, we designed and developed an extension of the *Web of Things* standard, which - with respect to plain *Web of Things* - permits dynamic and completely remote management of Things. As far as the conflict resolution process is concerned, we have developed a service that deals with it (*GiòMediator*) by exploiting an inference engine to analyse the preferences expressed by users and mediate them through the policies set by the administrator. Finally, we have developed two services regarding the user interface: one that shows the current state of the global environment (*GiòInterface*), the second that shows the details and history of the individual rooms (*GiòDashboard*).

The prototyped IoT device used to monitor temperature and brightness of the involved smart rooms, based on *Micro:bit*, is very flexible and simple and could be further extended to account for other possible architectures, before adopting dedicated hardware. Indeed, we believe that the rapid prototyping *Micro:bit* platform could allow experimenting more, before converging on a final solution.

The possibility to change hardware so easily is given by the fact that *GiòEnv* has been designed to permit changing the system components modularly, by reducing programming effort in case interfaces are kept as they are. In the case of *GiòButton*s , this is possible because we have established a simple protocol for the exchange of messages between IoT devices and the system if they cannot connect to the internet or otherwise through the *Web of Things* standard. In addition, the *Thing Description* is an abstraction of IoT devices, which allows the change of hardware with little work, necessary, to add or eliminate the sensors or actuators offered by that type of hardware. Last but not least, The proposed microservice architecture allows the various components to be exploited even in sectors and ways other than those proposed, thus guaranteeing good flexibility.

We believe that the *Web of Things*-based monitoring and actuation systems can be used independently of the decision system: the latter may not even exist or could be transformed into a semi-automatic one in which humans make decisions and machines execute them.

*GiòMediator* is among the most flexible components of *GiòEnv*, being independent of the particular application and usable in any automatic or semiautomatic decision-making process because exploits all the power and the flexibility of the logic programming.

As support to human decision-making and system monitoring, the dashboard service and the interactive map of the building allow the possibility of simply checking the entire state of the environment and, if necessary, requesting the details and history of the single rooms. The information offered is however limited, a greater quantity and variety of information would lead to better insights.

We believe that the most important contributions proposed with this thesis concern the *Web of Things Server* and the *GiòMediator*.

***Web of Things*** As for the *Web of Things Server* we believe that the most interesting aspects we proposed is the extension of the *Web of Things* REST API. This proposal was born from looking for an effective and uniform method for letting IoT devices and servers communicate.

We implemented an extension of the *Web of Things* standard because we believe that standardising also the communication between the device and its virtual counterpart can benefit the *Web of Things*, making it more solid and flexible. We need to investigate various aspects regarding the extension, above all from the point of view of security: in fact, our proposal allows the user to specify customs scripts to be executed, which requires appropriate caution. The possibility of remotely creating and deleting Things, enabled by the proposed extension, may have very interesting implications, such as the possibility of offering an infrastructure on which customers can create their own objects, perhaps also taking advantage of built-in components.

***GiòMediator*** This component takes care of the conflict resolution process. This service is an *LPaaS*, this improves flexibility and expressive power with respect to traditional procedural systems, by relying on first-order logic. Both users and the administrator can, therefore, express their goals through the writing of simple *Prolog* clauses and the service will take care of implementing the decision-making process. The flexibility of the system can lead it to adapt even to much more complex environments by simply (and carefully) changing the modeling made for the environment.

To conclude, we believe it is necessary to add that in this prototype phase it was not possible to perform a live deployment, despite having all the features and capabilities to be able to effectively manage an environment. For now, *GiòEnv* limits itself to perceiving the state of the rooms and sending the commands for updating, however, these commands go up to the *GiòButton*s but are not actually enacted. To make *GiòEnv* fully functional, it is therefore only necessary to connect the room actuators to the IoT devices.

## 5.4 Future Work

In this section we indicate five main lines to extend and improve the *GiòEnv* prototype presented in this thesis. Namely:

**Improving *Web of Things Server*** An important point to develop is undoubted that relating to the security of the *Web of Things Server*. From this point of view, the W3C still working to come up with a complete proposal [38]. Also adding different levels of security for different methods (as proposed for the management of the extension with PATCH) could be taken into account for a non-prototype version of the system. Another idea that we considered very interesting but that we have not developed as it is not inherent to the thesis work is certainly that of being able to use the *Web of Things* as a tool to perform the Remote Method Invocation in a simple, effective and interoperable way. With the *Web of Things* it would be possible to develop a technology for uniform and interchangeable RMI between the various programming languages: representing objects through *Thing Description* with actions that reflect the methods and properties the attributes.

Moreover, through the extension proposed by us of the *Web of Things* it is possible to create *Thing Description*s in a dynamically: all this is possible thanks to the new representation of the *Thing Description* that allow users to compose the elements of the description with the associated scripts. We therefore find it interesting to develop this idea in order to make the creation of the *Thing Description*s itself modular: creating a database of various components of the *Thing Description*s such as actions, property events, or a set of them. In this way when a *Thing Description* is created there is no need to write all the information but the user can link some parts directly to the element of the database he wants to use. Thus creating a dynamic, easy to use the system to bring reusability even in the *Web of Things*.

**Different Mediation Functions** Testing different mediation functions and therefore be able to measure which one achieves the right balance between energy saving and user satisfaction. In addition to standard mediation functions, we also consider it interesting to evaluate the possibility of implementing some through machine learning. Finally, introducing fuzzy logic to carry out mediation [8] can be a very interesting starting point towards more efficient and dynamic conflict management.

