ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Scuola di Scienze
Corso di Laurea Magistrale in Informatica

RED-Bridge: A Multiprotocol Tool Prototype for IoT Sensors

Supervisor:
Prof. Marco Di Felice

Co-supervisor:
Dr. Luca Sciullo

Candidate:
Dr. Riccardo Maffei

Session II
Academic Year 2021/2022

Information about licenses and credits can be found in Appendix A.

Ad meliora et maiora semper!

Abstract

The IoT is growing more and more each year and is becoming so ubiquitous that it includes heterogeneous devices with different hardware and software constraints leading to an highly fragmented ecosystem. Devices are using different protocols with different paradigms and they are not compatible with each other; some devices use request-response protocols like HTTP or CoAP while others use publish-subscribe protocols like MQTT. Integration in IoT is still an open research topic.

When handling and testing IoT sensors there are some common task that people may be interested in: reading and visualizing the current value of the sensor; doing some aggregations on a set of values in order to compute statistical features; saving the history of the data to a time-series database; forecasting the future values to react in advance to a future condition; bridging the protocol of the sensor in order to integrate the device with other tools.

In this work we will show the working implementation of a low-code and flow-based tool prototype which supports the common operations mentioned above, based on Node-RED and Python. Since this system is just a prototype, it has some issues and limitations that will be discussed in this work.

Contents

Introduction **1**

1 General Concepts and Technologies **3**
1.1 Node-RED . . . 3
1.2 InfluxDB . . . 3
1.3 Protocols . . . 4
1.3.1 HTTP . . . 5
1.3.2 CoAP . . . 7
1.3.3 MQTT . . . 8
1.4 W3C Web of Things . . . 9

2 Architecture and Functionalities **13**
2.1 Goals and Use Cases . . . 13
2.2 Requirements and Functionalities . . . 13
2.2.1 Additional and Experimental Functionalities . . . 14
2.2.1.1 Data Forecasting . . . 14
2.2.1.2 W3C Web of Things Support . . . 15
2.3 Architecture Overview . . . 15
2.3.1 Frontend . . . 15
2.3.2 Backend . . . 16

3 Implementation **17**
3.1 Backend . . . 17
3.1.1 Flow Overview . . . 17
3.1.2 Input Module . . . 18
3.1.2.1 MQTT . . . 18
3.1.2.2 CoAP . . . 19
3.1.2.3 HTTP . . . 19
3.1.2.4 WoT Property Read . . . 20
3.1.3 Processing Module . . . 20
3.1.3.1 Passthrough Pipeline . . . 21
3.1.3.2 Aggregation Pipeline . . . 21
3.1.3.3 Forecasting Pipeline . . . 22
3.1.4 Output Module . . . 22
3.1.4.1 File . . . 23
3.1.4.2 WebSocket . . . 24
3.1.4.3 MQTT . . . 24
3.1.4.4 CoAP . . . 24

3.1.4.5 HTTP . . . 25
3.1.4.6 WoT Property Write . . . 26
3.1.4.7 InfluxDB . . . 26
3.1.5 WoT Introduction . . . 27
3.2 Frontend . . . 28
3.2.1 Parameters and Configuration Handling . . . 29
3.2.2 Flow Manipulation and Deploy . . . 30
3.2.3 Console Handling . . . 31
3.2.4 Self Thing Description Generation . . . 31

4 Testing and Evaluation **33**
4.1 Manual I/O Testing . . . 33
4.1.1 MQTT . . . 33
4.1.2 CoAP . . . 33
4.1.3 HTTP . . . 34
4.1.4 WoT . . . 34
4.1.5 File . . . 34
4.1.6 WebSocket . . . 34
4.1.7 InfluxDB . . . 34
4.2 Manual Processing Testing . . . 35
4.2.1 Passthrough Pipeline . . . 35
4.2.2 Aggregation Pipeline . . . 35
4.2.3 Forecasting Pipeline . . . 35
4.3 Whole System Testing . . . 35
4.3.1 Edge Device Prototype . . . 36
4.3.1.1 TMP36 Sensor . . . 36
4.3.1.2 ESP32 Prototyping Board . . . 37
4.3.1.3 Temperature Reading . . . 37
4.3.1.4 MQTT, CoAP and HTTP Implementation . . . 38
4.4 Bridging Performance Evaluation . . . 38
4.4.1 Evaluated Protocols . . . 39
4.4.2 Evaluation Scenario . . . 39
4.4.3 Results . . . 41

5 Issues and Improvements **45**
5.1 Limiting Frontend . . . 45
5.2 I/O Format Assumption . . . 45
5.3 HTTP Server Port . . . 45
5.4 WoT: TD, Protocols and Authentication . . . 45
5.5 CoAP Input Observe . . . 46
5.6 Naive Forecasting Implementation . . . 46
5.7 Single Flow Limit . . . 46

6 Conclusions and Future Works **47**

Appendix A Licenses and Credits **49**
A.1 Third-Party Content . . . 49

Acknowledgments 51

References 53

List of Figures

1.1 Example of Node-RED flow. 3
1.2 InfluxDB GUI showing sample data. 4
1.3 Example of HTTP request-response. 5
1.4 CoAP message format. 7
1.5 Example of CoAP request-response with non-confirmable messages. . 8
1.6 Example of MQTT publish-subscribe architecture. 9
1.7 WoT interaction abstraction diagram. 10
1.8 Architectural aspects of a WoT Thing. 11
1.9 Abstract architecture of W3C WoT. 12

2.1 System architecture overview. 15

3.1 Flow overview diagram. 17
3.2 Flow input module configuration. 18
3.3 Flow processing module configuration. 20
3.4 Flow output module configuration. 23
3.5 “Get last data” subflow implementation. 25
3.6 WoT Introduction flow configuration. 27
3.7 RED-Bridge usage help. 29

4.1 Screenshot of InfluxDB data explorer during testing. 34
4.2 Screenshot of the test of the aggregation operation. 36
4.3 TMP36 output voltage vs. temperature. 36
4.4 Edge device prototype mounting diagram. 37
4.5 Performance evaluation diagrams. 40
4.6 Box plots of the performance evaluation results. 42
4.7 eCDF plots of the performance evaluation results. 43

List of Listings

1 Code of the function node inside the forecasting pipeline. 22
2 Code of the function node inside the subflow. 25
3 Code of the function node that formats InfluxDB fields and tags. . . . 26
4 System's self TD template. 28
5 Example of MQTT input JSON configuration. 30
6 Example of HTTP active output JSON configuration. 30
7 Example of InfluxDB JSON configuration. 30
8 `get_temp()` function implementation. 38
9 `get_sample()` function implementation. 38

List of Tables

4.1 Performance evaluation results table. 41

Introduction

According to a report by Morrish and Arnott [1], "at the end of 2021 there were 11.3 billion active IoT devices, a figure which will grow to 29.4 billion in 2030, a compound annual growth rate (CAGR) of 12%.". More and more physical objects are becoming "smart devices" connected to the internet including sensors and actuators used, for example, in smart homes, smart agriculture [2] etc. IoT is becoming so ubiquitous that it includes heterogeneous devices with different hardware and software constraints leading to an highly fragmented ecosystem. In particular, devices are using different protocols with different paradigms depending on constraints such as computing power, power consumption and network capabilities; some devices use request-response protocols like HTTP or CoAP [3] while others use publish-subscribe protocols like MQTT [4]. At time of writing, there are several works and studies about interoperability and bridging between these protocols such as the Eclipse Ponte project [5], the work from Khaled and Helal [6] and the review by Tayur and Suchithra [7]. Furthermore, the W3C is currently working on a set of standards, mostly known as "Web of Things" [8], seeking "to counter the fragmentation of the IoT by using and extending existing, standardized Web technologies" [9].

When handling and testing IoT sensors there are some common task that people may be interested in. The most basic one is reading and visualizing the current value of the sensor; another one is doing some aggregations on a set of values in order to compute statistical features; one more can be saving the history of the data, usually to a time-series database like InfluxDB [10]; another common task could be forecasting the future values to react in advance to a future condition; finally, one of the common task could be bridging the protocol of the sensor in order to integrate the device with other tools.

The goal of this work is to present the design, implementation and testing of a simple-to-use tool for IoT sensors supporting the common operations mentioned above and minimizing the integration efforts. The system will be able to handle sensors supporting HTTP, CoAP and MQTT protocol and also support consuming W3C WoT Things. We will present a simple architecture where the backend is a custom-built flow [11] running on a Node-RED [12] instance and the frontend is a Python script providing a command line interface to let the user configure and interact with the system. After going into the implementation details, we will discuss the tests, all passed by the system, including tests using command line tools, adhoc flows and simulated devices but also tests executed with the help of an IoT sensor prototype we custom designed and built. We also wanted to evaluate the performances of the protocol bridging measuring the transmission time in order to estimate the overhead introduced by the system; we will present the evaluation scenario and the analyzed results that confirmed that the overhead introduced by

the proposed implementation is minimum. As we will describe, the system we will present is just a prototype and still has several issues that make it unsuitable for production environment or important tasks.

The structure of this work is divided in chapters; in particular, in chapter 1 we will briefly list and present the technologies and general concepts used in this work; in chapter 2 we will describe the requirements, the architecture and the functionalities of the system; in chapter 3 we will analyze the actual implementation; in chapter 4 we will discuss the testing phase and the evaluation conducted on the system; finally, in chapter 5 we will describe the issues of the prototype and possible improvements before drawing conclusions in chapter 6.

1 | General Concepts and Technologies

In this chapter we will briefly introduce general concepts and technologies used in this work. For more details we advise to refer to the ample existing literature and materials about these topics.

1.1 Node-RED

According to the official website [12]:

> Node-RED is a programming tool for wiring together hardware devices, APIs and online services in new and interesting ways. It provides a browser-based editor that makes it easy to wire together flows using the wide range of nodes in the palette that can be deployed to its runtime in a single-click.

Node-RED allows flow-based development [11] connecting built-in, third-party and custom nodes to easily create complex integrations and functionalities.

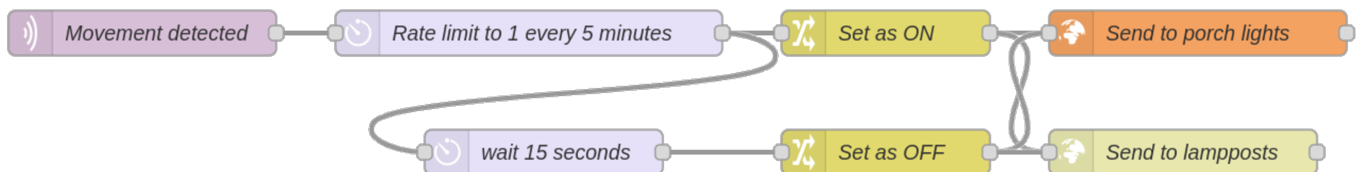


Figure 1.1: Example of Node-RED flow.

Suppose that we have some movement sensors in the garden that publish a message through MQTT when triggered. We want to turn on the smart lampposts and porch lights but they have to be controlled through HTTP and CoAP `POST`s, respectively. We want to limit this activation to one every 5 minutes and we want all the lights to be turned off after 15 seconds. In Figure 1.1 is shown a small implementation of this automation as a Node-RED flow.

1.2 InfluxDB

InfluxDB [10] is an open-source time-series database written in GO. It provides a graphical user interface, a command line interface and HTTP APIs. Starting from version 1.8, it supports Flux, a functional data scripting language designed for querying, processing, analyzing, and acting on data.
Some of the most important elements in InfluxDB are:

- **Bucket**: a container for all data, combining the concept of a database and a retention policy.
- **Measurement**: a container for fields, tags, and timestamps (similar to a table in RDBMS).
- **Timestamp**: a timestamp associated with the data, stored in epoch nanosecond format but shown in the format proposed in RFC 3339 [13].
- **Field keys and values**: names and values of actual data (similar to column names and values in RDBMS).
- **Tag keys and values**: optional indexed metadata (strings).
- **Series**: a logical group of data defined by shared measurement, field key and tag set.

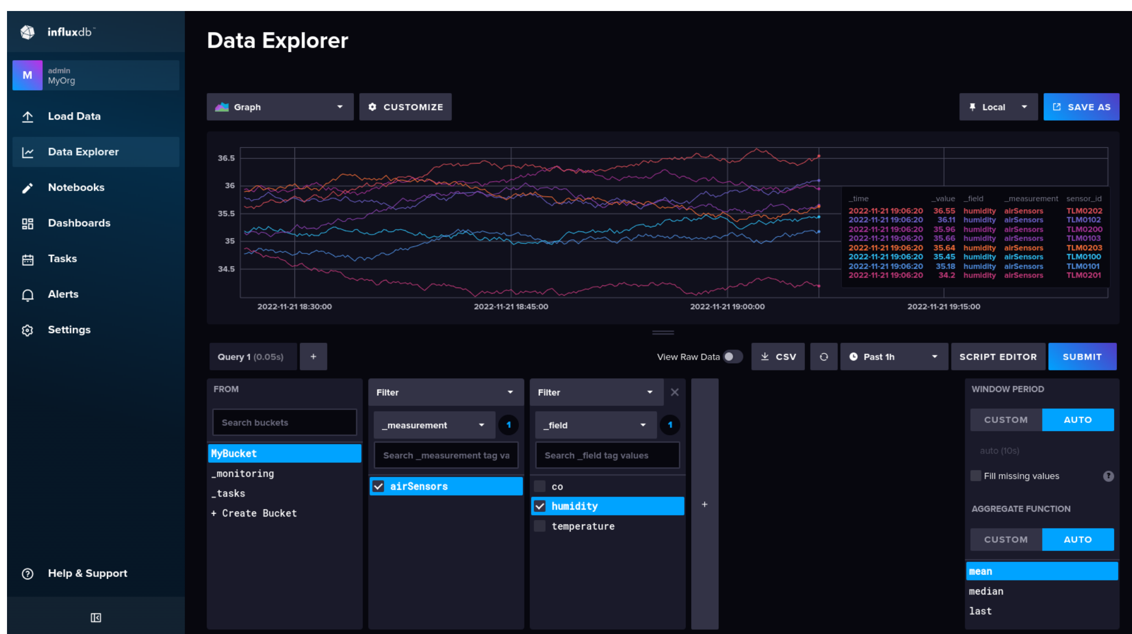


Figure 1.2: InfluxDB GUI showing sample data.

