Problem Statement: Serverless Iot Data Processing

Design and Implementation of a Framework for Smart HomeAutomation Based on Serverless Functions

1.Introduction

Completely automated smart homes are on the way to becoming a well-established re- ality. The ever-increasing number of smart objects in the smart home environment requires the definition of standardized and flexible protocols for commands and state information exchange, besides optimal strategies for processing a large number of commands. New technologies such as serverless computing and ad hoc communication protocols can be leveraged to manage a large fleet of smart objects, while also ensuring good accessibility and intuitive user interfaces. Serverless computing is an emerging cloud-computing paradigm that represents a promising development for many application fields, from micro- and web-services [1], to the the internet of things (IoT) [2,3], edge computing [4] and artificial intelligence (AI) [5], or even for domain-specific scientific applications [6,7]. Serverless com-puting operates at the crossing point of different models that have emerged alongside cloud computing, such as function as a service (FaaS) and pay-per-use models, and technologies such as event-driven programming, virtualization, and containerization [4]. Serverless computing comes with automated scalability and resource provisioning, meaning that resources assigned by the serverless platform scale depending on the number of functions that are running and the resource demand of each of those functions. These characteristics make serverless particularly appealing for applications that need to manage a large pool of users or, in the field of the internet of things and smart objects, a large number of devices [2]. The pioneering commercial platform for serverless computing was Lambda [8], developed by Amazon Web Services (AWS), the Amazon cloud platform.

2.Background

Many smart home automation papers propose APIs or practical implementationsof smart device use cases, employing protocols ranging from HTTP to MQTT to CoAP, which has also shown potential for home automation applications [21]. Crisan et al. [22]proposed and developed an API for smart home automation called qtoggle to control any IoT device in the smart home environment with a TCP/IP stack via HTTP requests, regardless of the type of smart object or device. Sarkar et al. [23] proposed a framework to manage IoT devices in smart buildings using the serverless paradigm. Their proposed system consists of gas and humidity sensors and smart bulbs, and the serverless platform of choice was the OpenFaas platform. Froiz-Míguez et al. [24] proposed a framework for fog computing in smart homes called Ziwi, which enables the coexistence of Zigbee and Wifi devices on the same smart home framework. MQTT is used for its small code footprint and for addressing compatibility between smart devices. The increasing usage of MQTT in machine-to-machine applications has been stressed by Biswajeeban et al. [25], stemming from its lightweight design

and its publish/subscribe architecture. Most frameworks employing this protocol require the implementation of security mechanisms (such as TLS) that add some extra overhead to the communication. As research on the topic of security in MQTT is increasing [26,27], various recent papers have also focused on the implementation of new security mechanisms designed specifically for MQTT-based smart home environments [28,29].

Table 1. Comparison of communication protocols for IoT and machine-to-machine applications.

	NB-IoT	LoRaWAN	Sigfox	LTE-M
Spectrum	Cellular (LTE)	ISM	ISM	Cellular (LTE)
Data Rate [46,47]	~200 kbps	~0.3–100 kbps	~100–600 bps	~1 Mbps
Payload (bytes) [47,48]	1600	243	12	1000
Tx Power (dBm) [48]	23	13	14	23
Consumption [48]	Medium low	Low	Very Low	Medium

MQTT

MQTT is an OASIS standard messaging protocol designed to be lightweight and to have a small code footprint [12]. MQTT is one the main protocols that is used for the IoT and for the implementation of middleware services, alongside COAP, AMQP and HTTP [49]. It is nowa- days used in many application scenarios, such as smart homes and smart buildings [50,51], smart cities [52], smart grids and energy management [53,54], smart agriculture [55], environ- mental monitoring and early warning [56], and so on. Furthermore, the main cloud platforms for the IoT use MQTT as their middleware service and brokering protocol [9].

