

# Lightweight Extensible Frameworks for IoT Applications

(Discussion Paper)

Giansalvatore Mecca<sup>1</sup>, Michele Santomauro<sup>1</sup>, Donatello Santoro<sup>1</sup> and Enzo Veltri<sup>1</sup>

<sup>1</sup>Università degli Studi della Basilicata (UNIBAS), Potenza, Italy

## Abstract

The current trend of automation and data exchange in manufacturing technologies is often called Industry 4.0 or the Fourth Industrial Revolution. This new era of industrialization is characterized by integrating cyber-physical systems, the Internet of Things (IoT), artificial intelligence (AI), and big data analytics to gain a competitive advantage. In this aspect, the *data* generated by IoT devices and other sensors has a huge impact on businesses. These insights can be used to optimize production processes, reduce downtime, and improve product quality. Furthermore, data analytics can help companies to identify new market opportunities, develop new products, and improve customer experiences. Nevertheless, even with the extensive accessibility of tools and technology, creating intelligent applications within the industrial framework still poses a challenging and costly undertaking. This paper proposes a lightweight framework that can facilitate the adoption of IoT and IIoT solutions in industry and domotics. The framework is designed to be extensible, scalable and declarative, thus allowing for a wide range of configurations with minimal user effort. We successfully adopted the system in real-life use cases to prove its applicability. We consider this a significant contribution because it paves the way for more widespread adoption of IIoT-enabling technologies in the industry.

## Keywords

embedded-system control, Internet of Things, open-source software, smart manufacturing, Big Data

## 1. Introduction

Industry 4.0 brings *smart factories* at the center of the technology spectrum [1, 2]. Usually, a smart factory integrates different systems to enable machine-machine and human-machine cooperation. A key enabling technology in this framework is the so-called *Internet of Things* (IoT) or, even better, its industrial counterpart, called *Industry Internet of things* (IIoT) [3, 4]. IoT and IIoT share two fundamental aspects: *i*) on the one side, they share the common feature of potentially generating *big data*, i.e., very large quantities of data that need to be collected, processed and analyzed often in near real-time, thus imposing strict requirements in terms of timing, frequency of operations and throughput. In fact, these architectures are considered paradigmatic sources of Big Data [5]. Therefore, it is crucial that frameworks conceived for these tasks are able to scale to such large volumes of data. *ii*) On the other side, we notice that

---


SEBD 2023: 31st Symposium on Advanced Database Systems, July 02–05, 2023, Galzignano Terme, Padua, Italy

✉ giansalvatore.mecca@unibas.it (G. Mecca); michele.santomauro@unibas.it (M. Santomauro); donatello.santoro@unibas.it (D. Santoro); enzo.veltri@unibas.it (E. Veltri)

ORCID 0000-0002-1189-1481 (G. Mecca); 0000-0002-5651-8584 (D. Santoro); 0000-0001-9947-8909 (E. Veltri)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

in both the larger context of IoT applications and in the more specific one of industrial IoT there is a strong need for generic tools that allow for quick integration of existing machinery.

This is true in domotics, where appliances are often not IoT-enabled and, therefore, require tools that can bridge the gap toward the goal of integration and remote control. However, it is especially true in industry. In fact, industrial machinery is not always equipped with sensors and/or actuators. Even when some sensors/actuators are available, they may not exhaust the needs of all possible IIoT scenarios. Sensors tend to be expensive and often difficult to configure when available. Finally, industrial sensors are usually not cloud-enabled and fail to meet the Big Data requirements discussed above.

Consequently, IIoT applications tend to be complex monolithic projects with high investments and increased design time. To facilitate the adoption of these technologies, we introduce IoT Helper [6], an easy-to-go framework that allows for quick prototyping in terms of IoT/IIoT-enabled applications. IoT Helper enables monitoring and controlling embedded devices. The framework is based on a generic architecture that can be used with many classes of sensors and actuators. Thus, the framework can be effectively used to facilitate the development of intelligent applications in domotics and industry. It requires minimum configuration and virtually no application logic to remotely access the data and control devices. So, application developers can focus on developing higher-level applications that use data collected by IoT Helper to gain insights. IoT Helper is cloud-enabled by default, using the publish-and-subscribe protocol for decoupling the production of data from its consumption and may leverage public-cloud platforms to scale to very large volumes of data. At the same time, coherently with its agile inspiration, it also allows for on-premise deployments that can be preferred in some scenarios due to data protection and privacy concerns.