Figure 1.2 shows a screenshot of the InfluxDB data explorer with sample data. In particular, the bucket "`MyBucket`" contains a measurement named "`airSensors`" with "`co`", "`humidity`" and "`temperature`" as fields but the query filtered only the humidity. The measurement has a tag named "`sensor_id`" whose value is the id of the sensor. The plot shows the different series, one for each combination of field key and tag set.

1.3 Protocols

1.3.1 HTTP

HTTP is probably one of the most known application layer protocol and the foundation of the World Wide Web. This protocol uses a request-response interaction model, as shown in Figure 1.3, and is usually used in IoT creating REST [14] APIs.

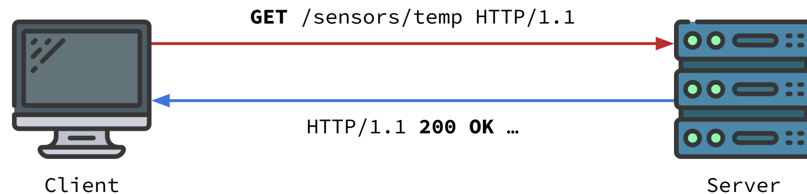


Figure 1.3: Example of HTTP request-response.

As of version 1.1, a request message is composed of:

- **a request line**, like "`GET /sensors/temp HTTP/1.1`", containing:
  - the case-sensitive request method name followed by a space;
  - the requested URL followed by another space;
  - the protocol version followed by a CR and a LF.
- **1 or more headers**, like "`Host: www.example.com`", containing:
  - the case-insensitive header name followed by a colon;
  - the header value (with optional leading/trailing space) followed by a CR and a LF.
- **an empty line** (CR and LF);
- **an optional message body**.

The protocol provides several request methods. Some of the most important for IoT are, for example:

- **GET**: this method is used to retriever the state of the resource, including its metadata; for example, this method can be used to retriever the value of a sensor.
- **HEAD**: this method is used like the previous one but retrievers only the metadata, without the content; for example, this method can be used to check the last modification date of the resource.
- **POST**: this method is used to request that the target resource handles the given representation according to its semantics; for example this method can be used to ask an IoT device to execute an action.

- **PUT**: this method is used to update the state of the resource with the one in the request; for example, it can be used to set the brightness of a smart light.

As of version 1.1, a response message is composed of:

- **a status line**, like "`HTTP/1.1 200 OK`", containing:
  - the protocol version followed by a space;
  - the response status code followed by another space;
  - an optional reason phrase followed by a CR and a LF.
- **0 or more headers**, like "`Content-Type: application/json`", containing:
  - the case-insensitive header name followed by a colon;
  - the header value (with optional leading/trailing space) followed by a CR and a LF.
- **an empty line** (CR and LF);
- **an optional message body**.

The protocol provides several 3-digit response status codes that can be divided in 5 classes according to the first digit:

- **1xx**: this class is used for informational responses; in this case the request was received and understood.
- **2xx**: this class is used for successful responses; in this case the request was successfully received, understood, and accepted.
- **3xx**: this class is used for redirection responses; in this case further actions are necessary in order to complete the request.
- **4xx**: this class is used for client error responses; in this case the request is not valid and cannot be fulfilled.
- **5xx**: this class is used for server error responses; in this case the server failed to fulfill an apparently valid request.

The main advantage of using HTTP as IoT protocol is that interactions can be done using widely available standard tools designed for the web. On the other hand, HTTP was not designed to be used in constrained scenarios where the energy efficiency and network availability and conditions must be taken in consideration. Therefore, sometimes HTTP cannot be used as messaging protocol for constrained IoT devices.

1.3.2 CoAP

Constrained Application Protocol is a lightweight application layer protocol for constrained devices, as defined by Shelby, Hartke, and Bormann [3]. The protocol has been designed to be easily translated to HTTP using the same interaction model while trying to minimize the requirements of the device and the network. For example, the protocol has a smaller header, is asynchronous and is based on UDP [15] but supports optional mechanisms to enhance reliability (e.g. confirmable messages and retransmission).

Byte	0								1								2								3							
Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Field	Ver		Type		TKL				Request/response code								Message ID															
	Token (if any, TKL bytes)																															
	Options (if any)																															
	1	1	1	1	1	1	1	1	Payload (if any)																							

Figure 1.4: CoAP message format.

Unlike HTTP, CoAP uses a binary message format, shown in Figure 1.4, that can be as small as 4 bytes.
In particular:

- **Ver**: indicates the CoAP protocol version;
- **Type**: indicates the message type that can be:
  - **0**: this message is a confirmable request and expects an acknowledgment;
  - **1**: this message is a non-confirmable request and doesn't expect an acknowledgment;
  - **2**: this message is an acknowledgment to a confirmable message;
  - **3**: this message indicates a "reset" because the received message couldn't be processed.
- **TKL**: indicates the length of the token (from 0 to 8 bytes);
- **Request/response code**: indicates the request method or the response code;
- **Message ID**: is a message identifier used to detect duplicates and to match messages of type acknowledgment/reset to messages of type confirmable/non-confirmable;
- **Token**: is a variable-length identifier used to match the requests with the responses;
- **Options**: are additional options (e.g. the content format);
- **Payload**: is the actual payload of the message (e.g. the value of the sensor).

CoAP supports several request methods including `GET`, `PUT` and `POST` with similar, but not identical, semantics to those of HTTP methods. CoAP response codes relate to a small subset of HTTP status codes with some CoAP-specific additions. A protocol extension [16] added support for resource observation. More information about this protocol can be found in the RFC [3].

In Figure 1.5 is shown an example of a coap request-response using non confirmable messages: the client sends a non confirmable `GET` request with id `0x1234`, and token `0x42`; the server, later, replies with a non confirmable response message including the same token and the result.

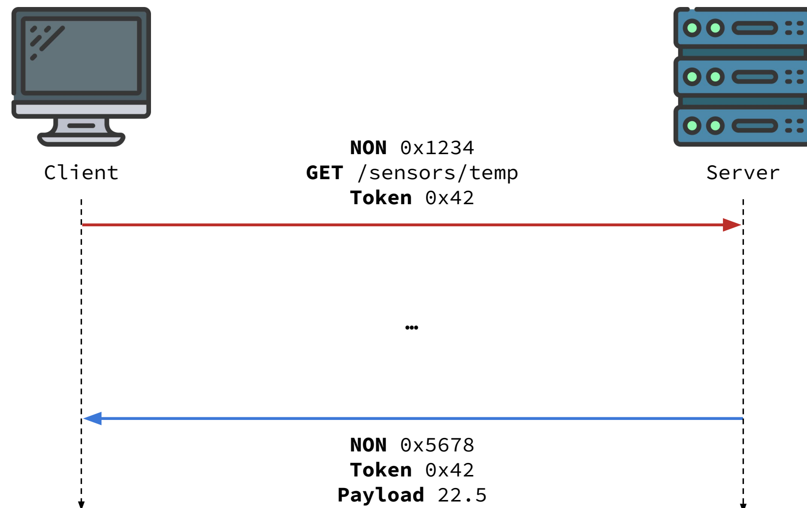


Figure 1.5: Example of CoAP request-response with non-confirmable messages.

1.3.3 MQTT

Message Queue Telemetry Transport is a lightweight application layer messaging protocol for resource-constrained environments standardized by OASIS [4]. Unlike HTTP and CoAP, this protocol is based on the publish-subscribe interaction model as shown in Figure 1.6. Clients connect to a mediator, an MQTT broker, and can publish messages with a string topic and/or declare their interest in one (or more) topic. The mediator allows many-to-many interactions and space/time decoupling. The connections to the broker are usually based on TCP [17] which guarantees delivery on the link between the client and the broker; however, messages can still be lost (e.g. if the broker application crashes). For this reason, the protocol provides 3 different levels of quality of service:

- **At most once (QoS level 0)**: with this configuration the message is sent only once (fire and forget).
- **At least once (QoS level 1)**: with this configuration the sender stores the message and keeps resending it until an acknowledgment is received (acknowledged delivery).

- **Exactly once (QoS level 2)**: with this configuration the sender and the receiver engage in four-part handshake to ensure that one and only one copy of the message is received (assured delivery).

The protocol provides some basic security mechanism for authentication and confidentiality but they mainly relies on lower layers or other components. The MQTT standard provides other configurations which are not described in this document.

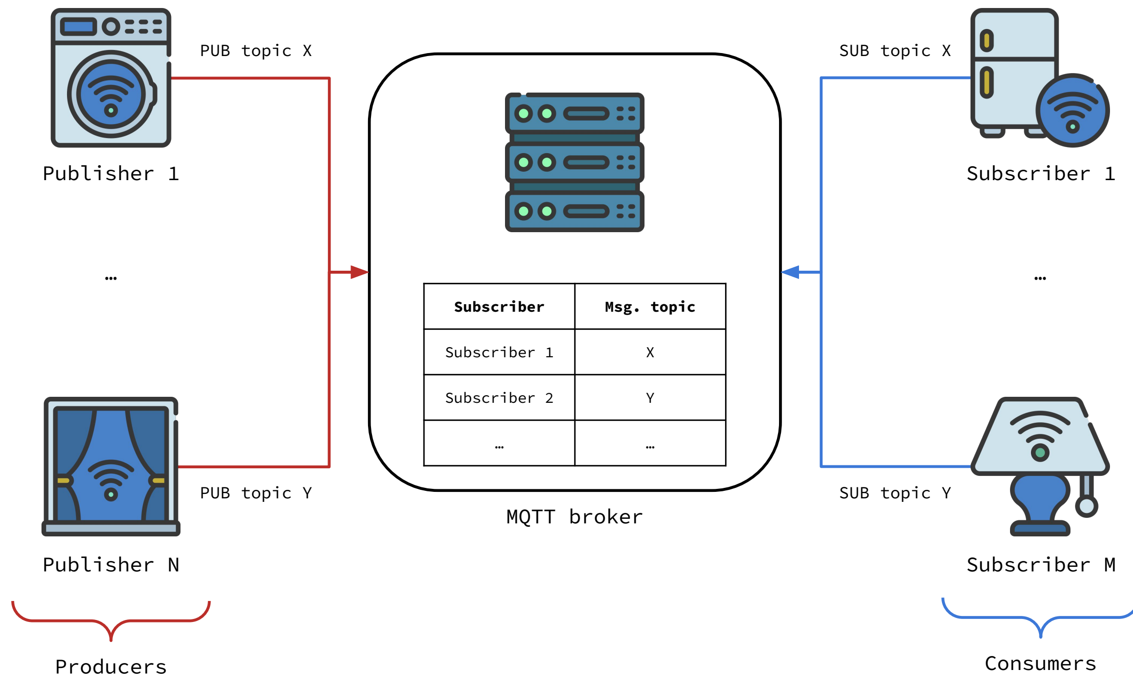


Figure 1.6: Example of MQTT publish-subscribe architecture.

1.4 W3C Web of Things

According to the official website [9]:

> The Web of Things (WoT) provides a set of standardized technology building blocks that help to simplify IoT application development by following the well-known and successful Web paradigm. This approach increases flexibility and interoperability, especially for cross-domain applications, as well as enabling reuse of established standards and tools.

The W3C published some normative deliverables, recommendations and drafts, with more information about the WoT including: the Architecture [8], the Thing Description [18] and the Discovery [19].

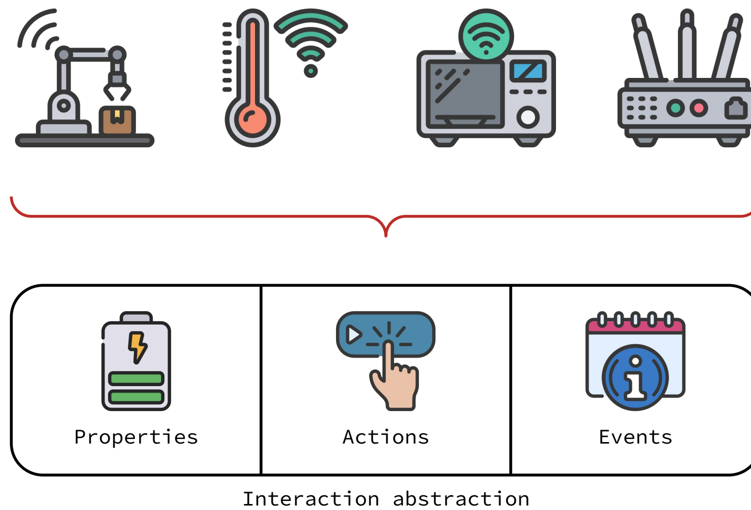


Figure 1.7: WoT interaction abstraction diagram.

One of the most important concepts introduced by the W3C WoT is an interaction abstraction, depicted in Figure 1.7, based on properties, actions and events:

- **Properties** are interaction affordances that expose some kind of state of the Thing. Some examples are: configuration parameters, sensor values, computation results etc.
- **Actions** are interaction affordances used to invoke a function of a Thing like starting a process (e.g. brewing a coffee), manipulating some hidden configurations etc.
- **Events** are interaction affordances that describe a source of events asynchronously sent by the Thing to the Consumer. Some examples are: alarms (e.g. a fire alarm), events (e.g. door opened) etc.

Another important WoT concept used in this work is the WoT Thing Description, one of the WoT central building blocks and the entry point of a Thing. A TD is a JSON-LD [20], usually provided by the IoT device itself or by an external repository (e.g. a TD Directory), describing the Thing and how it can be used. In particular, a TD instance has 4 main components:

- textual metadata about the Thing;
- a set of interaction affordances that indicate how the Thing can be used;
- schemas for the data exchanged with the Thing;
- web links to express any relation to other Things or documents on the Web.

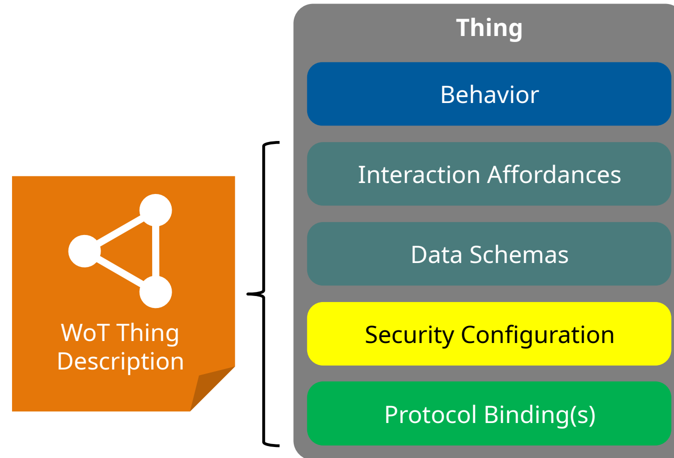


Figure 1.8: Architectural aspects of a WoT Thing.

