MQTTProvider: A flexible MQTT traffic generator for the performance evaluation of complex Smart Cities scenarios

Michel Sales[1], **Mateus P. Santos**[1], **Paulo A. L. Rego**[1]

[1]Universidade Federal do Ceará – CE – Brasil

`michelsb@ufc.br, mateussantos14@alu.ufc.br, paulo@dc.ufc.br`

Abstract. *In smart cities, designing intelligent spaces incorporating Internet of Things (IoT) applications is crucial for enhancing urban efficiency and quality of life. However, developing these applications requires real data at scale, which can be complex and non-trivial to obtain. A real traffic generator aids in modeling unpredictable traffic, validating IoT configurations, evaluating network performance, and improving cybersecurity and privacy solutions. In this sense, we introduce the MQTTProvider, an adaptable IoT traffic generator for simulating various smart spaces and supporting time-driven, event-driven, mobile, actuators, and complex devices. It was designed to integrate with IoT Middleware, such as FIWARE. Experiments indicate that MQTTProvider can be used to assess the scalability of IoT platforms.*

1. Introduction

The rapid evolution of Internet of Things (IoT) and support technologies (e.g., edge devices, storage) has caused an increase in the development of smart space ecosystems in many domains, including home and industrial automation, smart cities, healthcare, agriculture, connected vehicles, and more. Furthermore, there is a large diversity of IoT devices for many purposes (actuators, environment sensors, etc.), and these new uses considerably impact the network's performance [Chio et al. 2023]. According to Statista[1], the number of connected IoT devices nearly doubled between 2019 and 2023, rising from 8.6 million to 15.14 million. Furthermore, it is estimated that there will be more than 29 billion connected IoT devices in 2030, generating zettabytes of data.

In this scenario, designing smart spaces that provide safety, security, reliability, and sustainability for IoT applications requires real data at scale, and such data can be complex and non-trivial to obtain. The problem is related to the cost and complexity of getting such data, which generally involves purchasing and deploying devices [Chio et al. 2023]. In this context, depending on the size of the domain, the project becomes unfeasible. Moreover, sensor data has limitations. It might not capture all relevant scenarios, especially those involving dynamic changes like fluctuating occupancy, adaptable spaces, or unprecedented events like pandemics. These are difficult to replicate in controlled settings, hindering our understanding of human behavior in such situations. Finally, privacy considerations complicate the use of sensor data. Individuals and organizations may be hesitant to share sensitive, personal data, especially when it requires long-term storage or collection from many people [Martínez-Ballesté et al. 2013, Pappachan et al. 2017]. One powerful approach to tackle these challenges involves leveraging the capabilities of an IoT real traffic generator.

[1]https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

There are many benefits to using a real traffic generator, including modeling unpredicted traffic behavior, validating configurations in IoT platforms (performance tuning), evaluating the network's performance, and validating new solutions regarding cyber security and privacy. Moreover, it empowers the development of sophisticated machine learning-based anomaly detection systems by facilitating comprehensive IoT traffic characterization. This involves creating detailed IoT device behavior profiles, providing the foundation for effective identification and classification [Nguyen-An et al. 2020a, Nguyen-An et al. 2020b].

We found several IoT traffic generators in the literature. Based on these studies, we established a set of functional requirements for these generators. These requirements include the capability to simulate time-driven, event-driven, and mobile sensors; actuator simulation; the ability to evaluate large-scale distributed systems; and the creation of complex devices that integrate various sensors and actuators. However, we observed that existing approaches have limitations in generating realistic workloads for IoT platforms. Furthermore, to the best of our knowledge, no existing studies have comprehensively addressed the simulation of actuators and complex devices.

This paper presents the MQTTProvider, an IoT traffic generator for large-scale distributed systems that aims to simulate different types of smart spaces. It enables the simulation of various categories of sensors, which send messages simultaneously using an Message Queuing Telemetry Transport (MQTT) protocol. MQTTProvider also supports the simulation of actuators, which process commands and send status information. Such commands change the state of the actuator and can be sent manually or automatically from datasets. Furthermore, our tool allows the creation of complex devices, which may encompass different sensors and actuators. Finally, MQTTProvider was developed to be adaptable to the main existing IoT Middlewares. Therefore, its topic and data format were designed to be consumed directly by FIWARE [FIWARE 2024], one of the most used IoT Middlewares globally. However, it is worth highlighting that the format is generic enough to be adapted for any new consumer.

We performed experiments in two machine configurations to validate using MQTTProvider to assess an IoT platform's scalability properties. To do so, we defined some performance metrics and ran the tool, varying the number of devices. The results allowed us to identify performance bottlenecks at the software and hardware levels.

The remainder of the article is organized as follows. Section 2 presents work related to this research. Section 3 describes the MQTTProvider architecture and details how different types of devices are implemented. Section 4 describes the experiments performed to evaluate the MQTTProvider. Finally, Section 5 concludes the article.

2. Related Work

IoT and Smart city benchmarks like RIoTBench [Shukla et al. 2017] and CityBench [Ali et al. 2015], built from real device data, stress-test the distributed processing systems that power these complex ecosystems. RIoTBench offers four simulated city and fitness data streams, mimicking real-world sensor bursts from 500 to 10,000 messages per second with diverse arrival patterns. CityBench, in turn, dives deep into RDF stream processing with real-time sensor data, painting a vivid picture of Aarhus's smart city pulse. However, although they have taken an essential step toward the evaluation of smart domains on a large scale, these tools are limited to simple sensors that follow a particular frequency distribution or a dataset, not corresponding to the real scenario, which encompasses actuators and devices that monitor the behavior of citizens in the environment.