In the paper we present two concrete application scenarios: The first one is a typical domotics application for controlling the fan of a fireplace extractor chimney. The second one is a complex industrial application with respect to monitoring welding pliers of a robot arm.

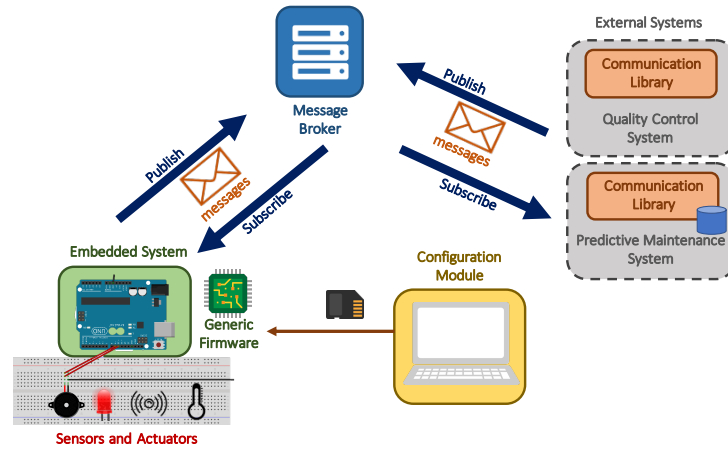
## 2. System Architecture

This section introduces the IoT Helper architecture for monitoring and controlling embedded devices. The flexibility of the approach allows for a very straightforward integration of such devices into *cloud-computing* architectures easily, but it can also be used in *fog-computing* or even *edge-computing* solutions.

Figure 1 shows the core of our proposed architecture. It is composed of the following: *i*) An *Embedded System* [7] that acts as the main controller of sensors and actuators deployed in the solution; *ii*) a *Configuration Module* that allows users to configure and customize the various sensors and actuators; *iii*) a *Generic Firmware* to manage operations, also called *commands* on connected devices; and *iv*) a *Message Broker* module that decouples the embedded system from external processors in order to scale up to large volume of data.

At the core of the **Embedded System** is the Arduino module [8, 9, 10]. Sensors or actuators can be plugged into its shield. Moreover, it is possible to re-engineer the shield to customize the hardware to the specific scenario with industrial components.

To achieve high extensibility and improve usability, the Embedded System needs to be easily



**Figure 1:** System architecture. Data collection.

configurable. We model sensors as generic inputs and actuators as generic outputs. The list of the sensors and actuators defines the *interface* of the embedded system, which can be observed as a *black box*. An external system that wants to communicate with the Embedded System needs to know only its interface without knowing the actual implementation, which might change over time. This approach allows for the creation of loosely coupled systems.

For example, suppose that we want to collect data from an analog temperature sensor and to control the electrical relay of an air conditioner. The interface of the Embedded System is an analog input `temp`, and a digital output `ac`. An external system, such as a mobile application that wants to read the sensor and control the actuator, will send a generic message "`read input temp`" or "`change the output of ac to 1`" without knowing any details about the physical connections on the embedded system or the actual circuitry of the sensors. As a consequence, we can change how the embedded system physically controls air conditioning from an electrical relay to an infrared actuator, keeping the interface unchanged without changing the user application.

The **Configuration Module** describes the inputs and the outputs managed by the embedded system. Each input is represented by the following: (i) the type, i.e., if it is analog or digital; (ii) a unique ID used in the firmware for controlling it; (iii) a description that is useful for documentation and human interpretation; and (iv) the pin(s) where it is connected to the embedded system. Each output is represented in the same manner as the input; however, in addition, since it represents an actuator, we also store information about the allowed values.

Table 1 contains a sample configuration. In this example, we have one temperature sensor with the name "`I1`". In addition, we have two actuators: a digital LED `O1` that can be switched on and off (values 0 or 1); and an analog fan controller `O2` that accepts values from 0 (off) to 6 (maximum speed).

The Configuration Module also allows for specifying network parameters in order to connect the Embedded System, typically to a WiFi network. All of the configurations are stored on file on an external SD card.

The basic operations executed on the inputs and output are as follows: (a) read an input value

**Table 1**

Configuration example.  $I1$  is a digital sensor,  $O1$  is a digital actuator and  $O2$  is an analog actuator.