**Automatic Recognition of Users & Prediction of Preferences** Automatic user recognition would make the whole system more effective and user friendly: being able to automatically know the user's position within the environment and the rooms, the system could automatically adapt its parameters based on the users present without them having to manually report their presence. By collecting data on each user, it may also be possible, thanks to machine learning, to predict the user's preferences without the need for them to enter them manually, if not in a first phase. Furthermore, these rules could easily adapt to the change of preferences of a certain user or still recognize some wishes that not even the user knew he had.

**User Interface** As illustrated, the *GiòDashboard* service allows the creation of dashboards on demand, but for the moment the format of the dashboards is very simple and does not allow for customisation. Adding various ways of visualising the same data would be necessary to actually release the software. Users would be able to ask how they want to view the data by specifying what type of charts they want to have. In this way, the same sequence of data could be viewed in different ways by different users: all without having to access the service but specifying at the time of the request what they want to see. Another interesting aspect is undoubtedly that of offering a GUI to express users' preferences.

**Live Deployment & Questionnaires** Deploying in a real environment in which to actually test the system is essential to properly measure the *GiòEnv* performance: both as regards energy consumption and as regards the satisfaction of office users, measurable with questionnaires. We also believe that this system can be used in other applications where shared resources need to be managed. For this reason, we also consider it interesting to evaluate the applicability of the prototyped solution to other use cases.

# Bibliography

[1] Gartner. *Leading the IoT*. 2017.

[2] Statista. *Internet of Things connected devices installed base worldwide from 2015 to 2025*. 2020.

[3] Yusuf Perwej et al. "An extended review on internet of things (iot) and its promising applications". In: *Communications on Applied Electronics (CAE), ISSN* (2019), pp. 2394–4714.

[4] Amelie Gyrard and Amit Sheth. "IAMHAPPY: Towards an IoT knowledge-based cross-domain well-being recommendation system for everyday happiness". In: *Smart Health* 15 (2020), p. 100083. URL: `http://www.sciencedirect.com/science/article/pii/S2352648319300479`.

[5] Antonio Fernández-Caballero et al. "Smart environment architecture for emotion detection and regulation". In: *Journal of Biomedical Informatics* 64 (2016), pp. 55–73. URL: `http://www.sciencedirect.com/science/article/pii/S1532046416301289`.

[6] S. Merabti, B. Draoui, and F. Bounaama. "A review of control systems for energy and comfort management in buildings". In: *2016 8th International*

*Conference on Modelling, Identification and Control (ICMIC)*. 2016, pp. 478–486.

[7] Eric Torunski et al. "A Review of Smart Environments for Energy Savings". In: *Procedia Computer Science* 10 (2012). ANT 2012 and MobiWIS 2012, pp. 205–214. URL: `http://www.sciencedirect.com/science/article/pii/S1877050912003869`.

[8] Ahmed Salih. "Fuzzy Expert Systems to Control the Heating, Ventilating and Air Conditioning (HVAC) Systems". In: *International Journal of Engineering Research and Technology* 4 (Aug. 2015).

[9] Paul Davidsson and Magnus Boman. "Saving Energy and Providing Value Added Services in Intelligent Buildings: A MAS Approach". In: vol. 1882. Jan. 2000, pp. 166–177.

[10] Darren Booy et al. "A Semiotic Multi-Agent System for Intelligent Building Control". In: (Feb. 2008).

[11] Rajesh Kumar, Rajeev Aggarwal, and Jyoti Sharma. "Energy analysis of a building using artificial neural network: A review". In: *Energy and Buildings* 65 (Oct. 2013), pp. 352–358.

[12] *IFTTT: If This Then That*. URL: `https://ifttt.com/`.

[13] *Google Home device specifications*. URL: `https://support.google.com/googlenest/answer/7072284`.

[14] *What Is Alexa?* URL: `https://developer.amazon.com/en-US/alexa`.

[15] Christian Becker et al. "Pervasive computing middleware: current trends and emerging challenges". In: *CCF Transactions on Pervasive Computing and Interaction* 1 (Feb. 2019).

[16] S. Dustdar, C. Avasalcai, and I. Murturi. “Invited Paper: Edge and Fog Computing: Vision and Research Challenges”. In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2019, pp. 96–9609.

[17] P. Habibi et al. “Fog Computing: A Comprehensive Architectural Survey”. In: *IEEE Access* 8 (2020), pp. 69105–69133.

[18] Flavio Bonomi et al. “Fog Computing: A Platform for Internet of Things and Analytics, Big Data and Internet of Things: 169 A Roadmap for Smart Environments, Studies in Computational Intelligence 546”. In: Mar. 2014.

[19] *GIÒ: a Fog computing testbed for research & education*. URL: `https://www.researchgate.net/project/GIO-a-Fog-computing-testbed-for-research-education`.

[20] Martin Cooper. “Meet the BBC Micro:Bit”. In: *ITNOW* 61.3 (2019), pp. 14–15.

[21] Sam Newman. *Building microservices: designing fine-grained systems.* ” O’Reilly Media, Inc.”, 2015.

[22] B. Butzin, F. Golatowski, and D. Timmermann. “Microservices approach for the internet of things”. In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016, pp. 1–6.