MQTT is a publish/subscribe message protocol that decouples data producers and data consumers through the use of a central entity called “broker”, a server responsible for dispatching messages among MQTT clients. brokers contain a list of topics, which are strings that are organized hierarchically. The MQTT client can either subscribe to a topic, publish a message on a topic, or both. When a client publishes a message on a certain topic, the broker will forward it to every client that has subscribed to that topic. MQTT includes a series of security mechanisms, including password and username authentication, and au- thentication with transport layer certificate exchange. Data encryption can be guaranteed with protocols such as the transport layer security protocol (TLS), as is performed in AWS, or it can be implemented at the application layer to provide end-to-end encryption.

3.Materials and Methods

The design proposed in this paper is applicable to any kitchen or home appliance due to the flexibility of the MQTT protocol and the serverless paradigm. It provides a valuable framework for the management of smart appliances that can be easily replicated in any smart home system. It leverages technologies in the field of serverless and machine-to- machine communication to create interactive smart objects with dedicated on-demand services, with good accessibility due to the use of vocal interfaces.

A description of the proposed framework for a smart appliance system follows in this section, and in particular, the design of the hardware and the serverless application for a smart kitchen extractor fan are outlined. The framework is completely applica- tion agnostic and can be replicated with any kitchen appliance or smart home device. The cloud-developed application makes use of Amazon Alexa to build a vocal interface and provides touchless interaction with the smart object. A phone application acting as an MQTT publisher can also be used to send commands to the appliance. MQTT makes the framework extremely flexible, and it makes the integration with cloud platforms (in this case, with AWS) simple and secure due to the use of certificates securely stored on the device by the use of a cryptochip. MQTT is also used to receive status updates from the smart object. The serverless framework provides efficient replicability and scalability properties to the application, and most importantly, it rationalizes resource allocation on the cloud when managing a large number of devices.

smart Extractor Fan Design

The kitchen extractor fan is controlled by a device whose block diagram is illustratedin Figure 2. It consists of a microcontroller (referred to as “micro-host”) that interfaces with various peripherals and controls the ventilation motor and a set of lights. A NB-IoT chip is used to receive commands from the cloud in MQTT format over NB-IoT. The MQTTprotocol guarantees a simple and secure interface toward the cloud through the nativeuse of TLS encryption and authentication. The AWS IoT core requires authentication over MQTT with X.509 certificates. A cryptochip is used to securely connect and authenticatethe AWS IoT core. Such a device can securely store certificates on the IoT device andauthenticate the AWS IoT core.

The device receives messages from the cloud, parses them, and executes commandsbased on the content of their payload. It also sends state updates toward the cloud. Firmware updates can be received over the air using MQTT, simplifying the deployment of new firmware on large fleets of devices.