SenSE (Sensor Simulation Environment) [Zyrianoff et al. 2017] is an open-source platform that simulates complex smart city environments, generating massive amounts of diverse sensor data from thousands of virtual devices. SenSE generates synthetic MQTT data simulating a sensor network. The user can choose the types (Time or Event-driven) and the number of sensors. Each type of sensor sends the data in a fixed time interval. Regarding data, in time-driven sensors, the data varies randomly, while in event-driven sensors, the data varies according to a probabilistic distribution (e.g., Poisson). However, even though it offers event-driven sensors, the data does not simulate the movement of vehicles and people in different routes. Besides, it does not simulate actuators or complex devices, which may encompass various sensors and actuators.

The IoT-Flock [Ghazanfar et al. 2020] is a freely available framework designed for generating IoT traffic, providing support for two commonly utilized IoT application layer protocols—MQTT and CoAP. This innovative framework empowers users to construct an IoT use case, integrate personalized IoT devices, and generate regular and malicious IoT traffic within a real-time network environment. However, IoT-Flock is limited to simple sensors that send the data in a fixed time interval, and the data varies randomly.

InterSCSimulator [de M. Del Esposte et al. 2019], developed within the InterSCity project, is a scalable and open-source simulation tool designed for large-scale smart city scenarios. It replicates the mobility patterns of cars and individuals across various routes, encompassing bus and subway systems. The simulator includes diverse mobility models for vehicles, pedestrians, buses, and subways. InterSCSimulator can simulate entire cities, like São Paulo, featuring over 10 million virtual software agents navigating tens of thousands of streets. The simulator took an essential step towards simulating the movement of vehicles and people in different routes; however, it does not simulate actuators or complex devices.

It is worth highlighting that only some approaches focus on both topic and message formats. It is essential to focus on the format of both so that the generator is easily adaptable to different types of IoT middleware and reduces error-prone tasks as the scenario inflates. For example, SenSE and IoT-Flock leave it up to the user to define the topic name. SenSE is the only solution that defines the format of the message. Inside the message, in addition to the payload, we have the sensor type, ID, and timestamp, which can provoke an overhead. In MQTTProvider, device identification is done in the topic, which follows the specific FIWARE format. Furthermore, the message body transmits only the payload. Also, it follows a FIWARE-specific format, the Ultralight (UL) [FIWARE 2024], a lightweight text-based protocol for constrained devices and communications with limited bandwidth and memory resources.

Finally, to the best of our knowledge, no previous work explores the simulation of actuators and complex devices (may encompass different sensors and actuators). We propose to solve this problem by providing all the needed devices. In the next section, we will describe our solution.

3. MQTTProvider

This section presents MQTTProvider, an MQTT traffic generator aimed at large-scale IoT domains. MQTTProvider is a better alternative to other generators because it supports the simulation of:

- A larger set of sensor types, including those that send random values within a range and at a fixed interval (**Time-driven sensors**), such as those whose data is generated from events caused by the movement of people in different spaces participating in different events (**Event-driven sensors**) or the movement of vehicles on a given route (**Mobile sensors**);
- **Actuators**: devices that receive commands and send state information. Such commands change the state of the actuator and can be sent manually or automatically. On the other hand, status information is sent periodically. For example, a smart door lock always reports whether its status is locked or unlocked. Furthermore, command traffic is automatically generated by a dataset. Upon receiving a command, our actuator will change its state. More details are presented in the subsection 3.3;
- **Complex devices**: devices that can encompass a set of sensors and actuators. Such devices are increasingly present in our daily lives. For example, smart light bulbs can have brightness sensors, light intensity, and color controls. Another example would be a smartphone, a complex device with sensors such as an accelerometer, gyroscope, magnetometer, GPS, and biometric sensors.

Moreover, different IoT middlewares must support the traffic generated. Therefore, our tool sends messages using MQTT, the most used communication protocol in IoT environments. Besides, we designed the topic and data format (Ultralight) to be consumed directly by FIWARE. In this case, devices must publish (measures) and subscribe (commands) to data on specific topics. **Measures** are sent on the ***/apiKey/deviceID/attrs*** topic. On the other hand, **Commands** are received on the ***/apiKey/deviceID/cmd*** topic. In this scenario, ***apiKey*** must be replaced by a code that describes a group of devices, and ***deviceID*** must be replaced by a unique device identification.

Furthermore, MQTT message data must follow the Ultralight (UL) format. UL is a lightweight text-based protocol for constrained devices and communications with limited bandwidth and memory resources. The payload consists of a list of key-value pairs separated by the pipe | character, as shown below.

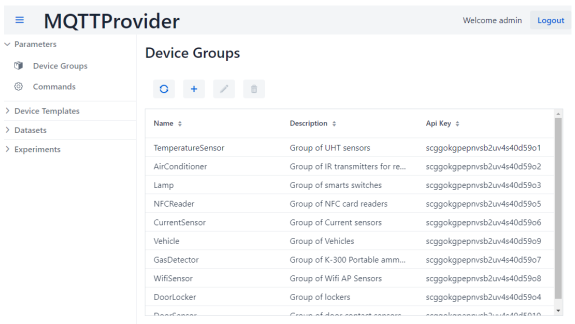
```
<key>|<value>|<key>|<value>|<key>|<value> ...
```

The formatting described above enables the validation of configurations on an IoT platform (performance tuning). It reduces the probability of errors (i.e., the definition of topics) in building a large-scale simulated infrastructure. It is worth highlighting that this format is generic enough to be adapted for any new consumer, unlike FIWARE.

Creating applications that seamlessly interact with databases is crucial in the ever-evolving software development landscape. Databases provide a structured way to organize and store data. Information is stored in tables with predefined columns, ensuring a consistent and organized format. This structure simplifies data retrieval and analysis. Of the related works, the only one that used a database to save sensor information was

IoT-Flock [Ghazanfar et al. 2020]. For MQTTProvider, we decided to turn to a powerful combination of Java programming language and PostgreSQL database management system.

