applied sciences

Article

IoT Helper: A Lightweight and Extensible Framework for Fast-Prototyping IoT Architectures

Giansalvatore Mecca †, Michele Santomauro †, Donatello Santoro *,† and Enzo Veltri *,†

Dipartimento di Matematica, Informatica ed Economia, Università Degli Studi Della Basilicata, 85100 Potenza, Italy; giansalvatore.mecca@unibas.it (G.M.); michele.santomauro@unibas.it (M.S.)
* Correspondence: donatello.santoro@unibas.it (D.S.); enzo.veltri@unibas.it (E.V.)
† These authors contributed equally to this work.

Abstract: Industry 4.0 is focused on the task of creating Smart Factories, which require the automation of traditional industrial processes and the fully connection and integration of different systems and devices. However, despite the wide availability of tools and technology, developing intelligent applications in the industry framework remains a complex and expensive task. This paper proposes a lightweight, extensible and scalable framework called IoT Helper to facilitate the adoption of IoT and IIoT solutions both in industry and domotics. The framework is designed to be highly flexible and declarative in nature, thus allowing for a wide range of configurations with minimal user efforts. To emphasize the practical applicability or our proposal, we present two real-life use cases where the framework was successfully adopted. We also investigate a crucial aspect of these applications, i.e., what level of scalability can be achieved with a lean generic framework based on inexpensive components such as ours. Comprehensive experimental results show the excellent cost-to-performance ratio of our solution. We consider this to be an important contribution because it paves the way for a more widespread adoption of IIoT-enabling technologies in industry.

Keywords: embedded-system control; edge computing; Internet of Things; open-source software; smart manufacturing

check for updates

Citation: Mecca, G.; Santomauro, M.; Santoro, D.; Veltri, E. IoT Helper: A Lightweight and Extensible Framework for Fast-Prototyping IoT Architectures. *Appl. Sci.* **2021**, *11*, 9670. https://doi.org/10.3390/app11209670

Academic Editors: Stanislao Patalano, Antonio Lanzotti, Bruno Siciliano and Luigi Villani

Received: 17 August 2021
Accepted: 11 October 2021
Published: 17 October 2021

1. Introduction

Industry 4.0, or the fourth industrial revolution [1], brings *smart factories* at the center of the technology spectrum. 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) [2,3].

IoT and IIoT differ in several respects, as we discuss in Section 2, but they share two fundamental aspects that motivate this work.

On the one side, IoT and IIoT architectures 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 as paradigmatic sources of Big Data [4]. Therefore, it is crucial that frameworks conceived for these tasks are able to scale to such large volumes of data.

On the other side, we notice that in both in 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 very often appliances are not IoT–enabled and, therefore, require tools that can bridge the gap towards the goal of integration and remote control. However, it is especially true in industry. In fact, the following is true:

- 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.
- When sensors are available, they tend to be quite expensive and often difficult to configure.
- Finally, industrial sensors are usually not cloud-enabled and, therefore, fail to meet the Big Data requirements discussed above.

As a consequence, IIoT applications tend to be complex, monolithic projects with high investments and increased design time. To facilitate the adoption of these technologies, we believe that for both IoT and IIoT there is a strong need for lean, generic solutions that may scale nicely up to Big Data scenarios. In this paper, we propose such a solution.

1.1. Contributions

We present an easy-to-go framework called IoT Helper that allows for quick prototyping in terms of IoT/IIoT-enabled applications. The main contributions made by IoT Helper are the following:

- We introduce a new framework for 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.
- The system requires minimum configuration and virtually no application logic in order to remotely access the data and control devices. As a consequence, application developers can focus on the development of higher-level applications that use data collected by IoT Helper in order to gain insights.
- IoT Helper is cloud-enabled by default. It is based on a publish-and-subscribe protocol for decoupling the production of data from its consumption and may leverage public-cloud platforms in order to scale to very large volumes of data. At the same time, coherently with its agile inspiration, the framework also allows for on-premise deployments that can be preferred in some scenarios due to data protection and privacy concerns.
- Declarative configuration and architectural flexibility are achieved by means of an abstract representation of the components of the frameworks, i.e., the embedded hardware, sensors, actuators, message endpoints and message formats. This conceptualization, described in more detail in the following sections, is a crucial component of our implementation.
- Finally, IoT Helper is based on inexpensive, open-source components from the Arduino [5] platform. This significantly lowers costs and adoption times and concretely fosters a more agile approach to IoT and IIoT. In this paper, we discuss how the choices made in designing the framework help users to quickly prototype working solutions and to test them. Then, after initial tests are completed, a final development phase of the missing app components can be started. During this phase, Arduino also allows for industrializing the production of embedded devices for the production phase, thus, further lowering costs.
- 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. In addition to these, we report comprehensive experimental results to study the scalability of our framework. In fact, one of the main questions we intend to investigate in this paper is what level of scalability can be achieved with a lean generic framework based on inexpensive components such as ours. We consider the answer to this question an important contribution of the paper because it proves that IIoT-enabling techologies can now be considered as a commodity; therefore, users may be more ambitious in experimenting with their usage.

1.2. Organization of the Paper

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 introduces the logical architecture. In Section 4, we present two real use cases where we deployed our solutions, and we describe the real architecture adopted for each of them. Section 5 presents scalability experiments where we show that low-profile embedded systems could scale from low throughput scenarios to high throughput scenarios. Finally, Section 6 concludes the paper.

2. Related Work

There is a vast number of proposals to address the class of applications discussed in this paper [6,7]. In this section, we classify first the relevant technologies and then propose a comparison of our proposal to other solutions on the market.

Embedded Systems Modularity, interoperability and real-time orientation are considered as primary requirements for smart factories [8]. Modularity is important because each subsystem is designed to perform highly specific tasks, but, at the same time, it should be able to be combined with other subsystems in order to extend its actions. Interoperability is mandatory in order for different subsystems to exchange information among each other. Real-time data analysis is crucial in order to quickly change production configurations and to react to events.

In this framework, *Embedded Systems* [9] play a central role in order for achieving a configurable and programmable architecture. Embedded systems are programmable modules typically based on low-end hardware that allow one to easily receive raw data from sensors and generate actions with the actuators. Data received by the sensors represent the *inputs* that the system can read. The *outputs* of the system are the commands sent to the actuators to generate actions. Thus, an embedded system can be observed as a *generic program* that manages inputs and outputs and can be easily configurable.