Figure 1.8 shows the main architectural aspects of a WoT Thing. In particular:

- the behavior of a Thing includes both the autonomous behavior and the implementation of the handlers for the interaction affordances;
- the interaction affordances provide a model of how the Thing can be used through abstract operations, but without reference to a specific network protocol or data encoding;
- the protocol bindings provide a mapping between the interaction affordances and the concrete protocol messages along with additional details; in general, a Thing may use different protocols to support different interaction affordances;
- the security configuration represents the access control mechanisms used to access the interaction affordances and the management of related public security metadata (e.g. the authentication mechanism) and private security data (e.g.: secrets, private keys etc.).

Figure 1.9 shows the abstract architecture of the W3C WoT and how its concepts can be applied and combined to address even the more complex use cases. In particular [8]:

> The concepts of W3C WoT are applicable to all levels relevant for IoT applications: the device level, edge level, and cloud level. This fosters common interfaces and APIs across the different levels and enables various integration patterns such as Thing-to-Thing, Thing-to-Gateway, Thing-to-Cloud, Gateway-to-Cloud, and even cloud federation, i.e., interconnecting cloud computing environments of two or more service providers, for IoT applications.

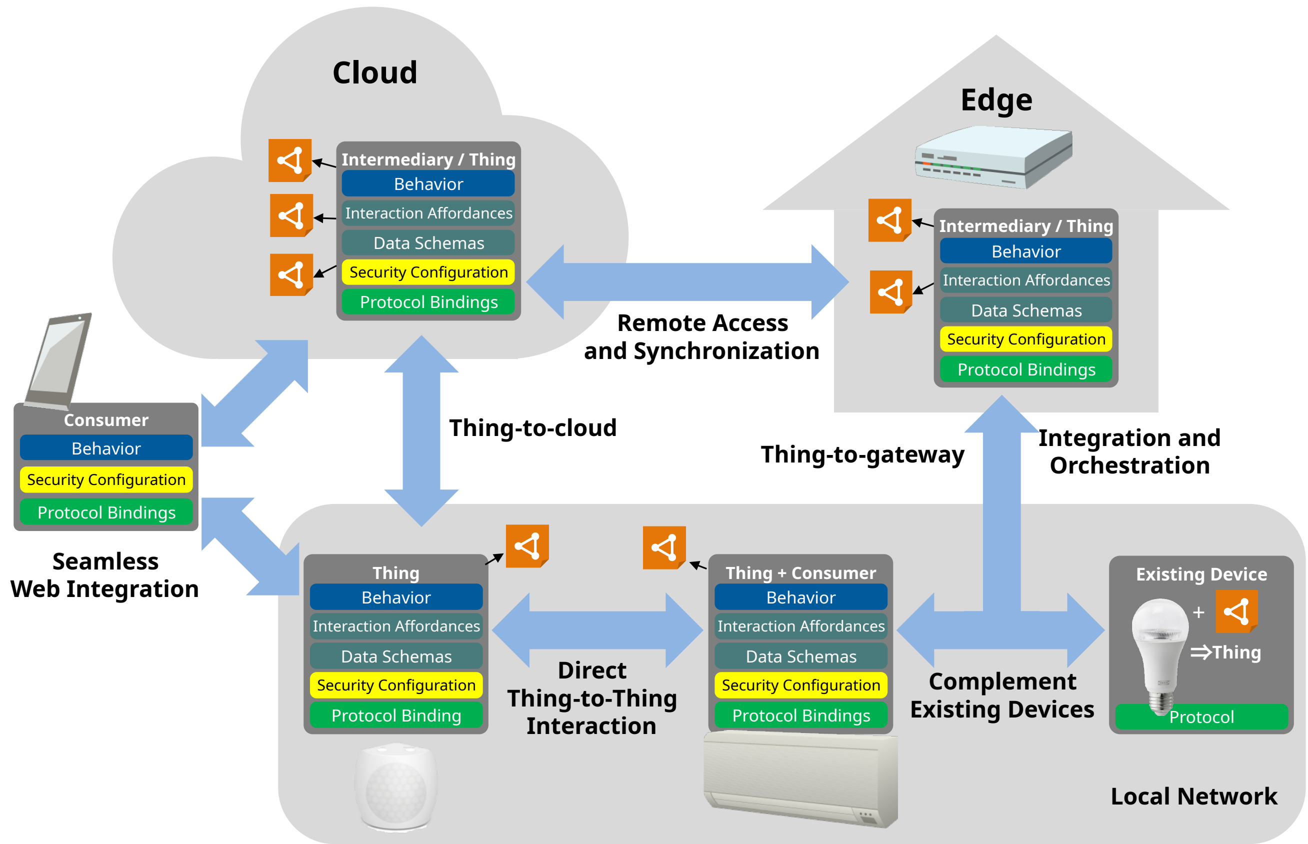


Figure 1.9: Abstract architecture of W3C WoT.

More information and details can be found in the normative deliverables.

2 | Architecture and Functionalities

In this chapter we will briefly describe the requirements, the functionalities and the architecture of the system. More details about the implementation in chapter 3.

2.1 Goals and Use Cases

The goal of the system is to provide an easy to use command line tool to handle IoT sensors that supports different protocols in order to support common tasks while reducing integration efforts. Follows a small set of examples of use cases.

Data visualization The system can be used, for example, to quickly access and visualize in real time the values coming from sensors with different protocols; this could be useful especially for diagnostic purposes.

Data storage The system can be used when the user needs to save to a time-series database the history of the values coming from sensors supporting different protocols (e.g.: temperature sensors, door sensors etc.).

Data forecasting The system can be used to create a data flow that forecasts the future value of a sensor in order to react in advance to a future condition. For example, the tool can output, in any of the supported protocols, the future temperature or humidity of a room (e.g. a wine cellar); in this way, a different existing component can access that flow and trigger the HVAC system.

Protocol bridging The system can be used to bridge the protocol of a sensor to another one supported by the existing IoT ecosystem; for example, an existing automation system may support MQTT while the sensor may expose a CoAP end-point. The two protocols can be easily bridged by the tool with a single, simple, command.

2.2 Requirements and Functionalities

The system has some requirements and functionalities described in this section and other additional and experimental functionalities described in their own subsection. The most important requirements and assumptions are:

- the system should provide a CLI frontend;

- the frontend should be used to configure the system and to show the output, when necessary;
- the system can assume that the input data format is a JSON [21] object containing a property named "`value`" (e.g. `{"value": ... }`);
- the system should support at least MQTT, HTTP and CoAP;
- the system should support the following basic operations:
  - **Data visualization**: the system prints the JSON documents acquired in real-time.
  - **Data aggregation**: the system computes and prints the statistical features of the sensory data, such as the maximum, minimum, average and standard deviation, computed every `n` observations, where `n` is a custom parameter provided by the user.
  - **Data storage**: the system stores the data on an InfluxDB [10] database. In this case, the user must provide a JSON file with the required configuration such as: the URL of the device hosting the InfluxDB instance, the access token, the name of the bucket and additional tags. As a consequence of this action, the time series of the data flow are automatically sent to InfluxDB and stored there.
  - **Data bridging**: the system bridges data between supported protocols. In this case, the user must specify the destination protocol as well as the protocol-specific parameters in a JSON configuration file. The system produces a data flow in output using the target protocol. The system is orthogonal compared to the input/output protocols, it's able to acquire data in input in any protocol and to produce a data flow in output in any other protocol.

2.2.1 Additional and Experimental Functionalities

The systems supports some additional experimental functionalities but their implementation, shown in chapter 3, is just a Proof of Concept and should not be used for important tasks.

2.2.1.1 Data Forecasting

As an additional functionality, the system supports data forecasting. In this case, the system produces in output a data flow in the specified target protocol, however, it returns the next value of the time series computed according to the ARIMA model [22]. The statistical parameters of the ARIMA model (i.e. `p`, `d`, `q` parameters) are defined by the users and passed as input of the command.

2.2.1.2 W3C Web of Things Support

As an additional functionality, the system experimentally supports W3C Web of Things [8]. In particular, the system can interact with "Things" reading their property as an input or writing to them as an output. In this case, the user should provide the name of the property and the Thing Description [18]. In some user-selected configurations the system exposes some endpoints that others can use to push or pull data. In this case the system can act itself as a Thing generating and exposing its TD [18, 19] including a property named "`data`".

2.3 Architecture Overview

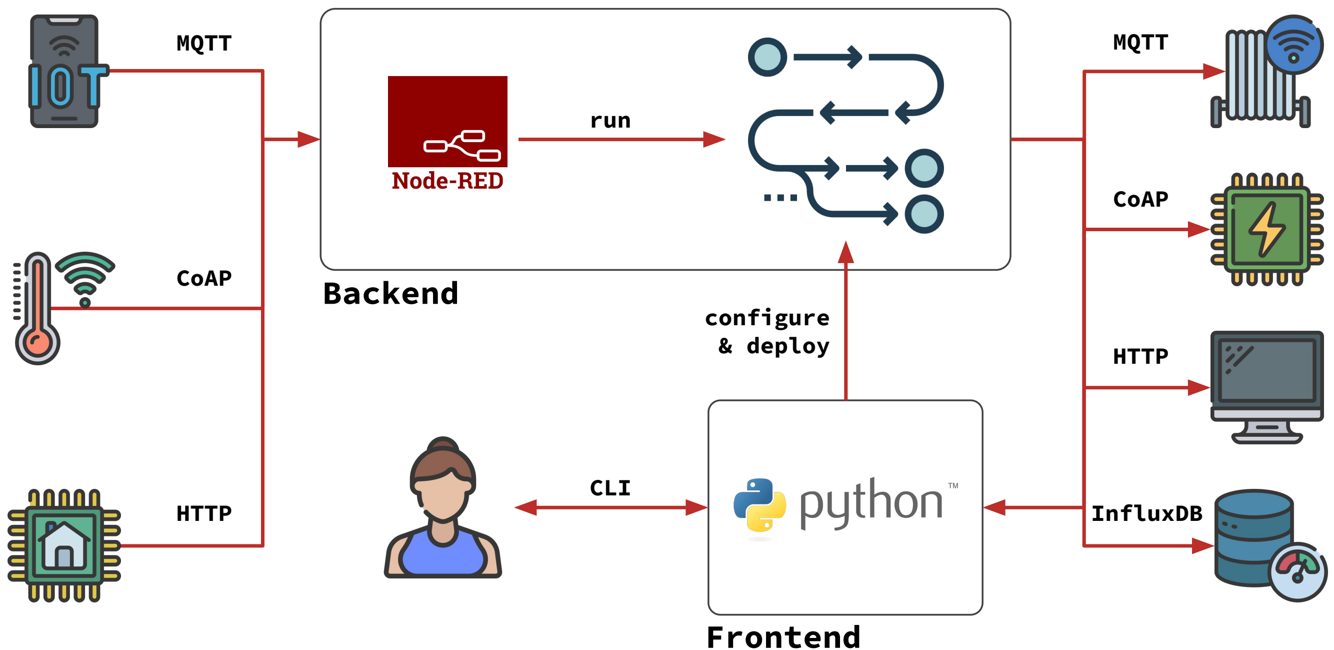


Figure 2.1: System architecture overview.

Figure 2.1 shows an overview of the architecture of the system along with some dummy IoT devices, InfluxDB etc.

2.3.1 Frontend

One of the main components of the system is a frontend application written in Python [23]. The application provides a command line interface used to configure the system. For example, the user can use the `command` parameter to specify the desired operation and provide all the necessary configurations (e.g.: ip addresses, credentials, protocols etc.) as JSON [21] configuration files. After parsing all the required configurations, the application generates and configure a Node-RED flow and automatically deploys it on the backend that will handle the actual operations. More information in chapter 3.

2.3.2 Backend

The backend is the core of the system. An instance of Node-RED [12] exposes its deployment APIs and receives the generated flow configuration from the frontend. The flow configuration is then executed in order to support the user-selected operations. If required, the backend can communicate with the frontend in order to show output in console. More information in chapter 3.

3 | Implementation

In this chapter we will briefly describe the implementation of the system and its additional components.

3.1 Backend

As stated in subsection 2.3.2, the backend has been designed creating a program, "flow" in jargon, deployed on a running Node-RED [12] instance. In order to implement all the functionalities, we used both built-in and third-party nodes. For some functionalities we used the "function" node that allowed us to write custom JavaScript [24] code. More details in the following sections.

3.1.1 Flow Overview

In order to implement all the functionalities and protocols, we decided to design a single big Node-RED flow and let the frontend take care of the activation and configuration of the appropriate nodes required to support the user-selected operations.

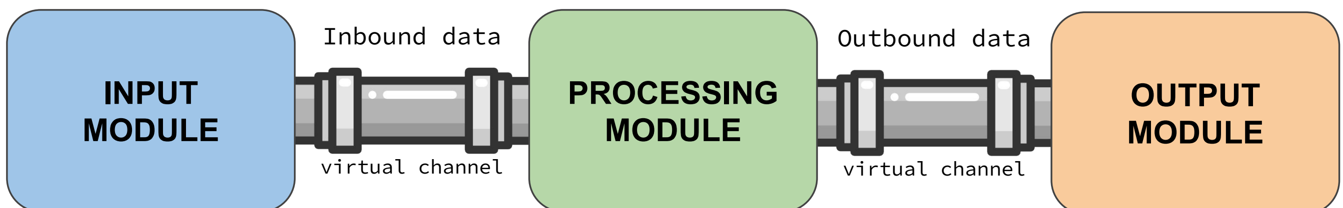


Figure 3.1: Flow overview diagram.

We designed the flow in a modular way. As shown in Figure 3.1, the flow has been divided in 3 modules communicating through 2 virtual channels called "Inbound data" and "Outbound data". The first module, shown in Figure 3.2, handles the input data and passes the result[1] to the following module; the second module, shown in Figure 3.3, handles the requested data processing (if necessary) and passes the result to the following module; the last module, shown in Figure 3.4, handles all kinds of data output[2]. This architecture aims to make modules independent from each other.

[1]Note that Node-RED nodes exchange messages (objects). The parsed input data is stored in the property "payload" of the message.

[2]Not limited to messaging protocols. This module supports also the InfluxDB, console etc.