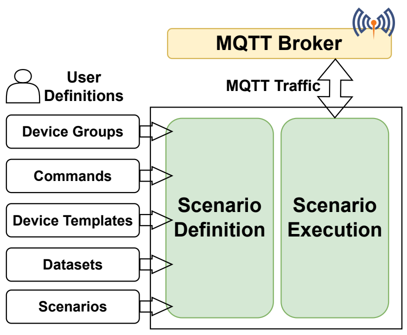
Figure 1(a) shows the MQTTProvider Graphical User Interface (GUI). It consists of a Web Application that uses Java Vaadin[2]. In Java Vaadin, **"View"** and **"Service"** are fundamental for effectively creating and managing web applications. These concepts help separate the concerns within the application, with Views primarily handling the user interface (**frontend**) and Services (**backend**) managing the business logic and data access. Concerning our application, Services are injected into Views using Spring-based dependency injection. This allows Views to directly call methods defined in Services. On the other hand, our Services use Spring Data JPA (repositories) to manage entities in the Postgres Database component. Since all interactions between MQTTrovider and Postgres are initiated by JDBC (Java Database Connectivity), the entities can be containerized and run from exposed ports. To keep things simple, both components run using Docker[3]. Docker is a container technology that isolates different components into their respective environments. The associated "docker-compose.yml" file shows the necessary configuration information.



Device Groups

Name	Description	Api Key
TemperatureSensor	Group of UHT sensors	scggokgpepnvsb2uv4s40d59o1
AirConditioner	Group of IR transmitters for re...	scggokgpepnvsb2uv4s40d59o2
Lamp	Group of smarts switches	scggokgpepnvsb2uv4s40d59o3
NFCReader	Group of NFC card readers	scggokgpepnvsb2uv4s40d59o5
CurrentSensor	Group of Current sensors	scggokgpepnvsb2uv4s40d59o6
Vehicle	Group of Vehicles	scggokgpepnvsb2uv4s40d59o9
GasDetector	Group of K-300 Portable amm...	scggokgpepnvsb2uv4s40d59o7
WifiSensor	Group of Wifi AP Sensors	scggokgpepnvsb2uv4s40d59o8
DoorLocker	Group of lockers	scggokgpepnvsb2uv4s40d59o4

(a) Graphical User Interface



(b) Architecture

Figure 1. The MQTTProvider.

Figure 1(b) illustrates the MQTTProvider architecture. It has a two-component architecture involving Scenario Definition and Scenario Execution. The **Scenario Definition** component serves as the foundation of the traffic generator, where the devices and traffic scenarios are conceptualized, specified, and configured by the user. On the other hand, the **Scenario Execution** component executes a defined scenario by generating MQTT traffic or processing the actuator's commands. The following subsections detail how the tool's main features are implemented.

3.1. Time-Driven Sensors

Time-driven sensors send data periodically to report a certain state. This state information consists of a value that varies randomly within a range (e.g., temperature, humidity, and light sensors, for example). In MQTTProvider, the implementation of time-driven sensors is similar to that of SenSE [Zyrianoff et al. 2017] and IoT-Flock [Ghazanfar et al. 2020].

[2]https://vaadin.com/

[3]https://www.docker.com/

Therefore, when creating a new device (*"Device Templates > Random Sensors"*), the user must define the following parameters:

- The sensor **name** and **description**;
- **Object ID**: consists of the parameter to which the value will be associated in the MQTT payload. FIWARE maps this parameter to a context attribute. All devices registered in FIWARE have context attributes, such as the "temperature" attribute on a UHT sensor that stores temperature values. Suppose we define the object id "t" for this attribute. When reading from an MQTT broker, the IoT Agent will search the payloads for object IDs. Once it finds the payload a "t" with an associated value (e.g.,"t|30"), it will store this value ("30") in the "temperature" attribute;
- **Data type**: the type of data that will be sent by the sensor, which can be chosen between basic types (int, float, boolean, and char);
- **Periodicity**: the frequency (in seconds) with which sensors send data (e.g., every 10 seconds);
- **Minimum** and **maximum** range values.

This kind of sensor is the simplest type and is the most implemented. It is suitable for carrying out performance tests. However, it presents limitations in generating realistic workloads for IoT platforms. For this, other types of sensors are necessary.

3.2. Mobile Devices

Simulation of Urban MObility (SUMO) is an open-source multimodal traffic simulation package capable of handling complex scenarios with different vehicles. SUMO also allows entities other than vehicles, such as pedestrians, and interfaces with tools for creating scenarios. After completing the Scenario, it is possible to carry out the simulation, which can generate a dataset (Extensible Markup Language (XML) file) that describes the position of each vehicle at each time point in the simulation.

Once the dataset is available, the MQTTProvider user must create an instance of a SUMO trace (table *sumo_trace*) in "*Datasets > SUMO*". Next, the user must import the XML file by clicking the "*Upload File*" button. Each dataset can be associated with one or more instances of Mobile sensors.

A Vehicle class was developed to monitor a specific vehicle based on its ID in the simulation to implement the Mobile sensor operation. It stores the vehicle's coordinates at a given instant and has the logic to update its position. Moreover, a MobileSensorDevice class plays the role of the sensor present in the vehicle. It receives simulated data from the corresponding Vehicle class and builds and publishes the message on the specific topic.

3.3. Event-Driven Devices and Actuators

Event-driven sensors and **Actuators** pave the way to generate realistic workloads for IoT platforms. In MQTTProvider, they simulate people's interaction with devices while moving along different routes. Event-driven sensors send data when a state changes caused by the occurrence of an event. For example, people's entry into space changes the state of motion and temperature sensors. On the other hand, Actuators are devices that convert energy into motion or mechanical force.