[23] Petar Krivic et al. “Microservices as Agents in IoT Systems”. In: *Agent and Multi-Agent Systems: Technology and Applications*. Ed. by Gordan Jezic et al. Cham: Springer International Publishing, 2017, pp. 22–31. ISBN: 978-3-319-59394-4.

[24] M. A. Razzaque et al. “Middleware for Internet of Things: A Survey”. In: *IEEE Internet of Things Journal* 3.1 (2016), pp. 70–95.

[25] Simon Duquennoy, Gilles Grimaud, and Jean-Jacques Vandewalle. "The Web of Things: Interconnecting Devices with High Usability and Performance". In: *Proceedings - 2009 International Conference on Embedded Software and Systems, ICESS 2009* (May 2009).

[26] Deze Zeng, Song Guo, and Zixue Cheng. "The web of things: A survey". In: *JCM* 6.6 (2011), pp. 424–438.

[27] A. Brogi et al. "Bonsai in the Fog: An active learning lab with Fog computing". In: *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*. 2018, pp. 79–86.

[28] *Micro:bit Developer Community-Hardware Description*. URL: `http://tech.microbit.org/hardware/`.

[29] *Micro:bit Bluetooth Low Energy*. URL: `https://lancaster-university.github.io/microbit-docs/resources/bluetooth/bluetooth_profile.html`.

[30] *Micro:bit Radio*. URL: `https://lancaster-university.github.io/microbit-docs/ubit/radio/`.

[31] *Micro:bit Docs*. URL: `https://makecode.microbit.org/reference/`.

[32] *Micro:bit Runtime*. URL: `https://lancaster-university.github.io/microbit-docs/`.

[33] *Micro:coin*. URL: `https://makecode.microbit.org/projects/micro-coin`.

[34] Web of Things Working Group. *Web of Things: Architecture*. Version W3C Recommendation. Apr. 2020. URL: `https://www.w3.org/TR/wot-architecture/`.

[35] Web of Things Working Group. *Web of Things: Thing Description.* Version W3C Recommendation. Apr. 2020. URL: https://www.w3.org/TR/wot-thing-description/.

[36] Web of Things Working Group. *Web of Things: Scripting API.* Version W3C Working Draft. Oct. 2019. URL: https://www.w3.org/TR/wot-scripting-api/.

[37] Web of Things Working Group. *Web of Things: Binding Templates.* Version W3C Working Group Note. Jan. 2020. URL: https://www.w3.org/TR/wot-binding-templates/.

[38] Web of Things Working Group. *Web of Things: Security and Privacy Guidelines.* Version W3C Working Group Note. Nov. 2019. URL: https://www.w3.org/TR/wot-security/.

[39] Douglas Crockford. *The application/json Media Type for JavaScript Object Notation (JSON). Internet informational RFC 4627.* July 2006.

[40] Roy Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. 2000. Chap. 5. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.html.

[41] D. Guinard, V. Trifa, and E. Wilde. "A resource oriented architecture for the Web of Things". In: *2010 Internet of Things (IOT).* 2010, pp. 1–8.

[42] Erik Wilde. "Putting things to REST". In: (2007).

[43] Alain Colmerauer and Philippe Roussel. "The birth of Prolog". In: (Nov. 1992). URL: http://alain.colmerauer.free.fr/alcol/ArchivesPublications/PrologHistory/19november92.pdf.

[44] Robert Kowalski. “Predicate Logic as Programming Language”. In: vol. 74. Jan. 1974, pp. 569–574.

[45] Ivan Bratko. *Prolog programming for artificial intelligence.* Pearson Education Canada, 2012. ISBN: 9780321417466.

[46] ROBERTA CALEGARI et al. “Logic programming as a service”. In: *Theory and Practice of Logic Programming* 18.5-6 (2018), pp. 846–873.

[47] R. Calegari et al. “Logic Programming as a Service (LPaaS): Intelligence for the IoT”. In: *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC).* 2017, pp. 72–77.

[48] Lynne E Parker. “Distributed Intelligence: Overview of the Field and its Application in Multi-Robot Systems.” In: *AAAI Fall Symposium: Regarding the Intelligence in Distributed Intelligent Systems.* 2007, pp. 1–6.

[49] Cesare Pautasso and Erik Wilde. “Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design”. In: *Proceedings of the 18th International Conference on World Wide Web.* WWW ’09. Madrid, Spain: Association for Computing Machinery, 2009, pp. 911–920. ISBN: 9781605584874. URL: https://doi.org/10.1145/1526709.1526832.

[50] Robert C Martin. *Agile software development: principles, patterns, and practices.* Prentice Hall, 2002.

[51] Ben Francis, ed. *Mozilla Web Thing API.* URL: https://iot.mozilla.org/wot/.

[52] *Thingweb.* URL: https://www.thingweb.io/.

[53] *ProbLog.* URL: https://dtai.cs.kuleuven.be/problog/.

[54] Franco Cicirelli et al. “Metamodeling of Smart Environments: from design to implementation”. In: *Advanced Engineering Informatics* 33 (2017), pp. 274–284. URL: http://www.sciencedirect.com/science/article/pii/S1474034616302063.

[55] Sareh Naji et al. “Application of adaptive neuro-fuzzy methodology for estimating building energy consumption”. In: *Renewable and Sustainable Energy Reviews* 53 (2016), pp. 1520–1528.

