ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Design and implementation of a chatbot for remote sensors reading and home automation.

Tesi di laurea in
CRITTOGRAFIA

Relatore
Dott. Luciano Margara

Candidato
Alessia Cerami

Sessione Unica di Laurea
Anno Accademico 2020-2021

This thesis is dedicated to my family and my companion of life and madness, Andrea. A special feeling of gratitude to my loving parents, Dario and Mirella, whose words of encouragement and push for tenacity ring in my ears. My brother, Pierguido, who has always supported me.

Acknowledgements

I thank my colleague Andrea Giordano for the help offered to me in this thesis. But, above all, for bearing with me and supporting me in this period.

I thank Professor Luciano Margara for the availability and understanding shown as a thesis supervisor, the passion and professionalism communicated as a teacher, but above all for always encouraging my choices in the professional field.

I thank all the colleagues I met who shared with me the joys and sacrifices of these years of study, filling them with laughter and making them unforgettable.

I thank all my friends for believing in me but above all for showing me continuously, even from a distance, their enormous affection.

I thank all my family. But above all I thank my parents and my brother because without them all this would not have been possible and also for having always shown me their unconditional love by encouraging and supporting me in all my choices.

Contents

1 Introduction **1**

2 Technologies and libraries **5**

2.1 Chatbot: What are we talking about? 6

2.1.1 How do chatbots work? 7

2.1.2 A brief history of chatbots 8

2.2 MQTT 10

2.2.1 MQTT: Key aspects 11

2.2.2 Advantages of MQTT and why we choose it 17

2.3 Ktor 19

2.3.1 Key aspects 19

2.3.2 How Ktor was used? 21

2.4 Main libraries 21

2.4.1 JetBrains Exposed 21

2.4.2 Gson 22

2.4.3 Node.js Telegram Bot API 22

2.4.4 jSerialComm library 23

2.5 Utilities 23

2.5.1 Security mechanisms 23

2.5.2 Raspberry PI 24

2.5.3 Arduino 26

2.5.4 Coroutines and concurrency in Kotlin 26

3 Requirements **29**

3.1 Ubiquitous Language 29

3.2 Non-functional requirements 30
3.3 Functional requirements 31

4 Design 33
4.1 Architecture 33
4.1.1 Description of the architecture inside the project 34
4.1.2 Application of the architecture in the project 36
4.1.3 Clean architecture 38
4.2 Sub-domains of the system 39
4.3 Core logic 41
4.3.1 Bounded Context 41
4.3.2 Core Finite State Machine 42

5 Implementation 45
5.1 JavaScript sub-project: Telegram Bot 46
5.2 Kotlin sub-project: CoreSystem 47
5.2.1 Creation of the Server 48
5.2.2 WhitePaper 50
5.2.3 State 50
5.2.4 UtilityHomeConnection 51
5.3 PublisherClient sub-project 52
5.3.1 InputAnalyzer 52
5.3.2 Detector 53
5.4 HomeAutomationController 53
5.4.1 Store 54
5.4.2 HomeComponentHandler 54

6 Conclusions and Future works 55

A Technologies 63
A.1 MQTT 63
A.1.1 MQTT Topic: # Wildcard 63
A.1.2 QoS 64
A.1.3 Creation of a MQTT Client 64

B Library **67**
B.1 Exposed 67
B.2 Gson 68

C Code representation **71**
C.1 Telegram Bot: Node.js Telegram Bot API 71
C.2 CoreSystem 72
C.3 PublisherClient 73

List of Figures

2.1 MQTT basic aspects 11
2.2 MQTT Topic 12
2.3 MQTT HiveMQ Broker 14
2.4 Ktor main entities 20

3.1 UL of the system 30

4.1 General schema of the chosen architecture 35
4.2 System architecture for sensors' reading. 37
4.3 System architecture for home automation management. 38
4.4 Sub-domain identification 39
4.5 Figure representing all identified bounded context inside Core . . . 41
4.6 Core Finite State Machine 43

5.1 Authentication phase 49

Listings

A.1 Example of multi-level topic in the MQTT Client 63
A.2 Creation of an MQTT Client 65
A.3 Connection to the MQTT Broker using credentials 65
B.1 Data Persistence through Exposed library 68
B.2 Use of Gson to implement (De)serialization 69
C.1 Implementation of the back-end of the Bot application 71
C.2 Creation of the Server using Ktor 72
C.3 Opening websocket 72
C.4 Method to handle websocket 73
C.5 Main function of the project with concurrent threads 74

Chapter 1

Introduction

IT development, and in particular that linked to mobile devices, allows the creation of more and more intelligent messaging applications aimed at making communication itself more convenient and practical.

In this sense, thanks to new technologies, it is possible to establish real communication with these applications that from now on we call **Chatbots**.

An important feature in developing chatbots is the use of Natural Language Processing and sentiment analysis that allows them to establish real communication with users. In fact, these technologies are able to communicate in human language by text or oral speech with humans or other chatbots.

There are several definitions, in literature, about Chatbots [6]. An example is that which is expressed in the Oxford Dictionary [7]: *a Chatbot is a computer program that can hold a conversation with a person, usually over the internet.*

Thanks to their flexibility and ease of use in the real world, the application areas of chatbots are various: they can be used as Customer Service services (FAQs, assistance ...) or to make reservations and recruitments, etc.

A possible use of this new technology is the one reported in this thesis. Here, the chatbot, **SensorReadingBot**, covers two important features:

Remote reading sensor information On the one hand, the chatbot undertakes to provide the user with information, which in turn comes from different sensors scattered around the city. When a request arrives from the user, the Bot application provides a response which varies according to the city requested by the user and the information that sensors send to the Bot application.

Governance of home automation On the other hand, the chatbot allows the user to interact with his home automation system by sending specific commands. These are suggested to the chatbot by each specific actuator and then communicated to the user. For example a user will be able to turn on or off a light bulb using commands that are comprehensible by the specific machine.

The purpose of this thesis is to create a complete distributed system in which the advantages of chat and the world of chatbots are combined with those of sensors, particularly adopted in the IoT field. To do so, two main activities will be performed:

- Design and implementation of the whole system as a container for other sub-systems.
- Design and implementation of a connection layer between sub-systems.

Thesis structure

Accordingly, the reminder of this thesis is structured as follows:

To give the reader a complete experience and to allow him to understand what we are talking about, chapter 2 discusses what a chatbot is, the areas of use and how they have changed up to today. Furthermore it discusses all the main technologies involved in that project thesis.

Chapter 3 discusses the requirements of the system both functional and non-functional.

Chapter 4 discusses the design we decide to adopt and thus explains the architecture we choose.

Chapter 5 discusses how the most important components of the whole system were implemented.

Finally, Chapter 6 concludes this thesis by summarising its main contribution and introducing future works and interesting topics to evaluate.

Chapter 2

Technologies and libraries

A fundamental part of this thesis is the realization, completely from scratch, of a distributed system [1, 21, 32] consisting of multiple software components installed on multiple computers but that runs as a single system.

The realization of the system includes two important steps: first of all we **design** the whole system, then we **develop** it. During this former step, we perform two relevant activities that are divided in research and understanding and study of technologies.

Research and understanding These activities are, in turn, divided into two main tasks. In fact we have to:

- Understand the target of users to whom the bot is intended;
- Understand the goal of the project;

Technologies That activity includes the:

- Analysis of different technologies and libraries;
- Selection of the ones that best suit in terms of reliability, modularity, scalability and performance.

During the latter step, the development one, given the analysis of the problem, we proceed to develop the solution. Thus, we proceed to set up a suitable system architecture and the components by which it is composed.

Architecture Analysis and, then, selection of the architecture of the system and the architectural style to pursue. In this case, the one that best suites is the Clean Architecture;

Components Identification and realization of the individual components belonging to the system.

In order to explain how this thesis has been developed, in the following sections we explain all the main technologies and libraries involved.

2.1 Chatbot: What are we talking about?

A Chatbot is an artificial intelligence application that is able to simulate a real conversation with a user in his natural language using a variety of input methods, such as voice, text, gesture and touch on websites, messaging applications, mobile apps, or telephone.
Thus, it is considered an automatic interface trained to give information to users that request them and it is able to manage requests without any sort of human support.

Using a chatbot has a lot of engagements and advantages, even in the work area and thus has permitted its growth in popularity. In fact, this technology offers: 24-hours availability, multitasking, multiple channels, improvement of user engagement and data tracking. A brief explanation about its utilities:

- Promote **24-hours** availability, allowing continuous communication between

the seller and the customer 24/7. Thus, it permits cost saving because a Business owner does not have to pay their employees to manage requests. Thus, chatbots are able to cut down on staffing expenses.

- Manage multiple requests in **multitasking**. So, chatbots can answer thousands of questions at the same time providing instant answers. In this way, chatbots eliminate the need for employers during online interaction with customers, allowing the reduction of workload and response time.
- Be accessible from **multiple channels**: For example, Telegram, Whatsapp, site web, Skype etc. In this sense, users can freely choose the preferred communication channel without any sort of external imposition.
- **Improve user engagement**: This means that chatbots only give data based on the input provided by users at each time so they do not end up annoying users with irrelevant information and keep the customers engaged for longer by sharing interesting information.
- **Track data**: Chatbots collect feedback from customers, and this information can help a company to improve their services.

2.1.1 How do chatbots work?

Chatbots use artificial intelligence to talk to people and give relevant content or suggestions.

The interaction with a chatbot can take place using voice or text. If voice is used, the chatbot first turns the voice data input into text (using Automatic Speech Recognition (ASR) technology), then analyses the text input. Doing so, a chatbot can identify the user intent and consider the best response to deliver back to the user. They can be categorized in two categories:

- Rules-based
- AI-based

Rules-based

They operate by means of specific commands generating a targeted conversation. For example, the user can interact with rule-based chatbots by clicking on buttons and using predefined options.

A chatbot that works based on rules is, usually, quite limited because it is designed to respond to fixed commands. So, if a person makes a malformed request, this type of chatbot will not understand what the question is, and therefore, will not provide an appropriate response. In this case, the intelligence of the bot solely depends on how it is programmed.

AI-based

An AI is trained in order to recognize what users need. This type of chatbot makes use of advanced technologies such as machine learning, AI and Natural Language Processing to increase its capacity of dialogue and interaction but also to understand and remember the context of the conversation and the user's preferences.

One important aspect of that chatbot is the engine, which is responsible for the transformation of natural language into machine-understandable actions.
A chatbot that uses machine learning understands not only commands but also human language. The user, therefore, does not have to use precise words to get accurate or useful responses. Thus, chatbot learns from interactions that it has with users and can deal with similar situations when they arise later.

2.1.2 A brief history of chatbots

Chatbots became more interesting after the introduction of the Internet as they started to be used to support customer service functions.

The first machine capable of speech using natural language processing was [35]

ELIZA, created in 1966, which simulates human conversation using basic natural language processing techniques [38]. One example of technique used is passing the words that users entered into a computer and then pairing them to a list of possible scripted responses.

Despite being relatively simple, ELIZA is capable of give the illusion of understanding the user's problems and successfully fooled a large number of people.