Arduino (https://www.arduino.cc/, (accessed on 15 October 2021)) is a highly extensible embedded system used in most laboratory experiments with a wide range of usages [5,10,11] that is also characterized by very low costs in the range of USD 30 or even less. A more recent alternative is Espruino (https://www.espruino.com/, (accessed on 15 October 2021)). Espruino is as equally inexpensive as Arduino with respect to higher-end solutions and introduces a number of interesting features, namely the use of the JavaScript programming language in place of C/C++ and an improved Integrated Development Environment with a nice graphical editor for code. Espruino is largely compatible with Arduino shields. Tasmota (https://tasmota.github.io/, (accessed on 15 October 2021)) is another alternative, based on ESP hardware, similarly to Espruino.

Higher-range devices include Raspberry Pi (https://www.raspberrypi.org/, (accessed on 15 October 2021)). Raspberry PI devices provide more flexibility with respect to Arduino-like ones both in terms of operating system, programming language and extensibility. They also have higher prices.

Our proposed architecture is agnostic of the hardware so that a variety of embedded system can be used. In our deployed version, we used Arduino because the primary goals of this paper is to investigate scalability issues on inexpensive hardware. However, we have an almost complete port of the framework for Raspberry Pi, and porting to Espruino is planned as future work.

Classification of application scenarios and architectures. Most of the proposed solutions are domain-specific, ranging from performance evaluation of employees [12] and supply-chain management [13] to production monitoring [14] using SOA [15] architectures. Early efforts have been made to generalize environmental monitoring applications [16]. Indeed, IIoT [2] differs from IoT in two several respects:

- The main goal of IIoT is the interconnection of industrial machinery rather than human-to-machine interaction;

- IIoT assumes the presence of a structured, centralized network in which most of the nodes—i.e., machines—are fixed and known in advance; therefore, there is little to no focus on extensible network architectures in which peers may freely join and leave;
- IIoT is usually deployed in mission-critical scenarios in which data integrity and preservation are a primary concern.

Industry networks [3] are composed of edge nodes, i.e., the peripheral nodes where sensors and actuators are located. Edge nodes are the ones that produce data. In addition to edge nodes, intermediary nodes manage a small portion of the produced data. Typically, such intermediate nodes are specialized in task-specific analyses. Finally, centralized nodes are responsible for collecting and processing the dataset in order to derive useful information from it.

Based on where data are processed, it is possible to have three types of network architectures [17]:

- The first type is the traditional *on-premise* or *cloud-based* architecture where data from edges nodes are directly processed and managed in servers, either locally or in the cloud. This category of solutions that is still the most widely adopted suffers from latency problems in real-time applications with very-high volumes of data due to the latency of sending the data to servers [17] and, therefore, might be unsuitable for some scenarios in which data must be processed in real time.
- The alternative is *edge computing*, where data are processed and consumed directly in edge nodes. This solves the latency problem but incurs other limitations. In fact, edge nodes suffer from low or limited resources that limit computational power.
- A further alternative is *fog computing* where computational tasks are demanded to intermediary nodes in the network. Edge nodes produce data. Intermediary nodes pre-process and consume data for real-time analysis and then send processed data to the cloud for further analysis or storage.

A flexible IIoT framework should not be based on a fixed network architecture, but it should be flexible enough to accommodate for mixed approaches and take advantages or the pros of each of them.

In this paper, we concentrate primarily on on-premise and cloud-based scenarios for several reasons. First, these still represent the most frequent case in both IoT and IIoT. Second, edge computing and fog computing impose both higher costs due to the need of more capable embedded systems and higher complexity in coding and debugging. However, the main reason is of a methodological nature. Our study aims at measuring the trade-offs between hardware costs and scalability. In order to measure scalability, we need to be able to assess the capability of the architecture in order to generate data at the edge node and to process them at the servers. In an edge computing scenario, the embedded hardware not only collects data but also needs to process them. Therefore, the overall data collection and processing rate depends on the complexity of edge computations.

Communication Protocols. Multiple protocols [18] were proposed to address the problem of communicating among modules of an IoT/IIoT application. Broadly speaking, it is possible to group protocols into two main categories, as described below.

- *Client-Server* protocols work in a point-to-point fashion and require direct communication between modules. To provide an example, in order to obtain the reading of a sensor, a client app needs to contact the embedded hardware directly through the network. The main ones are XMPP and CoAP. XMPP (Extensible Messaging and Presence Protocol) is based on the exchange of XML messages. It is widely used in device-to-people communication such as control remote devices. CoAP (Constrained Application Protocol) relies on REST messages. It is used on low power consumption devices and is recommended in scenarios with a low number of messages exchanged. These protocols are easier to implement since they do not require the introduction of mediators within the architecture but show serious limitations in terms of flexibility and scalability in real-time applications [19,20]. On the one side, they introduce significant overhead in those cases in which multiple application modules—e.g., different instances of a mobile–client app—need to communicate with the base embedded system. On the other side, the case where real-time sensors generate volumes of data at a pace that applications cannot handle is very frequent due to the need to store values and to process and render them for users.

- An alternative is represented by *Publish and Subscribe (P&S)* protocols such as AMQP or the most widely used MQTT, the Message Queue Telemetry Transport protocol. P&S architectures systems exchange messages through middleware components called *message brokers*, where publishers push messages and subscribers read them. This obviously incurs more complexity in setting up the overall architecture, but it solves the problem of point-to-point protocols. In fact, brokers decouple the generation of messages from their actual handling, thus allowing for much higher scalability. MQTT exchanges messages using TCP. A permanent connection between the clients and the broker is established to exchange those messages through queues or channels. MQTT is widely used in data communication scenarios such as data collection. AMQP (Advanced Message Queuing Protocol) adds a security layer based on TLS. It is often used for data analysis.

To tackle the proliferation of protocols and to render nteroperability between different systems, Open Mobile Alliance (OMA) proposed a new protocol, OMA Lightweight M2M (OMA-LwM2M), that introduces a middle layer between servers and clients deployed on the IoT devices. Recently, OMA-LwM2M also supports P&S using MQTT.

Comparison to Other Solutions. As discussed, many alternatives exist on the market both in terms of embedded devices, protocols and server or cloud-based solutions. Many of them require coding specific application logic for the problem at hand. On the contrary, as it will be discussed in the following sections, a distinguished feature of our approach is that the framework can be configured in a declarative manner by specifying sensors, i.e., input values, and actuators, i.e., actions to be performed on the hardware without any line of code.

ThingWorx (https://www.ptc.com/it/products/thingworx/, (accessed on 15 October 2021)) and Cumulocity IoT (https://www.softwareag.cloud/site/product/cumulocity-iot.html, (accessed on 15 October 2021)) are examples of cloud-based platforms for collecting and processing data gathered by smart embedded devices. However, they do not provide services in terms of declarative configuration of devices—each embedded devices needs to be configured offline by the user before connecting it to the plaform—or fast development of client solutions.

Two proposals that share common goals are Greengrass and Brainboxes. We discuss these proposals and how they compare to ours in the following.