[56] Wu Jian and Cai Wenjian. “Development of an adaptive neuro-fuzzy method for supply air pressure control in HVAC system”. In: *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0.* Vol. 5. IEEE. 2000, pp. 3806–3809.

[57] Francesco Calvino et al. “The control of indoor thermal comfort conditions: introducing a fuzzy adaptive controller”. In: *Energy and Buildings* 36.2 (2004), pp. 97–102. URL: http://www.sciencedirect.com/science/article/pii/S0378778803001312.

[58] William J Stevenson. *Using artificial neural nets to predict building energy parameters.* Tech. rep. American Society of Heating, Refrigerating and Air-Conditioning Engineers . . ., 1994.

[59] Nivine Attoue, Isam Shahrour, and Rafic Younes. “Smart building: Use of the artificial neural network approach for indoor temperature forecasting”. In: *Energies* 11.2 (2018), p. 395.

[60] Michael Wooldridge. *An Introduction to MultiAgent Systems.* 2nd. Wiley Publishing, 2009. ISBN: 0470519460.

[61] Huib Aldewereld, Virginia Dignum, and Wamberto W. Vasconcelos. “Group Norms for Multi-Agent Organisations”. In: *ACM Trans. Auton. Adapt. Syst.* 11.2 (June 2016). URL: https://doi.org/10.1145/2882967.

[62] Diane J Cook. “Multi-agent smart environments”. In: *Journal of Ambient Intelligence and Smart Environments* 1.1 (2009), pp. 51–55.

[63] Diane J. Cook, Michael Youngblood, and Sajal K. Das. “A Multi-agent Approach to Controlling a Smart Environment”. In: *Designing Smart Homes: The Role of Artificial Intelligence.* Ed. by Juan Carlos Augusto and Chris D. Nugent. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 165–182. ISBN: 978-3-540-35995-1. URL: https://doi.org/10.1007/11788485_10.

[64] C. Becker et al. “BASE - a micro-broker-based middleware for pervasive computing”. In: *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, 2003. (PerCom 2003).* 2003, pp. 443–451.

[65] M. Lippi et al. “An Argumentation-Based Perspective Over the Social IoT”. In: *IEEE Internet of Things Journal* 5.4 (2018), pp. 2537–2547.

[66] L. Amgoud, N. Maudet, and S. Parsons. “Modelling dialogues using argumentation”. In: *Proceedings Fourth International Conference on MultiAgent Systems.* 2000, pp. 31–38.

[67] A. Andrushevich et al. “Towards semantic buildings: Goal-driven approach for building automation service allocation and control”. In: *2010 IEEE 15th Conference on Emerging Technologies Factory Automation (ETFA 2010).* 2010, pp. 1–6.

[68] Javier Palanca et al. “Designing a goal-oriented smart-home environment”. In: *Information Systems Frontiers* 20.1 (2018), pp. 125–142.

[69] A. M. Rahmani, A. Jantsch, and N. Dutt. “HDGM: Hierarchical Dynamic Goal Management for Many-Core Resource Allocation”. In: *IEEE Embedded Systems Letters* 10.3 (2018), pp. 61–64.

[70] A. Jantsch et al. “Hierarchical dynamic goal management for IoT systems”. In: *2018 19th International Symposium on Quality Electronic Design (ISQED)*. 2018, pp. 370–375.

[71] Iván Marsá-Maestre et al. “A Hierarchical, Agent-based Approach to Security in Smart Offices.” In: *ICUC*. 2006.

[72] G. Fortino et al. “Enabling Effective Programming and Flexible Management of Efficient Body Sensor Network Applications”. In: *IEEE Transactions on Human-Machine Systems* 43.1 (2013), pp. 115–133.

[73] G. Wiederhold and M. Genesereth. “The conceptual basis for mediation services”. In: *IEEE Expert* 12.5 (1997), pp. 38–47.

[74] *Flask*. URL: `https://flask.palletsprojects.com/en/1.1.x/`.

[75] *Reliable UDP Algorithms*. URL: `https://io7m.com/documents/udp-reliable/`.

[76] Doan Thanh Tran and Eunmi Choi. “A reliable UDP for ubiquitous communication environments”. In: 2007.

[77] M. Masirap et al. “Evaluation of reliable UDP-based transport protocols for Internet of Things (IoT)”. In: *2016 IEEE Symposium on Computer Applications Industrial Electronics (ISCAIE)*. 2016, pp. 200–205.

[78] Fahad Al-Dhief, Naseer Sabri, and Musatafa Albadr. “Performance Comparison between TCP and UDP Protocols in Different Simulation Scenarios”. In: (Dec. 2018).

[79] Zhaojuan Yue, Yongmao Ren, and Jun Li. “Performance evaluation of UDP-based high-speed transport protocols”. In: *2011 IEEE 2nd International Conference on Software Engineering and Service Science*. 2011, pp. 69–73.

[80] D. Madhuri and P. C. Reddy. “Performance comparison of TCP, UDP and SCTP in a wired network”. In: *2016 International Conference on Communication and Electronics Systems (ICCES)*. 2016, pp. 1–6.

[81] Yongmao Ren et al. “Performance Comparison of UDP-based Protocols Over Fast Long Distance Network”. In: *Information Technology Journal* 8 (Apr. 2009).