Then during several decades, other chatbots followed ELIZA's approach such as *PARRY* [18] which imitates a patient with schizophrenia; Jabberwacky [34] which attempts to "simulate natural human chat in an interesting, entertaining and humorous manner"; Dr. Sbaitso [20, 19] which is an artificial intelligence speech synthesis development; *A.L.I.C.E.* [36, 37] (Artificial Linguistic Internet Computer Entity), a natural language processing chatbot, which uses heuristic pattern matching to carry conversations and which simulates a chat conversation with a real person over the Internet.

In 2001, another chatbot appeared, *Smarterchild* [13] which is available on AoL (America onLine) Instant Messenger MSN Messaging networks.

The chatbot offers personalized conversation and brings innovation as it is connected to a knowledge base and holds useful information for its users.

Finally, in the early years, virtual assistants, such as *Siri* [2], *Cortana* [24], *Google Now* [11], *Google Assistant* and *Alexa* arrived.

Siri is an intelligent personal assistant and learning navigator which uses a natural language UI and enables Apple users to command actions or queries from the device. This chatbot uses voice queries to answer questions, performs actions and makes recommendations according to the user's needs. The software is adaptable to users' individual language usages, searches, and preferences.

Google Now, developed in 2012, uses a natural language user interface to answer questions, makes recommendations, and performs actions by passing on requests

to a set of web services. Google Assistant replaced this chatbot in 2017.

In 2015, Alexa appeared. It is an intelligent personal assistant developed by Amazon working through voice communication. Thus, users can search the Web, play music, get news or weather reports, control your smart-home products and more.

In the same year, Cortana appeared. It is an intelligent personal assistant which recognizes natural voice commands, can set reminders and answer questions and perform tasks.

2.2 MQTT

MQTT [28] stands for Message Queuing Telemetry Transport. It is a TCP based **publish** and **subscribe** messaging protocol designed for lightweight machine-to-machine (**M2M**) communication or Internet of Things (**IoT**) types of connections.

The publish-subscribe model is an alternative to the traditional client-server model in which a client communicates directly with a server to request information. In that model, the publishers and subscribers never contact each other directly. In fact, they are not even aware that the other exists.

That means that, instead of communicating with a server, every client devices and applications, in an asynchronous manner, publish and subscribe messages.

Hence, when a node A wants to communicate with a node B, it does not happen synchronously, as if it were a phone call where questions and answers are immediately provided, but the message is published by the node A (*publish*) and is received by the nodes B that subscribed to receive the message (*subscribe*).

The MQTT protocol was originally developed by IBM in 1999 as they needed a protocol for minimal battery loss and minimal bandwidth to connect with oil pipelines via satellite.

Nowadays, the goal is no longer the original one but it is the world of the Internet of Things (IoT), which involves every aspect of our society.

The reference model of the MQTT protocol is the *hub-and-spoke*, which arranges service delivery assets into a network. This consists of an anchor establishment (hub), which offers a full array of services, and secondary establishments (spokes), which offer more limited service arrays. Furthermore, the latter route requests of users needing more intensive services to the hub.

2.2.1 MQTT: Key aspects

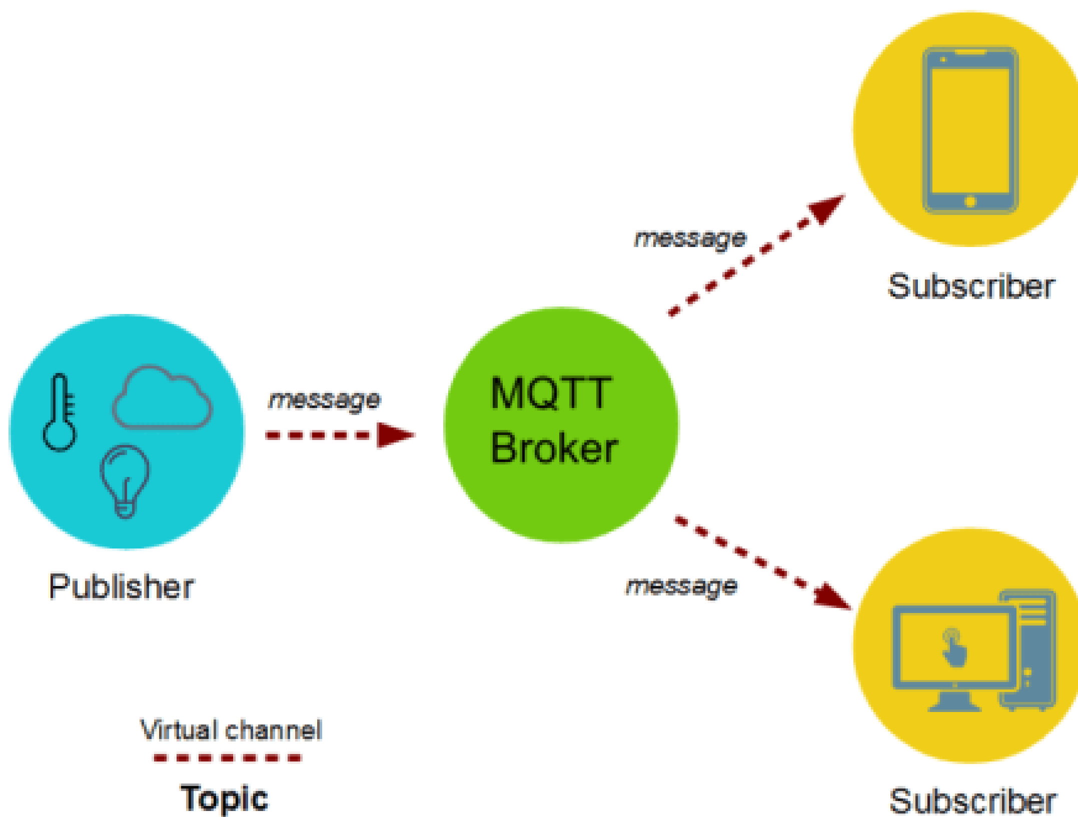

Figure 2.1: MQTT basic aspects

MQTT is a very light weight and binary protocol and, due to its minimal packet overhead, it works well, in comparison to protocols like HTTP, when transferring data over the wire.

As shown in fig. 2.1, the architecture of the MQTT protocol requires two fundamental entities: the *client* (section 2.2.1), which is both publishers and subscribers, and the *broker* (section 2.2.1). These two systems communicate by exchanging messages through a bidirectional channel called *topic* which allows to determine which message goes to which client.
In order to ensure the reader a complete understanding of the selected communication protocol, we explain, in the following paragraphs, the MQTT protocol key aspects and why it is involved in the thesis.

MQTT Topics

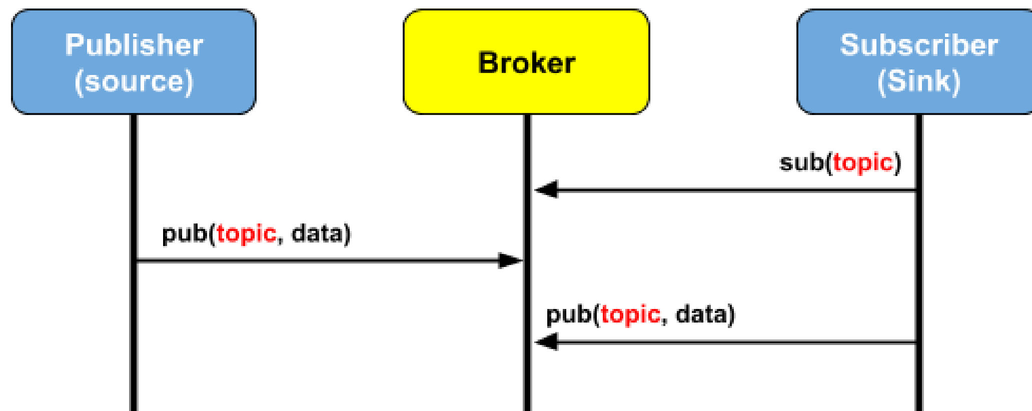

Figure 2.2: MQTT Topic

In MQTT, the word *topic* refers to hierarchical UTF-8 strings which are used by the broker to route messages to specific connected clients. Every topic consists of

one or more levels, separated by a forward slash, the topic level separator.

As shown in fig. 2.2, messages from publishers must include a topic. To receive the published message, clients that want to consume it must subscribe to the same topic channel.

Compared to the traditional exchange of messages through message queue, MQTT topics are lightweight: clients do not need to create the topic before they publish or subscribe to it. This is the broker's job which is responsible for receiving messages on a certain topic channel and transmitting them to the clients that are subscribed to that specific topic. It accepts each valid topic without any sort of initialization. In fact, when a broker receives data on a topic that does not exist, that topic is created and, then, clients may subscribe to the new topic.

Clients can subscribe to one or multiple topics. When subscribing to multiple topics, two wildcard characters can be used:

- #: It matches everything for arbitrary level depth from the current level. It is used as a multi-level wildcard and can only be inserted at the end of the topic.
- +: It matches everything at the current level. It is used as a single-level wildcard.

An example of the use of a multi-level topic can be examined in the appendix A.1.1.

MQTT Broker

One of the main entities of the MQTT protocol is the MQ Broker, which represents the core part of any publish/subscribe model and covers the role of a gateway, a receiver or a server. It allows clients, which are both publisher and subscribers, to communicate with each other by sending messages.

Basically, the MQ broker is responsible for receiving all messages, filtering the messages, determining who is subscribed to each message, and sending the

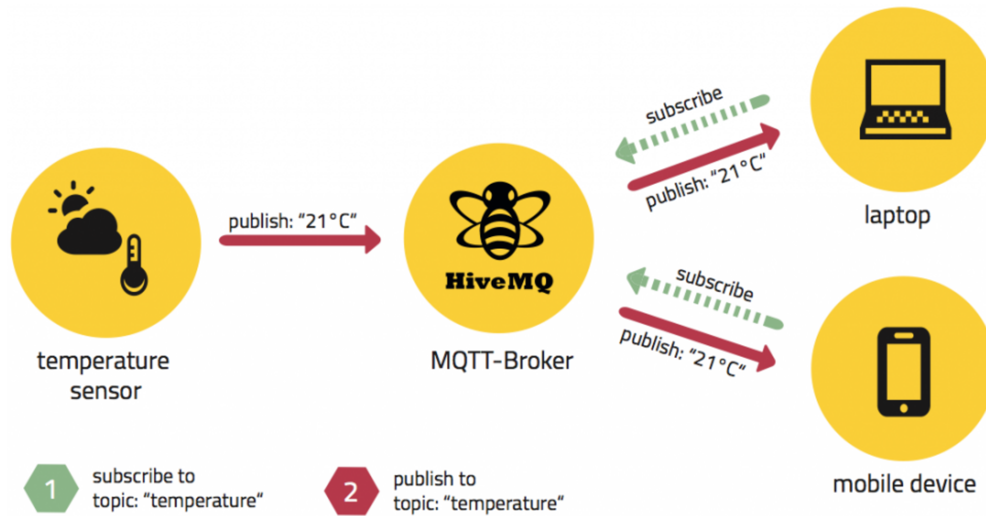

Figure 2.3: MQTT HiveMQ Broker

message to these subscribed clients. Furthermore, the MQ broker is responsible for holding all session datas about all clients that have persistent sessions, including subscriptions and missed messages.

Among the various broker implementations existing on the Net, for the realization of the MQTT protocol, the one we use in this thesis project is HiveMQ. In particular we decided to use **HiveMQ Cloud** [26, 12], a cloud implementation of a broker deployed on Azure. Instead of a classic on-premises broker implementation, which could be a time-consuming and demanding solution, we have chosen a cloud service, where for a minimal cost it is possible to quickly connect and start using the MQTT protocol.