Type	Typology	Name	Description	Pin	Values
Input	Digital	I1	Temperature Sensor	1	
Output	Digital	O1	Initialization Complete LED	10	{0,1}
Output	Analog	O2	Fan Speed	12	{0–6}

as the state of a sensor or actuator and (b) change the value of an output, i.e., execute an action on an actuator. **Generic Firmware** executes such operations with the following commands:

- **Read**—*read values of inputs  $I_0 \dots I_n$ .* For example, read temperature and humidity values from the respective sensors.
- **RepeatRead**—*read of inputs  $I_0 \dots I_n$  each  $n$  milliseconds.* This is useful when we need continuous read operations with a fixed frequency. This command accepts a parameter  $t$  representing time in milliseconds. This command is used to regularly collect data from the Embedded System and allows one to minimize the number of requests sent to the system.
- **Write**—*change values of output  $O_0$  to  $v_0, \dots O_n$  to  $v_n$ .* For example, switch the LED on and set the speed of the fan to four.
- **TimedWrite**—*change values of outputs after a fixed delay.* For example, switch off the fan in two hours.

When the system starts, the Generic Firmware runs some initialization operations: (1) it reads the configuration file, (2) it configures input and output pins, (3) it starts the WiFi connection, (4) it initializes the connection with the Message Broker and (5) it starts a timer for timed commands.

After initialization, Generic Firmware waits for commands and executes them. Each command is received as a request message and generates a response, as discussed in Section 2.1.

The final model we discuss is the **Message Broker**. It handles the exchange of messages among client apps and the Embedded System, according to the formats described in Section 2.1.

We adopt a Publish and Subscribe (P&S) communication model [11]. The **Message Broker** stands at the core of this protocol by decoupling the involved parties, i.e., the message sender (publisher) from the message receiver (subscriber). The publisher and the subscriber, therefore, do not need to establish a direct point-to-point connection. We can either have multiple publishers that publish messages to one subscriber or multiple subscribers that receive messages from one publisher at the same time. The broker is responsible for message routing and distribution.

The main benefits of the P&S approach are that the embedded systems do not know any other external application but communicates only with the message broker. Adding new subscribers will not need to modify the publisher's behavior when they join the architecture. In addition, publishers and subscribers do not need to be online and ready simultaneously. Still, the embedded system can publish new data as soon as they are ready without waiting for the clients. This allows achieving better performance in real-time applications. Finally, the only component that needs to be reachable is the Message Broker. This can be easily obtained by installing the module on a public server or by using a cloud solution.

In our architecture, we used two channels: (i) the *data channel* that manages read operations, i.e., this channel is used to publish sensors data from the embedded system to the broker; and (ii) the *command channel* that stores commands that come from external systems or devices to the embedded system. In essence, from the point of view of the embedded system, it registers itself as a subscriber to the data command channel because it needs to receive messages, process them using Generic Firmware, and execute the corresponding operations. Generic Firmware also registers itself as a publisher to the data channel because it sends out input values. The Message Broker registers itself as a publisher in the command channel because it dispatches messages to the embedded system and registers itself as a subscriber to the data channel in order to receive data from the embedded system. The complete architecture is depicted in Figure 1.

Any other external system, such as a database, a monitoring system or an anomaly detection system, can be easily added to the data and command channels. The external device registers itself as a publisher for the command channel and subscriber for the data channel. In this manner, it can send messages to the embedded system and receive data from the sensors. Of course, any other embedded systems can be plugged into the network using the same mechanism.

## 2.1. Data Model

Communication between the embedded system and other devices differs depending on the type of interaction. Data and command channels physically separate messages exchanged by the systems. This brings significant advantages in terms of simplicity, scalability and performance.

The command channel is the one used to send requests to sensors or actions to the actuators. Data format contains the following: (1) the command typology and (2) sensors/actuators involved. The format of a request is the following:

```
/COMMAND_NAME/LIST_OF_SENSORS_ACTUATORS&TOKEN
```

where

- `COMMAND_NAME` represents the requested action that we have already discussed, and the values are *read*, *repeatedRead*, *write* and *timedWrite*;
- `LIST_OF_SENSORS_ACTUATORS` represents the name of sensors and actuators to which we want to send the action. Sensors or actuators are separated by the special character "&." Moreover, the name used is the one used in the configuration step;
- `TOKEN` represents a key to match different request–response pairs. Since communication is asynchronous, there is the need to correlate the response to the request. In real-life scenarios, multiple clients might send requests simultaneously, and since they will wait for the response on the same channel, they need to filter the response. For this reason, in the request, the client generates a unique (or random) token that will be included in the response.