[82] Aaron Falk et al. “Transport protocols for high performance”. In: *Commun. ACM* 46 (Nov. 2003), pp. 42–49.

[83] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. Tech. rep. June 1999. URL: `https://doi.org/10.17487/rfc2616`.

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

[85] Zach Shelby, Klaus Hartke, and Carsten Bormann. “The constrained application protocol (CoAP)”. In: (2014).

[86] Yunhong Gu and Robert L. Grossman. “UDT: UDP-based data transfer for high-speed wide area networks”. In: *Computer Networks* 51.7 (2007). Protocols for Fast, Long-Distance Networks, pp. 1777–1799.

[87] B. Eckart, Xubin He, and Qishi Wu. “Performance adaptive UDP for high-speed bulk data transfer over dedicated links”. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008, pp. 1–10.

# Appendix A

# First Version of the *Web of Things Server*

## A.1 Mozilla's Web of Things

The first version of the *Web of Things Server* was based on the *Web of Things Server* proposed by Mozilla [51]. We chose this implementation because of its simplicity which made changes and extensions easy to do.

First, we decided to rewrite the server in Flask [74], while keeping the *Thing Description* code, but adapting it to the new infrastructure. We made this choice because, unlike the code offered by the basic implementation, we believe that a server in Flask is more flexible to manage and modify.

We also added the possibility to interact with the server through WebSockets, whose interactions are described but not implemented in the code.

The main problem we had to face was deciding how to make physical devices communicate with digital twins. This is because a way to update the status of a resource has not been standardised [36, 34, 37], if it was read-only (for example sensor values).

We decided to solve the problem through a client-server architecture: the *GiòButtons Manager* would have been the client and would have sent the updates to the *Web of Things Server* who would have managed them appropriately. The problem remained of deciding which protocol to use for communication: a reliable protocol was needed, so as not to lose requests for status changes, but at the same time fast for notifications and requests to arrive as soon as possible. Various options emerged from the literature [75, 76, 77, 78, 79, 80, 81, 82] but none fully satisfied us as we did not find anything suitable for a dynamic IoT environment such as *GiòEnv*. We, therefore, decided to implement a protocol that reflected all our needs: *ReTRo*.

This protocol covered all our needs: it was fast, it was reliable, it did not require initialisation or closure phases and it also performed well in dynamic topological environments. But, the simplicity of the Mozilla's server was beginning to make itself felt: the more the project went on, the more work was required only to adapt the server to our needs and also with the official standardization it was also necessary to readjust the code and the *Thing Description*s to the official requirements. All these considerations led us to re-analyse the available alternatives and we decided to switch to the server offered by Eclipse: ThingWeb[22] [52].

We will now analyse the proposed communication protocol.

[22]see Section 3.2.4.

## A.2 *ReTRo*

### A.2.1 Principles

*ReTRo* – UDP (Real-Time Reliability over UDP) is a communication protocol based on UDP. This protocol is designed for communication between IoT Devices. The principles behind the protocol are:

- a large number of IoT devices sends to each other small messages,
- the frequency of messages can be high or low,
- usually a message represents real-time information,
- messages must arrive at their destination as soon as possible,
- each device can send messages to many devices simultaneously,
- a device usually send different types of messages (e.g. status updates, actions, negotiation, requests),
- the system's topology can be client-server, peer-to-peer, or hybrid and can change dynamically. So, the communication needs to be fast and reliable, it must also allow the exchange of messages between many different devices.

Therefore, this protocol was designed for real-time sensitive IoT applications. The choices made a point in the direction of having fast, reliable communication to many devices simultaneously. Furthermore, given the dynamic nature of IoT systems, the protocol must be able to adapt to changes in topology in the system.

### A.2.2 Characteristics

*ReTRo* is:

- bidirectional,
- multiply-Connected (1 Port – N Connection),
- reliable (allows ordering and acknowledge),
- message oriented,
- p2p Friendly,
- w/o initialisation phase or closing phase,
- with flow control (RTO),
- with implicit window control (Channel),
- with implicit Flow label (Mailbox),
- with parallel management of flows,
- with 32 bit header.

The protocol does not require an initialisation phase and buffers are not used to handle unordered messages. In fact, all these features would slow down communication and considering the real time nature of the messages and their small size, they would not improve performance.

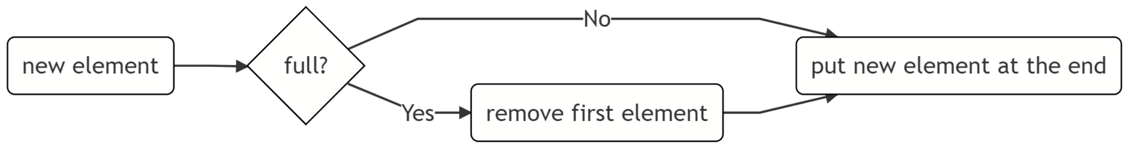


Figure A.1: A diagram showing how a Channel works.

### A.2.3 Definitions

- a User is defined by the pair (ip,port),
- when a User receives or sends a message to another User, not yet met, he creates a Connection: a set of Mailbox each identified by a number. A Connection allows bidirectional communication with a single User (ip, port),
- a Mailbox is defined by two Channels, one for the messages to be sent and the other for those received.,
- a Channel is a special queue: it has a maximum size (max_size) and contains only the max_size most recent messages (see Figure A.1),

### A.2.4 Implementation choices

Given the need for fast and possibly frequent communication of small messages, it was decided to use the UDP protocol instead of TCP.