Furthermore, it simplifies the deployment and management of a scalable, reliable and secure MQTT broker cloud service, thanks to an already configured security system on the server. HiveMQ's MQTT Cloud broker is the first cloud-native IoT messaging service which makes use of cloud resources. The use of MQTT reduces network bandwidth required for moving data to and from connected devices.

It is possible to use HiveMQ Cloud to connect up to 100 MQTT client devices at no cost.

It requires no installation and management. In fact, it is possible to create a cluster with just a few clicks and connect the IoT devices. We just need to configure the hostname and port to be able to receive connections from MQTT clients.

It automatically scales up and scales down to meet the demands of your IoT application. In order to ensure scalability and availability, we need to manage the cluster nodes.

It is easy to connect and integrate with no vendor lock-in. HiveMQ Cloud is 100% compliant with the MQTT specification, including QoS 1 and QoS 2 (section 2.2.1), retained messages, shared subscriptions, user properties and negative acknowledgements.

MQTT clients

MQTT clients are any device that runs a MQTT library and connects to a Broker over a network. Both publishers and subscribers are MQTT clients. In fact, when a MQTT client is connected to the broker, it can publish on a specific topic messages and other MQTT clients can subscribe to messages they want to receive.

The implementations of MQTT clients typically require a minimal footprint. So, are usually used for deployment on small devices and are very efficient in their bandwidth requirements.

Among the various MQTT Client libraries, for the realization of the MQTT clients, the one we choose to use within the thesis is HiveMQ [27, 15].

HiveMQ MQTT Client is a library that provides a fast, low-overhead, high throughput and modern MQTT library for Java. In doing so, it builds on modern frameworks like Netty, the one used, within this thesis, for handling connections.

Furthermore, the library provides three distinct flavours of API:

- *Blocking*, the one used in our project for quick start;
- *Asynchronous*;
- *Reactive*.

So, it is possible to choose a programming style which best meets project specifications. We report an example of the creation of a MQTT client in the appendix A.1.3.

Quality Assurance

One important functionality of the MQTT protocol is the Quality of Service (**QoS**). This refers to the degree of accuracy in the delivery of MQTT messages, which occurs in two steps:

1. Firstly, it is sent from a publisher to a broker. In this step, the publisher decides the QoS level associated with that message.
2. Secondly, the broker forwards this message to all clients that have a subscription for that message using the same QoS level.

We can choose between *three* different QoS levels:

- **QoS 0**: This level offers the minimum amount of data transmission, that means zero.
  This level is designed for maximum performance with less effort: in this scenario each message is delivered to a subscriber once with no confirmation. So, there is no guarantee of delivery. In fact, there is no way to know if subscribers received the message. This method is called *fire and forget* or *at most once delivery*.

- **QoS 1**: In this scenario, the broker attempts to deliver the message and, then, waits for a confirmation response from the subscriber. If a confirmation is not received within a specified time frame, the message is sent again.
  Using this QoS, the subscriber may receive the message more than once if the broker does not receive the subscriber's acknowledgment in time. This method is called *at least once delivery*.
- **QoS 2**: This is the most reliable method. The process involves a double bounce between sender and receiver, a four-step handshake, in order to confirm that the sender has actually received the messages and that it is received only once. This method is called *exactly once delivery*.
  In the appendix A.1.1, you can see how we use this QoS level.

2.2.2 Advantages of MQTT and why we choose it

MQTT is an ISO standard protocol and it is used as a data connection IoT protocol. It is based on the publisher-subscriber messaging model and allows a simple data flow between different devices. It has a lot of advantages. Some of them are provided below.

Minimal resources This communication protocol requires minimal resources since it is *lightweight* and *efficient*. In fact, it can support different application scenarios for IoT devices and services. This characteristic increases the amount of data that can be monitored or controlled.

Publish/subscribe protocol MQTT is a *publish/subscribe* protocol. So, it allows IoT devices to publish messages to a Broker. Clients connect to this Broker. Each device can subscribe or register to particular topics. When a client publishes a message on a subscribed topic, the broker forwards the message to any client that has subscribed. Furthermore, this model maximizes the available bandwidth.

Bidirectional exchange This communication protocol supports bidirectional messaging exchange between the connected devices and cloud. Furthermore, it maintains stateful session awareness.

Scalability MQTT protocol supports scalability. In fact, it can scale to millions of connected devices.

Delivery guarantee It guarantees delivery of messages through three different QoS levels.

Last Will and Testament Thanks to that feature, if a device loses connection, all subscribed clients will be notified with the *Last Will and Testament* feature of the MQTT server, so that, any authorized client in the system can publish a new value back to the device, maintaining a bidirectional connectivity.

Motivations

The reasons that led us to choose this protocol are various.The ones we considered the most relevant are listed below.

- It is pretty much the only standard protocol that makes it easy to send commands to the connected devices. It is possible even if the device needs to be controlled remotely.
- Thanks to MQTT, a device connects to the MQTT broker and can subscribe to a topic, and any other clients, which have the right credentials, can connect to the MQTT broker and publish messages to that topic.
- It offers simple methods that adapt well to IoT tasks. For example, subscriptions that recover connections after unexpected client disconnections. Compared to HTTP/HTTPS it is simpler to extract data from the package without any parser.
- It is a light protocol with a fast response time.

- It allows us to build efficient connections between devices whatever the number of the latter is.

2.3 Ktor

Ktor [17] is a Kotlin native web framework that, basically, allows building web applications, HTTP services, mobile and browser applications. Furthermore, it allows the creation of *asynchronous* client and server applications.
That library was developed by JetBrains and it is lightweight and it offers support for coroutines (section 2.5.4), thanks to which it is possible to express complex asynchronous constructs as if they were simple sequential code.
Furthermore, it also offers support for Kotlin Multiplatform and it allows cross-platform capabilities.
In order to ensure the reader a complete understanding of the selected framework, we explain, in the following sections, Ktor key aspects and how it is involved within this thesis.

2.3.1 Key aspects

As shown in fig. 2.4, the main entity in a Ktor project is the Application object. It accepts requests from the **servlet engine**, a class which is used to extend the capabilities of Servers that host applications, and returns responses.
When you build a Ktor web application, first of all you have to set up the engine. In Ktor many engines are supported:

Netty: It is the one that we have chosen as the engine in this thesis. It is an asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers & clients.

Jetty: It provides a web server and servlet container, additionally providing support for HTTP/2, WebSocket. It is easily embedded in devices, tools, frameworks, application servers, and modern cloud services.

Tomcat: It provides a *pure Java* HTTP web server

CIO: Coroutine-based I/O is a fully asynchronous coroutine-based engine that can be used for both JVM and Android platforms. It supports only HTTP/1.x for now.

Furthermore, many applications require common functionality that is out of scope of the application logic. This could be things like serialization and content encoding, compression, headers, cookie support, etc. All of these are provided in Ktor by means of what we call **Plugins** (also known as Features).

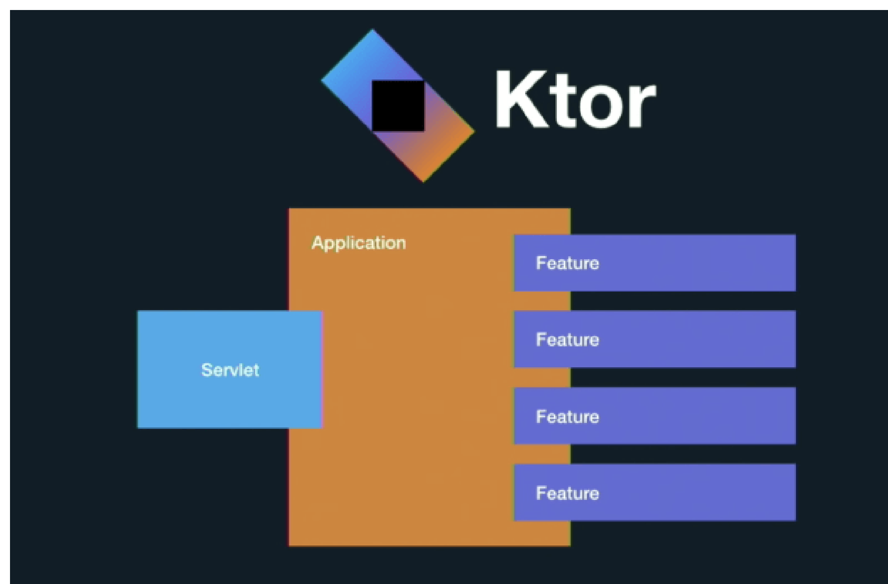

Figure 2.4: Ktor main entities

2.3.2 How Ktor was used?

In this thesis, we use Ktor both as a Server and as a Client.

The former one is meant to work with the application, the Telegram Bot. Thus, it is used to read information coming from various sensors, installed into various cities, and then to communicate that information to the SensorReadingBot application which makes it visible to users.

The latter one, indeed, is meant to work with the various controllers. In doing so, it instantiates the websokets which are used to communicate commands to the various actuators, like light bulb, washing machine etc.. in short, our home automation.

2.4 Main libraries

An important but also hard step is that relating to the selection of libraries that best suited our needs.

Below, we report those which we choose in order to allow us to implement different mechanisms, such as (de)serialization, data persistence etc. In addition, we also report how these mechanisms are used within this thesis.

2.4.1 JetBrains Exposed

In order to acquire data persistence inside the application, we decided to use, as a library for back-end application development, the *Exposed Library* [16].

It is a lightweight SQL library, provided by JetBrains, for the Kotlin language, which relies on database access.

Every database access using Exposed is started by obtaining a connection and creating a transaction.

We reported a very simple example in the appendix B.1. The example explains how to use that library in a real context.

2.4.2 Gson

Gson [14] is a Java library which enables the conversion of Java Objects to an equivalent JSON representation and, conversely, enables the conversion of a JSON string to an equivalent Java object. Thus, it is used to implement serialization and deserialization mechanism.
In this thesis, we use this library both on the Server-side and Client-side.

In the former, we use it to answer requests coming from the Application Bot, thus we use it to serialize messages. That response is a set of:

- Message to display to user;
- Various commands which are serialized as a JSON object and sent to the Application Bot.

In the latter, we use Gson to deserialize messages received from the actuator controller, which manages all user home automation. These messages arrive as JSON strings, so they need to be converted into String format. We report an example of serialization and deserialization using Gson in the appendix B.2.

2.4.3 Node.js Telegram Bot API

In order to ensure the communication between the Telegram Bot application and its respective Server side it is, first of all, necessary to develop the application component of the Bot. This is done thanks to BotFather, a bot provided by Telegram to create and manage personal bots. At the time of the creation of the Bot, a Token is provided. This will allow us identify it and to communicate with it from the Server side.

There are different ways and languages to develop the service but we choose NodeJS because it is better suited to develop the connection between our Bot and server in a short time. Furthermore, to ensure communication we adopted a NodeJS module, [25], which allows interaction using the Telegram Bot API.

An example of use of that module is reported in an extract of the realization of the bot backend in the section 5.1.

2.4.4 jSerialComm library

For simplicity of testing, not having available all the utilities necessary for the real configuration of the network, the communication between Arduino (section 2.5.3) and Raspberry PI (section 2.5.2) passes through the serial port.