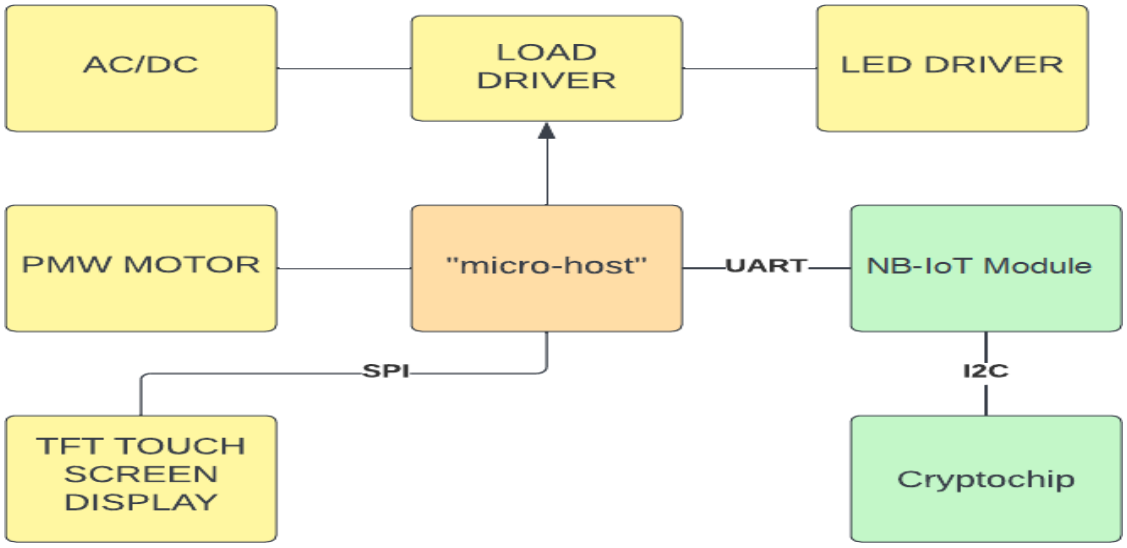

Figure 2. Block diagram of the smart extractor fan,

Architecture of the Serverless Cloud Application

An application was designed for the management and control of smart objects. The ap- plication is developed using AWS Lambda and the MQTT protocol to provide flexible interaction between the users and the extractor fan. Amazon Alexa is used to implement a vocal interface for the control of the smart object. State updates are also sent by the fan using the custom commands protocol and are processed by a dedicated Lambda function on the cloud to keep track of the state of each device.

The application was developed on AWS, and the serverless platform of choice was Amazon Lambda. Two Lambda functions were implemented in NodeJS (JavaScript) using the AWS SDK: one to forward user commands to devices, either by an application or Alexa skill, and the other to handle state updates from the devices. JavaScript, being a high-level interpreted language, lends itself quite naturally to the development of serverless or FaaS applications and, alongside Python, is one of the most used languages for the development of serverless applications [3]. Lambda functions comprised one or more handlers that are used to manage triggers, which can be received by various outside services, including Amazon Alexa or AWS IoT core. Each handler was defined on the basis of the custom command protocol and of the intents that were implemented in the Alexa skill. This way, users are able to control and monitor all the admissible states of the devices.

The complete description of the smart appliance system is illustrated in Figure 4. When a user invokes the Alexa skill with a valid intent, i.e., an Alexa skill intent for which a handler was defined, the skill triggers Lambda and passes a JSON message to it, containing both the UserID and a string identifying the handler required to manage that request. The Lambda function queries DynamoDB for the identifier(s) of the device(s) owned by that user. If the user owns more than one device, then a custom slot in the Alexa intent is used to distinguish between their multiple devices and retrieve the correct identifier from DynamoDB. Once Lambda knows the device to send the command to, and on the basis of the intent, it publishes an MQTT message on AWS IoT core following the commands protocol for that specific device.

The device, having subscribed to its individual topics at start-up, will receive the message, parse its payload, and execute a command according to its content. Whenever it receives a command, the device will also publish an update on its status on its dedicated OutState topic. It is also possible to configure a "refresh time" at the monitoring topics, i.e., a time at which the devices sends periodic updates on the topic. An AWS IoT core rule is implemented to handle state updates and to invoke a second Lambda function, which updates a DynamoDB tablewith the new state information.

The state information is used to monitor devices and their activities, but it also allows users to ask the Alexa skill for the state of the fan, and it uses "feedback" commands that require the Lambda function to know the current state of the device (e.g., "lower the ventilation", "raise the ventilation", and similar commands). The implementation of feedback intents makes the communication between the smart appliance and the user feel more "natural". At the same time, periodic refreshes at short intervals amplify the application load on the cloud and the number of transmissions, which might not be suitable for certain devices or communication protocols. For such use cases,it might be required to set a longer refresh time or to only enable refreshes when there is a state update via outside command. The refresh option has been used for the experimental evaluation of the smart object work load on the cloud.