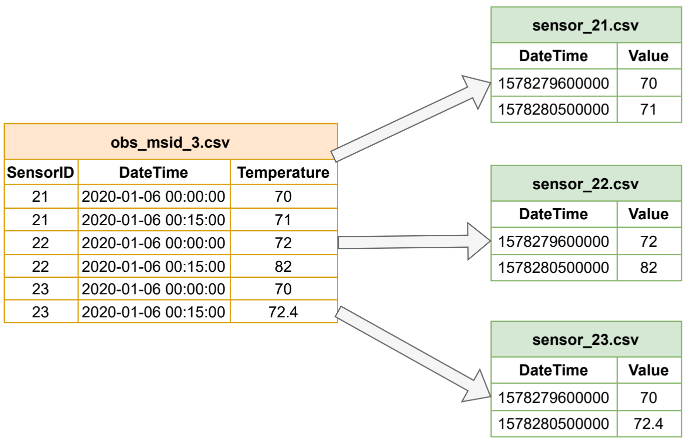
To simulate Event-driven sensors (e.g., Radio Frequency Identification (RFID), Passive Infrared Sensor (PIR), temperature, pressure, door/window contact, motion, and more) and actuators, MQTTProvider allows one to import data from the dataset generated by the SmartSPEC tool. SmartSPEC [Chio et al. 2023] employs an event-driven methodology to produce realistic and customizable datasets for smart spaces, encompassing elements such as occupancy, trajectories, and sensor data within real-world environments at both building and city scales. The framework adopts a versatile semantic model representing a smart space ecosystem, including spaces, individuals, events, and sensors. To streamline model definition, SmartSPEC utilizes available sensor data (e.g., WiFi/Bluetooth) as input, employing machine learning algorithms to derive higher-level metamodels that outline the characteristics of a smart space. Subsequently, these metamodels harness semantics to generate synthetic datasets, capturing space occupancy, occupant trajectories, and corresponding sensor observations.

Once the dataset is available, in MQTTProvider, the user must create an instance of a SmartSPEC dataset (table *smart_spec_dataset*) in "*Datasets > SmartSPEC*", providing only a name and a description. Next, the user must import some files from the dataset by clicking the "*Add Files*" button and then "*Upload Files...*"). The following files must be imported:

- **MetaSensors.json**: Defined by the SmartSPEC user. This file contains a list of sensor categories (e.g., Temperature, Door, WiFi APs, etc.);
- **Sensors.json**: Defined by the SmartSPEC user. This file contains a list of sensors, each one with its coordinate and being part of a category defined in "MetaSensors.json";
- **Spaces.json**: Defined by the SmartSPEC user. This file contains a list of spaces in the Scenario that will be simulated. Each space contains a description, capacity, coordinates, and a list of other neighboring spaces;
- **obs_msid_x.csv**: Dataset file generated by SmartSPEC. Based on the data provided by users, SmartSPEC will generate a file for each sensor category (we will have a file ("x") for each id of "MetaSensors.json"). In each of these files, we will have a list of data sent by the sensors of the same category, organized in time.
- **data.csv**: Dataset file generated by SmartSPEC. This file reports the start and end times of people's participation in certain events in certain spaces. In other words, it tells us the story of people's interaction with the spaces in the setting. In SmartSPEC, this data is used to generate the data reported by the sensors. MQTTProvider will be used to simulate the triggering of commands to the actuators.

Once these files have been uploaded, the next step will be to import the data into the database and generate the necessary files to create the event-driven sensor and actuator templates. In this case, on the "*Datasets > SmartSPEC*" screen, the user must click on the "Generate Data" button for the respective dataset and then click on "Generate Data". Once this is done, MQTTProvider will perform two macro-activities. Are they:

- **Creating instances in the Database:** MQTTProvider will create instances in the tables *smart_spec_meta_sensor*, *smart_spec_sensor* and *smart_spec_space* , according to the records in the files "MetaSensors.json", "Sensors.json" and "Spaces.json", respectively. Such data will be used to create files in the next macro-activity and create new templates for event-driven sensors and actuators.



obs_msid_3.csv		
SensorID	DateTime	Temperature
21	2020-01-06 00:00:00	70
21	2020-01-06 00:15:00	71
22	2020-01-06 00:00:00	72
22	2020-01-06 00:15:00	82
23	2020-01-06 00:00:00	70
23	2020-01-06 00:15:00	72.4

sensor_21.csv	
DateTime	Value
1578279600000	70
1578280500000	71

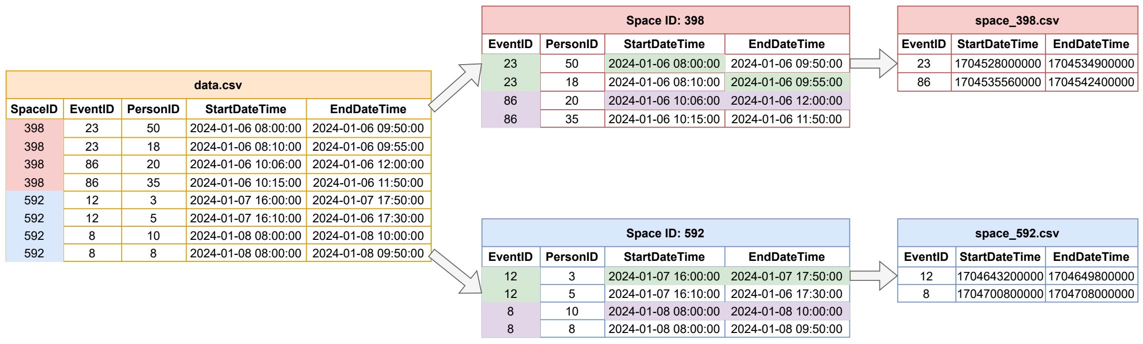
sensor_22.csv	
DateTime	Value
1578279600000	72
1578280500000	82

sensor_23.csv	
DateTime	Value
1578279600000	70
1578280500000	72.4

Figure 2. Generating Event-driven sensors dataset

- **Creation of new files in the dataset repository:** From the dataset files "obs_msid_x.csv" and "data.csv", MQTTProvider will derive other CSV files necessary to simulate the data generated by the event-driven sensors and the triggering of commands to the actuators, respectively. Figura 2 illustrates the first case. Each "obs_msid_x.csv" derives a "sensor_x.csv" file for each sensor ID X. In each of these new CSVs, we will have two columns. The first consists of the timestamp in which the sensor generated the data, while the second column consists of the generated value (e.g., temperature, person ID, and more). Each dataset can be associated with one or more instances of Event-driven sensors. We will detail the process of creating actuator datasets later.