The library that we decided to adopt is the *jSerialComm* [10] which provides a way to access standard serial ports without requiring external libraries, native code, or any other tools.

2.5 Utilities

Below are all the utilities that became necessary during the construction of the whole system.

2.5.1 Security mechanisms

A relevant aspect of this thesis is that of security. In fact, since we are dealing with confidential information it is necessary to adopt an encryption algorithm.

In particular, we adopt the ***AES*** symmetric block cipher algorithm [29, 30, 31] which we decide to use to encrypt a plain text and decrypt a cipher text. In particular, this algorithm is used to protect the credentials which we use to connect to the HiveMQ Broker.

Another security mechanism is the one that we realize through the token check system.

This check is realized to prevent a malicious person from overlapping bots in the conversation with the Core, so it is realized to make sure that Bot is really Bot.

Assumptions At this moment, we assumed that the communication channel between the Bot and the Core is secure as we use the HTTPS protocol. So, any message sent cannot be read by an attacker.

2.5.2 Raspberry PI

Another component of the system is the Raspberry PI which hosts two projects: publisherClient and HomeAutomationController.

publisherClient

The former project simulates an MQTT Client that connects to the MQTT Broker and publishes, on the topic of interest, the results relating to the reading of the sensors' information, which are sent to it by Arduino. It is divided into three main entities: Detector, Input Analyzer and Publisher Client.

- **Detector class**: When invoked, it opens the serial port */dev/ttyACM0* and, in loop, it reads all the messages, byte per byte, which are written on the serial line by the Arduino microcontroller. The reading ends when it encounters the $ terminator character of the message. Now, it sends the message over the communication channel, which is shared with the Input Analyzer.
- **Input Analyzer**: In a loop, it pulls out the message from the input channel and, after verifying that this message respects a certain pattern (given by

the specified regex), it sends IT to the MQTT Publisher Client.

- **Publisher Client**: It takes care of establishing the connection with the MQ Broker and, then, it publishes the messages to the MQ Broker under a certain topic.

HomeAutomationController

The latter project, instead, deals with managing the connection and, then, the communication with the home automation components.
It is divided into three main entities, which are reported below.

HomeComponent It is the effective actuator on which the requested command by the user is executed.
For simplicity, in this project, we implement only a SmartLight, a light bulb that has other functionalities besides ON and OFF.Once the command is executed, the actuator communicates which are the next possible commands and its status: On or Off.

HomeComponentHandler It is the entity responsible for the management of all the HomeComponent existing in the system. When invoked for the first time, it creates a new component and assigns to it a set of:

- Initial commands,
- Name,
- Initial status.

HandleSerialMsg It is the entity responsible for the management of communication with home automation. In fact, it takes care of sending on the serial port the command executed by the user on a particular actuator. This command is received from the actuator that, once executed, it sends back a list of new possible

commands on the serial port as response. The HandleSerialMsg entity is also responsible for reading that response and communicating it to the Component entity and, then, to the user via the Application Bot.

2.5.3 Arduino

The part relating to the management of sensors and actuators was carried out using the open-source electronics platform Arduino [3] on which we put some sensors. In order to separate concepts well, two Arduinos were used.

On the former, the program relating to the reading of the sensors runs.
In that program, every sensor, at each delay (`500`), reads the information and sends it to CoreSystem, which represents our Server and maintains the just received information into a data structure.

On the latter, the program relating to the management of home automation of an apartment runs.
In this case, the program, in a loop, checks if new messages arrived on the serial port and, based on the type of message,

- It executes the received command and
- Updates the Server with the possible other actions which can be performed on a particular home automation actuator.

If the message is not recognized, it is simply ignored.

2.5.4 Coroutines and concurrency in Kotlin

To explain how we exploit concurrency in this thesis, firstly we expose how Kotlin can support it. Following, we introduce the main constructs available, how they work and how they interact.

Coroutines

Coroutines are computer program components which generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed. Kotlin, as a language, provides only minimal low-level APIs in its standard library to enable various other libraries to utilize coroutines. Kotlin's concept of suspending function provides a safer and less error-prone abstraction for asynchronous operations than futures and promises.

A coroutine is an instance of suspendable computation. It is conceptually similar to a `thread`, in the sense that it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one. These can be thought of as light-weight threads with a number of important differences that make their real-life usage very different from threads.

These follow a principle of structured concurrency which means that new coroutines can be only launched in a specific `CoroutineScope` which delimits the lifetime of the coroutine. Structured concurrency ensures that coroutines are not lost and do not leak. An outer scope cannot complete until all its children's coroutines are complete. Structured concurrency also ensures that any errors in the code are properly reported and are never lost.

Coroutines always execute in some context represented by a value of the `CoroutineContext` type, defined in the Kotlin standard library. The coroutine context is a set of various elements. The main elements are the Job of the coroutine and its dispatcher. The coroutine dispatcher determines what thread or threads the corresponding coroutine uses for its execution. The coroutine dispatcher can confine coroutine execution to a specific thread, dispatch it to a thread pool, or let it run unconfined.

When a coroutine is launched in the `CoroutineScope` of another coroutine,

it inherits its context and the `Job` of the new coroutine becomes a child of the parent coroutine's job. When the parent coroutine is cancelled, all its children are recursively cancelled, too.

Channels

Coroutines are the components that allow execution to be suspended and resumed and, thus, let the programmer exploit concurrency. Often these components need to communicate thus, a stream, `Channels` come into play.

A *Channel* is a non-blocking primitive for communication between a sender and a receiver. It is conceptually very similar to a `BlockingQueue`. One key difference is that instead of a blocking put operation it has a suspending `send`, and instead of a blocking take operation it has a suspending `receive`. Unlike a queue, a channel can be closed to indicate that no more elements are coming. Conceptually, a close is like sending a special close token to the channel. The iteration stops as soon as this close token is received, so there is a guarantee that all previously sent elements before the close are received

Chapter 3

Requirements

In this chapter, we report the steps we pursue to tackle the domain analysis of this thesis.

3.1 Ubiquitous Language

The fundamental step to be carried out during the problem analysis is to define the *Ubiquitous Language* [33]. We model it within a Limited context, where the terms and concepts of the business domain are identified and they are expressed without ambiguity in order to eliminate inaccuracies and contradictions.

We then used the defined Ubiquitous Language, which we report in fig. 3.1, to clearly express all system requirements. In the Ubiquitous Language terms are associated with their own definition of the domain of interest. Looking at the figure, we define:

- **Sensors**: They are the devices meant to detect or measure a physical property and, then, record or indicate it.
- **Core System**: It is the system component which takes care of connecting the Bot application and the Raspberry component and, thus, allows estab-

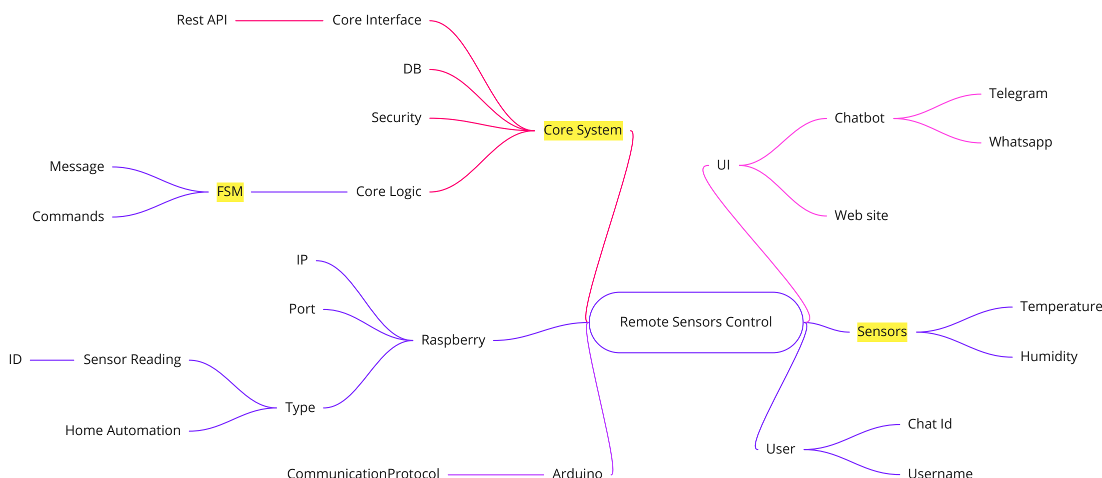

Figure 3.1: UL of the system

lishing communication between them.

- **FSM**: It is a behaviour model which consists of a finite number of states. Based on the current state and a given input, the machine performs state transitions and produces outputs.

3.2 Non-functional requirements

Non-functional requirements capture operational characteristics, architecture, technical specifications and design. We distinguish two non-functional requirements: product and external one.

Product requirements

- **Usability**: The system has to adopt a simple language in order to allow everyone to understand it. Furthermore, the user interface has to be easy to use and easy to understand too.
- **Scalability**: The system has to support a growing number of members and

has to manage their simultaneous access.

- **Reliability**: The system has to be able to react to every type of error and, if unable to handle it, it has to restart. In addition, it has to try to isolate, as much as possible, the error into the specific sub-portion of the system in which it occurred, trying to leave the other parts of the system operative.
- **Portability**: The system has to work regardless of the device members decide to use.

External requirements

- **Interoperability**: The system consists of several subsystems, which are defined during the design phase. These have to interact and inter-operate with each other as if they were a single system.
- **Security**: The system provides access using credentials. Sensitive data need to be protected, in fact these are encrypted, using symmetric encryption.

3.3 Functional requirements

Functional requirements are used to express the service that the software must offer. some of them are expressed below:

The Bot application should be always available and usable from any device. No need to install any application to use the Bot. The first implementation will provide an executable application that will be runnable via the Telegram application. In the future, it will be possible to use the system via a web interface or other applications.

Users should be able to interact with sensors. In particular,

1. The Bot application should invoke the Server.
2. Server connects to the Broker.
3. Broker sends requests to Arduino that reads information.
4. User should be able to receive the information in real-time from sensors.

Users should be able to access his home automation. In particular,

1. The Bot application should invoke the Server.
2. Server connects to Websocket.
3. Raspberry receives messages on the channel and sends them to Arduino that executes those messages as commands.
4. User should be able to receive the information in real-time from actuators.

Chapter 4

Design

4.1 Architecture

In order to apply the previously identified requirements of the system, in particular the functional ones, we decide to develop a distributed system. This expects both data and transaction processing to be split between one or more computers connected by a network, each of which plays a specific role in the system.

The motivations that bring us to the choice of developing a distributed system are that this system offers: availability, durability, efficiency, scalability and redundancy.

- **Availability**: This requirement determines how long your IT System can be unavailable without impacting operations..
- **Durability**: The expected life of a system.
- **Efficiency**: The extent to which the software system handles capacity, throughput, and response time.
- **Scalability**: The system can easily be expanded by adding more machines.
- **Redundancy**: Various machines can provide the same services, so if one is unavailable, work does not stop.

During the design phase, we consider various architectural choices but the decided one, at the end, is the *Client-Server* architecture as it is the basis for distributed systems.

The Client-Server architecture allows data and service integration and, furthermore, allows Clients to be separated from system complexity, for example the establishment of a communication protocol.

The simplicity of that architecture allows Clients to make requests. These, made in the form of *transactions*, are routed to the appropriate Server. Customer transactions are often SQL procedures and functions that access databases and services.