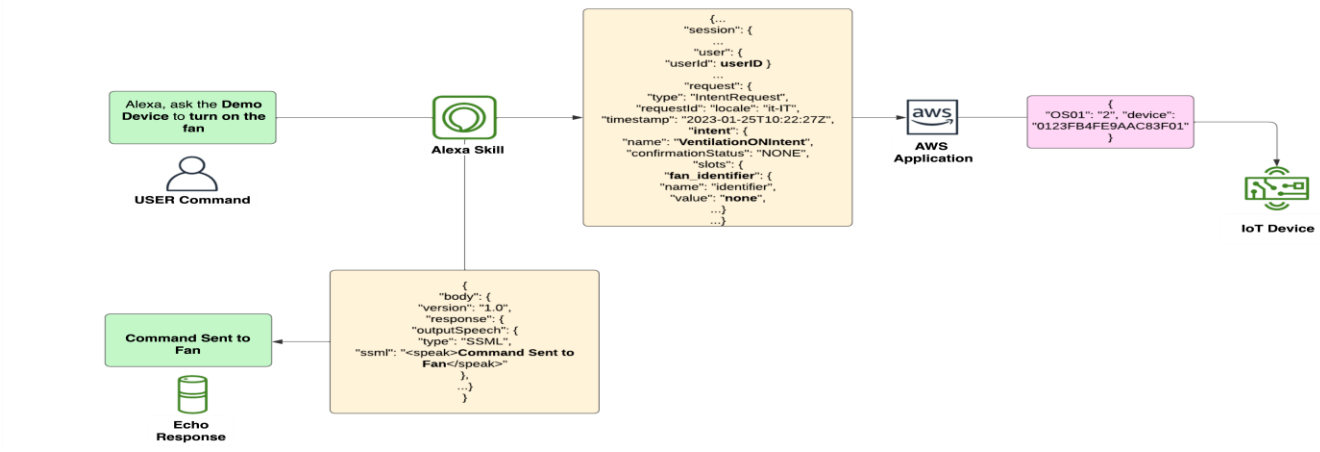

Figure 3. Messages flow from the user command up to the MQTT message generated by AWS that is forwarded to the device