To simulate the commands for the actuators, we use data from the "data.csv" file. The general idea is that, for each event in space, the first person to arrive activates the actuators (e.g., unlock a door, turn on the lights and air conditioning). The last person to leave also activates them (e.g., lock a door, turn off the lights and air conditioning). In this sense, the MQTT Provider generates a CSV file for each space ID from "data.csv". This organization enables the association of actuators to a given space and the simulation of commands. Figure 3 shows how the algorithm works.



data.csv				
SpaceID	EventID	PersonID	StartDateTime	EndDateTime
398	23	50	2024-01-06 08:00:00	2024-01-06 09:50:00
398	23	18	2024-01-06 08:10:00	2024-01-06 09:55:00
398	86	20	2024-01-06 10:06:00	2024-01-06 12:00:00
398	86	35	2024-01-06 10:15:00	2024-01-06 11:50:00
592	12	3	2024-01-07 16:00:00	2024-01-07 17:50:00
592	12	5	2024-01-07 16:10:00	2024-01-06 17:30:00
592	8	10	2024-01-08 08:00:00	2024-01-08 10:00:00
592	8	8	2024-01-08 08:00:00	2024-01-08 09:50:00

Space ID: 398			
EventID	PersonID	StartDateTime	EndDateTime
23	50	2024-01-06 08:00:00	2024-01-06 09:50:00
23	18	2024-01-06 08:10:00	2024-01-06 09:55:00
86	20	2024-01-06 10:06:00	2024-01-06 12:00:00
86	35	2024-01-06 10:15:00	2024-01-06 11:50:00

space_398.csv		
EventID	StartDateTime	EndDateTime
23	1704528000000	1704534900000
86	1704535560000	1704542400000

Space ID: 592			
EventID	PersonID	StartDateTime	EndDateTime
12	3	2024-01-07 16:00:00	2024-01-07 17:50:00
12	5	2024-01-07 16:10:00	2024-01-06 17:30:00
8	10	2024-01-08 08:00:00	2024-01-08 10:00:00
8	8	2024-01-08 08:00:00	2024-01-08 09:50:00

space_592.csv		
EventID	StartDateTime	EndDateTime
12	1704643200000	1704649800000
8	1704700800000	1704708000000

Figure 3. Generating actuators dataset.

For each space ID, select all entries grouped by event ID and order them by start date time. The order by start date time aims to create a timeline of people ingressing in a specific event and space. The result consists of a Map in which each key represents an event id, and each value is a list of filtered and ordered entries.

Based on the above result, create the entries for the specific “space_x.csv”, where x is a space id. Each entry in the CSV consists of an event that happens in that space, as well as the timestamp when the first person arrives and the last person leaves. Furthermore, such events are ordered in time, from the first to the last, that occurs in a given space. Each dataset can be associated with one or more instances of actuators. Furthermore, MQTTProvider also allows the creation of commands, which can be associated with one or more actuators. Every command has a name and a state that it generates. For example, the “on” command generates the “ON” state in the actuator.

It is worth mentioning that for both event-driven sensors and actuators, timestamps are used by the MQTTProvider to determine when to send MQTT messages. Furthermore, to create instances (*“Device Templates > ED Sensors”* and *“Device Templates > Actuators”*), the user must provide a name, description, object ID (even the time-driven one), and dataset that will be used. In the case of actuators, we will still have to fill in the periodicity field, which indicates the frequency in seconds at which it will send the current state, and the default state field, which means the initial state of the actuator. Finally, actuators can also be created without a dataset, i.e., commands must be triggered manually from an IoT platform.

3.4. Complex Devices and Scenarios

To start the traffic generator (*“Experiments > Run Generator”*), the user must specify the MQTT broker address, the port to establish the connection, the execution time in seconds, and, most importantly, the Scenario. The **Scenario** is nothing more than a representation of the IoT infrastructure, which encompasses all devices with their sensors and actuators, which in turn send and process MQTT messages.

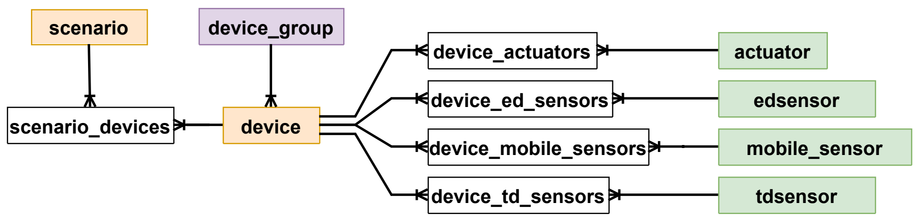


Figure 4. Complex Devices and Scenarios

In this context, Figure 4 shows the Entity Relationship (ER) Diagram that illustrates how MQTTProvider defines a traffic scenario. In MQTTProvider, a Scenario (*“Experiments > Scenarios”*) instance contains one or more Device instances. The Device (*“Experiments > Devices”*) represents a complex device, encompassing one more instance of the sensors and actuators described previously. For FIWARE compatibility, each Device instance must be associated with a Device Group. A Device Group defines a group of devices of the same category, such as a set of UHT sensors, WiFi APs, and more. Each Device Group instance contains a name, a description, and an API Key, which must be unique for each instance. The API Key will be used to form the topic, as described at the beginning of this section.