3.1.2 Input Module

As mentioned before, this module handles the input of the data in all supported protocols assuming that the data itself is a JSON object containing a property named “`value`” (e.g. `{"value": ... }`). The obtained data is eventually pushed in the virtual channel “inbound data”.
At time of writing, the following protocols are supported:

- MQTT;
- CoAP;
- HTTP;
- WoT property read;

More details in the following sections.

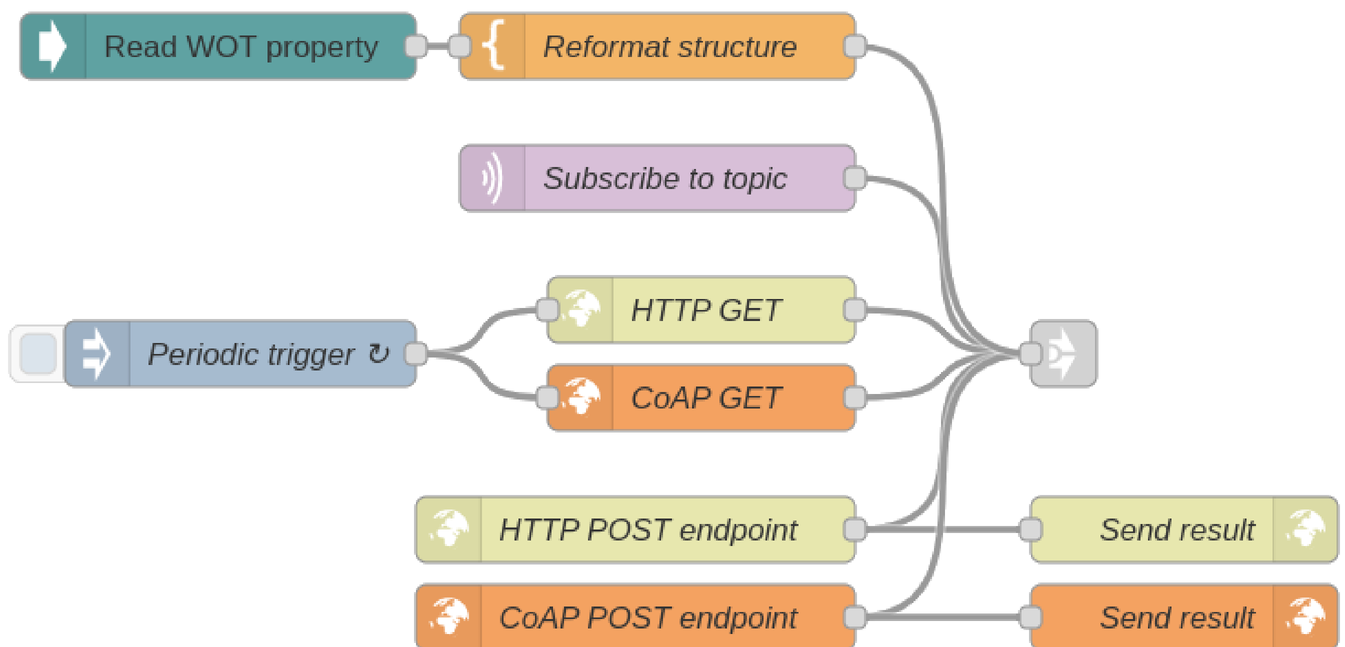


Figure 3.2: Flow input module configuration.

3.1.2.1 MQTT

Receiving data through MQTT essentially means subscribing to a topic and receiving updates. We used 2 useful built-in nodes:

- a global configuration node “mqtt-broker”;
- a standard node “mqtt in”.

The former takes care of the connection to the input[3] MQTT broker with the user configuration (e.g.: the host, the credentials etc.); the latter actually subscribes to the given topic with the give parameters (e.g. the QoS level) and emits a new Node-RED message when a new message is published to the subscribed topic. The payload of the message is the parsed JSON data.

3.1.2.2 CoAP

We decided to support input data through CoAP protocol both actively and passively. In the active way the system sends a `GET` request to the user-configured URL polling periodically[4]. In the passive way the system exposes an endpoint on a local CoAP server and handles incoming `POST` requests.

In order to implement this functionality, we used `node-red-contrib-coap` [25], a third-party node set that adds CoAP support to Node-RED. In particular we used:

- a global configuration node "coap-server";
- a standard node "coap in";
- a standard node "coap response";
- a standard node "coap request";
- a standard node "inject".

The global configuration node creates a locally running CoAP server with the user configuration (e.g. the port) while the "coap in" node sets up the `POST` endpoint with the given configuration (e.g. the path). The "coap request" node sends a `GET` request to the configured host and path when triggered by the "inject" node. Both the "coap request" and "coap in" node emit a Node-Red message with the parsed JSON data as payload but the latter must send a response to the received request. The response is handled by the "coap response" node and the status code is set to `2.04 Changed` as described by Shelby, Hartke, and Bormann [3].

3.1.2.3 HTTP

We decided to support input data through HTTP protocol both actively and passively. The implementation is completely analogous to the CoAP one described in subsubsection 3.1.2.2 and uses the following built-in nodes:

- a standard node "http in";
- a standard node "http response";
- a standard node "http request";

The HTTP server hosting the endpoint is the default one used by Node-RED, more information in chapter 5.

[3] Input and output can be done through different brokers if necessary.

[4] The node also supports the "observe" flag. More information in chapter 5.

3.1.2.4 WoT Property Read

As described in subsubsection 2.2.1.2, the system experimentally supports consuming a WoT Thing and reading a property as input. In order to implement this functionality, we used `node-red-contrib-web-of-things` [26], a third-party experimental node set that adds WoT consuming support to Node-RED. In particular we used:

- a global configuration node "consumed-thing";
- a standard node "read-property";
- a standard node "template".

The global configuration node holds the information about the consumed thing including its TD and the credentials for basic authentication[5], if enabled. The "read-property" nodes actually reads the configured property from the consumed thing[6] and emits a Node-RED message. At time of writing, the "read-property" nodes unwraps the value from the received JSON. To be consistent, the "template" node wraps it back as `{"value":...}`.

3.1.3 Processing Module

This modules handles all kinds of data processing required to implement the functionalities of the system described in chapter 2. We implemented different "pipelines", all of them receive data from the "Inbound data" channel and push the result to the "Outbound data" channel. For this reason, only one of them should be enabled at the same time.

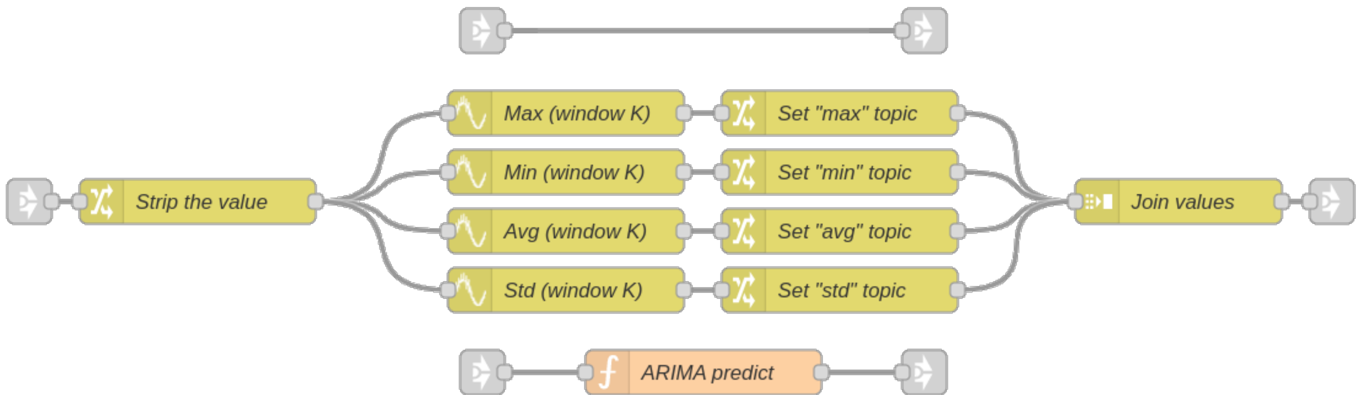


Figure 3.3: Flow processing module configuration.

[5]Only basic type of authentication is currently supported. More information in chapter 5.
[6]Not all protocols are supported, more information in chapter 5.

3.1.3.1 Passthrough Pipeline

This pipeline is the one at the top in Figure 3.3 and is essentially a "no-op". "Inbound data" channel is directly linked to "Outbound data" channel. In this way all Node-RED messages are just forwarded. This pipeline is used by the system to implement the operations described in chapter 2 that doesn't require a data processing such as:

- **data visualization**, data is printed as-is;
- **data storage**, data is saved as-is in an InfluxDB database along with additional user-defined tags;
- **data bridging**, data is bridged to a different protocol without any processing.

3.1.3.2 Aggregation Pipeline

This pipeline is shown in the middle in Figure 3.3 and is used to implement the **data aggregation** operation as described in chapter 2. In particular the system considers non-overlapping windows of size `K`, where `K` is configured by the user, and computes the following statistical features:

- maximum;
- minimum;
- average;
- standard deviation.

In order to implement this functionality, we used `node-red-node-smooth` [27], a node set that provides a node that implements several simple smoothing algorithms for incoming data values. In particular we used:

- some standard nodes "change";
- some standard nodes "smooth";
- a standard node "join".

The first "change" node is used to unwrap the value from the payload JSON as required by the following nodes. The message is then quadruplicated and passed to 4 "smooth" nodes. Those nodes are configured to automatically compute the right statistical feature over the received payload of the previous `K` messages and emit[7] a new message with the result. Before the end of the pipeline, the 4 messages should be merged back in a single one. To do that, 4 "change" nodes set the name of the statistical feature in the message property named "topic" and then the messages are passed to a single "join" node. This last node is configured to wait 4 messages and then combine their payload to create a key/value object using the value of the topic as the key and the value of the payload as the value (e.g. `{"max":..., "min":..., "avg":..., "std"...}`). Finally, the new message is pushed to the "Outbound data" channel.

[7]one message for each non-overlapping window of `K` values.

3.1.3.3 Forecasting Pipeline

This pipeline is shown at the bottom in Figure 3.3 and is used to experimentally implement the **data forecasting** operation as described in subsubsection 2.2.1.1. This pipeline should produce in output the next value of the time series computed according to the ARIMA [22] model with the statistical parameters `p`, `d` and `q` chosen by the user. In order to implement this functionality, we used a standard "function" node and a JavaScript library `arima` [28]. When a message arrives to the node, the function code shown in Listing 1 is invoked.

```
//get the list of previous data (or create a new empty)
let allData = flow.get("allData") || [];
//push the new value and save the list
allData.push(msg.payload.value);
flow.set("allData", allData);
//set the options
let options = {p: P_VALUE, d: D_VALUE, q: Q_VALUE};
//train, predict and save to payload
[[msg.payload.value], [msg.payload.pError]] =
                        new ARIMA(options).train(allData).predict(1);
return msg;
```

Listing 1: Code of the function node inside the forecasting pipeline.

This implementation is experimental and naive, more information in chapter 5. Whenever a new message arrive:

1. a list of previous data is retrieved from the flow context or created if it doesn't exist;
2. the received value is appended to the list and the context is updated;
3. the ARIMA model is trained on the whole dataset with the user-configured given options;
4. the next value is predicted and saved in the message overwriting the received value;
5. the prediction error is added to the payload;
6. the edited message is emitted.

Note that "`P_VALUE`", "`D_VALUE`" and "`Q_VALUE`" are placeholders replaced by the frontend with values provided by the user.

3.1.4 Output Module

This module handles all outputs of the system. Data is received from the previous modules through the "Outbound data" channel and is handled to be outputted according to the selected operation and given configuration. Regardless of the selected

operation, the message is passed to a "change" node configured to store the payload in the flow context before forwarding the message. This "cached" value will be necessary for those kinds of outputs that are asynchronous like the passive versions of CoAP and HTTP output.
At time of writing, the following outputs are supported:

- File;
- WebSocket;
- MQTT;
- CoAP;
- HTTP;
- WoT property write;
- InfluxDB.

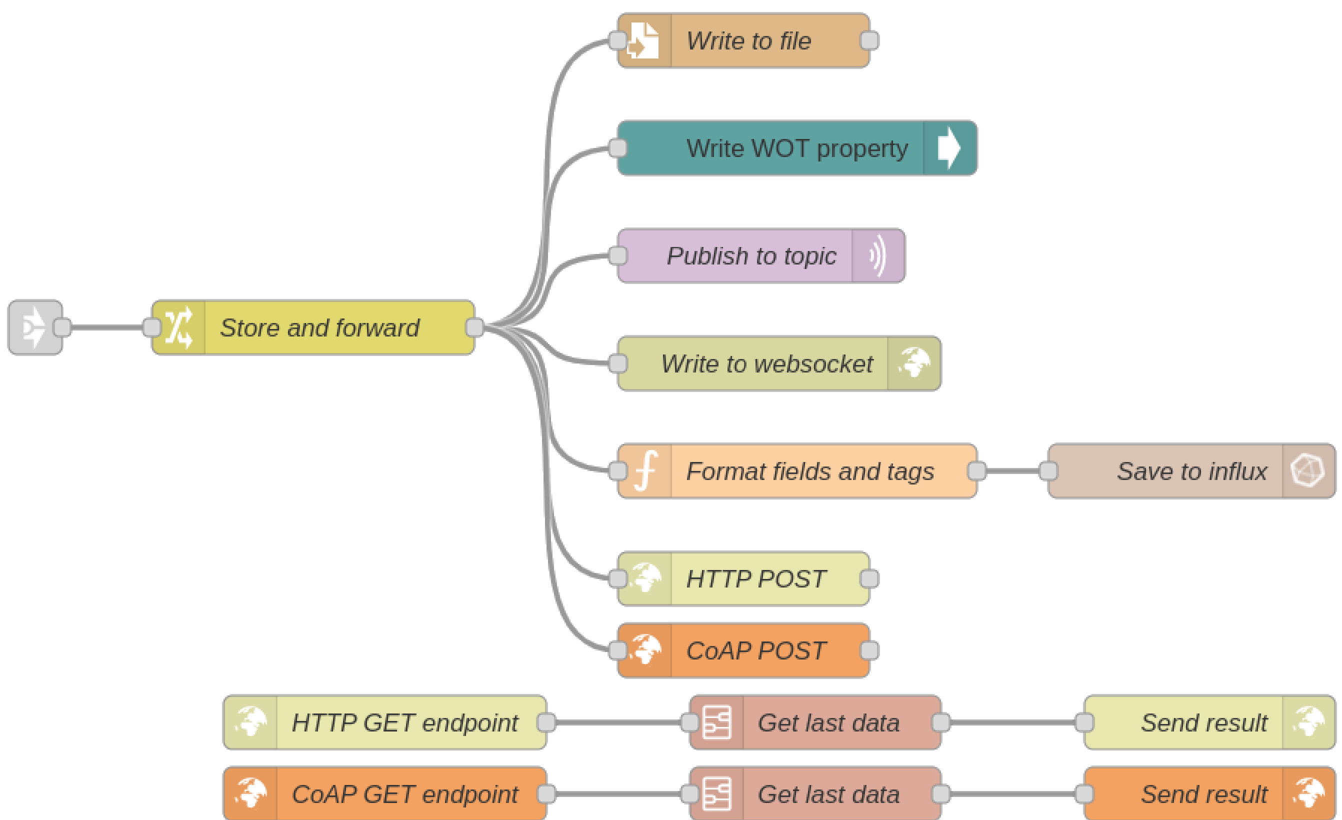


Figure 3.4: Flow output module configuration.