Greengrass (https://aws.amazon.com/it/greengrass/, (accessed on 15 October 2021)) is a framework for developing IoT applications on the AWS cloud platform. It leverages the low-level IoT provided within AWS IoT. In essence, in order to develop a Greengrass application, users need to configure the embedded devices and join them to the project. Then, they can use Greengrass commands to monitor and collect data from devices. This can also be performed with an edge-approach by pushing code to embedded devices. A range of protocols and connectors for different hardware are provided.

The BB400 from Brainboxes (http://www.brainboxes.com/, (accessed on 15 October 2021)) is a commercial, ready-to-use solution for industrial IIoT apps. The BB400 is powered by a custom Raspberry Pi device integrated with an Arduino board for digital and analog sensor and actuator control. It has a number of features specifically designed for industrial environments, such as a protective case for high temperature and a UPS battery. The box is configured with a proprietary firmware that provides similar features to IoT Helper: (i) it allows for a declarative configuration of sensors and actuators without the need of writing code; (ii) it supports the MQTT protocol; and (iii) it offers a high level interface to ease the development client code.

Both Greengrass and BB400 are similar to our approach in many respects, but there are some significant differences. The main ones are the following:

- Greengrass requires the adoption of the AWS cloud platform, while our framework can also be deployed on-premise; this is an important requirement in many industrial applications in which security and data protections are primary concerns.
- On the contrary, BB400 provides several connectors and also allows for configuring on-premises architectures. Still, while IoT Helper uses a declarative approach to define inputs and outputs and hides the complexity of the underlying architecture (message format, topics, MQQT brokers and so on), leaving the user to focus on the client side only, the Brainboxes framework requires the user to explicitly define the communication flow between the edge node and the server using Node-RED, a browser-based visual flow editor based on JavaScript.
- One clear difference is represented by the computing power of the required embedded devices: Greengrass relies on the Java platform and, therefore, needs at least a Raspberry Pi device; BB400 uses a Debian Linux distribution and, therefore, runs on a combination of both a Raspberry Pi and and Arduino. In both cases, costs are higher than those incurred by our framework. In fact, a BB400 device is one order of magnitude more expensive than our solution. Since it is typically needed to configure many of these devices, often in the order of the dozens, these differences in costs may be significant.
- Another important aspect is that there is no available data about scalability of these platforms. On the contrary, a major goal of this paper is to investigate the scalability limit that can be reached with inexpensive hardware. In fact, we consider this one of the main contributions of the paper.

To summarize, these products are more suited for higher-end apps with complex edge-based logic and significantly larger budgets, but they may be unsuitable for rapid-prototyping of IIoT scenarios where great flexibility is required to design, implement and test the app in the early stages before committing to a larger investment. On the contrary, we consider IoT Helper as an ideal solution for this purpose, thus, nicely complementing the offer of solutions on the market.

3. System Architecture

This section introduces the IoT Helper extensible 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, as discussed in Section 2.

Figure 1 shows the core of our proposed architecture. It is composed of the following:

- An *Embedded System* that acts as the main controller of sensors and actuators deployed in the solution;
- A *Configuration Module* that allows users to configure and customize the various sensors and actuators;
- A *Generic Firmware* to manage operations, also called *commands* on connected devices;
- 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. Indeed, any sensors or actuators can be plugged into an Arduino shield. Moreover, it is possible to re-engineer the shield to customize the hardware to the specific scenario with industrial components. One of the cons of Arduino is the small memory size. Thus, the firmware that runs in the Flash Memory should be minimal.

To achieve high extensibility and improve usability, the Embedded System needs to be easily 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, well known in other fields such as software engineering, allows for the creation of loosely coupled systems.

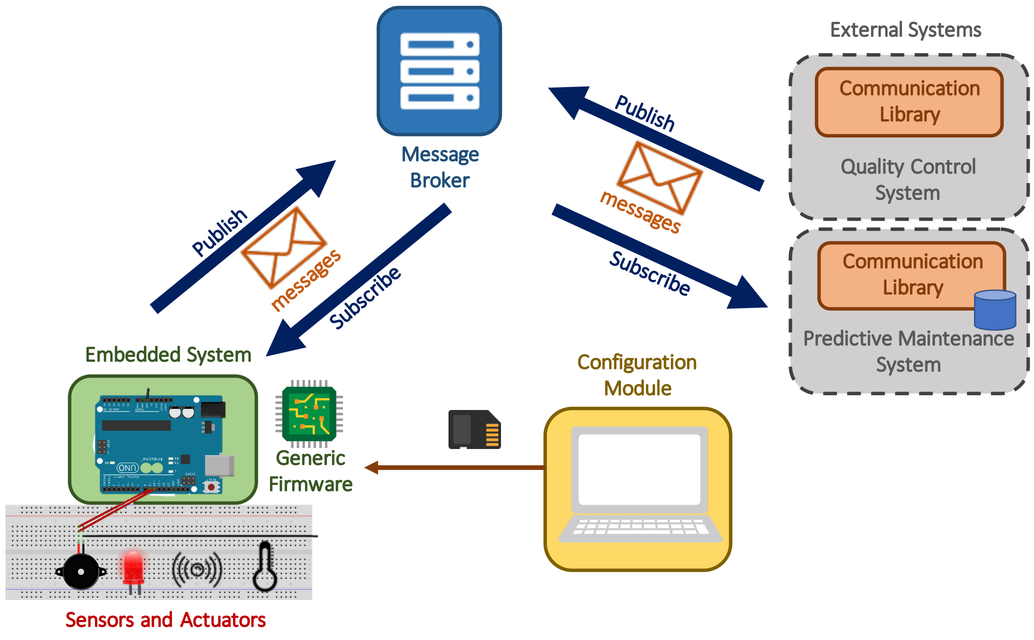

Figure 1. System architecture. Data collection.

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 "*I*1". In addition, we have two actuators: a digital LED *O*1 that can be switched on and off (values 0 or 1); and an analog fan controller *O*2 that accepts values from 0 (off) to 6 (maximum speed).

Table 1. Configuration example. *I*1 is a digital sensor, *O*1 is a digital actuator and *O*2 is an analog actuator.

Type	Typology	Name	Description	Pin	Values
Input	Digital	*I*1	Temperature Sensor	1	
Output	Digital	*O*1	Initialization Complete LED	10	{0,1}
Output	Analog	*O*2	Fan Speed	12	{0–6}

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. For practical reasons, we provide a desktop application with an easy-to-use user interface in order to specify configuration values.

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

- **Read**—*read values of inputs* $I_0 \ldots I_n$. For example, read temperature and humidity values from the respective sensors.
- **RepeatRead**—*read of inputs* $I_0 \ldots I_n$ *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* $v_0, \ldots O_n$ *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.

