ABSTRACT

Despite the high growth of the Internet of Things and the multitude of applications that use the information generated, it is estimated that 90% of these data are not yet fully used. This is because IoT systems are based on heterogeneous networks and devices with resource restrictions, which cannot execute significant processing. So one of the alternatives that have been addressed is the use of cloud applications, where the data can be appropriately processed, stored, and analyzed. In the cloud, applications can then employ resource elasticity and be developed in microservice architectures.

In state-of-the-art, the main efforts to provide portability and efficient treatment of IoT applications are described. However, a platform that provides protocol portability, resource elasticity using microservices architecture for the efficient execution of IoT applications has not yet been implemented. In this context, this work presents SaaSport, which among its contributions, will allow the efficient treatment of data generated by IoT devices that use the MQTT and CoAP protocols. The final result of this work demonstrates that the model provides an efficient treatment of the data generated in IoT environments.

KEYWORDS

IoT, Cloud Computing, MQTT, CoAP, Portability

1. INTRODUCTION

As described in [Pierleoni et al. 2020], the Internet of Things (IoT) aims to connect the real world composed of devices, sensors, and actuators to the virtual world to interconnect devices, generating information from the collected data. However, the devices, in general, have the computational power and limited storage capacity. Cloud computing allows access to a set of shared and configurable resources offered as services, with an almost unlimited capacity in terms of storage and computing. One of the main challenges faced in IoT is the high degree of heterogeneity in terms of the communication resources of the devices, protocols, technologies, and hardware [Yacchirema Vargas and Palau Salvador 2016], users and applications, to interpret the data of the devices need to know details of each protocol, which is not trivial, as it requires time. With each launch or upgrade of a protocol, there is a new learning effort.

With the continuous development and evolution of IoT, monolithic applications have become much more extensive and even more complex. This leads to low scalability, extensibility, and maintainability. In response to these challenges, the microservice architecture must be introduced in IoT applications due to its flexibility, light, and flexible coupling [Sun et al. 2017]. Still, there is a concern with software engineering and scalability so that the number of users and IoT devices do not interfere with the system's quality of service. As [Bansal and Kumar 2020] describe, the main concerns and research areas on IoT platforms are scalability, personalization, and security.
Studies carried out in the area focus on implementing gateways for the interconnection of devices with cloud environments that do not implement elasticity and have a monolithic architecture. These works explore the Message Queuing Telemetry Transport (MQTT), Constrained Application Protocol (CoAP) protocols, or specific application domains. Among the works, it is possible to mention [Dizdarevic’ et al. 2019], [Khaled and Helal 2019], and [Lai et al. 2019]. However, it is noted that the related works do not deal with the interoperability and interconnection of heterogeneous devices at the device, protocol, and data level. Moreover, as described [Lai et al. 2019], traditional centralized architectures do not have the necessary flexibility to deal with heterogeneous devices efficiently.

Among the gaps that the work seeks to fill are the efficient and high-performance treatment of communication portability for devices that use the protocols defined in the context of IoT networks. In this context, the SaaSport model was developed, a new middleware for IoT and cloud computing, which offers an abstraction layer of communication between devices using the MQTT and CoAP protocols, with Elasticity and Microservices so that through an API, it is possible for users to have access to information generated by devices and sensors, in the cloud. Thus, the objective of this work is to develop a model to allow the portability of the MQTT and CoAP protocols used in IoT. With this, it will be possible for users to access the data collected by the devices through an API available in the cloud, using microservices and elasticity.

This work is organized in a way. Section 2 presents related works. Subsequently, in section 3, the SaaSport model is described. Right after, in section 4, the model evaluation methodology is treated, where the aspects related to methods applied for model validation are specified. Furthermore, in sections 5 and 6, we present the experiments, conclusions, and perspectives of future works, respectively.

2. RELATED WORK

In this section, we present some initiatives, works, and solutions related to the proposed model. The works were selected based on the criteria: (I) origin IEEE1, ScienceDirect2, ResearchGate3, and ACM4, (II) publications in the period from 2016 to 2020, (III) search result for the keywords "Cloud Computing", "IoT", "MQTT", "CoAP", "Portability", "Interoperability" and "Microservices".

In the work of [Yacchirema Vargas and Palau Salvador 2016] a new Smart IoT Gateway was implemented, designed to allow interconnection and interoperability between heterogeneous devices in the IoT. The proposed gateway offers advantages such as: connectivity of different protocols and traditional communication technologies (Ethernet) and wireless (ZigBee, Bluetooth, Wi-Fi); uses a flexible protocol that translates all the data obtained from the different sensors into a uniform format, performing the analysis of the data obtained from the rules based on the environment related to the different types of sensors; uses a lightweight and ideal protocol for using devices with limited resources to provide an information environment; and provides local data storage for later use and analysis.