3.1.4.1 File

Following the philosophy that in UNIX "everything is a file descriptor or a process" (Torvalds [29]) we decided to design the backend to be able to write to a file.

This output is extremely powerful especially when combined with synthetic file systems [30]. To implement this functionality we used a built-in "write file" node in append mode. The frontend can configure the node with the path of a file that can represent: a regular file, a pipe, a socket etc. More information in section 3.2.

3.1.4.2 WebSocket

In order to improve the console output implementation, more information in section 3.2, we added the support for WebSocket [31] output using 2 useful built-in nodes:

- a global configuration node "websocket-listener";
- a standard node "websocket out".

The former creates the WebSocket listener on the configured path; the latter actually writes the payload of the received message to the WebSocket.

3.1.4.3 MQTT

Sending data through MQTT essentially means connecting to a broker and publishing the data to a topic. We used 2 useful built-in nodes:

- a global configuration node "mqtt-broker";
- a standard node "mqtt out".

The former takes care of the connection to the output MQTT broker with the user configuration (e.g.: the host, the credentials etc.); the latter actually publishes the payload of the received message to the given topic with the given parameters (e.g. the QoS level). The payload of the message is serialized as JSON.

3.1.4.4 CoAP

We decided to support output through CoAP protocol both actively and passively. In the active way the system sends a `POST` request to the user-configured URL with the data. In the passive way the system exposes a CoAP endpoint and handles incoming `GET` requests.

In order to implement this functionality, we used the same third-party node set used for the input as described in subsubsection 3.1.2.2. In particular we used:

- a global configuration node "coap-server";
- a standard node "coap in";
- a standard node "coap response";
- a standard node "coap request";
- a custom-built subflow node.

The "coap request" node executes the POST request to the configured path for the active output. The global configuration node is the same used for the input and the "coap in" node sets up the GET endpoint with the given configuration (e.g. the path) for the passive output. The incoming requests are passed to our custom subflow shown in Figure 3.5. This subflow encapsulate a single "function" node, shown in Listing 2, that retrieves the last cached data and saves it inside the message payload. If the data is not available (e.g. because we haven't received anything in input yet) a 404 status code is set inside the Node-RED message.

Figure 3.5: "Get last data" subflow implementation.

```
//get last data from parent flow context
msg.payload = flow.get("$parent.lastData");
//set 404 return return code if empty
msg.statusCode = (!msg.payload) ? "404" : msg.statusCode
return msg;
```

Listing 2: Code of the function node inside the subflow.

The response to the incoming request is handled by the "coap response" node. This node sends back the JSON encoded data and sets the response status code according to the value stored inside the received message[8].

3.1.4.5 HTTP

We decided to support output through HTTP protocol both actively and passively. The implementation is completely analogous to the CoAP one described in subsubsection 3.1.4.4 and uses the following built-in nodes:

- a standard node "http in";
- a standard node "http response";
- a standard node "http request";
- a custom-built subflow node.

The HTTP server is the same used for the input.

[8]Note that HTTP status codes are translated to the CoAP equivalent (e.g. a 404 code is sent as a 4.04 Not Found [3]).

3.1.4.6 WoT Property Write

As described in subsubsection 2.2.1.2, the system experimentally supports consuming a WoT Thing and writing a property as output. This is done using the same node set used for the input and described in subsubsection 3.1.2.4. In particular we used:

- a global configuration node “consumed-thing”;
- a standard node “write-property”;

The global configuration node holds the information about the consumed thing just like the one used for the input[9]. The “write-property” nodes actually writes the data to the configured property of the consumed thing.

3.1.4.7 InfluxDB

This output is used to implement the **data storage** operation described in chapter 2. In order to implement this functionality, we used `node-red-contrib-influxdb` [32], a third-party node set that provides nodes to save and query data from an InfluxDB time-series database. In particular we used:

- a global configuration node “influxdb”;
- a standard node “function”;
- a standard node “influx out”.

The global configuration node handles the connection to the database with the user configuration such as the connection details and the credentials while the “influx out” writes the received data to the database. As described in chapter 2, we wanted to support writing custom tags along with the data. According to the documentation if the payload of the message received by the “influx out” node “is an array containing two objects, the first object will be written as the set of named fields, the second is the set of named tags” [32]. Listing 3 shows the code of a “function” node that formats the payload as required. Note that “`TAGS_HERE`” is a placeholder and is replaced by the frontend with tags provided by the user.

```
msg.payload = [
    msg.payload,
    TAGS_HERE
]
return msg;
```

Listing 3: Code of the function node that formats InfluxDB fields and tags.

[9] Things used for input and for output can be different.

3.1.5 WoT Introduction

As described in subsubsection 2.2.1.2, the system experimentally supports behaving like a consumable W3C WoT Thing. Following the W3C Draft [19] we decided to support the introduction mechanism through Well-Known URIs [33] creating an HTTP `GET` endpoint at `/.well-known/wot` that returns the Thing Description [18] of the system. The backend implementation, shown in Figure 3.6, is straightforward and uses the following built-in nodes:

- a standard node "http in";
- a standard node "http response";
- a standard node "template";

The "http in" node sets up the endpoint and passes the request to the "template" node that simply adds the configured TD JSON to the message payload. Finally the "http response" node sends the payload of the message as response setting the `Content-Type` header as `application/ld+json`.

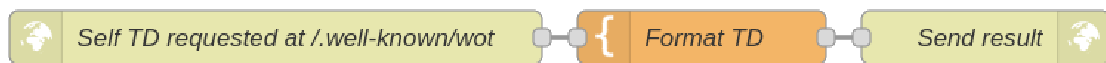


Figure 3.6: WoT Introduction flow configuration.

The TD is generated by the frontend starting from the template shown in Listing 4. A property "data" is declared and the right interaction affordance forms are added by the frontend according to the configuration of the enabled inputs and outputs. No security is enabled.
Note that this TD uses fictional CoAP and MQTT Protocol Bindings as they are not available at time of writing. The id of the thing is a fictional URI starting with the real domain of the Computer Science department.

```
{
    "@context": [
        "https://www.w3.org/2022/wot/td/v1.1",
        {
            "cov": "http://www.example.org/coap-binding#",
            "mqv": "http://www.example.org/mqtt-binding#",
            "unibo_disi": "http://disi.unibo.it/"
        }
    ],
    "id": "unibo_disi:projects/RED-Bridge",
    "title": "RED-Bridge",
    "securityDefinitions": {
        "nosec_sc": {
            "scheme": "nosec"
        }
    },
    "security": "nosec_sc",
    "properties": {
        "data": {
            "description": "RED-Bridge data",
            "type": "object",
            "properties": {
                "value": {
                    "type": "number"
                }
            },
            "forms": []
        }
    }
}
```

Listing 4: System's self TD template.

3.2 Frontend

As described in subsection 2.3.1, the frontend is a Python script that provides a CLI to configure the system. The application handles the configurations provided by the user in order to create and configure the flow before deploying it on the Node-RED instance. In addition, the application handles the console output for those operations that require it.
Schematically the application:

1. parses the command line arguments (e.g.: the selected operation, the configuration files, the parameters etc.);

2. loads the specified configuration files (e.g. the protocols configuration);

3. configures the flow as required to support the selected operations and given configurations;
4. deploys and start the flow on the backend;
5. waits for the signal `SIGINT` from the user before undeploying the flow and terminating.

3.2.1 Parameters and Configuration Handling

As shown in Figure 3.7, the frontend application accept several arguments parsed by `argparse` [23] module. Input, output and InfluxDB configuration should be provided passing the path of a JSON file as parameter. Configuration file examples are shown in Listing 5, 6 and 7.

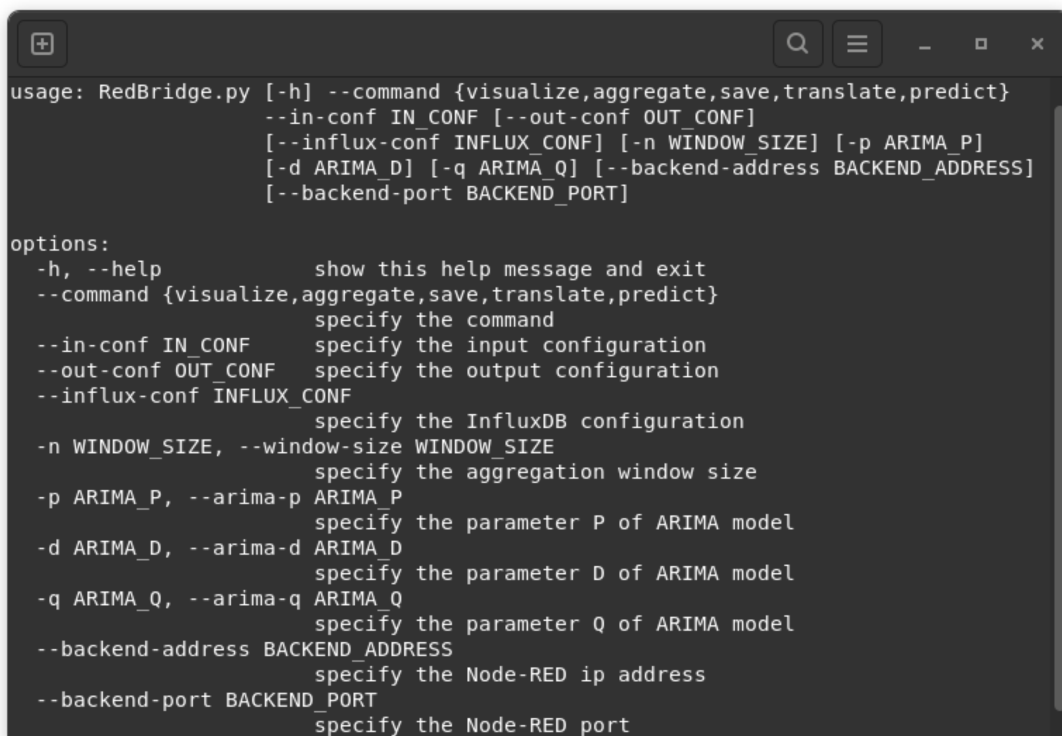


Figure 3.7: RED-Bridge usage help.

```json
{
    "protocol": "mqtt",
    "host": "test.mosquitto.org",
    "port": "1883",
    "user": "",
    "password": "",
    "topic": "ExampleTopic",
    "qos": "0"
}
```

Listing 5: Example of MQTT input JSON configuration.

```json
{
    "protocol": "http",
    "host": "example.org",
    "port": "8080",
    "path": "/data/path",
    "method": "POST"
}
```

Listing 6: Example of HTTP active output JSON configuration.

```json
{
    "host": "example.org",
    "port": "8086",
    "token": "...",
    "org": "Example Organization",
    "bucket": "Example Bucket",
    "measurement": "Example Measurement",
    "tags": {
        "Example Tag Name": "Tag Value"
    }
}
```

Listing 7: Example of InfluxDB JSON configuration.

3.2.2 Flow Manipulation and Deploy

In order to dynamically configure the flow, we disabled most nodes (e.g. the entry point for the processing pipelines and I/O nodes) then we exported the entire flow to a JSON file. The application:

1. loads the entire flow from the JSON file;

2. according to the user requirements:

- enables all required nodes;
- configure all required nodes;

3. deploys the configured flow to the Node-RED instance using its HTTP API.

For example, if the selected input is MQTT, the node for the input MQTT broker and the one for the MQTT subscription are enabled and configured with the user configurations; if the aggregation operation is selected, the nodes for the aggregation pipeline are enabled and configured with the given window size; if the selected output is passive CoAP, the global CoAP server node and the CoAP `GET` endpoint node are enabled and configured with the user configurations.

All the flow configurations for all the supported operations and inputs/outputs are done in a similar way.

3.2.3 Console Handling

Originally, the console output was implemented leveraging the file output support of the backend. At time of writing, the console output is implemented using the WebSocket output support as described in subsubsection 3.1.4.2 and the `websocket-client` [34] Python library. The frontend starts a WebSocket client on a new thread; this client is connected to the backend output and prints received messages to the console.

3.2.4 Self Thing Description Generation

If one or more of the following inputs/outputs are enabled the system can behave like a WoT Thing:

- MQTT input/output;
- CoAP passive input/output;
- HTTP passive input/output.

In this case, as described in subsection 3.1.5, the frontend should enable the WoT introduction endpoint and configure the "template" node with the self Thing Description. In order to generate the TD, the frontend loads the TD template shown in Listing 4 and adds the forms according to the enabled inputs and outputs and their configurations.

4 | Testing and Evaluation

In this chapter we will briefly describe tests and evaluations conducted on the system. The backend is designed in a modular way as described in section 3.1; for this reason, we have been able to test inputs, outputs and processing pipelines independently before testing the entire running system and measuring performances. More information in the following sections.

4.1 Manual I/O Testing

In order to manually test the various inputs and outputs, we disabled the processing module, added a "debug" node to the "Inbound data" virtual channel and an "inject" node to the "Outbound data" virtual channel; the "debug" node is used to see what is received by the input module while the "inject" nodes is used to inject some data into the output module.