The overall flow-chart of Generic Firmware is shown in Figure 2. 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.

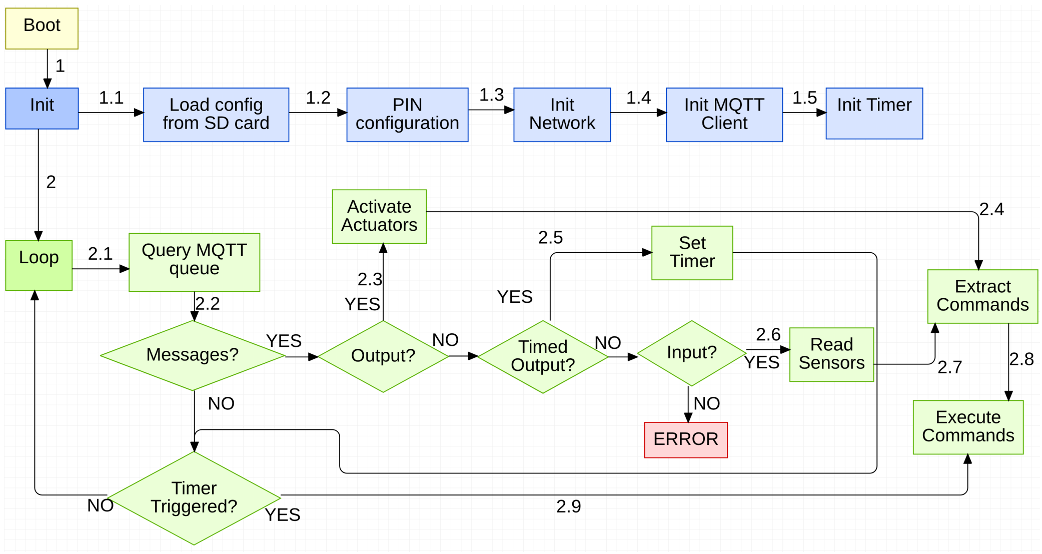

Figure 2. Generic firmware flowchart.

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 3.1.

Generic Firmware is developed in a C dialect and is released as open source.

It is important to emphasize that the Firmware is designed to be completely generic, i.e., it is agnostic of the specific sensors and actuators and can be configured to work with any of these. Indeed, all application-specific logic is coded within the configuration and, of course, in client applications. Client apps, either Web or mobile, embed all custom logic and translate input and output values into the specific scenario. This has several advantages: it makes it easy and relatively inexpensive to turn non-embedded devices and machinery into IoT-enabled systems. At the same time, it enables quick prototyping of an end-to-end solution. In fact, in the early stages, the client app can be simulated by using generic clients, thus allowing users to easily test messages. Then, at later stages, once tests have been completed and a decision to consolidate the solution has been made, real client apps can be developed using high-level programming languages and platforms, such as, for example, Java in decoupled manner from the embedded system. In Section 3.2, we discuss the support library provided as part of the IoT Helper framework to simplify the development of client apps.

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 3.1. Several communication models can be used to connect nodes in a distributed environment. A simple one consists of a *point-to-point* approach, where the Embedded System directly communicates with every external system. This model, however, presents several limitations that might have a negative impact on the overall performance of the architecture. In particular, the following limitations are listed:

- *Performance issues*: The number of messages is directly dependent on the number of nodes in the architecture. Since in a common scenario several applications need to communicate with the embedded system, the latter needs to be able to handle multiple connections at the same time. This can be considered as the normal behavior of high-end servers. However, an embedded system, such as Arduino or Raspberry Pi, has limited concurrent capability. In such systems, every new connection will add considerable overhead to the system, with a result of unacceptable response time, especially in a real-time scenario;
- *Network issues*: In order to establish a direct connection, the embedded system must be reachable from external applications. If we consider a common *domotics* scenario, the embedded system will be connected to the local wireless network, and it can be easily reached by any other devices connected to the same network. However, to be visible to an external device, such as a smartphone connected through a mobile network, an external public IP address is needed, and the routing table of the local network needs to be properly configured. Both of this steps require complex operations that make the adoption of the system in small or home environments burdensome.

To solve these problems, we adopt a Publish and Subscribe (P&S) communication model [21]. 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 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.

Message exchanges are performed using *channels* (also called *queues* or *topics*). A subscriber asks the broker to be notified when a message is published in a channel. Publishers send messages to channels. The Message Broker module acts as a register of all channels and their subscribers and publishers.

Several different implementations of the P&S paradigm can be used to deploy the Message Broker component. One of the most popular is the MQTT open standard [22], a lightweight protocol designed to be efficient even on small micro-controllers. The protocols run over TCP/IP, and messages are optimized to reduce network bandwidth. Popular open-source implementations of MQTTs include Mosquitto, HiveMQ, EMQX and VerneMQ [23–25]. By using one of these implementations, a Message Broker can be deployed on a local server, on virtual private server or even on a low budget device.

As an alternative to achieving higher scalability and reducing the maintenance cost of this component, a cloud solution can be adopted. Some well-known software such as service solutions includes PubNub (https://www.pubnub.com/, (accessed on 15 October 2021)), Google Cloud Pub/Sub (https://cloud.google.com/pubsub/, (accessed on 15 October 2021)) and Ably (https://ably.com/, (accessed on 15 October 2021)). Most of them also support the MQTT protocol and, therefore, can be seamlessly used within our framework.

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.

3.1. 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.

For example, the following represents the response to command #1.

```
{``data'':
    {
     ``I1'': 23,
     ``I2'': 0.67
    },
 ``token'': ``T001'',
 ``timestamp'': ``20210506T13:38:00''
}
```

It contains the raw data from sensors $I1$ and $I2$. Raw data are wrapped in the block "data." In addition, it contains the token related to the request and the time when the response is generated.

By executing command #2, we can receive a response such as the following.

```
{``data'':
    {
      ``result'': ``SUCCESS''
    },
 ``token'': ``T002'',
 ``timestamp'': ``20210506T13:39:41''
}
```

3.2. Support Library

External systems can be seen as *clients* that require services from the Embedded System—acting as a *server*—through the Message Broker. Clients apps may be implemented by using any high-level language or platform, such as Java Desktop applications, Android or iOS mobile applications or low-level firmware. Software developers need to implement messaging functionalities as described in Section 3.1 to and from the Message Broker.

To ease the development of such clients, we designed and implemented a support library in Java and Objective-C. This library offers a high-level interface to communicate with the framework. The Application Level Interface (API) consists in three main components: `IClient`, `Command` and `PinValue`. The UML class diagram [26] is reported in Figure 3.