It was chosen to use the Channels because, given the real-time nature of the information contained in the messages, it would not make sense to store too old information. The choice of using multiple Mailboxes for communication with the same User allows differentiating, already at the communication level, different messages' type. Furthermore, the management of messages in different Mailboxes is completely independent.

The *ReTRo* protocol has the following characteristics to ensure reliability:

- each message has a sequence number and a mailbox number,
- when a message is received an ack message is sent in response,
- if an ack is not received within a certain time limit the message is sent back,
- if the sequence number is less than or equal to the last message received in the same Mailbox, the message is discarded,
- for each Mailbox: as long as you do not receive an ack message for the last message sent, no other messages are sent from the same mailbox until the ack is received or the message is discarded from the Channel because it is too old.

These mechanisms allow us to have a real-time, fast and, reliable protocol. The multi-Mailbox system allows you to manage messages to the same User independently: while waiting for an ack in a specific mailbox, the others can continue communication independently (see Figures A.7 and A.6). The sequence number also allows you to manage the sorting of messages from the same mailbox.

Furthermore, if the last message arrived is for example 42, and arrives 44 before 43, it is not important to manage the situation, because the correct use of Mailboxes allows managing only one type of message for each Mailbox. So message 44 will be an "update" of 43, therefore the information lost in message 43, which even if it arrives at its destination will be discarded, is not important.

### A.2.5 Header

The 32-bit header is composed as follows:

SYN $\rightarrow$ 1 bit

ACK $\rightarrow$ 1 bit

Sequence/Ack number $\rightarrow$ 16 bit

Global number $\rightarrow$ 10 bit

Mailbox number $\rightarrow$ 4 bit

### A.2.6 Summary

*ReTRo* is a protocol designed for dynamic IoT systems, in which the reliability and speed of communications are essential. To implement all these features we have chosen to use the UDP protocol by adding some useful features for our purposes. To start, with a single pair (ip, port) it is possible to communicate, bidirectionally, with many other devices, thanks to the Connection concept, which also allows communication to be started without any initialisation phase. Each connection is also defined by a series of Mailboxes, this allows the user to be able to mark different messages in a different way, implementing a rudimentary Quality of Service system (like Flow labels in IPv6) different mailboxes will manage different message flows independently and therefore parallel to the others mailbox. While messages in the same mailbox will be sent sequentially and will have a sequence number, this allows for reliable and orderly sending of messages within the same mailbox, in addition to the sequence number, there is also a global one, which is not used in the management of communications but can be used by the user to make comparisons between messages in different mailboxes. Finally, a simple and fast window control mechanism is implemented thanks to the Channels, the concept behind this structure is that in a channel there are messages concerning the same flow and in case the channel is full you can discard the older messages because those new will “overwrite” that information and thanks to the classic RTO, we implement flow control.

### A.2.7 Related Work

With *ReTRo* we have tried to implement a reliable IoT data communication protocol. For this reason, we will now mention, to the best of our knowledge, the most used protocols for reliable data communication for IoT systems. We can divide the protocols into two categories: those at the application level and those at the transport level [77, 79].

Among the existing application-level protocols, HTTP [83], MQTT [84] and CoAP [85] are relevant to our study. At the cost of greater communications overhead, these protocols allow great expressiveness and simplicity in their use compared to lower-level protocols.

As far as the transport layer protocols are concerned, the following deserve particular attention: R-UDP [76], UDT [86] and PA-UDP[87]. These protocols are created following the principle of guaranteeing reliability and at the same time the highest possible performance. For this, connection setup mechanisms are used which, in the case of sending small packets not below, may, however, lead to efficiency losses.

### A.2.8 Diagrams

In this section, some diagrams show the functioning of *ReTRo* in various phases.

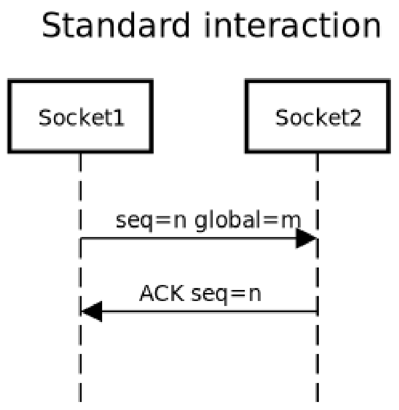


Figure A.2: A standard interaction.

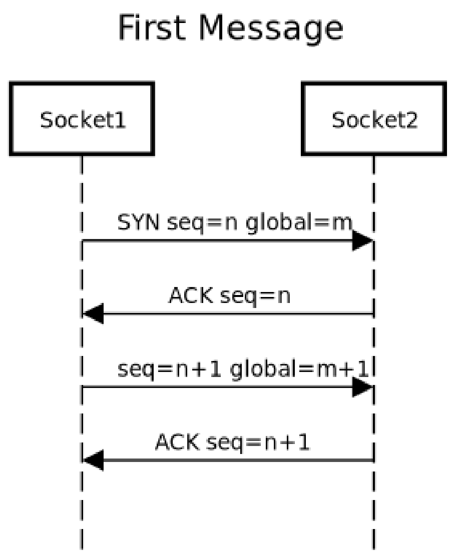


Figure A.3: The first message must have the SYN flag at 1.