4. Experiments

This section presents our experiments for generating synthetic MQTT traffic for Smart Campus environments using our packet-level traffic generator. In this sense, we validate MQTTProvider through its use to assess the scalability properties of the FIWARE platform [FIWARE 2024].

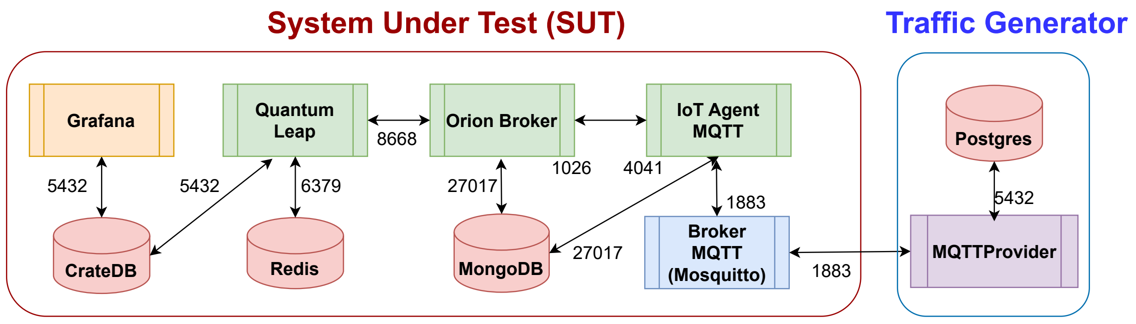


Figure 5. Test System

To conduct the experiments, a production-like instance of the FIWARE platform pre-populated with the resources (devices) available on the Campus and their initial states is required. For this, we set up an infrastructure capable of collecting and storing sensor data in a time series database and provide a visualization tool to retrieve time-based aggregations of such data. Figure 5 shows our Test System. In one machine, we configure the MQTTProvider that publishes its data using the Ultralight 2.0 format over MQTT. On another machine, we deploy our System Under Test (SUT). It consists of **Eclipse Mosquitto**, which will process messages published from MQTTProvider and three FIWARE Generic Enablers: (i) the **Orion Context Broker**, which will receive requests using NGSI-v2 and manage context data; (ii) the **IoT Agent** which will receive northbound measurements from Mosquitto in Ultralight 2.0 format and convert them to NGSI-v2 requests for the Orion; and (iii) the **QuantumLeap** that subscribes to context changes and persisting them into a **CrateDB**[4] database. Then, we use **Grafana**[5] as a time series analytics tool. It is worth mentioning that since HTTP requests initiate all interactions between the elements, those entities can be containerized and run from exposed ports. Therefore, to keep things simple, all components run using Docker[6].

Moreover, to enable service monitoring for collecting metrics, we raised **Prometheus** (integrated with Grafana) and three (3) Prometheus exporters: (i) **cAdvisor** (Container Advisor) to measure the resource usage and performance characteristics of the running containers; (ii) **Node Exporter** to collect hardware and OS metrics; and (iii) **SQL Exporter** to exposes metrics gathered from CrateDB.

In this scenario, we evaluated the scalability of our SUT using two machines. The first, referred to as **"Desktop"**, simulates a device operating at the network's edge, featuring a Dell Desktop with a 1x Intel Core i7-2600 3.40GHz 8MB (4-core), 8GB RAM, a 500GB SATA-III hard drive, a Gigabit Ethernet NIC, and Debian 11.6 (Bullseye). The second, known as **"Server"**, is a high-capacity Dell R610 Power Edge server equipped

[4]https://cratedb.com/
[5]https://grafana.com/
[6]https://www.docker.com/

		1000		2000		3000	
Metric	**Target**	**Mean**	**Error**	**Mean**	**Error**	**Mean**	**Error**
CPU Usage (%)	Desktop	33.84	(32.41, 35.28)	36.14	(35.38, 36.90)	42.12	(41.15, 43.09)
	Server	10.20	(9.46, 10.94)	12.14	(11.87, 12.41)	17.13	(16.79, 17.46)
RAM Usage (GiB)	Desktop	3.568	(3.565, 3.571)	3.802	(3.792, 3.811)	3.958	(3.949, 3.967)
	Server	17.013	(16.983, 17.042)	17.218	(17.202, 17.233)	17.347	(17.334, 17.360)

Table 1. Pyshical resources usage.

with 2x Intel Xeon E5645 (6-core) processors, 64GB RAM, a 500GB SATA-III SSD, a Gigabit Ethernet NIC, and Debian 11.6 (Bullseye).

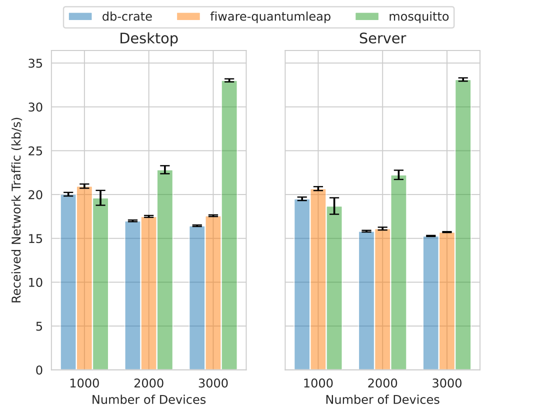
The MQTTProvider is a multithreaded application (one thread for each simulated device) and requires a machine with high computational capacity to run thousands of devices (> 1000). Therefore, we create one Virtual Machine with the following configurations: 15 vCPUs, 20 GB RAM, 70GB disk space, NIC Gigabit Ethernet, and Ubuntu 22.04. With this setup, we were able to simulate more than 3000 devices.