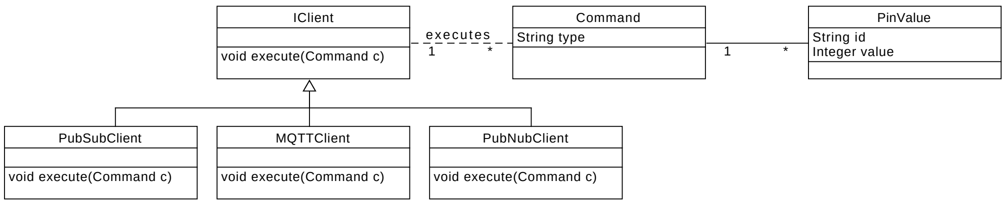

Figure 3. Support Library UML class diagram.

The `Command` represents the action that the client wants to execute. A command is identified by a type, which can be `READ`, `REPEATED_READ`, `WRITE` or `TIMED_WRITE`, and a list of `PinValue`. In the case of a read request, each PinValue contains only the identifier of the sensor, while the value will be populated after executing the command. For a write command, the client needs to specify the identifier of the actuator and its new value. The communication with the Message Broker is managed by the `IClient` interface that has several implementations, one for each supported Message Broker. In the current version of the library, we support MQTT and two SaaS cloud solutions, PubNub and Google Cloud Pub/Sub. Using this library, clients will not have to create messages and parse responses, and the integration with the embedded system will be simpler. In addition, the `IClient` hides the asynchronous communication exposing a synchronous interface, which is usually easier to handle. When the client performs the `execute` method, a request message with a new *token* is sent to the appropriate topic and the execution flow is interrupted until a message for that token is published on the response topic.

4. 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.

4.1. 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 4.

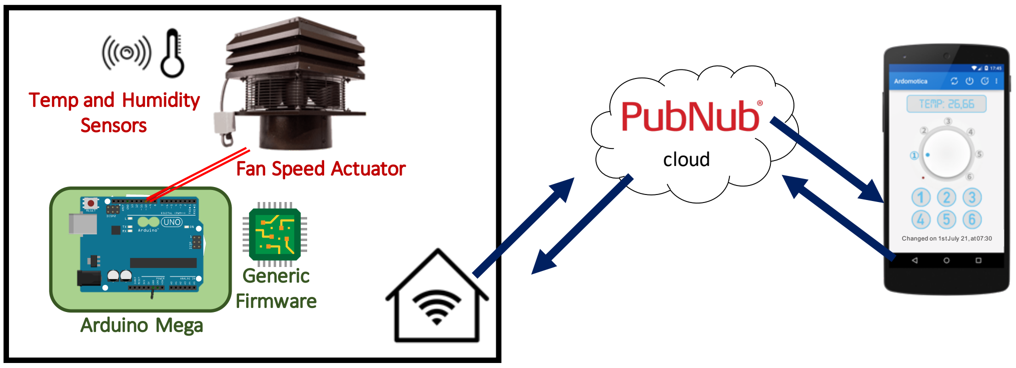

Figure 4. Fan control scenario architecture.

Since the product was designed to be used by final users, we had several conditions to meet:

- The additional hardware cannot be expensive in order to avoid a significant increase in the market price of the chimney;
- The chimney needs to be accessible both at home, i.e., from the local WiFi network, or from outside, i.e., from the internet;
- 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 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, accessed on 15 October 2021).

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 (https://play.google.com/store/apps/details?id=it.unibas.ardomotica, accessed on 15 October 2021) and one for iOS devices). All of them were developed by using the supporting library, allowing developers to send commands to the Embedded System without knowing any technical details of the board and its physical connections or on the message format.

4.2. 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 [27–29].

The quality control process starts with an operator that places a welded workpiece on a custom workbench, as shown in Figure 5. 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.

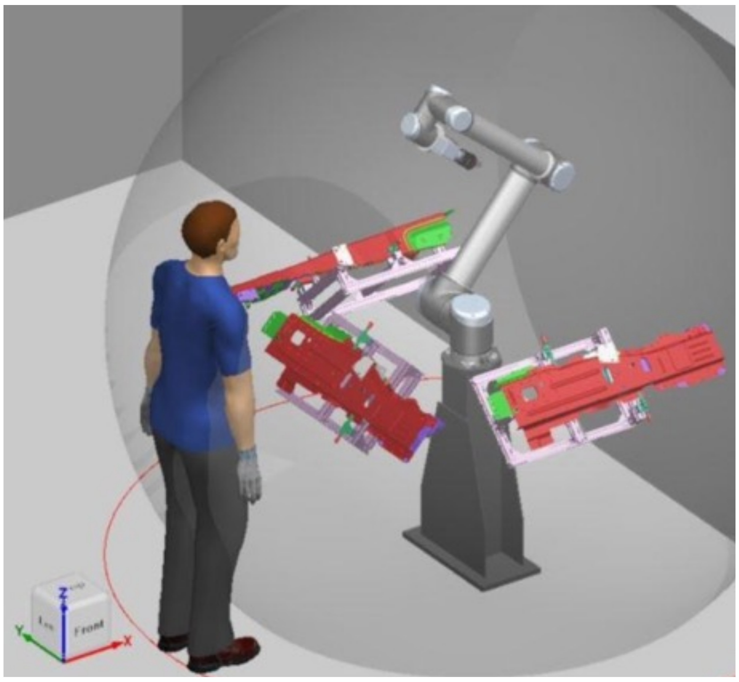

Figure 5. Quality control scenario.

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 6. These sensors are then wired connected to an Arduino Mega 2560 that runs the generic firmware described in the paper. The operator will check the sensors interacting with a custom controlling software that will communicate with Arduino using our supporting 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.

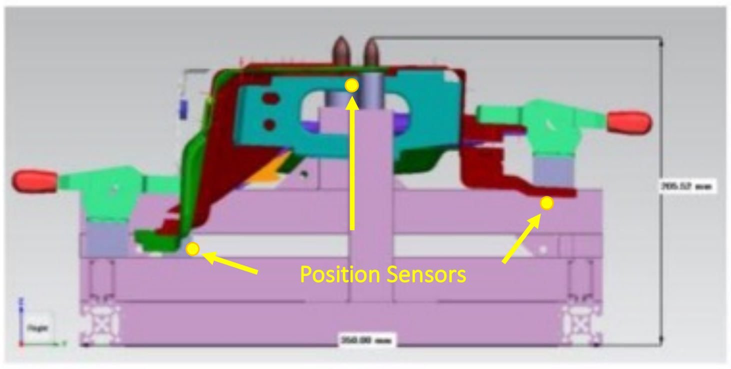

Figure 6. Workbench enriched with position sensors.

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. However, thanks to the flexibility of our solution, we were able to deploy a local Message Broker based on the MQTT protocol. 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 7.