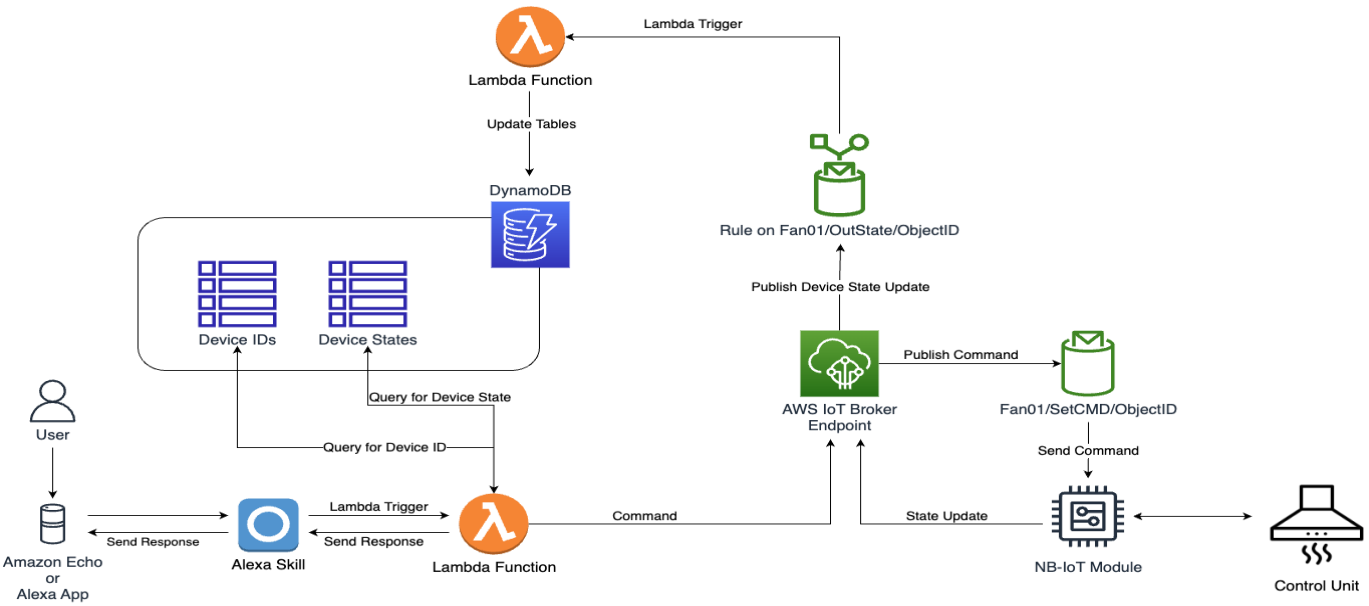

Figure 4. Architecture of the proposed smart fan application

4.Experimental Evaluation

An experiment was carried out to evaluate the use of narrowband IoT to dispatch MQTT messages across MQTT clients. The goal of the experiment was to provide an estimation of the time needed to forward a message to/from an MQTT client over NB-IoT to show whether the protocol is suitable for applications, such as the one developed in this paper, that require an acceptable latency performance to ensure a good user experience.

The experiment involves the following nodes

. An MQTT Client (Client 1) that publishes messages on a dedicated topic named Topic1;

. An MQTT broker;

. A NB-IoT module subscribed to Topic1 and publishing on a dedicated topic named Topic2;

. A second MQTT Client (Client 2) subscribed to Topic2.

This experimental setup is shown in Figure 5. The two MQTT clients might represent cloud applications sending or receiving messages from a smart object, such as vocal com- mands or state updates, while the NB-IoT device stands for the smart object itself. This setup reflects the interaction between users and the smart extractor fan system, where client 1 might be a Lambda function publishing a command on the broker to control the NB-IoT module/smart fan, which will then immediately publish a status update on the OutState topic once it sets the correct output variables.

At the start of the evaluation, the board registers to the NB-IoT network, connects to the MQTT broker, and it subscribes to Topic1. At the same time, the two clients also establish a connection to the MQTT broker, and Client 2 subscribes to its assigned topic. After waiting a short time to ensure that each MQTT client

has successfully connected to the broker, Client 1 will publish a first message on Topic1, containing a timestamp *T*1 representing the start of the communication. The broker then forwards the message to the evaluation board. The board will then publish a new message on Topic2 containing *T*1 and the time elapsed between reception of the first message from the broker and the transmission of the second message. The broker will forward this message to Client 2, which will save the timestamp of reception (*T*2). This information, alongside the one obtained from the first client, the board, and the broker, is used to evaluate latency at different sections of the communication. The procedure is repeated multiple times at intervals of 10 s between each message.