To generate traffic, we created ten (10) Device Groups in MQTTProvider divided between sensors (UHT, Gas, Door Contact, Mobile, Current, WiFi, and NFC) and actuators (Door Lock, Lamp, and Air Conditioner). It is worth mentioning that the devices created for the WiFi and Door Contact groups are Event-driven sensors, while those made for the Mobile group are Mobile Sensors; that is, their data comes from datasets generated by SmartSPEC and SUMO, respectively. Concerning SmartSPEC, we used one of the datasets available that describes a Smart Campus scenario. In the case of SUMO, we simulated a scenario describing the Federal University of Ceara bus routes. For the other groups, we created complex devices composed of multiple time-driven sensors or actuators with time-driven sensors. For example, a UHT device sends temperature (t) and humidity (h) measurements, while a door lock listens for lock or unlock commands and periodically sends information about its current state.

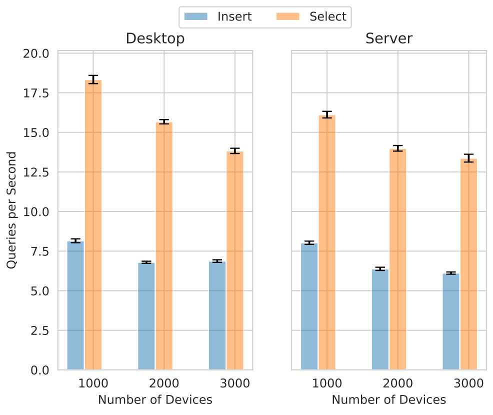
We performed three (3) rounds of the experiment for each SUT machine (Desktop and Server). The first round simulates 1000, the second 2000, and the third 3000 devices. It is worth mentioning that, for each simulation, we created the same number of devices for each of the 10 Device Groups. For example, for 1000 devices, we simulate 100 devices from each group. The idea of this workload composition is to generate different traffic profiles to evaluate essential aspects of the system and to ensure that the observed results are reasonable estimates for the general behavior of the system. Each round ran for 40 minutes and, from Grafana, we collected samples from the last 15 minutes, as we consider that the first 25 minutes is the time for the steady state, given that we need to create thousands of threads and the SUT needs to adapt to all the traffic generated.

It is worth noting that we collected data from the following metrics: (i) **Physical Resource Usage**: CPU and RAM memory; (ii) **Received Network Traffic Rate (kb/s) per Container**: the total rate (per second) of network bytes received over a 5-minute interval; (iii) **Queries per Second**: the total per-second rate, over a 1-minute interval, of different types of queries in CrateDB; (iv) **Disk Writes Completed Rate (io/s)**: the rate of disk write operations completed per second over a 1-minute interval; and (v) **Average Query Duration (ms)**: average query duration over the last 1 minute.

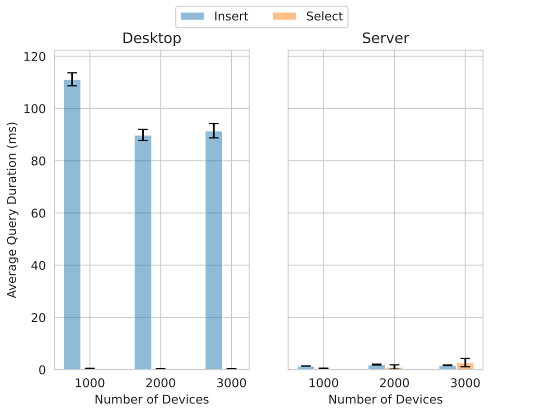
Table 1 shows the average physical resource usage. Regarding the CPU, we observed a constant increase on both targets as we increased the number of devices. Moreover, on neither machine, we could not even remotely exhaust their processing capabilities (maximum 42.12% on Desktop), which could cause instabilities in the SUT operation. We observed the same situation with memory consumption. However, the variation from one experiment to another is ridiculous, which shows that the SUT operates with minimal RAM requirements. From this result, we could conclude that the system behaves well with this variation in device demand. Nevertheless, we will see later that this is not the case.



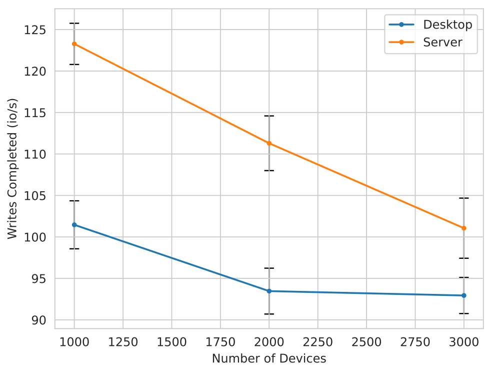
(a) Received Network Traffic per Container (kb/s)



(b) Queries per Second in CrateDB



(c) Average Query Duration (ms)



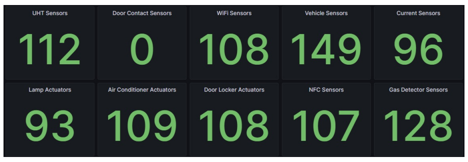
(d) Disk Writes Completed (io/s)

Figure 6. Experiment results.

Figure 6(a) shows the rate (kb/s) of received network traffic per container. For this metric, we only present the results for the Mosquitto, Quantum Leap, and CrateDB containers, as we only want to visualize the beginning (data input) and the end (data available to Grafana) of the data flow in the SUT. The results indicate that as we increase the number of devices, the traffic at Mosquitto increases but decreases in the other containers. Furthermore, the two machines present similar values, as shown in Figure 6(b). In this case, the rate of Insert requests in CrateDB offers similar values on the Desktop and Server. As the number of devices increases, the variation in this rate is ridiculous, giving an idea of constancy. Such behavior may indicate a performance bottleneck in Orion Context Broker, which sends notifications to Quantum Leap whenever updates to device data occur. Such a bottleneck can cause a noticeable delay in making sensor data available to

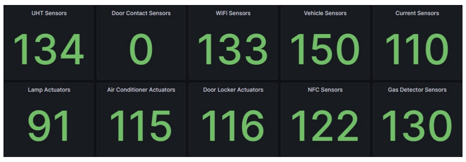
Grafana.