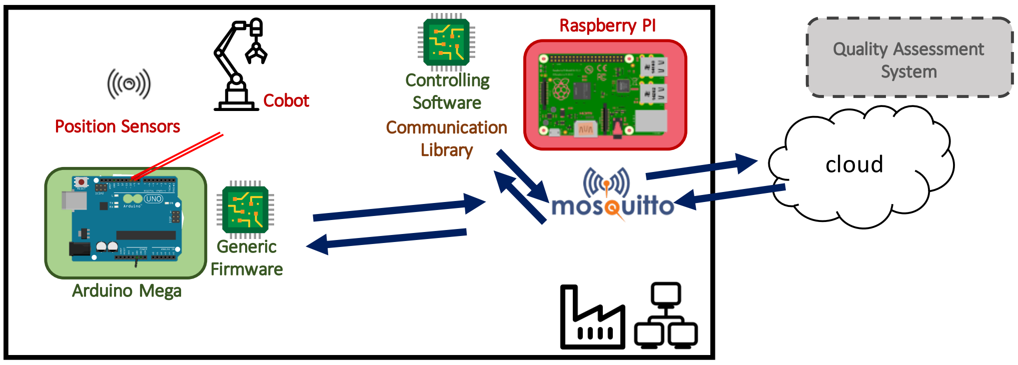

Figure 7. Quality control scenario architecture.

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.

5. Experiments

In this section, we report a number of experimental results with IoT Helper. Given the low cost and the limited hardware profile of the Arduino module, we are especially interested in investigating two essential questions:

- Our first goal is to investigate the capability of IoT Helper to collect potentially large volumes of data in real-time applications.
- Our second goal is to study scalability in terms of number of sensors, actuators and clients that IoT Helper can handle.

To find answers to the previous questions, we conducted two main groups of experiments in a controlled environment.

Pull-Mode Experiments: These experiments measure performance in a data-collection scenario where a client application uses IoT Helper to collect increasingly high-volumes of data from sensors. We measured the number of messages sent from the embedded system to a client subscribed to the data channel.

We fixed seven different frequencies of data collection. Then, we measured the number of the messages received by the client in a period of fixed time (10 min) with different numbers of sensors. For each experiment, we have an *expected number of messages*, corresponding to the total number of data items collected by the sensors in the time interval and the *actual number of messages* received by the client. We report the comparison of these two numbers to measure the effectiveness of IoT Helper.

For example, in the scenario with the frequency of one message per second, we expect that after 10 min 600 data items have been collected ad published on the channel. If the actual number of messages received by the client is lower than 600, IoT Helper is dropping values and, therefore, generating errors.

Command-Mode Experiments: These experiments measure the performances of IoT Helper in a scenario where clients send commands and receive data. Our goal is to evaluate the number of clients and commands that the embedded system is able to effectively handle.

We fixed the number of sensors and actuators plugged into the Arduino shield. We then simulate four different scenarios with 2 to 5 clients. Clients sent both read and write commands with different time intervals. We measured the average time of response of the embedded system. We also measured the errors, i.e., messages lost or handled after 10 s.

This section is organized as follows: (i) First, we introduce the setting in which experiments were conducted. (ii) Then, we discuss the results of Pull-Mode Experiments and (iii) the Command-Mode Experiments, respectively. (iv) Finally, we conclude this section by discussing some insights.

5.1. Experimental Environment

To evaluate the performances of IoT Helper, we deployed the following architecture in a controlled environment:

- The embedded system was based on an Arduino Mega 2560 shield that can be bought on the Web for much less than 50 US dollars;
- We plugged several LEDs into the Arduino shield to simulate actuators and a series of different sensors, analog and digital (temperature, potentiometer, photoresistor and distance sensors), to collect data;
- The Message Broker was running locally on a docker MQTT image;
- We developed a Java application using the Support Library to simulate different clients: clients that perform repeated read operations and clients that send commands (read or write) periodically to the embedded system. The Java application also collect and stores performance stats.
- We connected all nodes in a local WAN.

5.2. Pull-Mode Experiments

We performed six different experiments related to four different scenarios, as shown in Table 2. Each scenario uses a different number of sensors and actuators:

- Scenario 1 contains only one sensor;
- Scenario 2 contains two sensors and two actuators;
- Scenario 3 contains three sensors and three actuators;
- Scenario 4 contains five sensors and five actuators;

We subscribed a Java application to receive data from IoT Helper for each scenario. Each experiment tests all four scenarios for a total of 10 min with different acquisition frequencies ranging from 0.1 data points collected per seconds (one data point every 10 s; the slowest frequency) to 8 data points collected per second (one data point collected every 125 ms; the fastest frequency). Table 2 reports the total expected number of messages and those actually received by the client app in the four scenarios. Note that for the purpose of this experiment the number of clients is irrelevant since we are essentially measuring the capacity of the embedded system to collect data and push messages in real time to the data channel.

Table 2. Messages received for each scenario with different frequencies.

Freq.	Expected	Scenario 1	Scenario 2	Scenario 3	Scenario 4
0.1 msgs/s	60	60	60	60	60
0.2 msgs/s	120	120	120	120	120
1 msgs/s	600	598	597	596	595
2 msgs/s	1200	1195	1193	1189	1190
4 msgs/s	2400	2312	2313	2301	2292
8 msgs/s	4800	4632	4623	4580	4534

The results demonstrated that when messages are sent with low frequency, for example, one message every 10 or 5 s (the first two rows in Table 2), all the messages are completely handled, even with five sensors and five actuators. With higher frequencies, a delay appears. All messages are sent and received eventually, but some of them occur after the 10 min time frame of the experiment. This is due essentially to the limited processing power of the Arduino Mega CPU, which was not able to dispatch data in real time.

Figure 8 shows the percentage of messages that were dispatched after the deadline. As discussed, with low frequencies (0.1 and 0.2), there were no errors in any scenario. With frequencies of one and two messages per second, the amount of errors is very limited (less than 1%) even in Scenario 4, with five sensors and five actuators; with higher frequencies, the number of missed messages increases to 5%, especially in Scenario 4.