4.1.1 MQTT

In order to test the MQTT input functionality, we configured the input nodes to connect to a broker and subscribe to a test topic; we then used Mosquitto [35] CLI clients to manually publish data to the test topic. Testing the output works in a similar way: we configured the output nodes to connect to a broker and publish data to a test topic; we used the clients to subscribe to the test topic and the "inject" node to push some data.

4.1.2 CoAP

In order to test the CoAP passive input/output functionality, we configured the nodes with test parameters (e.g.: the path, the port etc.). We used the CoAP CLI client provided by `libcoap` [36] to send `POST`s and `GET`s to the input and output endpoints, respectively.
In order to test the CoAP active input/output functionality, we needed to set up 2 CoAP endpoints. We created and deployed a new Node-RED flow with the CoAP endpoints; the `GET` endpoint returns hard-coded data while the `POST` one prints the received value through a "debug" node. Finally we configured the nodes of our system to use the newly created endpoints as input and output.

4.1.3 HTTP

Testing the HTTP input/output functionality is completely analogous to the one for CoAP described in the previous section. The CLI client is `curl` [37] and the endpoints in the test flow are created using built-in HTTP nodes.

4.1.4 WoT

In order to test the WoT input/output functionality, we used `node-wot` [38] to create a simple WoT Thing with a readable and writable property. The read handler returns an hard-coded value while the write handler prints the received value. We then configured our WoT input/output nodes to consume this newly created Thing. In order to test the system's ability to be consumed as a WoT Thing, we enabled and configured the WoT introduction endpoint and generated the self TD. We then created a simple script using `node-wot` that fetches the TD of the system, consumes it and tries to push or pull data.

4.1.5 File

In order to test the file output functionality, we simply configured the path of a test file in the "file" node and read that file externally.

4.1.6 WebSocket

In order to test the WebSocket output functionality, we configured the nodes with a test path. We then used the same Python package used in the frontend to connect to the WebSocket as described in subsection 3.2.3.

4.1.7 InfluxDB

In order to test the InfluxDB output functionality, we installed and configured a local InfluxDB instance. We then enabled and configured the nodes with additional tags and connection settings. As shown in Figure 4.1, we used the InfluxDB data explorer to check if fields and tags had been saved correctly.

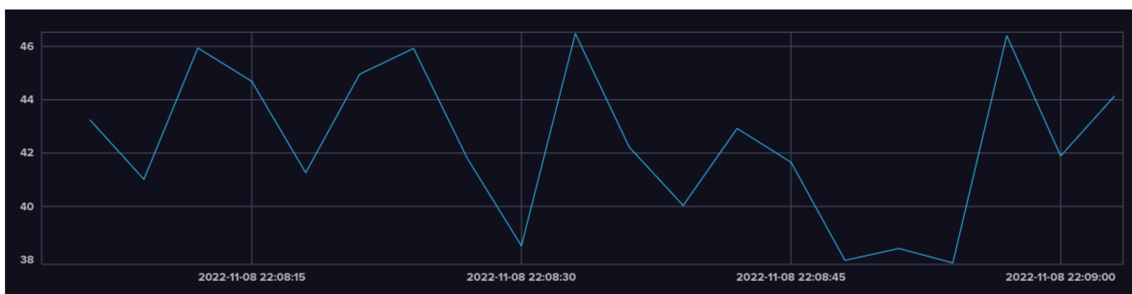


Figure 4.1: Screenshot of InfluxDB data explorer during testing.

4.2 Manual Processing Testing

In order to manually test the various processing pipelines, we disabled the other modules, added an “inject” node to the “Inbound data” virtual channel and a “debug” node to the “Outbound data” virtual channel; the “inject” nodes is used to inject some data into the pipeline while the “debug” node is used to see the result of the processing. We enabled and tested one pipeline at a time.

4.2.1 Passthrough Pipeline

Testing this pipeline is trivial.

4.2.2 Aggregation Pipeline

In order to test this pipeline, we configured the “smooth” nodes with a window size set as 5 and injected 5 different values. The “debug” node allowed us to check the aggregation results.

4.2.3 Forecasting Pipeline

In order to test this pipeline, we configured the “function” node configuring the options for the ARIMA model. We injected some values and checked the forecasting results.

4.3 Whole System Testing

When the system was completed, we started a whole system testing phase. Unlike in the manual testing phase, in this phase we tested the system using only the frontend without tampering with the backend.
In particular we tested:

- every supported operation;
- every supported input/output.

During this phase we used:

- the CLI tools used before;
- the additional flows used before;
- the WoT scripts used before;
- the MQTT broker and InfluxDB instance used before;
- a purpose-built edge device based on ESP32.

For example, Figure 4.2 shows the output of the test of the aggregation operation using MQTT as input protocol and 5 as window size.

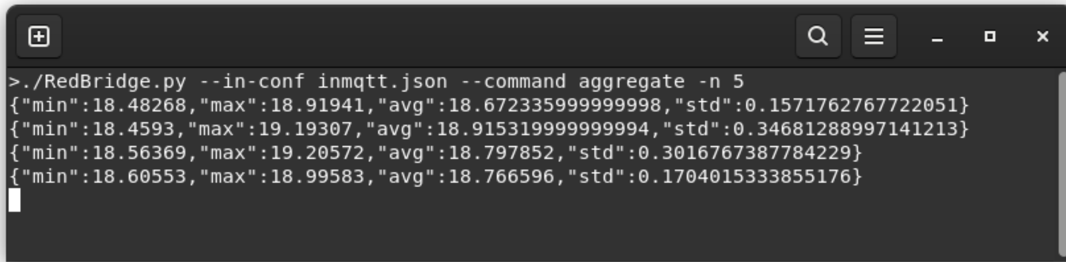


Figure 4.2: Screenshot of the test of the aggregation operation.

4.3.1 Edge Device Prototype

In order to test the system with a real edge device, we designed and built a simple prototype that works as a temperature sensor and supports MQTT, CoAP (server) and HTTP (server).

4.3.1.1 TMP36 Sensor

As temperature sensor we used a TMP36 [39], a low voltage precision centigrade temperature sensor that provides a voltage output linearly proportional to the sensed temperature, as shown in Figure 4.3.

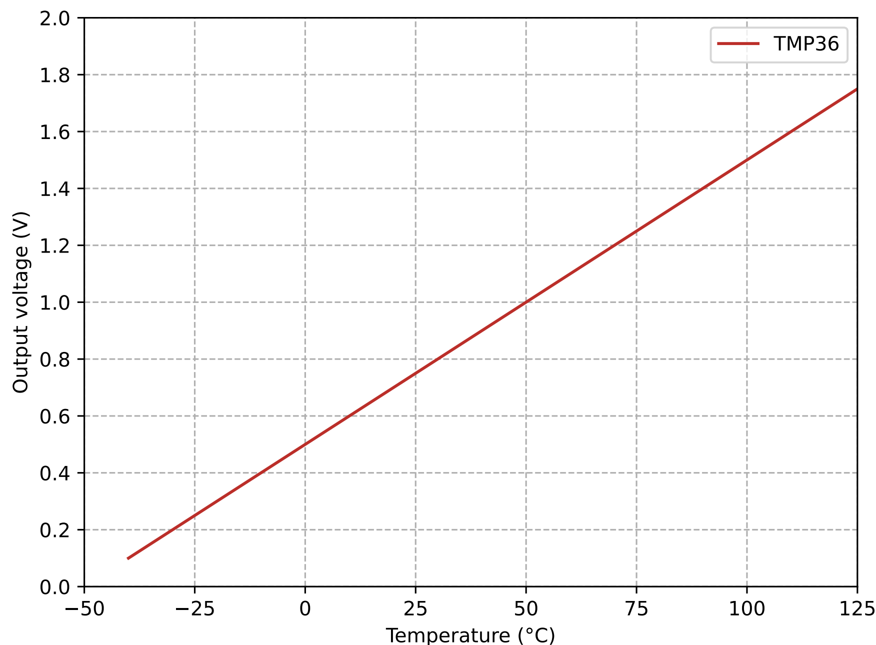


Figure 4.3: TMP36 output voltage vs. temperature.

After analyzing the datasheet, we derived the following formula to compute the temperature T given the sensor V_{out} reading:

$$T = 100 \cdot V_{out} - 50 \tag{4.1}$$

4.3.1.2 ESP32 Prototyping Board

We used an ESP32 [40] prototyping board, flashed the latest stable version of MicroPython [41] and connected the sensor through an analog pin, as shown in Figure 4.4. We initially used the REPL to do some testing then we developed a Python script and persistently uploaded it to the board. This script implements the required functionalities as described in the following sections.

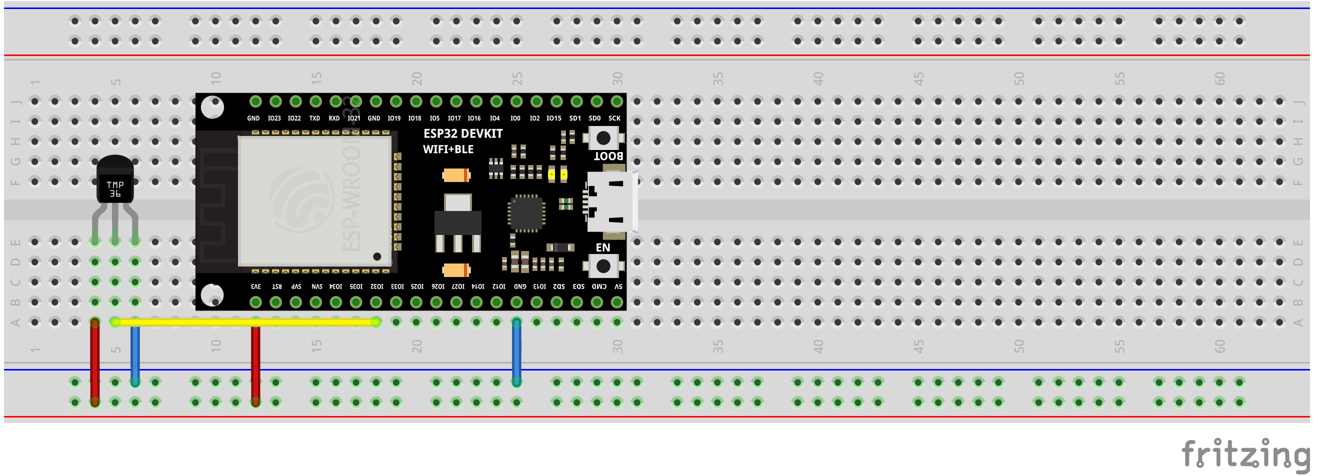


Figure 4.4: Edge device prototype mounting diagram.

4.3.1.3 Temperature Reading

In order to read the temperature, we needed to read the analog value coming from the sensor and then apply the formula shown in Equation 4.1. The ESP32 ADCs can measure analog voltages from $0V$ to V_{ref} where V_{ref} varies among different chips (the median is $1.1V$). Using `espefuse.py` [42], the official tool provided by Espressif, we were able to retrieve the calibrated value $V_{ref} = 1.114V$ from the chip. The ESP32 supports up to 12bit resolution (4096 levels) and supports attenuations to measure analog values higher than V_{ref}.
Converting raw readings to a voltage is done following Equation 4.2 where L_{read} is the raw value, V_{max} is the maximum value and L_{max} is the maximum level.

$$V_{out} = L_{read} \cdot V_{max}/L_{max} \tag{4.2}$$

Using $0dB$ attenuation and maximum resolution we have $V_{max} = V_{ref} = 1.114V$ and $L_{max} = 4095$. For this reason, we can compute the (calibrated) analog voltage as shown in Equation 4.3.

$$V_{out} = L_{read} \cdot 1.114/4095 \tag{4.3}$$

Combining Equation 4.1 and Equation 4.3, we implemented an helper function to get the temperature value from the analog pin; this function is called from another helper function that encapsulates the temperature in a dictionary. They are shown in Listing 8 and Listing 9, respectively.

```
def get_temp():
    # read analog value and compute temperature
    return 100 * (1.114 * temp_in.read() / 4095) - 50
```

Listing 8: `get_temp()` function implementation.

```
def get_sample():
    # do sample and return dictionary with "value" property
    return {"value": get_temp()}
```

Listing 9: `get_sample()` function implementation.

4.3.1.4 MQTT, CoAP and HTTP Implementation

The support for MQTT, CoAP and HTTP has been implemented using the MicroPython libraries `umqtt.simple` [43], `microcoapy` [44] and `microdot` [45], respectively.

MQTT After initializing the client and connecting to the broker, in the main loop a new sample is periodically retrieved using the `get_sample()` function, serialized it as JSON and published with a specific topic.

CoAP After initializing the client, we defined and registered a callback that handles incoming requests for a specific path. This callback gets a new sample using the `get_sample()` function, serializes it as JSON and sends the answer back with the right Content-Format [3]. We then started the client and handled incoming requests cooperatively in the main loop.

HTTP The Flask-inspired library is not designed to handle request cooperatively; for this reason, the web server is started on a new thread. After initialization, we defined and registered a callback to a specific path. This callback simply gets a new sample using `get_sample()` and returns it. The sample is automatically serialized as JSON and the right content type is automatically set.

4.4 Bridging Performance Evaluation

As part of the testing and evaluation phase, we decided to try to evaluate the performances of the protocol bridging operation of the system. In particular, we wanted to estimate the overhead introduced by the system comparing the total message transmission time using the bridging with the total message transmission time using a native protocol.

4.4.1 Evaluated Protocols

As described in the previous chapters, the system supports several input/output combinations but some of them involve polling, either by the system or the destination device. Polling time is user-configured and introduces a delay that is probably at least a couple orders of magnitude larger than the overhead we wanted to measure. For this reason, we decided to evaluate only the "push" versions of the supported protocols combinations listed below:

- CoAP[10] to MQTT;
- HTTP[10] to MQTT;
- MQTT to CoAP[11];
- HTTP[10] to CoAP[11];
- MQTT to HTTP[11];
- CoAP[10] to HTTP[11];

and compare them with native transmissions through:

- MQTT;
- CoAP[12];
- HTTP[12].