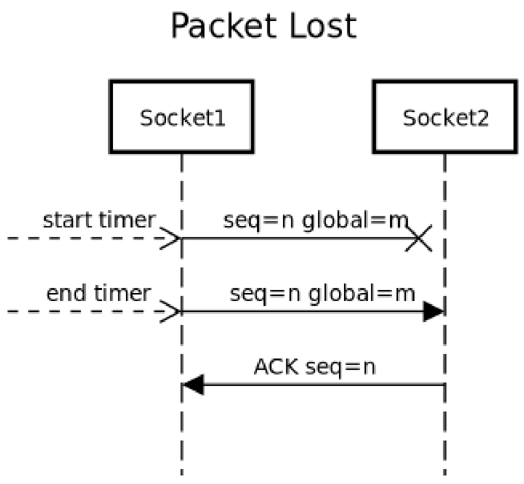


Figure A.4: If a packet is lost, the message is sent again.

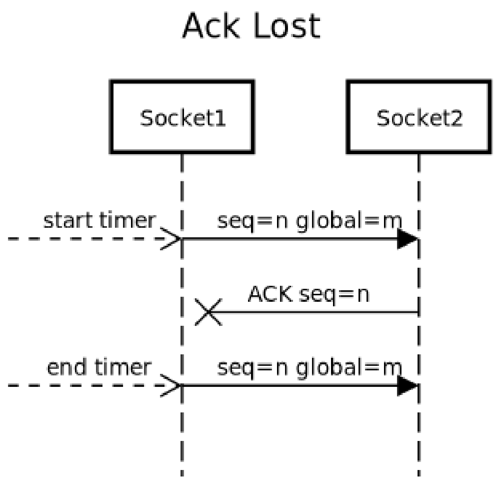


Figure A.5: If an ack is lost, the message is sent again.

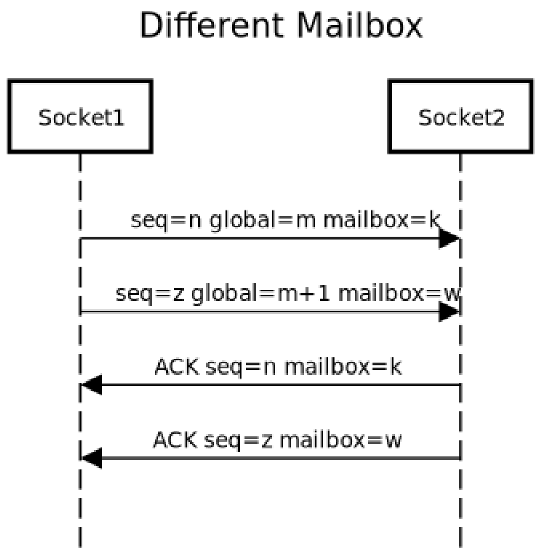


Figure A.6: Messages in different mailboxes can be sent in parallel.

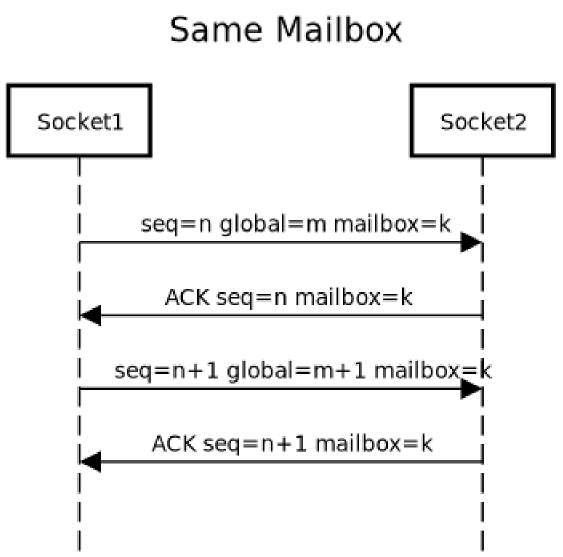


Figure A.7: Messages in the same mailbox must be sent sequentially.

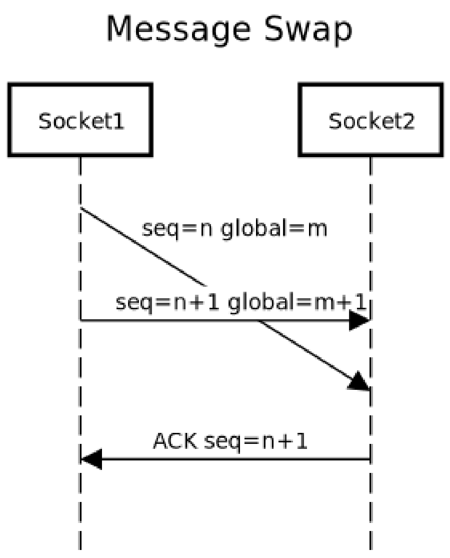


Figure A.8: In the event of an exchange of messages, the last one received (with a lower sequence number) is ignored.

# Appendix B

# Thing Descriptions

Digital twins are implemented through *Thing Descriptions*, as per the standard[23]. Before delving into the description of the rooms and *Micro:bit* , we would like to anticipate that, although the standard envisages the use of security protocols [38], we have chosen not to use it in our implementation, this to facilitate development and because security does not concern the purpose of this thesis, but is foreseen in possible future work.