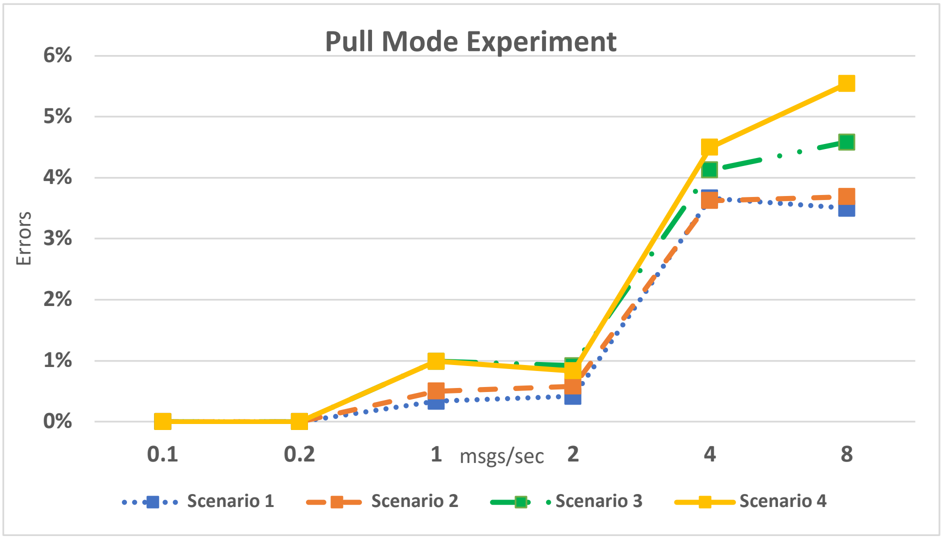

Figure 8. Pull-Mode experiment. Percentage of errors with respect to different frequencies and different scenarios.

We emphasize that most typical data collection tasks generate data with frequencies that are below 2 data points per second. Therefore, we can conclude that IoT Helper can be effectively used in these cases.

5.3. Command-Mode Experiments

The purpose of this group of experiments is to study the capacity of IoT Helper to handle multiple clients that send commands concurrently. Moreover, in this case, we considered four scenarios with different numbers of clients:

- Scenario 1 with two concurrent clients;
- Scenario 2 with three concurrent clients;
- Scenario 3 with four concurrent clients;
- Scenario 4 with five concurrent clients.

We varied the frequency with which clients generate commands from 0.1 commands per second to 8 commands per second, as in Pull-Mode experiments in Section 5.2.

To measure performance, we measured the *transaction time* of each command, i.e., the time in ms between the creation of the request by a client and the reception of the corresponding response notifying that the command had been completed with success by IoT Helper. Then, we calculated the mean among all the messages. We call an error a transaction that does not complete, i.e., the command is sent but execution is not acknowledged. For each test, we also report errors in percentage. The results are shown in Figure 9, which also reports error ratios.

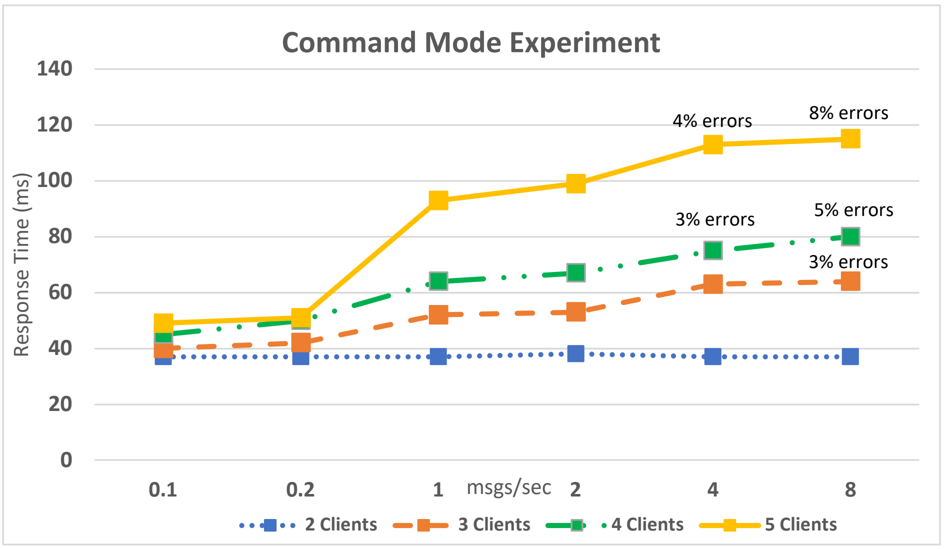

Figure 9. Command -mode experiment. Average response time with different clients and different frequencies of requests. We report errors, if any, near each scenario.

With low frequencies, such as one message every 10 or 5 s, the average transaction time is in the range of 40–50 ms, and no errors occur. When frequencies increase, transaction times also increase, and errors begin to appear. However, we notice that IoT Helper was able to handle all transactions without errors and an average transaction time below 100 ms in all scenarios up to the frequency of 2 commands per second. This, again, is a remarkable result considering the limited processing power of the Arduino CPU and confirms the practical applicability of the framework in many industrial scenarios.

5.4. Discussion

Based on the experimental results reported in the previous sections, we believe that we can provide answers to the questions we intended to investigate.

First of all, our experiments show that IoT Helper performs very well as long as message frequency does not exceed two message per second. In these tasks, even with a relatively high number of sensors to read (up to five in our experiments) or concurrent clients sending commands (also up to five in our experiments), the system performed very well, with very limited delay in data acquisition, low average transaction times to handle commands and no errors.

We consider the 2 msg/s limit to be very promising. In fact, it is definitely adequate to handle the initial design and fast-prototyping phases of any applications. In addition to this, we notice that many applications in Industry 4.0, even real-time ones, still require handling frequencies that are below this limit; therefore, we may conclude that IoT Helper may be a solution in these cases even in a production environment.

Performance degrades when frequencies rise above two messages per second, or the number of concurrent clients increases above five. Notice, however, that these are quite different problems. In fact, handling a larger number of clients can be easily performed by deploying multiple copies of IoT Helper, each of which receives commands from a portion of the clients. Given the relatively inexpensive nature of the framework, this can be considered as a practical solution to the problem of increasing clients.

Handling very high-frequency acquisitions is more complex. If applications are non-mission critical, i.e., a small percentage of data points (in the order of 5%) can be dropped without impacting the overall functionality of the system, IoT Helper can still be a solution. On the contrary, for real-time mission-critical applications with zero-drop requirements, more powerful embedded systems are required in place of Arduino shields.

6. Conclusions

We introduced IoT Helper, a lightweight, generic framework for IoT and IIoT applications. As discussed in the paper, 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 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.