4.1.1 Description of the architecture inside the project

As represented in fig. 4.1, the user of the Bot Application, the first accessible component of the system, interacts with a User Interface, in this case Telegram Bot, that only forwards commands to the Core, the main component of the system.

In fact, the entire logic of the system is managed by the Core System. It takes care of opening the connections to the Raspberry pi, keeping the connection active, managing the communication protocol, etc.

As far as communication between these two elements is concerned, this can take place in two ways:

1. User wants to know about sensors' information, which came from the various Arduinos. That information is only accessible in a *reading* way, so the user can not use it as commands. We achieve this type of communication through the publish/subscribe MQTT protocol (section 2.2). We choose it for its lightness and because we assume that much information passes through the communication channel. In fact, we assume that, in a real use

of the application, we do not manage just two sensors but many more. So, this protocol seems to us the most suitable solution.

2. User wants to interact with the sensors in a *writing* way. In fact, the application allows the user to manage his home automation and, then, we want to give the user the ability to interact with a particular actuator by activating and deactivating it. For this type of communication we use, as a means of communication, Websocket. We choose it in order to guarantee security and to allow a steady bidirectional flow of data.

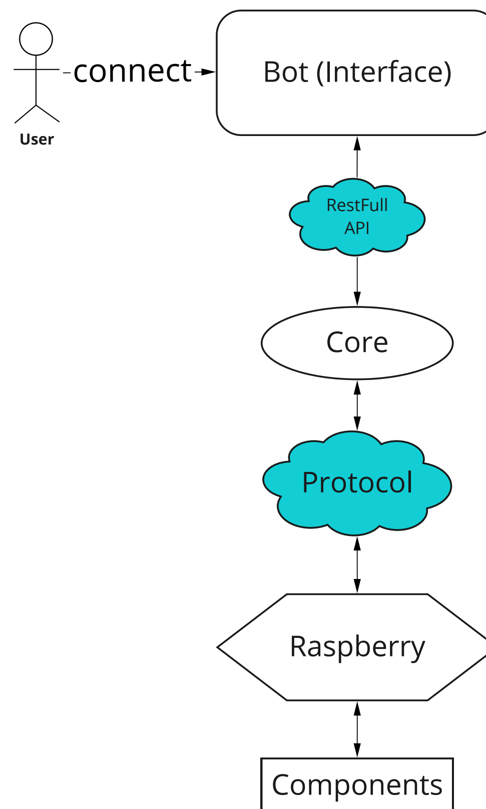

Figure 4.1: General schema of the chosen architecture

Looking at the fig. 4.1, we can notice another component, the Raspberry pi, which has two responsibilities. It takes care of:

- Managing the results of the sensors that reside in its geographical area. This component maintains the acquired datas in an appropriate data structure and then, when the number of data is adequate, it makes an average of the corresponding values and sends it to the Core System.
- Allowing the user to interact with his home automation. To do so, that component has to address user commands to the following component, Arduino.

Finally, the last component in our system is the Arduino. It deals with, from one side, various sensors that continuously read information and send these data to Raspberry, from the other side, it deals with actuators that execute the received message from the Raspberry pi component and send it back the corresponding response.

4.1.2 Application of the architecture in the project

In this project, the chosen architecture has a different meaning according to the observer's point of view and changes according to the role that the system plays in a specific moment.

Regarding the sensors' reading, as we can see in fig. 4.2, the user considers the Bot application as if it were the Server to which it sends commands. Instead, the Bot application, from its point of view, is the Client of the Core to which it forwards requests.

This latter, in the same way, becomes Client of the Broker as well as the Raspberry pi component. Thus, in this portion of the system, the Hive MQ Broker is considered a Server.

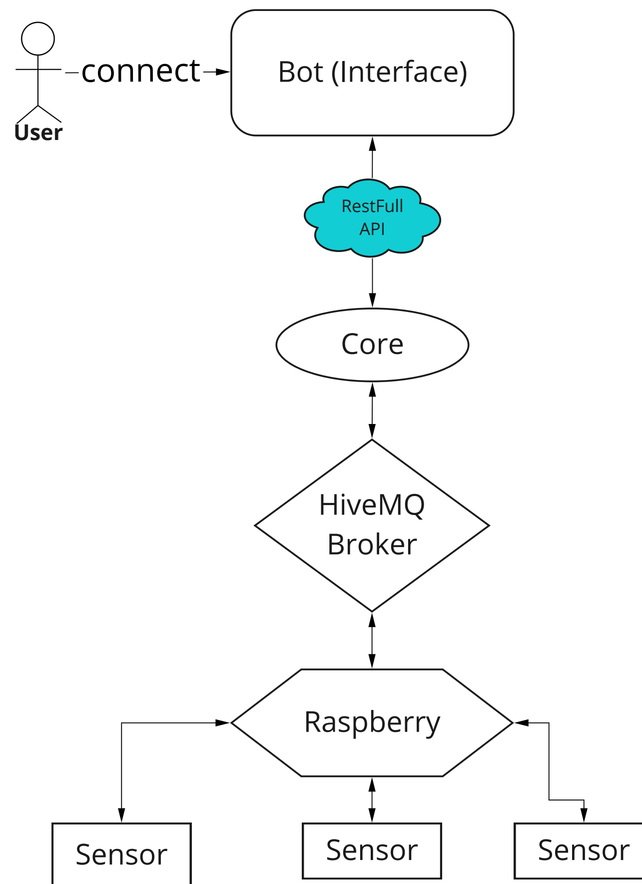

Figure 4.2: System architecture for sensors' reading.

A change occurs when we want to manage our home automation. This is clearly illustrated in fig. 4.3. First portion of the system is identical to the previous one, what changes is the communication between Core and Raspberry pi which directly communicate. In that case, Core is the Client that makes requests through Websocket, while Raspberry pi is our Server that receives requests and sends them to the actuators.

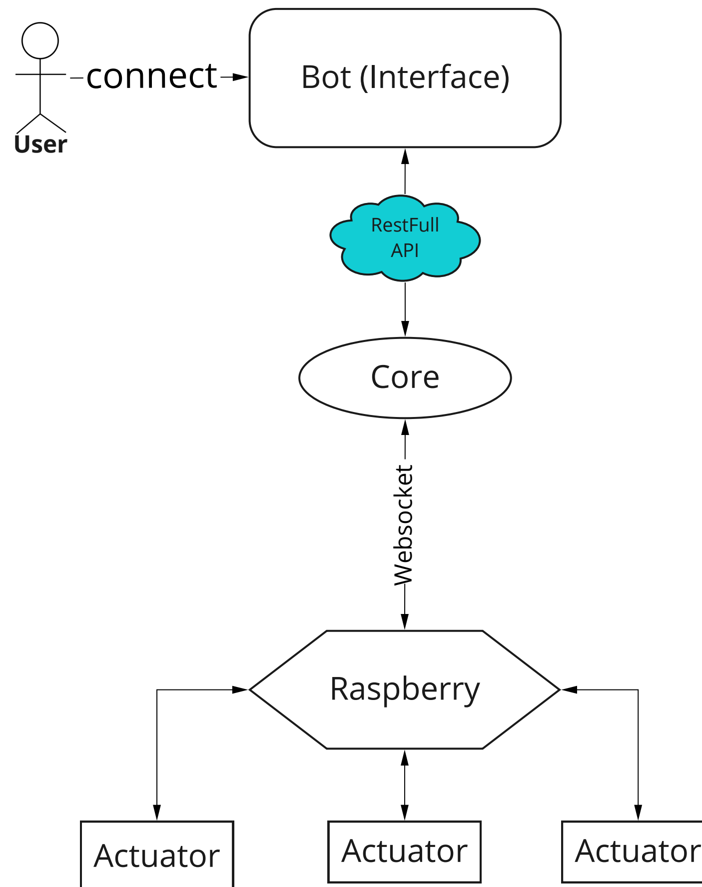

Figure 4.3: System architecture for home automation management.

4.1.3 Clean architecture

In order to achieve clarity into the structure of the code and to improve separation of duty of each part of this system, we adopt, as an architectural style, the *Clean Architecture* [22, 23].

It expects that any layer can only reference a layer below it and know nothing about what is going on above. This architectural style divides the system into four layers:

Entities - These are the business objects and should not be affected by any change external to them.

Use Cases - These implement and encapsulate all of the business rules.

Interface Adapters - These convert and present data to the use case and entity layers.

Frameworks and Drivers - These contain any frameworks or tools needed in the application.

4.2 Sub-domains of the system

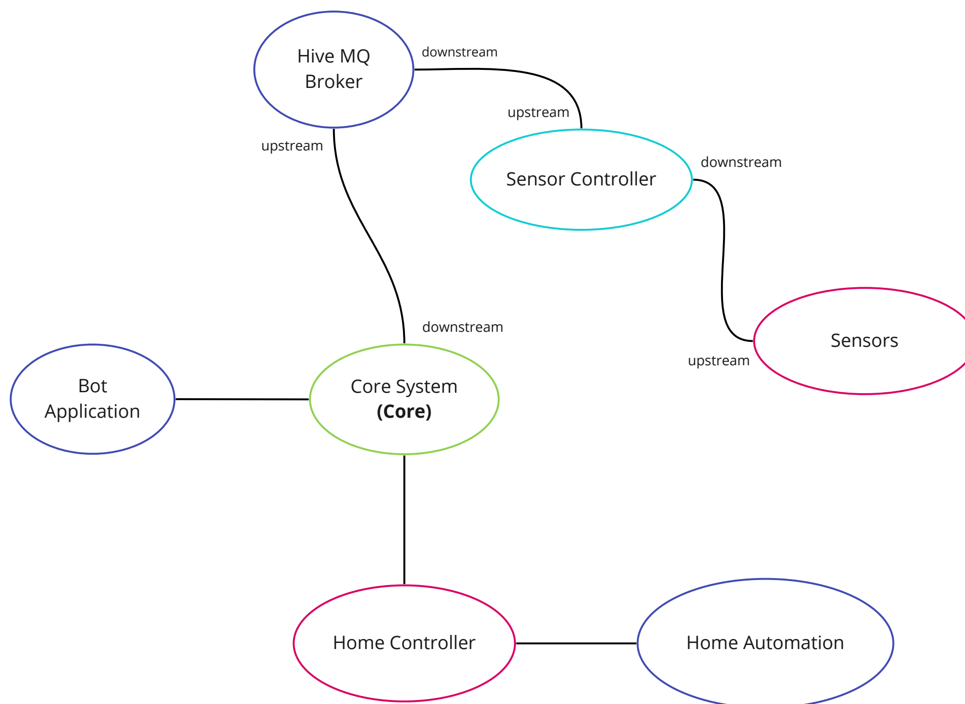

Figure 4.4: Sub-domain identification

The division of the domain into the various sub-domains is guided by the idea of separating the behavior and functionalities that emerged during the analysis phase as clearly as possible. The identified sub-domains and the relationships with each other are clearly reported in fig. 4.4.

Bot Application: This context represents the application part that links the user and the application itself. It provides a simple User Interface. It has a relationship of both upstream and downstream towards the Core System context.

Hive MQ Broker: This is a third party context that is responsible for maintaining a communication between publisher and subscriber clients. It has a relationship of downstream towards both the Core System context and the Sensor Controller one.