Commands cannot be mixed, i.e., send *read* and *write* operations cannot be combined at the same time. For example, we have the following commands:

```
#1 /READ/I1&I2&TOKEN=T001 ;  
#2 /WRITE/O1=1&O2=255&TOKEN=T002 ;  
#3 /TIMEDWRITE/TIMER=100&O3=1&TOKEN=T018 .
```

Command #1 represents a reading example from two sensors ( $I1$  and  $I2$ ) associated with a unique token  $T001$ . For example,  $I1$  and  $I2$  could be, respectively, temperature and humidity sensors. Command #2 represents a written example to two actuators ( $O1$  and  $O2$ ). For each actuator, we specify the value to send. With respect to digital actuators, the admitted values are zero or one (such as  $O1$ ). For analog actuators, the admitted values depend on the actuators themselves.  $O2$  is an example of an analog actuator, and we send a value of 255. Finally, command #3 represents a timedWrite operation on actuator  $O3$ . The operation is executed after 100 seconds. For each of the above commands, a token identifies the corresponding request generated from the client.

After command execution, the response is published on the data channel. The response depends on the request type. If the command is a *read* type, then the response contains raw data acquired by sensors. If the command is a *write* type, then the response contains information about the received message. Figure 2 contains example of possible responses.

<pre> {" data ":   {     " I1 ": 23,     " I2 ": 0.67   },   " token ": " T001 ",   " timestamp ": " 20210506T13:38:00 " } </pre>	<pre> {" data ":   {     " result ": " SUCCESS "   },   " token ": " T002 ",   " timestamp ": " 20210506T13:39:41 " } </pre>
---	--

**Figure 2:** Response to command #1 on the left, and response to command #2 on the right.

### 3. Use Cases

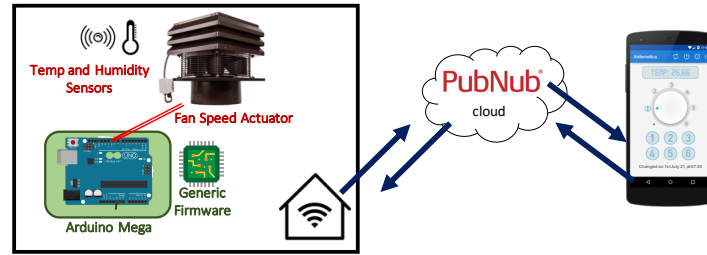
This section presents two real use cases: a small application for monitoring and managing the fan of a fireplace extractor chimney and an industrial application used in the ICOSAF project (PON R&I 2014–2020) to control a resistance spot welding process.

#### 3.1. Fan Control Scenario

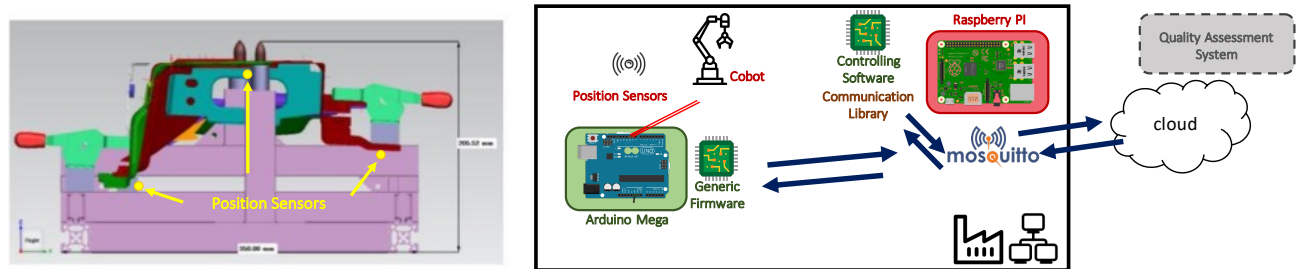
We deployed this app for a small factory that produces extractor chimneys. The final goal was to turn their traditional chimneys into smart appliances by allowing remote control from a mobile application. The deployed architecture is shown in Figure 3.

Since the product was designed to be used by final users, we had several conditions to meet: 1) The additional hardware cannot be expensive in order to avoid a significant increase in the market price of the chimney; 2) The chimney needs to be accessible both at home, i.e., from the local WiFi network, or from outside, i.e., from the internet; 3) Configuration and installation steps need to be as easy as possible, without the need of an IT expert.