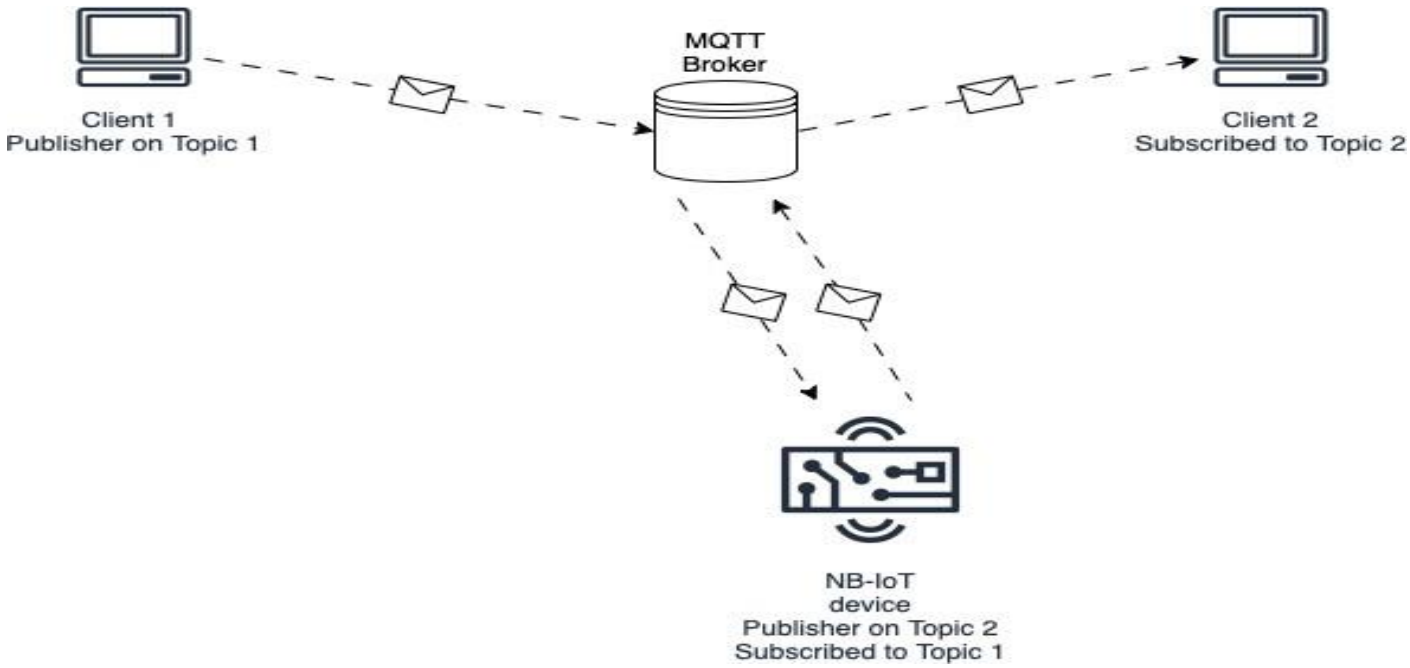

Figure 5. Diagram of the evaluation testbed used to evaluate NB-IoT latency for dispatching MQTT messages.

5.Results

The performance of the developed smart fans was monitored using Amazon Cloud- watch and forcing a situation with frequent calls to Lambda. By setting a refresh time of 1 min for the OutState topic, the Lambda function responsible for updating the state database is called once every minute by the AWS IoT rule that handles messages received on the OutState topic. The function simply parses the JSON message forwarded to it by AWS IoT core, containing the new states and the identifier of the smart object, and updates a DynamoDB table with this new information, also increasing a usage atomic counter on the table to keep track of the usage of each smart device. Such a frequent refresh time might be incompatible with NB-IoT, given the high transmission load compared to the relatively low throughput expected of applications using this protocol, but more suitable to WiFi implementation. Furthermore, configuring the device so that the OutState topic is refreshed only when there is a state variable change is an implementation choice with an acceptable load for NB-IoT, and possibly a more realistic work condition for the smart home object. In the case of the Cloudwatch evaluation, the goal was to

evaluate a situa- tion with frequent calls to the serverless Lambda function; therefore, the number of state updates was incremented.

The results of the evaluation are shown in Figures 6 and 7. The device was left on for about 24 h and the CloudWatch Lambda insights service was used to keep track of various parameters. Figure 6 shows the median, 90th percentile, and maximum duration of Lambda function calls, computed in intervals of 5 min. Figure 7 shows the maximum CPU usage time and network load during 24 h, computed in 5-min intervals.

The spikes in duration correspond to “cold starts” that were also recorded by Cloud- watch. A cold start is the time required to allocate cloud resources to instantiate a Lambda function environment for the first time. Each Lambda function is only kept “warm” for a certain amount of time, requiring a new, prolonged startup to instantiate a new execution environment once the execution environment turns “cold”. Setting up provisioned concur- rency can help mitigate the effect of cold starts when there are multiple devices invoking the Lambda function.