Core System: This is the core part of our system. It has a relationship of downstream towards Hive MQ Broker and both upstream and downstream towards Home Controller context.

Sensor Controller: It is the context responsible for pushing information on the topic channel managed by the Broker. So, it has a relationship of downstream towards the Broker context.
Furthermore, it has the responsibility of maintaining all information coming from Sensor, so it has a relationship of downstream towards Sensors context.

Sensors: It is the part that takes care of sending information on the serial port. It has a relationship of upstream with Sensor Controller context.

Home Controller: It is the context responsible for requesting information about home automation status. So, it has a relationship of both upstream and downstream towards the Home Automation context.

Home Automation: It is the specific smart house component. It has a relationship of both upstream and downstream towards the Home Controller context.

4.3 Core logic

Given the analysis of the problem of the domain, we proceed to develop the solution following the philosophy of *Domain Driven Design* [8], especially in the development of the main entity.
We deepen the Core System design due to its higher degree of complexity.

4.3.1 Bounded Context

First of all, we identify the Bounded Context [9] related to the Core sub-domain. As we show in fig. 4.5, we identified three Bounded Context:

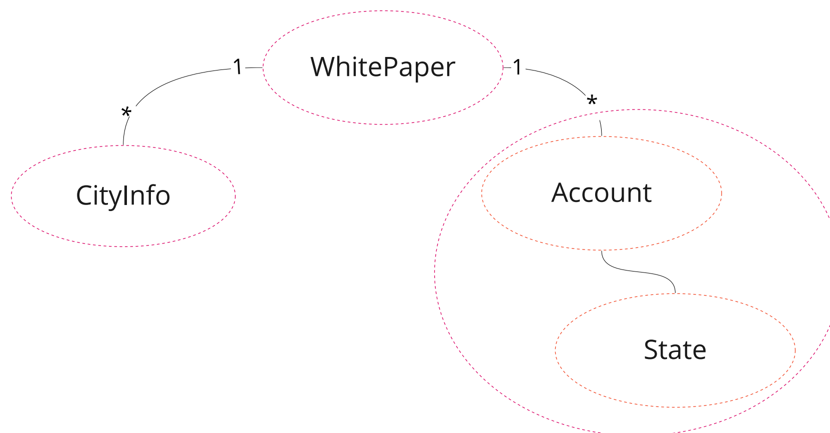

Figure 4.5: Figure representing all identified bounded context inside Core

WhitePaper: It is the Bounded Context meant to manage all information, inside the Core Context, about users and cities.

CityInfo: It is the Bounded Context which takes care of maintaining in memory and then storing in a database all information related to a particular city. In the design phase, we choose H2 as our DB.

Account + State: It is the Bounded Context which takes care of managing the states in which an application can be based on the user requests. It simulates a Finite State Machine in which a new request from the user determines a change of state.

4.3.2 Core Finite State Machine

In this section we explain how we use the FSM inside the Core logic, giving the reader an overview of the identified states.

As shown in fig. 4.6, when the user starts the Bot application, the initial state is `StartState`. From this one, there are two possibilities:

1. User can press "Sensors" changing the position of the FSM to `SensorState`. There, the user can only request information about sensors. Even in this state, the user has two choices:
   - He can ask for temperature information changing the position of the FSM to `TemperatureState`;
   - Or, he can ask for humidity information changing the position of the FSM to `HumidityState`;

   In both states, the only accepted requests can be: "city", the specific manually inserted city of which the user wants to know about, and "cityList, by pressing a specific button which returns, as response, the list of all cities.
2. Otherwise, the user can press *Home Automation* moving the position of the FSM to *HomeState*. If the user has not yet registered the IP address of his home automation, FSM moves its state to a temporary one, the *RegisterState*, in which the insertion of the address is requested. Once the user ends the registration step, FSM moves its state to *HomeState*.

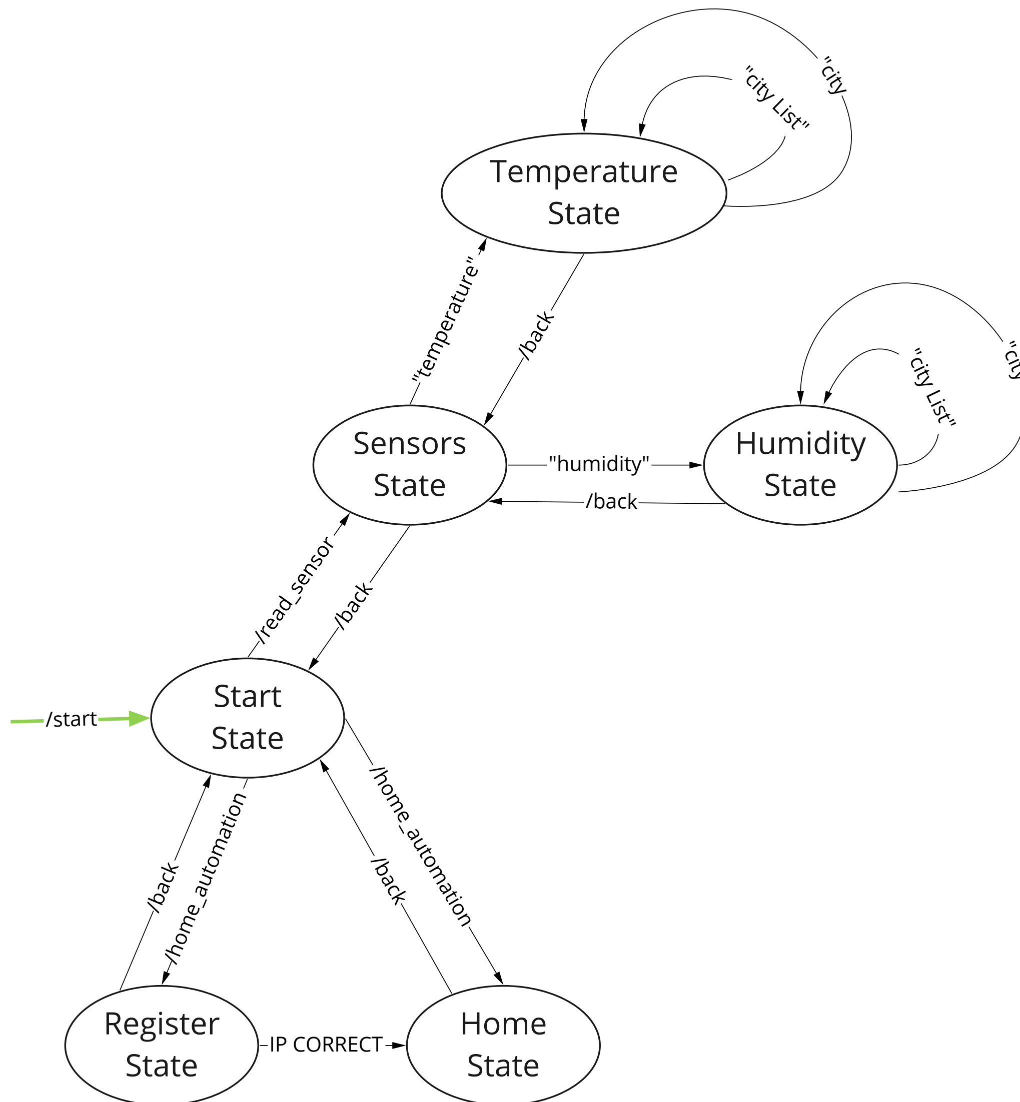

Figure 4.6: Core Finite State Machine

Chapter 5

Implementation

At the implementation level, our thesis is divided into six sub-projects: a JavaScript sub-project, two Arduino sub-projects and three Kotlin sub-projects.

JavaScript sub-project - It represents our application Bot backend. This sub-project exploits a NodeJs module, the *node-telegram-bot-api*, to enable communication between Telegram Bot and our backend.

Arduino sub-projects

- The first sub-project is a very simple one and takes care of sensors' reading. This sub-project realizes a starting setup in which it sets the data rate in bits per second (baud) for serial data transmission. Afterwards, in a loop it reads the value from the specified analog pin and then it sends this data on the Serial port.
- The other sub-project, instead, is specific for the component we want to query. For example, if our home automation includes components with different functions, a washing machine, light bulbs, air conditioner and other components we should have many other Arduino programs: one for each component.

  In our case, for simplicity of development, we provide a single light bulb but

with different functions.

What this second program does is, in a loop, to check for new messages on the serial port. These messages have to respect a certain pattern, otherwise they are ignored. When a message respects that pattern, the relative command is executed and the status of the respective component is updated. Finally, the modified information is sent on the Serial line.

Kotlin sub-projects

- *CoreSystem* is the most complex sub-project. It has two main tasks: connect Server to the Bot Application backend and, also, connect itself to the Hive MQ Broker.
- *PublisherClient* is the sub-project meant to simulate a client's behaviour that, after connecting to the Broker, he publishes on the latter's topic channel.
- *HomeAutomationController* is the sub-project which interfaces with the Arduino sub-project related to the interaction with the actuators.

We declare all linked snapshots in the Appendix sections.

5.1 JavaScript sub-project: Telegram Bot

As explained before, one of the sub-project is the JavaScript one. It uses the `node-telegram-bot-api` module, which is the NodeJs module that we use to connect Telegram Bot with our Bot backend.

The creation of Telegram Bot UI is simplified by BotFather, a Telegram Bot used to create other bot accounts and manage them. So, we:

1. Request to BotFather to initiate a new Bot providing the name to be assigned, SensorReadingBot, and an identifier.

2. Save the token provided by BotFather. This token is the access key to the Telegram API. In our case, that token is saved into the variable *token*, as shown in listing C.1 at line 4.
3. Use the `node-telegram-bot-api` module.

The most interesting aspect is that the Bot will be identifiable by Telegram anywhere as long as it initiates the dialogue via node-telegram-bot-api using the token.

As can be seen in listing C.1, TelegramBot is initialized by passing the token generated by the BotFather as the constructor argument.

This instance is an `EventEmitter` object which emits the *message* event when a new incoming message of any kind is received from the bot. The event is caught thanks to the `on` method. First of all, the message text is extracted via `msg.text` and, depending on the text, it executes a specific action. In this case it sends a request to the Server which processes it and replies back. Once the response is received, line 15, it is sent to the TelegramBot via `sendMessage` Telegram API.

For sending requests to the Server we use the Axios client library [5], a client HTTP JavaScript library used to make HTTP requests from NodeJs from the browser. Furthermore, that library supports the Promise API that is native to JS.

We choose it because it makes it easy to send asynchronous HTTP requests to REST endpoints and perform CRUD operations.

5.2 Kotlin sub-project: CoreSystem

As introduced in this chapter, the CoreSystem sub-project is the most complex among other sub-projects. In order to explain its functioning, we offer an explanation of the most relevant entities.

5.2.1 Creation of the Server

One of the main tasks of CoreSystem is to open a connection with Bot. To do so, we use the technology declared in section 2.3, Ktor. Thanks to it, it is possible to instantiate a Server in just a few lines.

As we show in listing C.2, first of all we configure the server parameters in the code via the `embeddedServer()` function. The `Netty` Engine and the port `8000` where it expects to receive requests, are the Server configuration parameters we choose.

The embeddedServer method allows us to quickly run our application, invoking the `start()` method on the just created Server. The boolean value inside the start method means that this function exits if the application engine stops and exits.
The execution of the Server happens on a separate thread with respect to the main one in order to avoid blocking the main thread.