Since the chimney has six fan speeds, we represented them as six actuators on the Embedded System. In addition, we connected two sensors to monitor temperature and humidity in the proximity of the chimney. This allows us to create simple rules to change the speed of the



**Figure 3:** Fan control scenario architecture.



(a) Workbench enriched with position sensors.

(b) Quality control scenario architecture.

**Figure 4:** Quality Control Scenario.

fan based on the chimney status. For example, “turn on the chimney when the temperature is higher than 28 °C.”

For the Embedded System, we used Arduino Mega 2560 because it has a good balance between RAM size (256 KB) and price. To enable system discovery over the network, we used the Zero Configuration Network protocol by using a Bonjour implementation for Arduino ([https://github.com/adafruit/Adafruit\\_CC3000\\_Library](https://github.com/adafruit/Adafruit_CC3000_Library)).

For the Message Broker, we opted for Software-as-a-Service (SaaS) solution, PubNub. This provides two important benefits: (a) since it is based on cloud architecture, it offers potentially unlimited scalability, and (b) it reduces the costs of dedicated hardware and maintenance.

Finally, for the remote control, we implemented different client versions: a Java desktop application and two mobile versions (one for Android devices and one for iOS devices).

### 3.2. Quality Control Scenario

We also tested the effectiveness of the approach in an Industry 4.0 scenario. The experiments were conducted with Centro Ricerche FIAT (CRF) within the activities of the “Integrated Collaborative System for Smart Factory (ICOSAF)” Project.

The main goal of the experiment was to support quality assessment on Resistance Spot Welding (RSW) used to assemble car body parts. A typical car contains more than 5000 welding spots of different materials and thicknesses. Assuring the quality of this process is crucial to guarantee the solidity of the assembled vehicle. Several offline and online tests were proposed to evaluate the quality of the final welded workpieces [12, 13, 14].

The quality control process starts with an operator that places a welded workpiece on a custom workbench. This bench is designed so that all the welding spots are reachable by a collaborative robot (cobot) provided with an ultrasound probe that will read the dynamic resistance curves of the spots. Before starting the probe, it is important to verify the correct placement of the workpiece. Since it has a very flexible and uneven shape, it is hard to fasten with clamps. Reading data from a misplaced location will generate dirty data that might negatively impact the quality check algorithm.

To overcome this problem, we placed several digital position sensors in correspondence with the contact points between the shape and the bench, as shown in Figure 4a. These sensors are then wired connected to an Arduino Mega 2560 that runs IoT Helper. The operator will check the sensors interacting with a custom controlling software that will communicate with Arduino using our library. Using an HMI, the operator starts the process. The controlling software will publish a READ command for all sensors to the Message Broker. After receiving the sensor states, if all of them are evaluated as *pressed*, the cobot is started. The dynamic resistance curves of the welding spots are then read and stored in order to be processed using quality assessment techniques. From the architectural point of view, we adopted a hybrid approach by using edge computing to control the sensors and the cobot while dynamic resistance curves are processed on a cloud architecture. Since the company has strict security policies, we cannot use a SaaS solution for the Message Broker, so we opted to deploy a local Message Broker based on the MQTT protocol [15, 16, 17, 18]. The MQTT Broker was installed using a docker image on a Raspberry Pi connected to the same local network of the Arduino. The complete architecture is described in Figure 4b. These scenarios prove that our architecture can be applied within a wide range of cases and can not only be adopted to deploy rapid and affordable data collection in the control scenario but also in industrial and commercial cases.

## 4. Conclusions

We presented IoT Helper, a lightweight, generic framework for IoT and IIoT applications. As discussed, the main contribution of IoT Helper consists in the generic architecture that allows users to quickly prototype smart applications both in domotics and industrial scenarios. We introduced two such scenarios in which the framework has been tested with success and reported experimental data that show how, despite the high flexibility and low costs that come with the framework, it was able to handle a large volume of data and scale up nicely to real-time applications.

IoT Helper was conceived to simplify data collection, and it fits nicely into data analytics application scenarios. We believe that such a framework could be used also in medicine to collect data for ML algorithms [19, 20]. An interesting direction to extend the framework would be to integrate basic analytics features into the firmware in order to enrich and improve the generation of indicators during usage and possibly predictions based on simple machine learning models in order to push analytics to the edge of the architecture. This would have clear benefits in terms, for example, of anomaly detection and robustness of the solution.