Author Contributions: Conceptualization, G.M., M.S., D.S. and E.V.; methodology, G.M., M.S., D.S. and E.V.; software, G.M., M.S., D.S. and E.V.; testing, G.M., M.S., D.S. and E.V.; writing-original draft preparation, G.M., M.S., D.S. and E.V.; writing-review and editing, G.M., M.S., D.S. and E.V.; supervision, G.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Acknowledgments: The authors wish to acknowledge the ICOSAF project "Integrated Collaborative Systems for Smart Factory" (PON Ricerca e Innovazione 2014-2020 ARS01_00861), financially supported by MIUR (Ministero dell'Istruzione, dell'Università e della Ricerca). We would like to thank Centro Ricerche FIAT and, in particular, Mariano Quagliano e Gianmarco Genchi for their precious collaboration to the development of the second test case.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Sanders, A.; Elangeswaran, C.; Wulfsberg, J.P. Industry 4.0 implies lean manufacturing: Research activities in industry 4.0 function as enablers for lean manufacturing. *J. Ind. Eng. Manag.* **2016**, *9*, 811–833. [CrossRef]
2. Sisinni, E.; Saifullah, A.; Han, S.; Jennehag, U.; Gidlund, M. Industrial internet of things: Challenges, opportunities, and directions. *IEEE Trans. Ind. Inform.* **2018**, *14*, 4724–4734. [CrossRef]
3. Boyes, H.; Hallaq, B.; Cunningham, J.; Watson, T. The industrial internet of things (IIoT): An analysis framework. *Comput. Ind.* **2018**, *101*, 1–12. [CrossRef]
4. Cai, H.; Xu, B.; Jiang, L.; Vasilakos, A.V. IoT-Based Big Data Storage Systems in Cloud Computing: Perspectives and Challenges. *IEEE Internet Things J.* **2017**, *4*, 75–87. [CrossRef]
5. D'Ausilio, A. Arduino: A low-cost multipurpose lab equipment. *Behav. Res. Methods* **2012**, *44*, 305–313. [CrossRef]
6. Babun, L.; Denney, K.; Celik, Z.B.; McDaniel, P.; Uluagac, A.S. A survey on IoT platforms: Communication, security, and privacy perspectives. *Comput. Netw.* **2021**, *192*, 108040. [CrossRef]
7. Laghari, A.A.; Wu, K.; Laghari, R.A.; Ali, M.; Khan, A.A. A Review and State of Art of Internet of Things (IoT). *Arch. Comput. Methods Eng.* **2021**, *28*, 1–19.
8. Mabkhot, M.M.; Al-Ahmari, A.M.; Salah, B.; Alkhalefah, H. Requirements of the smart factory system: A survey and perspective. *Machines* **2018**, *6*, 23. [CrossRef]
9. Heath, S. *Embedded Systems Design*; Elsevier: Amsterdam, The Netherlands, 2002.
10. Pineño, O. ArduiPod Box: A low-cost and open-source Skinner box using an iPod Touch and an Arduino microcontroller. *Behav. Res. Methods* **2014**, *46*, 196–205. [CrossRef]
11. Spinelli, G.M.; Gottesman, Z.L.; Deenik, J. A low-cost Arduino-based datalogger with cellular modem and FTP communication for irrigation water use monitoring to enable access to CropManage. *HardwareX* **2019**, *6*, e00066. [CrossRef]
12. Kaur, N.; Sood, S.K. Cognitive decision making in smart industry. *Comput. Ind.* **2015**, *74*, 151–161. [CrossRef]
13. Verdouw, C.; Beulens, A.J.; Reijers, H.A.; van der Vorst, J.G. A control model for object virtualization in supply chain management. *Comput. Ind.* **2015**, *68*, 116–131. [CrossRef]

14. Li, Q.; Luo, H.; Xie, P.X.; Feng, X.Q.; Du, R.Y. Product whole life-cycle and omni-channels data convergence oriented enterprise networks integration in a sensing environment. *Comput. Ind.* **2015**, *70*, 23–45. [CrossRef]
15. Perrey, R.; Lycett, M. Service-oriented architecture. In Proceedings of the 2003 Symposium on Applications and the Internet Workshops, Orlando, FL, USA, 27–31 January 2003; pp. 116–119.
16. Dobrilovic, D.; Brtka, V.; Stojanov, Z.; Jotanovic, G.; Perakovic, D.; Jausevac, G. A Model for Working Environment Monitoring in Smart Manufacturing. *Appl. Sci.* **2021**, *11*, 2850. [CrossRef]
17. Bierzynski, K.; Escobar, A.; Eberl, M. Cloud, fog and edge: Cooperation for the future? In Proceedings of the 2017 Second International Conference on Fog and Mobile Edge Computing (FMEC), Valencia, Spain, 8–11 May 2017; pp. 62–67.
18. Ponnusamy, K.; Rajagopalan, N. Internet of things: A survey on IoT protocol standards. *Prog. Adv. Comput. Intell. Eng.* **2018**, *564*, 651–663.
19. Adler, R. Distributed coordination models for client/server computing. *Computer* **1995**, *28*, 14–22. [CrossRef]
20. Adebayo, O.; Neilson, J.; Petriu, D. A Performance Study of Client-Broker-Server Systems. In Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, CASCON '97, Toronto, ON, Canada, 10–13 November 1997; IBM Press: Indianapolis, IN, USA,1997; p. 1.
21. Eugster, P.T.; Felber, P.A.; Guerraoui, R.; Kermarrec, A.M. The Many Faces of Publish/Subscribe. *Acm Comput. Surv.* **2003**, *35*, 114–131. [CrossRef]
22. *Information Technology—Message Queuing Telemetry Transport (MQTT) v3.1.1*; Standard; International Organization for Standardization: Geneva, Switzerland, 2016.
23. Light, R.A. Mosquitto: Server and client implementation of the MQTT protocol. *J. Open Source Softw.* **2017**, *2*, 265. [CrossRef]
24. Hunkeler, U.; Truong, H.L.; Stanford-Clark, A. MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks. In Proceedings of the 2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08), Bangalore, India, 6–10 January 2008; pp. 791–798.
25. Bender, M.; Kirdan, E.; Pahl, M.O.; Carle, G. Open-Source MQTT Evaluation. In Proceedings of the 2021 IEEE 18th Annual Consumer Communications Networking Conference (CCNC), Las Vegas, NV, USA, 9–12 January 2021; pp. 1–4. [CrossRef]
26. Rumbaugh, J.; Jacobson, I.; Booch, G. *Unified Modeling Language Reference Manual*, 2nd ed.; Addison-Wesley Professional: Boston, MA, USA, 2004.
27. Capezza, C.; Centofanti, F.; Lepore, A.; Palumbo, B. Functional clustering methods for resistance spot welding process data in the automotive industry. *arXiv* **2020**, arXiv:stat.AP/2007.09128.
28. Raoelison, R.; Fuentes, A.; Rogeon, P.; Carré, P.; Loulou, T.; Carron, D.; Dechalotte, F. Contact conditions on nugget development during resistance spot welding of Zn coated steel sheets using rounded tip electrodes. *J. Mater. Process. Technol.* **2012**, *212*, 1663–1669. [CrossRef]
29. Martín, Ó.; Pereda, M.; Santos, J.I.; Galán, J.M. Assessment of resistance spot welding quality based on ultrasonic testing and tree-based techniques. *J. Mater. Process. Technol.* **2014**, *214*, 2478–2487. [CrossRef]