In addition to the performance problem described above, which is caused by software, we identified another whose cause is the difference in hardware on the two machines. From Figure 6(c), we can verify that the average query duration on the Desktop is approximately 100 times greater than on the Server. From Figure 6(d), we can conclude that the problem lies in the disk storage technologies used, as the number of disk write operations is always bigger on the Server. While the Desktop works with a Hard Disk (HD), the Server uses a Solid-State Drive (SDD), which has proven to perform better, even though both operate on SATA-III. This type of bottleneck will cause the delay described above to be greater on the Desktop.



UHT Sensors	Door Contact Sensors	WiFi Sensors	Vehicle Sensors	Current Sensors
112	0	108	149	96
Lamp Actuators	**Air Conditioner Actuators**	**Door Locker Actuators**	**NFC Sensors**	**Gas Detector Sensors**
93	109	108	107	128

(a) Desktop



UHT Sensors	Door Contact Sensors	WiFi Sensors	Vehicle Sensors	Current Sensors
134	0	133	150	110
Lamp Actuators	**Air Conditioner Actuators**	**Door Locker Actuators**	**NFC Sensors**	**Gas Detector Sensors**
91	115	116	122	130

(b) Server

Figure 7. Number of reported devices per Device Group (last 5 minutes).

Figure 7 shows the effect of this delay. It presents Grafana panels for the Desktop (7(a)) and Server (7(b)) that indicate the number of devices that reported data to CrateDB in the last 5 minutes, measured at the end of the experiment with 2000 devices. It is possible to verify that both targets suffer from the FIWARE delay. However, the Server reports a greater number of devices, as it operates with SSD.

Finally, after thorough experimentation and detailed performance assessments, MQTTProvider has demonstrated its effectiveness in evaluating the scalability of IoT platforms, pinpointing critical performance bottlenecks, and providing a powerful tool for the development and testing of IoT systems. Our research primarily examines a Smart Campus scenario. Yet, due to its versatile device creation capabilities, we argue that one can broaden the use of this tool to include a range of applications such as smart city planning, IoT network optimization, and the creation of intricate IoT solutions, thereby making a significant impact on IoT research and development.

5. Conclusion

The MQTTProvider is a versatile and powerful IoT traffic generator designed for large-scale distributed systems, capable of simulating a wide range of smart spaces with diverse sensor categories and actuators. Its compatibility with various IoT middlewares, particularly FIWARE, enhances its adaptability for IoT ecosystems. We conducted experiments on two different machine setups to verify the use of our tool in evaluating the scalability characteristics of an FIWARE platform. The outcomes enabled us to pinpoint performance bottlenecks at both the software and hardware levels. The MQTTProvider source code can be found in GitHub[7], as well as detailed guidance on system architecture, minimal hardware requirements, installation, use, and experiment replication. For future work, it would be beneficial to extend the MQTTProvider's capabilities to include more

[7]https://github.com/SmartCampus-UFC/SmartUFC-MQTTProvider

advanced features, such as integrating AI and machine learning algorithms for predictive analysis and adaptive traffic generation. This could involve developing models more accurately mimicking real-world IoT traffic patterns based on historical data and real-time analytics.

References

Ali, M. I., Gao, F., and Mileo, A. (2015). Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In *The Semantic Web - ISWC 2015*, pages 374–389, Cham. Springer International Publishing.

Chio, A., Jiang, D., Gupta, P., Bouloukakis, G., Yus, R., Mehrotra, S., and Venkatasubramanian, N. (2023). Smartspec: A framework to generate customizable, semantics-based smart space datasets. *Pervasive and Mobile Computing*, page 101809.

de M. Del Esposte, A., Santana, E. F., Kanashiro, L., Costa, F. M., Braghetto, K. R., Lago, N., and Kon, F. (2019). Design and evaluation of a scalable smart city software platform with large-scale simulations. *Future Generation Computer Systems*, 93:427–441.

FIWARE (2024). FIWARE - Open APIs for Open Minds — fiware.org. `https://www.fiware.org/`. [Accessed 08-01-2024].

Ghazanfar, S., Hussain, F., Rehman, A. U., Fayyaz, U. U., Shahzad, F., and Shah, G. A. (2020). Iot-flock: An open-source framework for iot traffic generation. In *2020 International Conference on Emerging Trends in Smart Technologies (ICETST)*, pages 1–6.

Martínez-Ballesté, A., Pérez-Martínez, P. A., and Solanas, A. (2013). The pursuit of citizens' privacy: a privacy-aware smart city is possible. *IEEE Communications Magazine*, 51(6):136–141.

Nguyen-An, H., Silverston, T., Yamazaki, T., and Miyoshi, T. (2020a). Generating iot traffic: A case study on anomaly detection. In *2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN*, pages 1–6.

Nguyen-An, H., Silverston, T., Yamazaki, T., and Miyoshi, T. (2020b). Generating iot traffic in smart home environment. In *2020 IEEE 17th Annual Consumer Communications Networking Conference (CCNC)*, pages 1–2.

Pappachan, P., Degeling, M., Yus, R., Das, A., Bhagavatula, S., Melicher, W., Naeini, P. E., Zhang, S., Bauer, L., Kobsa, A., et al. (2017). Towards privacy-aware smart buildings: Capturing, communicating, and enforcing privacy policies and preferences. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 193–198. IEEE.

Shukla, A., Chaturvedi, S., and Simmhan, Y. (2017). Riotbench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):e4257. e4257 cpe.4257.

Zyrianoff, I., Borelli, F., and Kamienski, C. (2017). Sense–sensor simulation environment: Uma ferramenta para geração de tráfego iot em larga escala. *Simpósio Brasileiro de Redes e Sistemas Distribuídos (SBRC).*