`Routing` is the core Ktor plugin for handling incoming requests in a Server application. When a client makes a request to a specific URL, for example, `/token`, `/` or `/freeText`, the routing plugin defines how the request has to be served.

/token route This route is used to verify the "secret" shared between Bot and Server: in this case a password. We show the process of authentication in fig. 5.1. If this secret is the same on both sides, then a token, generated via the JWT library [4], is sent back to the Bot. From this moment, all requests coming from the Bot and directed to the Server must have the generated token among the other parameters. If this is not present, the request is not valid; otherwise the Bot is considered reliable and communication can take place.

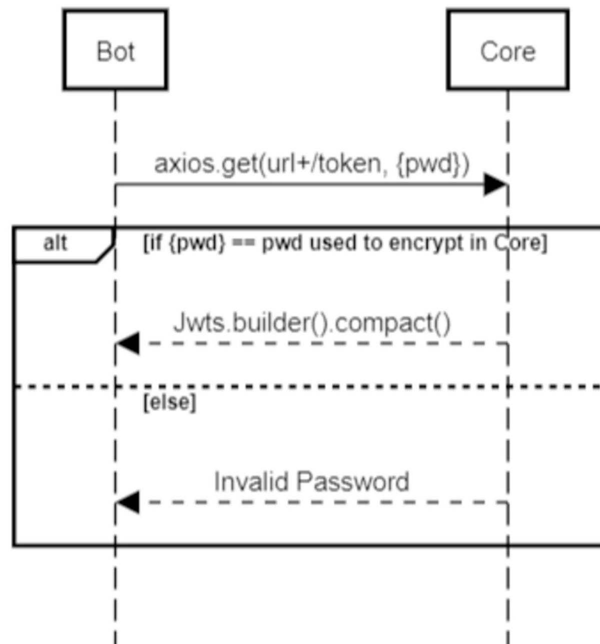

Figure 5.1: Authentication phase

Therefore:

- Bot sends a request on `/token` route by adding, as a parameter, the password that was chosen to configure the bot.
- CoreSystem receives the request and extracts the parameter. If the received message is the same as the password the core uses to encrypt, then Bot is really Bot. So it generates a token and sends it back to the Bot as a response.
- At this point, the token is inserted into the Axios header and all future requests will have it as the header.

/ route This route is used when a user starts the Bot application. When requested, this route firstly checks query parameters and then, if parameters are correct, creates a user account, if not already exists. At this point, it can send to the user the welcome message and a possible list of commands that he can invoke.

/freeText route This route is used when the bot sends a request to the Server with any message other than the initial one, which is the default message "/start". In this case, the received text must be parsed. This check is done via `handleText(chatId, text)` method and it needs to avoid SQL injection. If the text is valid, then the request is processed and the response is sent to the Bot.

5.2.2 WhitePaper

It is the entity which manages all data about users and sensors inside CoreSystem. This entity contains all the logic of loading information from the database, when starting the application, and the logic of handling users' accounts. In particular, it exposes the methods:

- `addNewAccount(chatId: String, username: String)`: The first thing that we do, as soon as the application is started, is to check the existence of the user within the system. If he does not exist then, we add him to the database and we set his initial state.
- `updateUserAccount(chatId: String, update: (Account) -->Account): Account`: Given a specific `chatId`, this function return a new account with updated information about that user.
- `handle(chatId: String, text: String) : Pair<String, List<String>>`: This function, given a `chatId` and a `text`, returns the new message and the list of commands.

5.2.3 State

Our FSM features six different states, only one of each is initial. We have no finite states as this machine takes care of keeping track of the possible actions that users can do. These states expect some parameters, which are:

- `message`: What is shown to the user. It is provided by the state itself.
- `account`: Object used to keep track of user's account information.
- `back` method: To go back to the previous state. It returns a new account where the status is updated. This latter does not change according to the previously performed action but with respect to the FSM.
- `next` method: To go to the next state in the FSM. Given a *command*, it returns a new account with an updated state, which is the next state in the FSM.
- `getCommandList` method: It returns a list of commands to show to users. These are provided by the state itself except in the case of StateHomeAutomation in which this list is provided by the Raspberry pi component that manages the home automation.

5.2.4 UtilityHomeConnection

This is the entity that takes care of managing the connection between CoreSystem and Raspberry pi. It is launched by the `StateRegisterInfoHomeAutomation` when, for the first time, the user registers the IP address of his home automation.

When an instance of UtilityHomeConnection is created, as we show in listing C.3, it opens the websocket connection specifying the IP address, the port and the routing path which is used to forward requests. The opening process is made in a separate thread from the main one because this operation is blocking.

Once the connection is established, the first operation executed is the *handshake*. In this phase, CoreSystem sends to Raspberry the shared *secret* to assure Raspberry about his identity. If the handshake succeeds, a message and a list of commands is given as a response.

After this phase, UtilityHomeConnection manages the websocket. As shown in listing C.4, it uses two different threads:

- One is used to catch all incoming messages that arrive in the `ReceiveChannel` from the user. Then, they are sent to Raspberry pi.
- The other one is used to catch messages coming from Raspberry pi and to send them to Bot.

5.3 PublisherClient sub-project

This sub-project is intended to simulate MQTT clients which publish data on the topic channel of the MQ Broker.

Each PublisherClient has a unique identifier and uses credentials to connect to the Broker. In this project we only have one publisher, the *temperatureClient*, which pushes on topic channel data of a specific city.

5.3.1 InputAnalyzer

Data shared with CoreSystem comes from another important entity inside this sub-project: *InputAnalyzer*. This component is started by the main class, as shown in listing C.5, and its task is to analyze all incoming messages. So, it takes messages received on channel and verifies that they respect a certain pattern, `str:str:str$`. If the message is valid, it drops the last character and splits that message into three pieces, two of which are used by InputAnalyzer:

- The central one that specify the type of message and
- The last one that contains the value that the sensor reads.

Based on the type, the value is stored in a specific data structure. When it reaches 10 elements, InputAnalyzer calculates the value to share and then sends a specific message to PublisherClient containing the data that needs to be pushed on the specific topic channel.

5.3.2 Detector

The channel from which InputAnalyzer reads data is shared with another entity inside that project, which is *Detector*. This takes care of opening the serial port through which communication occurs with the Arduinos that read sensors.

The message arrives in byte format, thus the Detector has to convert it in a format that is comprehensible by InputAnalyzer.

Both InputAnalyzer and Detector are started by the main class and, due to their blocking behaviour, they are launched with their own coroutine.

5.4 HomeAutomationController

This sub-project is intended to simulate our home automation controller. In order to manage the house, HomeAutomationController communicates with CoreSystem using Websocket. Incoming accepted messages can be: a password or a string in a specific form.

- `pwd`: This means that, for the first time, CoreSystem connects to HomeAutomationController. Thus, the mechanism of the handshake takes place. In this case HomeAutomationController sends to CoreSystem, through the websocket, a message containing:
  - A message to show to the user.
  - The names of all components.
  - The status of the components.
- A string in the form `name:command$`: This means the user requests the execution of a particular command on a home automation component. In this case Arduino sends a reply message to HomeAutomationController containing

- The new list of commands to be executed on that component,
- The updated status of these components.

5.4.1 Store

It is the object that contains a map where the key is the name of a component, while the value is the component itself. The latter is the entity that simulates home automation on which one command at a time can be executed. The command to execute is requested by the user. When HomeComponent receives the command to execute, it writes it to the serial port and sends it to the specific home automation component.

5.4.2 HomeComponentHandler

The entity which takes care of opening the serial port with Arduino is *HomeComponentHandler*. When invoked for the first time, it sends to Arduino an "info" message on the serial port. The response from Arduino, which contains the list of possible commands that a user can invoke on a home automation component, arrives in byte format. Thus, it needs to be converted into a Triple. This latter, is, then, split in three pieces. This allows HomeComponentHandler to create a HomeComponent.

Chapter 6

Conclusions and Future works

This thesis focuses on providing a complete distributed system over the network. At first, the focus is on the design of the system in all its components.
The work on this thesis is marked by two important phases: firstly the research and then the development.

The former involves the research and analysis phase in order to know the application domain and which may be the most suitable technologies to create our system.

The latter, instead, refers to the effective development of the system, during which, given the analysis of the problem, we proceed to develop the solution. So, to apply the decided architecture, Client-server, we realize all individual components of the system in disjoint sub-projects.

Several improvements and interesting topics in different areas are still available. One component available for improvements is SensorReadingBot. For example, it is possible to redefine the model by accepting other formats such as audio streams, images, etc.

Another improvement of the bot could be the combination of AI and Natural Language Processing to increase its capacity of dialogue and interaction. In fact,

at the moment, our bot is rule-based, so it accepts only a limited number of commands which are predefined during the implementation phase. As mentioned in chapter 1, a Bot could be AI-based and, thus, it is possible to train our bot through a machine learning process to autonomously recognize user requests.

A further area of improvement could be the inclusion of a push notification system. Since our Bot is also used for home automation management, an important feature would be to notify the user about an anomalous detection or a malfunction of a component. For example, the user wants to switch on the light bulb but it has burned out. The Bot should notify the user about the problem of the light bulb. In this way the user can have a direct relationship with the Bot and, therefore, experience a more natural interaction with its home as if it were a real human.

Bibliography

[1] Mukesh Singhal Ajay D. Kshemkalyani. Distributed computing: Principles, algorithms, and systems. https://eclass.uoa.gr/modules/document/file.php/D245/2015/DistrComp.pdf.

[2] Apple. Virtual assistants: What is apple siri. https://yakbots.com/virtual-assistants-what-is-apple-siri/#more-1442, 2010.

[3] Arduino. Arduino - introduction. https://www.arduino.cc/en/guide/introduction.

[4] auth0.com. Json web token introduction. https://jwt.io/introduction.

[5] Advanced encryption standard. https://www.tutorialspoint.com/cryptography/advanced_encryption_standard.html.

[6] Definition of chatbot - it glossary. https://www.gartner.com/en/information-technology/glossary/chatbot.

[7] Oxford Dictionary. Chatbot—meaning & definition for uk english—lexico.com. https://www.lexico.com/definition/chatbot?locale=en/, 2021.

[8] Eric Evans. Domain driven design. https://martinfowler.com/bliki/DomainDrivenDesign.html.

[9] Eric Evans. Domain driven design- bounded context. https://martinfowler.com/bliki/BoundedContext.html.

[10] Fazecast/jSerialComm. jserialcomm: Platform-independent serial port access for java. https://github.com/Fazecast/jSerialComm.

[11] Google. Virtual assistants: What is google now. https://www.androidcentral.com/google-now, 2012.

[12] Hivemq cloud :: Hivemq documentation. https://www.hivemq.com/docs/hivemq-cloud/introduction.html#guide, 2021.

[13] ActiveBuddy Inc. Chatbot history: What is smarterchild. https://yakbots.com/chatbot-history-what-is-smarterchild/, 2001.

[14] Java. Gson: A java serialization/deserialization library to convert java objects into json and back. https://github.com/google/gson.

[15] Java. hivemq/hivemq-mqtt-client: Hivemq mqtt client. https://github.com/hivemq/hivemq-mqtt-client.

[16] JetBrains. Kjetbrains/exposed: Kotlin sql framework. https://github.com/JetBrains/Exposed.