In [Martins et al. 2017] RAISE middleware was reengineered, proposing a new architecture with microservices and the use of cloud computing. The change envisaged an increase in the availability and reliability of the application. The authors point out that the resources offered by the cloud are practically unlimited and using load balancing and client redirection strategies, it is possible to provide middleware services around the clock for intelligent objects. Infrastructure automation is an essential tool to provide service elasticity. The infrastructure can be scaled to meet the specific needs of each microservice without affecting another microservice. The use of containers ensures the reproducibility of the software in the way the environment was defined. Another interesting aspect is that it allows the standardization of the service execution environment since aggregating other images construct the images.

[Sun et al. 2017] a general structure of the microservices system is proposed for the IoT application, which is a better scalable, extensible and sustainable architecture, the authors present the system design and related microservices and emphasize the leading service and communication of the device, from the service layer to the physical layer. It has a better ability to support interoperability and accommodate heterogeneous objects. In addition, this structure can quickly achieve more application integration, such as automation, intelligence, geographic service and Big Data.
The proposal by [Righi et al. 2018] includes a literature review, where the gap in addressing extensibility and interoperability in the IoT scenario is observed. The authors explored extensibility by providing an independent IoT protocol model, working with different communication models, including synchronous and asynchronous semantics. The scientific contribution appears in not providing another API but in maintaining the current ones, thus allowing communication between different technologies effortlessly. A prototype was developed that includes HTTP, MQTT, and CoAP technologies. The tests revealed a small overhead of IoT++ in the translation and forwarding of messages between the aforementioned protocols.

The structure proposed by [Pratik et al. 2018] provided a solution to the challenge of interoperability between objects that use the MQTT, CoAP protocols, and OIC data models. A client outside the network can obtain data from the sensor and actuator arriving at CoAP and sending GET and POST requests. In contrast, local clients (or actuators in the network) can obtain data from the MQTT broker through a topic subscription for taking local decisions. Since MQTT and CoAP have built-in security features, such as authentication, encryption, etc., they can add security and make the structure secure. Furthermore, as the data is stored in a tuple store database, SPARQL queries can be sent via CoAP requests in the future, assisting in troubleshooting.

In [Khaled and Helal 2019], the authors describe that systems research in the IoT is changing priorities to explore the explicit “architecture of things” that promote and allow friction-free interactions. They introduce the Atlas IoT communication structure, allowing interactions between things that speak similar or different communication protocols. The translator allows continuous communication between the CoAP, REST, HTTP, and MQTT protocols. A framework has been designed to facilitate interoperability between devices without taxing the performance of communicating homogeneously. The framework uses the concept of the topic and uses a meta-topic hierarchy to map and guide translations. The work described the architecture and a detailed benchmarking study measuring energy consumption and the characteristics of different aspects of the structure on real hardware platforms.

In the work of [Lai et al. 2019], the authors describe how microservice architectures can be adopted to create IoT services for multi-mobility in a smart city. Microservice architectures implement small, limited resources in a running process; Independent microservices can be deployed separately in a distributed system. An architectural draft has been proposed for general-purpose Internet of Things applications. Thanks to the choice of the microservice paradigm, the architecture can interact with a wide range of heterogeneous IoT devices while implementing scalability by design. On this basis, a Web application has been developed with a set of mobility services in mind for the multi-mobility of citizens in a smart city.

As described [Pamborisi et al. 2020], due to the heterogeneity of devices, the complexity of developing applications requiring the collection and sharing of data across multiple IoT devices is high, as developers need to be familiar with a diverse set of services and supported APIs. The authors developed a flexible and lightweight middleware that offers a unified API to help develop applications that use multiple heterogeneous IoT devices. It abstracts much of the complexity involved in orchestrating different devices at run time. At the same time, it avoids the aforementioned warnings of existing approaches through a simple and efficient design, but one that offers a rich set of resources to developers.

This section presented the works related to the research, several possible solutions were verified. However, as described [Dizdarevic et al. 2019], current solutions are far from ideal, which creates challenges and exciting opportunities in new architectures that will undoubtedly need to combine IoT, Fog, and Cloud Computing systems to meet the requirements of future applications. In Table 1, it is possible to perceive a summary view of them, encompassing items such as capacities, protocols, and target architecture. Making a comparative analysis concerning the works related to this project, a gap is observed where there is still no proposal for portability of the MQTT and CoAP protocols using the elasticity of cloud resources and microservice architecture.
3. SAASPORT MODEL

This section introduces the SaaSport model, which will explore the portability of protocols, resource elasticity, and microservice architecture for efficient execution of IoT applications. To describe it adequately, this section has been divided into three other subsections. Section 4.1 presents the project decisions, in section 4.2, the proposed architecture is presented, which has as a legacy the models described in the section of related works, which incorporated the concept of microservices and scalability, and the description of the other definitions of configuration in the development of the work. Finally, section 4.3 demonstrates how the model works.