## B.1 *Virtual Room*

The status of a room can be determined either by the same *Micro:bit* or by two different *Micro:bit*s (one for the temperature the other for the brightness) for which the reference is kept to which *Micro:bit* deals with a certain parameter. It is im-

[23]The *Thing Description* the standard [35] provides for the definition of the properties of the thing, the actions and events that that thing can launch and for each of them the possible ways with which it is possible to interact. The properties form the state of the thing and each of them can be read-only or write-only or readable and writable, moreover, they can be observable (the user is notified when the value changes).

portant to note that the *Micro:bit*s in the state are not those that read the sensors but those that deal with the implementation of the requests. The reference is necessary so that when mediation is activated the *Virtual Room* knows to that *Virtual GiòButton* to forward the request.

The *Virtual Room* status will therefore described by the following:

- `dashboard`: the url of the associated dashboard (read-only),
- `temperature_microbit`: the url of the *Virtual GiòButton* that takes care of temperature management (read-only),
- `light_microbit`: the url of the *Virtual GiòButton* that takes care of brightness management (read-only),
- `last_indoor_update`: timestamp of the last update received (read-only and observable),
- `last_outdoor_update`: timestamp of the latest weather update (read-only and observable),
- `users`: users in the room with their preferences (read-only and observable),
- `temperature`: the last temperature's value received (read-only and observable),
- `temperatureL`: label (`very_low, low, medium, high, very_high`) of the last temperature's value received (read-only and observable),
- `light`: the last brightness' value received (read-only and observable),

- `lightL`: label (`low, medium, high`) of the last brightness' value received (read-only and observable),
- `time`: the current time (read-only and observable),
- `timeL`: label (morning, afternoon, evening, night) of the current time (read-only and observable),
- `outdoor_temperature`: the last outdoor temperature's value received (read-only and observable),
- `outdoor_temperatureL`: label (`very_low, low, medium, high, very_high`) of the last outdoor temperature's value received (read-only and observable),
- `outdoor_light`: the last outdoor brightness' value received (read-only and observable),
- `outdoor_lightL`: label (`low, medium, high`) of the last outdoor brightness' value received (read-only and observable).

We have chosen to associate labels with state properties because we believe that users express their preferences better and more easily through them. For the temperature `very_low` means $< 18$ degrees Celsius, `low` $< 20$, `medium` $< 22$, `high` $< 24$ and `very_high` $>= 24$. For the light (performed in an interval between 0 and 255) `low` means $< 25$, `medium` $< 80$ and `high` $>= 80$. For the time: from 8 to 13 it's morning, until 19 it's afternoon, until 22 it's evening, and then it's night.

We point out that not only internal parameters but also external parameters are considered: this allows the expression of more precise rules and more targeted and efficient management of the rooms. In addition to the environmental parameters, internal and external, the time of day and the number of users[24] present are also taken consideration for the same reasons of expressiveness and efficiency.

Regarding the actions:

- `refresh`: force updating the weather data,
- `enter`: indicates that a certain user has entered the room,
- `leave`: indicates that a certain user has leaved the room,
- `mediate`: starts the mediation process.

Finally, the *Virtual Room*s do not launch events.

## B.2 *Virtual Mediator*

When the `mediate` action of a room is invoked, the status of that is sent to the *Virtual Mediator* which updates its information and starts the mediation process by interacting with *GiòMediator*.

The relative *Thing Description* ha the following properties:

- `rooms`: for each *Virtual Room* there is the most recent status reported,
- `users`: contains each user's preferences and administrator policies.

The actions are:

- `setRules`: set the preferences of a user,

[24]When the room status is sent to the *GiòMediator* the user's list is replaced by only the number of users present.

- `getRules`: get the preferences of a user,
- `mediate`: start a new mediation process,
- `solve`: resolve all conflicts interacting with *GiòMediator*,
- `deploy`: notifies the various *Virtual GiòButton* of the decisions taken.

The *Virtual Mediator* does not launch events.

## B.3 *Virtual GiòButton*

Having used only *Micro:bit* as physical device we will focus on their *Thing Descriptions*, any other type of hardware will however require little adaptation work.

A *Virtual GiòButton* is identified by is friendly name and its status is formed as follows:

- `serial_number`: the serial number of this *Micro:bit* (read-only),
- `dashboard`: the url of the associated dashboard (read-only),
- `light`: the last value recorded by the light sensor (read-only and observable),
- `temperature`: the last value recorded by the temperature sensor (read-only and observable).

The actions associated with a *Virtual GiòButton* are:

- `set`: receives as input a new setting for a parameter and launches the related event.

Finally, the events:

- `setup_temperature`: a new temperature setting is notified,
- `setup_light`: a new brightness setting is notified.

It is precisely through the `setup`s events that the *GiòButtons Manager* and therefore *Micro:bit*s become aware of the results of the mediation and therefore of the new states to perform. For each event of a *Virtual GiòButton*, the *GiòButtons Manager* creates an "observer" who waits, through long polling, for the publication of new events which are then sent through the *Micro:bit server* to the *Micro:bit* affected by that event. Among the various options, we have chosen long polling as it is among the fastest and simplest and therefore among the most supported also in devices with limited resources.

We have chosen to use distinct events for distinct parameters to allow the simultaneous observation of events relating to different parameters.