## References

- [1] A. Sanders, C. Elangeswaran, J. P. Wulfsberg, Industry 4.0 implies lean manufacturing: Research activities in industry 4.0 function as enablers for lean manufacturing, *J. Ind. Eng. Manag. (JIEM)* 9 (2016) 811–833.
- [2] M. M. Mabkhot, A. M. Al-Ahmari, B. Salah, H. Alkhalefah, Requirements of the smart factory system: A survey and perspective, *Machines* 6 (2018) 23.
- [3] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, M. Gidlund, Industrial internet of things: Challenges, opportunities, and directions, *IEEE Trans. Ind. Inform.* 14 (2018) 4724–4734.
- [4] H. Boyes, B. Hallaq, J. Cunningham, T. Watson, The industrial internet of things (iiot): An analysis framework, *Comput. Ind.* 101 (2018) 1–12.
- [5] H. Cai, B. Xu, L. Jiang, A. V. Vasilakos, Iot-based big data storage systems in cloud computing: Perspectives and challenges, *IEEE Internet Things J.* 4 (2017) 75–87. doi:10.1109/JIOT.2016.2619369.
- [6] G. Mecca, M. Santomauro, D. Santoro, E. Veltri, Iot helper: A lightweight and extensible framework for fast-prototyping iot architectures, *Applied Sciences* 11 (2021). doi:10.3390/app11209670.
- [7] S. Heath, *Embedded systems design*, Elsevier, 2002.
- [8] A. D’Ausilio, Arduino: A low-cost multipurpose lab equipment, *Behav. Res. Methods* 44 (2012) 305–313.
- [9] O. Pineño, Arduipod box: A low-cost and open-source skinner box using an ipod touch and an arduino microcontroller, *Behav. Res. Methods* 46 (2014) 196–205.
- [10] G. M. Spinelli, Z. L. Gottesman, J. Deenik, A low-cost arduino-based datalogger with cellular modem and ftp communication for irrigation water use monitoring to enable access to cropmanage, *HardwareX* 6 (2019) e00066.
- [11] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe 35 (2003) 114–131. URL: <https://doi.org/10.1145/857076.857078>. doi:10.1145/857076.857078.
- [12] C. Capezza, F. Centofanti, A. Lepore, B. Palumbo, Functional clustering methods for resistance spot welding process data in the automotive industry, 2020. arXiv:2007.09128.
- [13] R. Raelison, A. Fuentes, P. Rogeon, P. Carré, T. Loulou, D. Carron, F. Dechalotte, Contact conditions on nugget development during resistance spot welding of zn coated steel sheets using rounded tip electrodes, *J. Mater. Process. Technol.* 212 (2012) 1663–1669. doi:<https://doi.org/10.1016/j.jmatprotec.2012.03.009>.
- [14] Óscar Martín, M. Pereda, J. I. Santos, J. M. Galán, Assessment of resistance spot welding quality based on ultrasonic testing and tree-based techniques, *J. Mater. Process. Technol.* 214 (2014) 2478–2487. doi:<https://doi.org/10.1016/j.jmatprotec.2014.05.021>.
- [15] ISO/IEC 20922, Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1, Standard, International Organization for Standardization, Geneva, CH, 2016.
- [16] R. A. Light, Mosquitto: Server and client implementation of the mqtt protocol, *J. Open Source Softw.* 2 (2017) 265.
- [17] U. Hunkeler, H. L. Truong, A. Stanford-Clark, Mqtt-s—a publish/subscribe protocol for wireless sensor networks, in: 2008 3rd International Conference on Communication

Systems Software and Middleware and Workshops (COMSWARE'08), IEEE, 2008, pp. 791–798.

- [18] M. Bender, E. Kirdan, M.-O. Pahl, G. Carle, Open-source mqtt evaluation, in: 2021 IEEE 18th Annual Consumer Communications Networking Conference (CCNC), 2021, pp. 1–4. doi:10.1109/CCNC49032.2021.9369499.
- [19] P. Lapadula, G. Mecca, D. Santoro, L. Solimando, E. Veltri, Greg, ml – machine learning for healthcare at a scale, *Health and Technology* 10 (2020) 1485 – 1495. doi:10.1007/s12553-020-00468-9.
- [20] P. Lapadula, G. Mecca, D. Santoro, L. Solimando, E. Veltri, Humanity is overrated. or not. automatic diagnostic suggestions by greg, ml, *Communications in Computer and Information Science* 909 (2018) 305 – 313. doi:10.1007/978-3-030-00063-9\_29.