3.1 Project Decisions

It is relevant that the data collected by IoT devices are available for general purposes. However, a platform that provides protocol portability, resource elasticity using microservices architecture for the efficient execution of IoT applications has not yet been implemented. With this it is expected to contribute by modeling the SaaS platform in which the data collected by the IoT devices are made available in the cloud through an API, which in order to break the structural paradigm of monolithic systems, using microservices, which are an architectural and organizational model of development in which software is composed of small independent services, in architecture with a high level of interoperability and elasticity, to develop an advanced model and significantly add to the studies and proposals for the integration of IoT and Cloud Computing.

The model is a SaaS, as it will provide a service that explores the portability of IoT protocols, resource elasticity, and microservice architecture for the efficient execution of the data processing generated in IoT applications. The MQTT Broker and CoAP server are expected to send all messages collected on their networks to the cloud using a Gateway. The Gateway will subscribe to the MQTT topics and use the CoAP Observer (RFC 7641), to get the messages and relay them to the cloud via the HTTP protocol. The data will be received in the cloud through the API Gateway, responsible for forwarding the data through the MQTT-messages and COAP-messages queues, processed and translated by the MQTT Translator and CoAP Translator microservices. After being processed, the data is sent to the store-messages queue, where the Store Messages microservice will receive and persist such messages in the InfluxDb database. Then the messages will be available so that users and external applications can read data from different IoT environments through a single REST API.

3.2 Architecture

Microservices form the basis of the architecture, are the managers of the business data and provide a REST API to other services interested in consuming information made available by them. Among the functions that the architecture will perform are: (a) collecting the messages exchanged by the IoT devices, (b) forwarding
the messages, (c) processing and translating the data, (d) making the data available through an API. In order to illustrate how the communication between microservices will be implemented, Figure 1 was designed, highlighting the main components present in the model, as well as which actions each of them will trigger. In Figure 2, the most detailed architecture of the proposed solution can be seen. In which the IoT devices, allocated in their respective environments, will send the data to the Gateway, which will therefore handle the data in the cloud architecture, and make the data available through the model API for users and applications. The cloud architecture implements elasticity, replicating the instances according to the use of the service.

![Proposed Communication Model](image)

**Figure 1. Proposal communication model**

### 3.3 Operation

Data processing consists of performing tasks in the SaaSport model. The platform is designed to abstract the capture of messages through a Gateway, which makes publications in a queue, and will forward the data of each of the queues to be processed by the microservice that implements its due treatment. The components for communicating the model are detailed below:

1. **API Gateway** - responsible for transmitting messages collected to the cloud, using the HTTP protocol.
2. **RabbitMQ** - an open-source messaging server, which will control requests in queues during processing. There are 5 queues on the server, which are:
   a. `mqtt-messages` - stores the messages that will be consumed by the MQTT message translation microservice.
   b. `coap-messages` - stores the messages that will be consumed by the CoAP message translation microservice.
   c. `store-messages` - queue containing messages normalized to the key-value pattern, which will be consumed by the microservice responsible for storage.
   d. `api-client` - queue containing Rest API requests.
3. **Translate CoAP** - microservice responsible for processing data received by CoAP devices.
4. **Translate MQTT** - microservice responsible for processing data received by MQTT device.
5. **Store Messages** - microservice responsible for recording messages handled by Translates.
6. **API Rest** - responsible for making the data treated by the model available.
In order to represent the best functioning, the sequence diagram of the functioning of the model was drawn in Figure 3. In which "IoT devices" will trigger HTTP POST requests to the "gateway", which will add the request to "queue", the records in the queue will be processed and translated by "translation", will be returned to the "queue" and then stored in the "store". API Clients users will make HTTP GET requests for "endpoint", which will immediately go to the "queue" that will communicate with the database ("store") and return to the user.
4. EVALUATION METHODOLOGY

In this section, the evaluation method of the SaaSport model was documented. The section was divided into 5 parts. Where in section 5.1 the prototype is presented, in section 5.2 the infrastructure used in the tests was detailed. In section 5.3, the input data was segmented, in the sequence in section 5.4, the metrics for evaluating the model were defined. And to limit the scope of the test, the test scenarios were defined in section 5.5.