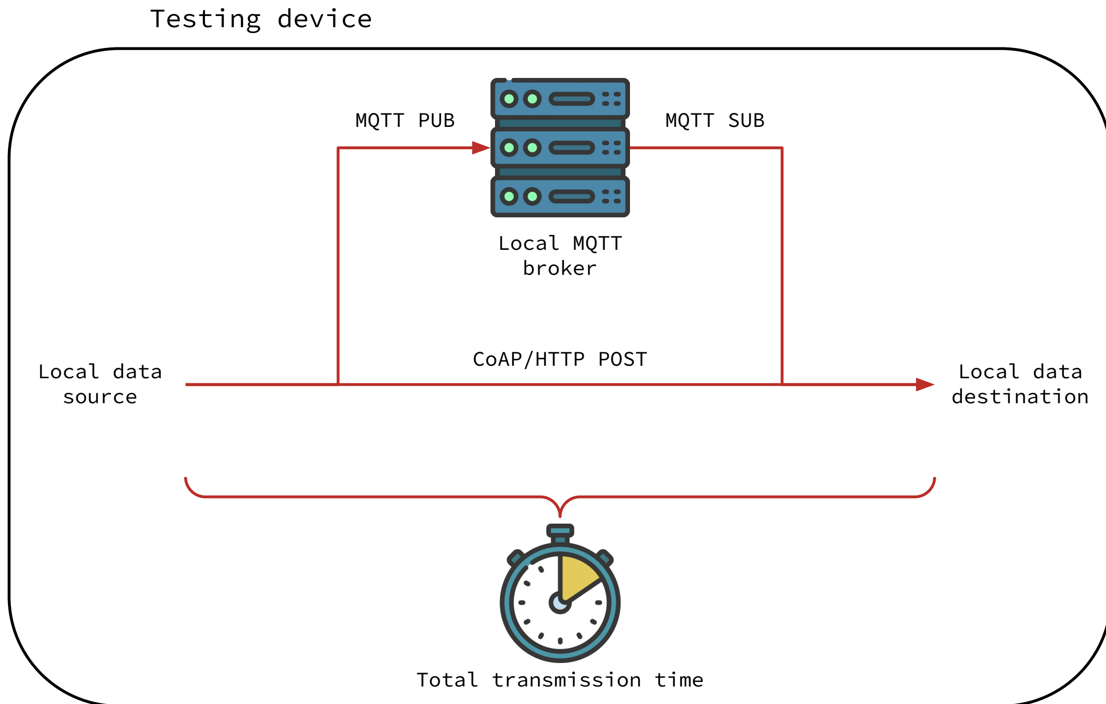
4.4.2 Evaluation Scenario

The total transmission time depends on several factors including the network conditions, the involved protocols and the system translation overhead. In order to minimize the influence of the network conditions, we decided to:

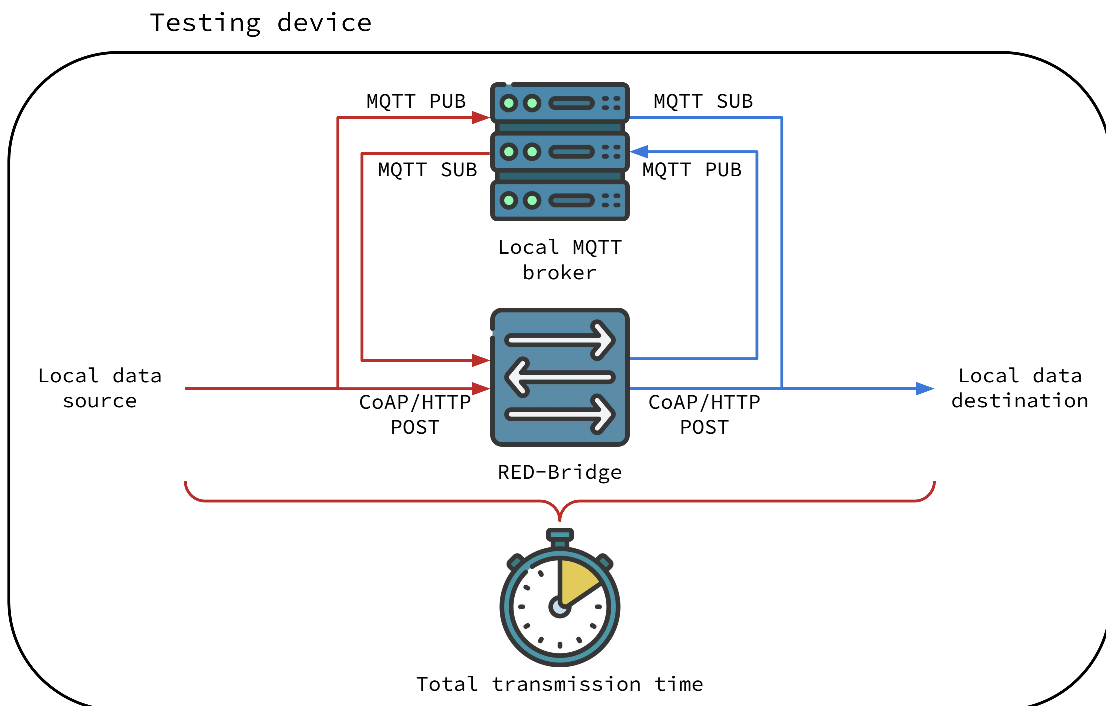
- deploy the system in a testing device;
- implement producers and consumers in the same testing device;
- deploy an MQTT broker in the same testing device.

In this way, the measured total transmission time of the native transfers, shown in Figure 4.5a, can be fairly compared with the total transmission time of those with bridging, shown in Figure 4.5b. For each of the protocols, we implemented producers and consumers as a Node-RED flow. The producers send 500 messages, 1 second apart from each other, including the timestamp with millisecond resolution; the consumers compute the time difference and save the result to a file.

[10] Receiving `POST`s as input.
[11] Sending `POST`s as output.
[12] Producers send `POST`s to consumers.



(a) Native transmission.



(b) Transmission with bridging.

Figure 4.5: Performance evaluation diagrams.

4.4.3 Results

For each of the protocols combinations listed in subsection 4.4.1, we collected 500 transmission times for a total of 4500 measurements. We saved the raw results in a CSV file and created a Python script to analyze them.
As shown in Table 4.1, we first analyzed them computing:

- minimum;
- maximum;
- average;
- median;
- standard deviation.

Then we plotted and analyzed the Box Plots [46], as shown in Figure 4.6, and the eCDFs [47], as shown in Figure 4.7.

	MQTT native	CoAP native	HTTP native	CoAP to MQTT	HTTP to MQTT	MQTT to CoAP	HTTP to CoAP	MQTT to HTTP	CoAP to HTTP
Min	0	0	3	1	4	1	4	3	3
Max	6	8	9	6	14	4	15	8	11
Avg	1,11	1,04	3,82	2,15	6,15	1,63	5,63	4,22	4,49
Median	1,00	1,00	4,00	2,00	6,00	2,00	5,00	4,00	4,00
St. dev.	0,43	0,56	0,88	0,61	1,17	0,60	1,27	0,76	0,82

Table 4.1: Performance evaluation results table.

As expected, the native versions of CoAP and MQTT are extremely fast on average. The former uses long-lived TCP [17] connections and the latter uses UDP [15]. The native version of HTTP native is almost 4 time slower, as expected, because the TCP connections are short-lived.
We expected the measurements for combinations with bridging to be, on average, at least the sum of the following factors:

- the transmission time of the native version of the input protocol;
- the overhead introduced by our system;
- the transmission time of the native version of the output protocol.

Analyzing all the results we concluded that this seems to be the case and, in particular, that the overhead of the system seems to be extremely low. For example, on average, bridging HTTP to MQTT seems to be only about 1ms slower than the sum of the native protocols.

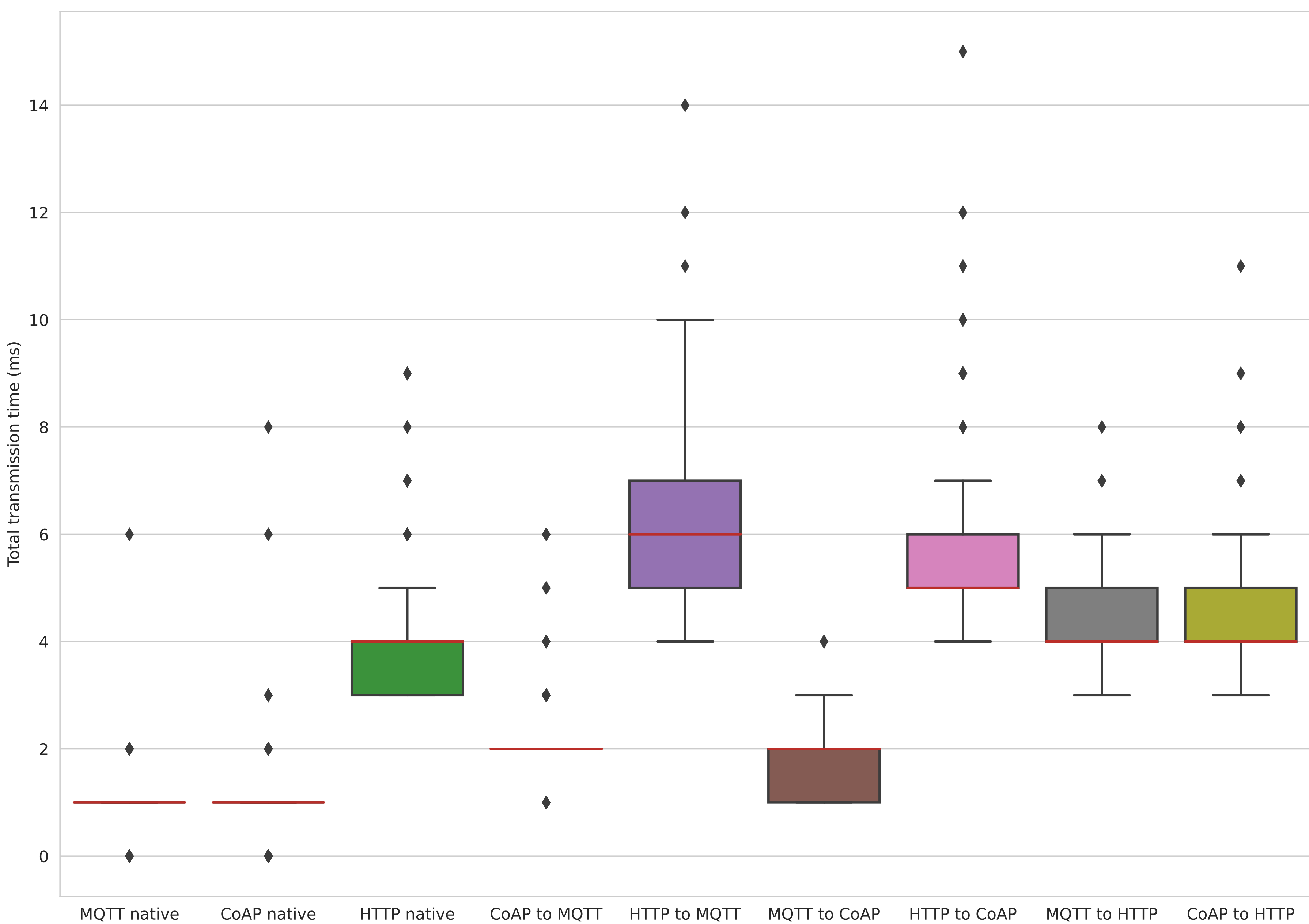


Figure 4.6: Box plots of the performance evaluation results.

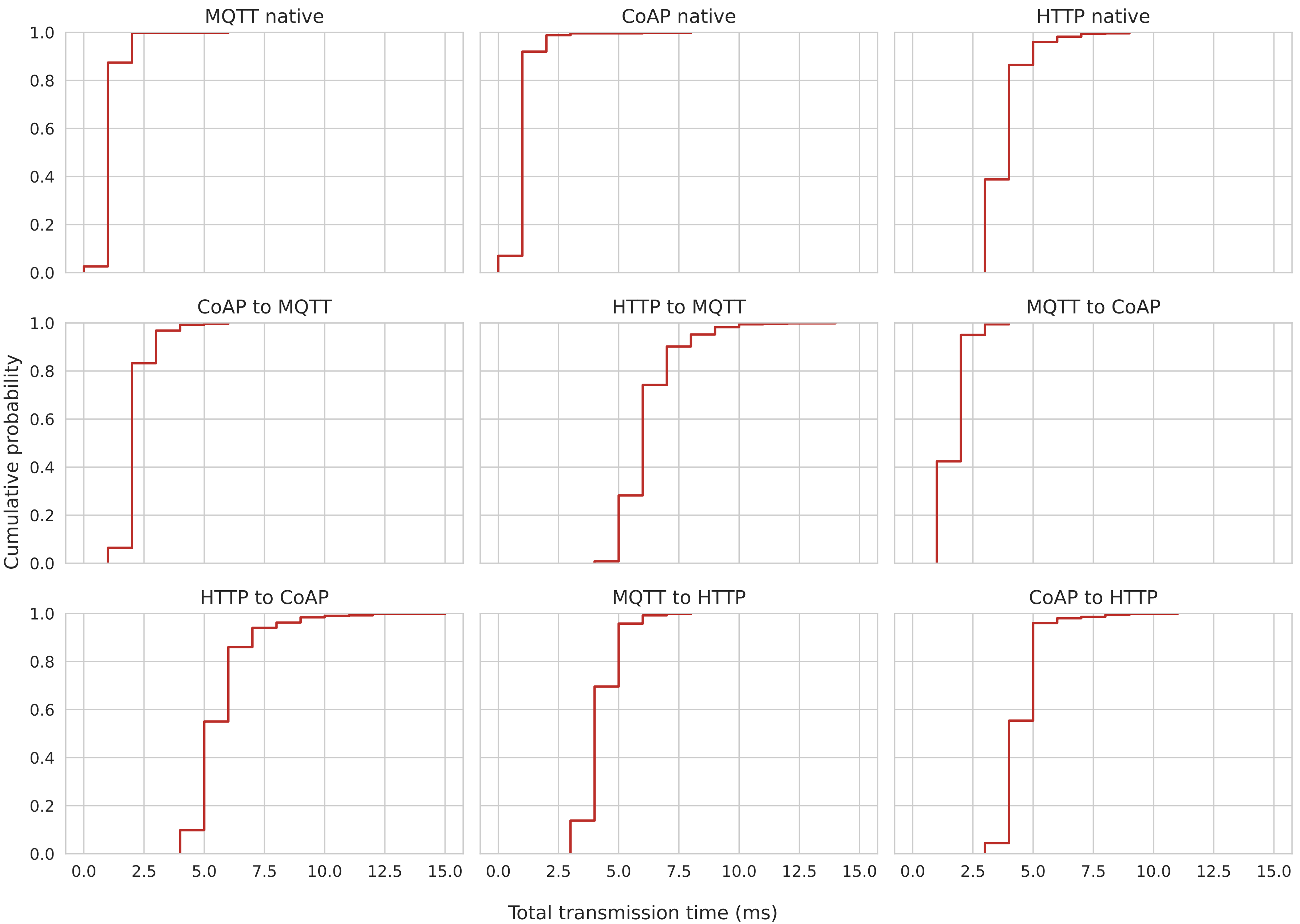


Figure 4.7: eCDF plots of the performance evaluation results.