[17] Ktor: Build asynchronous servers and clients in kotlin. https://ktor.io/.

[18] Karolina Kuligowska. Chatbot parry, kenneth mark colby — virtual assistant parry — virtual agent parry — chat bot parry — conversational agent parry. https://www.chatbots.org/chatbot/parry/, 1972.

[19] Creative Labs. Chatbot history: What is dr. sbaitso. https://yakbots.com/chatbot-history-what-is-dr-sbaitso/, 1992.

[20] Creative Labs. Chatbot jabberwacky, icogno — virtual assistant jabberwacky — virtual agent jabberwacky — chat bot jabberwacky — conversational agent jabberwacky. https://classicreload.com/dr-sbaitso.html, 1992.

[21] Leslie Lamport and Nancy A. Lynch. Distributed computing: Models and methods. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 1157–1199. Elsevier and MIT Press, 1990.

[22] Robert Martin. A quick introduction to clean architecture. https://www.freecodecamp.org/news/a-quick-introduction-to-clean-architecture-990c014448d2/.

[23] Robert Martin. Clean architecture: A craftsman's guide to software structure and design. https://booksvooks.com/nonscrolablepdf/clean-architecture-pdf-robert-c-martin.html, 2018.

[24] Microsoft. Cortana, 2015. https://yakbots.com/virtual-assistants-microsoft-cortana-chatbot/#more-18145, 2015.

[25] Node.js. node-telegram-bot-api: Telegram bot api for nodejs. https://github.com/yagop/node-telegram-bot-api.

[26] Dominik Obermaier and Ian Skerret. Introducing hivemq cloud. https://www.hivemq.com/blog/introducing-hivemq-cloud/.

[27] Dominik Obermaier and Ian Skerret. Introducing hivemq mqtt client. https://hivemq.github.io/hivemq-mqtt-client/.

[28] Dominik Obermaier and Ian Skerret. Mqtt: The messaging and data exchange protocol of the iot. https://www.hivemq.com/mqtt-protocol/.

[29] Vincent Rijmen and Joan Daemen. axios/axios: Promise based http client for the browser and node.js. https://www.tutorialspoint.com/cryptography/advanced_encryption_standard.htm.

[30] Vincent Rijmen and Joan Daemen. Kotlin- rsa, aes, 3des encryption and decryption with example. https://www.knowledgefactory.net/2021/01/kotlin-aes-rsa-3des-encryption-and.html.

[31] Vincent Rijmen and Joan Daemen. What is the aes algorithm? https://www.educative.io/edpresso/what-is-the-aes-algorith.

[32] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition.* Pearson Education, 2007.

[33] Developing the ubiquitous language. https://thedomaindrivendesign.io/developing-the-ubiquitous-language/.

[34] Edwin van Asch. Chatbot jabberwacky, icogno — virtual assistant jabberwacky — virtual agent jabberwacky — chat bot jabberwacky — conversational agent jabberwacky. https://www.chatbots.org/chatterbot/jabberwacky/, 1988.

[35] Erwin van Lun. Chatbot eliza, joseph weizenbaum — virtual assistant eliza — virtual agent eliza — chat bot eliza — conversational agent eliza. https://www.chatbots.org/chatbot/eliza/, 1966.

[36] Richard Wallace. Chatbot a.l.i.c.e., a.l.i.c.e. a.i foundation — virtual assistant a.l.i.c.e. — virtual agent a.l.i.c.e. — chat bot a.l.i.c.e. — conversational agent a.l.i.c.e. https://www.chatbots.org/chatbot/a.l.i.c.e/, 1995.

[37] Richard Wallace. Chatbot history: The alice chatbot. https://yakbots.com/chatbot-history-the-alice-chatbot/, 1995.

[38] J. Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9:36–45, 1966.

Appendix A

Technologies

A.1 MQTT

A.1.1 MQTT Topic: # Wildcard

Listing A.1: Example of multi-level topic in the MQTT Client

```
mqttTempClient.toAsync().subscribeWith()
    .topicFilter("sensors/#") //Multi-level topic
    .qos(MqttQos.EXACTLY_ONCE) //QoS level
    .send()
```

In MQTT, the word *topic* refers to hierarchical UTF-8 strings. Each topic level is separated by a forward slash, the topic level separator.

A client can subscribe to individual or multiple topics using wildcard characters.

In listing A.1, we show an example of a multi-level topic wildcard, which is represented by the hash symbol, the multi-level wildcard. It must be placed as the last character to allow the Broker to determine which topics match. Furthermore, when

a client subscribes to a topic with a multi-level wildcard, it receives all messages of a topic that begins with the pattern before the wildcard character.

A.1.2 QoS

In listing A.1, another important aspect of the MQTT protocol is reported, that is the QoS. As already mentioned, there are 3 levels of QoS:

- **QoS 0**: "at most once delivery."
- **QoS 1**: "at least once delivery".
- **QoS 2**: "exactly once delivery".

In this case, we show the declaration of the QoS level we decide to use, that is to say that of level 2. This latter involves a four-step handshake in order to confirm that the sender has actually received the messages and that it is received only once.

A.1.3 Creation of a MQTT Client

In listing A.2, we show the process of creation of an MQTT client. An utility that we use is that of the `builder`, which is accessible from the `MqttClient` class existing in the `com.hivemq.client.mqtt.*` package.

Thanks to the builder, it is possible to configure and create an MQTT client, which can be used to connect to Broker, subscribe to topics and, then, publish messages.

The created client has its own UUID and connects to the MQTT cloud Broker using SSL protocol, a specific port (the `8884`), and a specific version (the `MQTT 5 protocol`).

Once we create the MQTT client, the next step is to connect it to the Broker and the procedure is shown in listing A.3.

Listing A.2: Creation of an MQTT Client

```
1  fun initialize() {
2      var mqttPublisherClient: Mqtt5BlockingClient =
          MqttClient.builder()
3          .useMqttVersion5()
4          .identifier("temperatureClient_\${UUID.
              randomUUID()}")
5          .serverHost(host)
6          .serverPort(8884)
7          .sslWithDefaultConfig()
8          .webSocketConfig()
9          .serverPath("mqtt")
10         .applyWebSocketConfig()
11         .buildBlocking()
12 }
```

Listing A.3: Connection to the MQTT Broker using credentials

```
1     //Connect securely with username, password.
2      mqttPublisherClient.connectWith()
3          .simpleAuth()
4          .username(username)
5          .password(Charsets.UTF_8.encode(pwd))
6          .applySimpleAuth()
7          .cleanStart(false)
8          .send()
```

Appendix B

Library

B.1 Exposed

One of the most popular libraries for working with databases is the *Exposed* library. It is an ORM framework for Kotlin which offers two levels of database access: a typesafe SQL-wrapping DSL, the one we choose, and a lightweight data access object (DAO).

Every database access, using Exposed, is started by obtaining a connection and, then, creating a transaction. First of all, we need to define how to connect to a database. This is done via the `Database.connect()` function, as can be seen in listing B.1 (line 2). This function will not create a real database connection but only provide a descriptor for future usage. A real connection will be instantiated thanks to the transaction lambda, a CRUD operation which encapsulates a set of DSL operations.
The database we have chosen to save data is H2 while the saving method is "in file" with SQL mode.

Listing B.1: Data Persistence through Exposed library

```
//Database connection
Database.connect(
"jdbc:h2:./main/resources/cityTempAndHumidity;MODE=MySQL
   ",
"org.h2.Driver"
)
transaction {
    SchemaUtils.create(CityTemperature)
    SchemaUtils.createMissingTablesAndColumns(
        CityTemperature)
}
```

B.2 Gson

Gson is typically used by first constructing a Gson instance and then invoking `toJson(Object)` or `fromJson(String, Class)` methods. Gson instances are Thread-safe so we can reuse them freely across multiple threads without causing any blocks.

In listing B.2, the `sendSuccessResponse` method (line 11)is used to serialize the `ComplexMsgWithParams` data structure in a Json object and, then, send it to the Bot which renders the message and the list of commands to the user.

An example of the use of Gson to execute deserialization is reported in listing B.2 (line 29). In this case, the message that is received in the communication channel, received as a Json String, is converted into an Object of the specified class, the `ComplexMsg` class.

Listing B.2: Use of Gson to implement (De)serialization

```
@Serializable //Serializable data structure
data class ComplexMgsWithParams(
    val message: String?,
    val command: List<String>?,
    val contentType: ContentType,
    val httpStatusCode: HttpStatusCode
)
//Serialization method
var gson = Gson() //Gson instance

private fun sendSuccessResponse(message: String,
   commands: List<String>): String {
    return gson.toJson(
        ComplexMgsWithParams(
            message,
            commands,
            ContentType.Text.Plain,
            HttpStatusCode.OK
        )
    )
}
//Deserialization method
@Serializable
data class ComplexMsg(
    val message: String,
    val command: List<String>
)

val cMsg: ComplexMsg =
    gson.fromJson(
        channelOutput.receive(),
        ComplexMsg::class.java
    )
```

Appendix C

Code representation

C.1 Telegram Bot: Node.js Telegram Bot API

Listing C.1: Implementation of the back-end of the Bot application

```
const TelegramBot = require('node-telegram-bot-api');
var token = "" //Use the real telegram token
const bot = new TelegramBot(token, {polling: true});
bot.on('message', (msg) => {
  const msgReceived = msg.text.toString()
  if (msgReceived === "/start") {/* Welcome */
    const params = { chatId: msg.chat.id};
    return axios.get(url, { params }).then((res) => {
      bot.sendMessage(msg.chat.id, res.data.message, {
        reply_markup: {keyboard: [res.data.command]}
      });
    })
  } });
```

C.2 CoreSystem

How to declare a server-side application

Listing C.2: Creation of the Server using Ktor

```
fun createServer(pwd: String) {
    embeddedServer(Netty, port = 8000) {
        routing {
            get("/token") { ... } //Authentication API
            get("/") { ... } //START API
            get("freeText") { ... } //Message handling
               API
        }
    }.start(false)
}
```

UtilityHomeConnection: Open websocket

Listing C.3: Opening websocket

```
scope.launch {
  client.webSocket(method = HttpMethod.Get, host =
     ipAddress, port = 8080, path = "/home") {
    handshake()
    handleWS()
  }
}
```

Listing C.4: Method to handle websocket

```
private suspend fun DefaultClientWebSocketSession.
   handleWS() {
  try {
    val inputCh: ReceiveChannel<String> = channelInput
    val outputCh: SendChannel<String> = channelOutput
    val jobIn = scope.launch {
      while (true) {
        val input = inputCh.receive()
        send(input)
      }
    }
    val jobOut = scope.launch {
      for (message in incoming) {
        message as? Frame.Text ?: continue
        val receivedText = message.readText()
        outputCh.send(receivedText)
      }
    }
    jobIn.join()
    jobOut.join()
  } catch (e:Exception){
    e.printStackTrace()
  }
  client.close()
}
```

UtilityHomeConnection: Manage websocket

C.3 PublisherClient

Listing C.5: Main function of the project with concurrent threads

```
suspend fun main() = coroutineScope {
  val channel: Channel<String> = Channel()
  val publisher = PublisherClient()
  publisher.initialize()
  launch {
    Detector.initialize(channel, serialName)
    Detector.runDetector()
  }
  launch {
    InputAnalyzer(channel, publisher).run()
  }
}
```