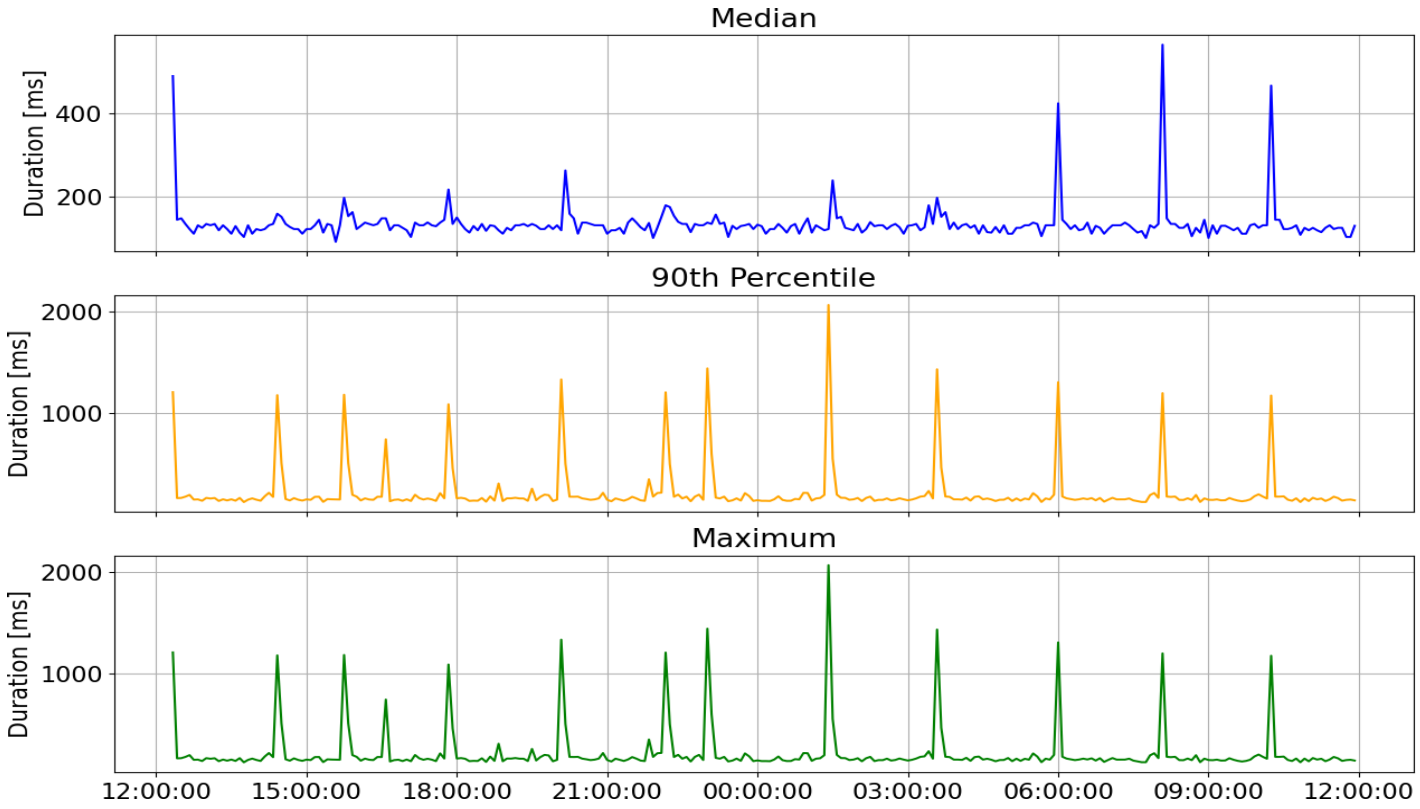

Figure 6. Median, 90th percentile, and maximum duration of a lambda function calls, computed in intervals of 5 min.