5 | Issues and Improvements

This work is just a prototype and has several limitations and issues that could be improved. Some of them are described in the following sections.

5.1 Limiting Frontend

As described in section 3.1, the backend flow has been designed to be modular and as much general as possible. For this reason, technically the backend supports receiving inputs from different sources at the same time, applying a processing pipeline and then sending the result to more than one output simultaneously. For example, the processing result can be simultaneously printed by the frontend, saved to an InfluxDB database and outputted in all supported protocols. Even though the backend supports these simultaneous operations, at time of writing the frontend limits the user and configures the backend to enable only one input and output for the purpose of supporting only the basic operations described in chapter 2. The backend supports even combined operations such as doing data aggregation (aggregation operation) and saving them to InfluxDB (storage operation) but the frontend prevents these configurations.

5.2 I/O Format Assumption

The system currently assumes that the input/output data format is a JSON object with a property named "`value`". The system could be improved relaxing this requirement in order to support arbitrary formats.

5.3 HTTP Server Port

At time of writing, every HTTP endpoint is served by the same server hosting the Node-RED editor. For this reason, the endpoints share the same port and cannot be customized. The system could be improved to support HTTP endpoints running on their own HTTP server instance.

5.4 WoT: TD, Protocols and Authentication

At time of writing, the node set used to implement WoT property read/write [26] is still experimental and supports only basic authentication. Furthermore, the frontend

currently enables only MQTT, HTTP and CoAP protocol support. Finally, the self TD generation and the whole system's WoT support is experimental and could be improved.

5.5 CoAP Input Observe

The CoAP request node used for the input, as described in subsubsection 3.1.2.2, supports observing resources [16]. In this case the periodic trigger should be disabled in order to prevent multiple requests. This case is not currently handled by the frontend and could be improved.

5.6 Naive Forecasting Implementation

The current implementation of the forecasting operation, described in subsubsection 3.1.3.3, is extremely naive and is just an example of how a forecasting pipeline can be added to the system. The current implementation stores the entire dataset in the flow context, there is no limit on the size of the dataset and the model is retrained every time before forecasting the next value. The implementation should be rewritten in a more robust and efficient way (e.g. using a more robust storage).

5.7 Single Flow Limit

At time of writing, Node-RED doesn't provide an API for managing individual subflows; in order to manage subflows, we should get the global flow configuration, edit it and then save it back but this is not currently implemented in the system. The current implementation of the frontend assumes that the Node-RED instance is not running any other flow; it deploys the flow using the "flows" API which pushes the entire configuration overwriting any other configured flow. Consequently, the system backend cannot be run if the Node-RED instance is running other flows.

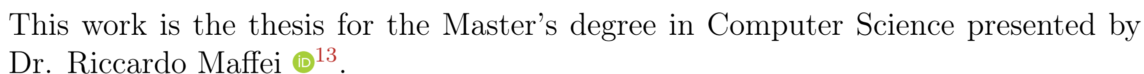
6 | Conclusions and Future Works

The goal of this work was to present the design, implementation and testing of a tool supporting some common tasks when handling IoT sensors communicating with different protocols like HTTP, CoAP and MQTT or sensors supporting the W3C WoT.

At the beginning of this work we briefly presented the technologies used in this system along with the basics of the supported protocols. We then listed the requirements and the supported functionalities of the system including: the data visualization, aggregation, storage, forecasting and the protocol bridging. We showed an overview of the system architecture, in particular we described the backend, which is a Node-RED instance running a custom-built flow, and the frontend, which is a Python script that provides a CLI to let the user configure and interact with the system. We dedicated some space to the implementation details of the system; in particular, we presented a completely modular flow composed of three independent modules: input, processing and output modules. The entire flow is implemented using both built-in and third-party nodes in order to support all the required functionalities. The implementation of the frontend is simpler: it parses the commands and the configurations provided by the user, generates and deploys a custom flow to the backend then handles the console output, if necessary. We tested the system several times, both during and after the implementation. We first tested the system with the help of command line tools, ad-hoc flows and simulated devices; then, we designed and built an edge device prototype, based on ESP32, and used it as input for the tests. Finally, we wanted to evaluate the protocol bridging performances estimating the overhead introduced by the system comparing the total message transmission time using the bridging with the total message transmission time using a native protocol. The analyzed results confirmed that the overhead introduced is minimum, as we expected.

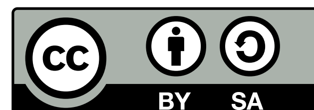
The system presented in this work successfully passed all the tests and evaluations but is just a prototype and is not meant to be used in production environments. In the future, with more time and resources, this work could be further improved trying to mitigate the issues described in the previous chapter, rebuilding the system in a more robust and flexible way and expanding the support to other protocols.

A | Licenses and Credits



This work is the thesis for the Master's degree in Computer Science presented by Dr. Riccardo Maffei [13].

Except otherwise noted, this work is licensed under a Creative Commons "Attribution-ShareAlike 4.0 International" license.



Except otherwise noted, the code is licensed under the GNU Affero General Public License either version 3 or any later version.

A.1 Third-Party Content

In this work we used third-party content excluded from the licenses mentioned above. We also used third-party tools that we want to acknowledge.
In particular:

- most of the icons used in figures are made by juicy-fish, Becris and Freepik from `flaticon.com` and are used in compliance with the Flaticon license;
- the Python logo in Figure 2.1 is property of the Python Software Foundation and can be used under nominative fair use;
- the Node-RED logo in Figure 2.1 is property of the OpenJS Foundation and can be used under nominative fair use;
- Figure 1.8 and Figure 1.9 are copied from the W3C Recommendation [8] and are copyrighted by the W3C;
- the diagram in Figure 4.4 has been created with Fritzing [48] and uses the ESP32 part made by Fedorov [49];
- some of the plots have been created using `matplotlib` [50] and `seaborn` [51];

[13] `https://orcid.org/0000-0002-6392-9701`

Acknowledgments

I want to thank my thesis supervisor, Professor Di Felice, and my co-supervisor, Dr. Luca Sciullo, for their support during the development of this thesis. I also want to thank all the Professors who have guided me and helped me during my course of studies.

Thanks to my friends, classmates and all my teammates with whom I created great projects and wonderful memories. Thanks to Beatrice, Dorotea, Federico, Fox, Nicholas, Oleksandr, Ossama, Riccardo, Samuele and Teresa.

Thanks to that part of my family that supported me in these tough years.

Finally, a thanks goes to those who deserve one but may not like to be directly mentioned here.

References

[1] Jim Morrish and Matt Arnott. *Global IoT Forecast Report, 2021-2030*. July 25, 2022. URL: https://transformainsights.com/research/reports/global-iot-forecast-report-2030.

[2] G. Sushanth and S. Sujatha. "IOT Based Smart Agriculture System". In: *2018 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*. 2018, pp. 1–4. DOI: 10.1109/WiSPNET.2018.8538702.

[3] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014. DOI: 10.17487/RFC7252. URL: https://www.rfc-editor.org/info/rfc7252.

[4] Andrew Banks et al. *MQTT Version 5.0*. OASIS Standard. Mar. 7, 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

[5] *Eclipse Ponte GitHub repository*. URL: https://github.com/eclipse/ponte.

[6] Ahmed E. Khaled and Sumi Helal. "Interoperable communication framework for bridging RESTful and topic-based communication in IoT". In: *Future Generation Computer Systems* 92 (2019), pp. 628–643. ISSN: 0167-739X. DOI: 10.1016/j.future.2017.12.042. URL: https://www.sciencedirect.com/science/article/pii/S0167739X17317387.

[7] Varun M Tayur and R Suchithra. "Review of interoperability approaches in application layer of Internet of Things". In: *2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*. 2017, pp. 322–326. DOI: 10.1109/ICIMIA.2017.7975628.

[8] Kunihiko Toumura et al. *Web of Things (WoT) Architecture*. W3C Recommendation. W3C, Apr. 2020. URL: https://www.w3.org/TR/2020/REC-wot-architecture-20200409/.

[9] *W3C Web of Things website*. URL: https://www.w3.org/WoT/.

[10] *InfluxDB OSS 2.0 Documentation*. URL: https://docs.influxdata.com/influxdb/v2.0/.

[11] Wikipedia contributors. *Flow-based programming — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Flow-based_programming&oldid=1120112618. [Online; accessed 17-November-2022]. 2022.

[12] OpenJS Foundation and Contributors. *Node-RED*. URL: https://nodered.org.

[13] Chris Newman and Graham Klyne. *Date and Time on the Internet: Timestamps*. RFC 3339. July 2002. DOI: 10.17487/RFC3339. URL: https://www.rfc-editor.org/info/rfc3339.

[14] Wikipedia contributors. *Representational state transfer — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=1122785111. [Online; accessed 20-November-2022]. 2022.

[15] J. Postel. *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/info/rfc768.

[16] Klaus Hartke. *Observing Resources in the Constrained Application Protocol (CoAP)*. RFC 7641. Sept. 2015. DOI: 10.17487/RFC7641. URL: https://www.rfc-editor.org/info/rfc7641.

[17] Wesley Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. Aug. 2022. DOI: 10.17487/RFC9293. URL: https://www.rfc-editor.org/info/rfc9293.

[18] Sebastian Käbisch et al. *Web of Things (WoT) Thing Description*. W3C Recommendation. W3C, Apr. 2020. URL: https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/.

[19] Michael McCool et al. *Web of Things (WoT) Discovery*. W3C Working Draft. W3C, Aug. 2022. URL: https://www.w3.org/TR/2022/WD-wot-discovery-20220810/.

[20] Pierre-Antoine Champin, Gregg Kellogg, and Dave Longley. *JSON-LD 1.1*. W3C Recommendation. W3C, July 2020. URL: https://www.w3.org/TR/2020/REC-json-ld11-20200716/.

[21] Felipe Pezoa et al. "Foundations of JSON schema". In: *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2016, pp. 263–273.

[22] Dimitros Asteriou and Stephen G Hall. "ARIMA models and the Box–Jenkins methodology". In: *Applied Econometrics* 2.2 (2011), pp. 265–286.

[23] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[24] David Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, Inc., 2006.

[25] *node-red-contrib-coap GitHub repository*. URL: https://github.com/JKRhb/node-red-contrib-coap.

[26] *node-red-contrib-wot GitHub repository*. URL: https://github.com/thingweb/node-red-contrib-web-of-things.

[27] *node-red-node-smooth GitHub repository*. URL: https://github.com/node-red/node-red-nodes/tree/master/function/smooth.

[28] *arima GitHub repository*. URL: https://github.com/zemlyansky/arima.

[29] Linus Torvalds. *Re: [PATCH] Futex Asynchronous Interface*. Email. June 9, 2002. URL: https://groups.google.com/g/fa.linux.kernel/c/nqB38TbjVug/m/YvfBUvRrfiwJ.

[30] Wikipedia contributors. *Synthetic file system — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Synthetic_file_system&oldid=1091200575. [Online; accessed 7-November-2022]. 2022.

[31] Alexey Melnikov and Ian Fette. *The WebSocket Protocol*. RFC 6455. Dec. 2011. DOI: 10.17487/RFC6455. URL: https://www.rfc-editor.org/info/rfc6455.

[32] *node-red-contrib-influxdb GitHub repository*. URL: https://flows.nodered.org/node/node-red-contrib-influxdb.

[33] Mark Nottingham. *Well-Known Uniform Resource Identifiers (URIs)*. RFC 8615. May 2019. DOI: 10.17487/RFC8615. URL: https://www.rfc-editor.org/info/rfc8615.

[34] *websocket-client on PyPI*. URL: https://pypi.org/project/websocket-client/.

[35] Roger A. Light. "Mosquitto: server and client implementation of the MQTT protocol". In: *Journal of Open Source Software* 2.13 (2017), p. 265. DOI: 10.21105/joss.00265.

[36] *libcoap GitHub repository*. URL: https://github.com/obgm/libcoap.

[37] *curl website*. URL: https://curl.se.

[38] *node-wot GitHub repository*. URL: https://github.com/eclipse/thingweb.node-wot/.

[39] *TMP35/TMP36/TMP37: Low Voltage Temperature Sensors Data Sheet*. Rev. H. Analog Devices, Inc. Oct. 2002. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/TMP35_36_37.pdf.

[40] *ESP32-WROOM-32E Datasheet*. Espressif Systems (Shanghai) Co., Ltd. URL: https://espressif.com/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf.

[41] Donald Norris. *Python for Microcontrollers: Getting Started with MicroPython*. McGraw-Hill Education, 2016. ISBN: 9781259644542.

[42] *espefuse.py documentation*. URL: https://docs.espressif.com/projects/esptool/en/latest/esp32/espefuse/index.html.

[43] *umqtt.simple GitHub repository*. URL: https://github.com/micropython/micropython-lib/tree/master/micropython/umqtt.simple.

[44] *microCoAPy GitHub repository*. URL: https://github.com/insighio/microCoAPy.

[45] *microdot GitHub repository*. URL: https://github.com/miguelgrinberg/microdot.

[46] Wikipedia contributors. *Box plot — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Box_plot&oldid=1118420858. [Online; accessed 15-November-2022]. 2022.

[47] Wikipedia contributors. *Empirical distribution function — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Empirical_distribution_function&oldid=1108690264. [Online; accessed 15-November-2022]. 2022.

[48] André Knörig, Reto Wettach, and Jonathan Cohen. "Fritzing: A Tool for Advancing Electronic Prototyping for Designers". In: *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*. TEI '09. Cambridge, United Kingdom: Association for Computing Machinery, 2009, pp. 351–358. ISBN: 9781605584935. DOI: 10.1145/1517664.1517735.

[49] Andrey Fedorov. *Fritzing-parts GitHub repository*. URL: https://github.com/Warlib1975/Fritzing-parts.

[50] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[51] Michael L. Waskom. "seaborn: statistical data visualization". In: *Journal of Open Source Software* 6.60 (2021), p. 3021. DOI: 10.21105/joss.03021.