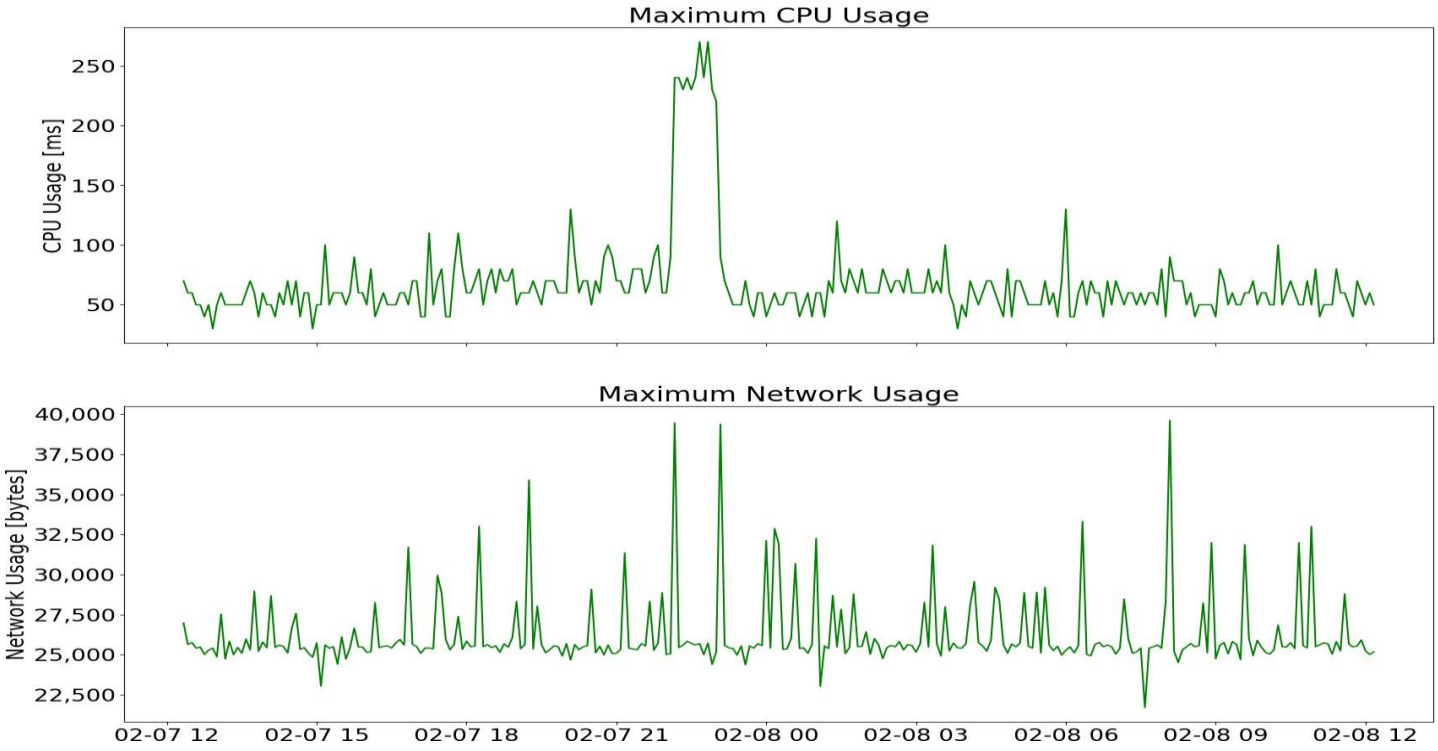

Figure 7. Maximum CPU (**top**) and network (**bottom**) usage during the Lambda function testing, computed in intervals of 5 min.

6.Conclusions

In this paper, a generalized and flexible framework for smart object applications in a smart home environment was proposed, leveraging MQTT, the serverless computing platform Amazon Lambda, Amazon ASK, and NB-IoT. The framework consists of smart objects receiving messages from the cloud through a dedicated message-exchange protocol and interactive vocal interfaces, with serverless functions deployed on the cloud to monitor and control the objects. A practical use case based on this framework was also developed, prototyped, and tested. The smart object, a smart kitchen fan, is equipped with a NB-IoT module and makes use of a custom command exchange protocol based on MQTT. A vocal interface was developed using Amazon Alexa and Lambda, enabling vocal control of all the smart object's controllable states. The suitability of NB-IoT for MQTT messages dispatching was tested and showed a good latency performance, excluding some peaksthat were evaluated and traced back to the formation of MQTT queues on the sender due to lost packets. Excluding cold starts, Lambda execution times are below 200 ms, positively contributing to an overall quasi-real time interaction between users and the smart object. Such a connected object could integrate additional customized services such as predictive maintenance and consumption control, enabled by cellular connectivity and simplified by resource provisioning on the cloud.