4.1 Prototype

The SaaSport model was implemented in the JavaScript language, in the multiplatform Node.js, according to details described in the model description section. The prototype had its implementation definitions made based on the technologies that have been widely used in the implementation of services in the cloud. A public Git repository was created for this work and made available at the URL <https://github.com/wagnerdevel/tcc-cloud-computing>, containing: (i) readme explaining the repository; (ii) The file structure with the source code of the architecture and experiments; (iii) Presentation (slides) of the work; (iv) Other links and support material.

4.2 Infrastructure

The model was implemented in the public cloud provider Amazon Web Services (AWS), which like [Pierleoni et al. 2020] describes one of the end-to-end Cloud-IoT platforms that currently lead the global market. Using Amazon Elastic Compute Cloud (Amazon EC2), configured with 1 instance of type T2.micro, which has 1 CPU and 1 GB of RAM, with a storage size of 8 GB. To monitor the services and generate the results, Amazon CloudWatch was used, which collects monitoring and operations data in logs, metrics and events, offering a unified view of the AWS resources, applications, and services executed. Communication between microservices uses the RabbitMQ messaging system through Amazon MQ. The data will be stored in InfluxDB, an open-source time-series database developed by InfluxData. This is optimized for fast, high availability storage and retrieval of time series data in monitoring operations, application metrics, sensor data, and real-time analysis.

ECS Auto Scaling provided the elasticity from AWS. Such service was configured using scalability policies in stages and also two CloudWatch alarms. The alarms observe a single metric of CPU utilization and send messages to EC2 Auto Scaling when the metric violates the defined threshold, thus determining when to scale the group. For example, to expand the auto-scaling group's capacity, a threshold higher than 60% of the average CPU utilization was considered. When this 60% is exceeded, an alarm will be generated for EC2 Auto Scaling, which will execute the expansion policy, adding two units of capacity to the group. The reduction in the capacity of the auto-scaling group, on the other hand, was considered a threshold below 40% of the average CPU utilization. When the CPU usage is less than 40%, an alarm will be generated for EC2 Auto Scaling, which will execute the reduction policy, removing a unit of capacity from the group.

4.3 Input Data

To validate the model, experiments were performed simulating a network of IoT devices. In which the exchange of messages is carried out, of the MQTT and CoAP application layer protocols. For testing purposes, loads were generated using scripts developed in Python with the Locust library, which represented the test scenarios, and executed in the clustered environment created on the Google Cloud Platform.

4.4 Evaluation Metrics

Metrics are ways of measuring variables and trends in behavior over a period of time, using the data collected to assess the performance of a model. In order to evaluate the SaaSport model, the metrics referring to the input data, CPU utilization, latency and average throughput were considered, observing the quality of the service with the exponential increase of the environments.
4.5 Test Scenarios

The test scenarios were designed to generate upward, downward and undulating loads. The parameters regarding the number of requisitions and scale were modified in each scenario in different periods. The following are highlighted the scenarios created to validate the model:

1. Ascending scenario where the number of requests was gradually increased during the test period.
2. Descending scenario - where the number of requests was gradually reduced during the test period.
3. Ripple scenario - where the number of requisitions was gradually increased and reduced during the test period.

5. RESULTS

To obtain the results, the SaaSport model was implemented and subjected to a battery of tests simulating IoT environments through scripts, as described in the evaluation methodology section. The data flow in the architecture was monitored with a simple load and verified through the logs if the information was following what was expected. The API Gateway abstracts the acquisition of raw data from the IoT networks to the cloud. It then sends the formatted data to the translator microservice queue, according to the application layer protocol of the request. The data is then translated and stored. The input data are created arbitrarily by each test user to simulate the data generated by IoT sensors in JSON format obtained by the gateway.

According to Auto Scaling Group policies, the prototype was configured to use a limit of up to 8 EC2 instances. The test scenarios submitted to the model obtained the results described below. The graphs of each test contain the Input data, CPU utilization, and Total instances, where it is possible to observe the behavior typical to the scenarios submitted. The X-axis of the graphs represents the interval at which the test was performed. In the Data Entry and Return Return Data graphs, the Y axis represents the number of Bytes of the loads, for which the blue line in the center traces the exact number of Bytes in each instant, which increased and decreased respectively. In the CPU Utilization graph, the Y-axis represents the percentage of usage. Moreover, in the Total Instances graph, the Y-axis represents the number of instances.

To compare the proposed model, tests were also carried out in an environment with no elasticity, composed of only one instance of type T2.micro. With the test load generated in this environment, it was possible to visualize that the latency increased according to the number of requests per second. When reaching the number of 2447 requests per second, downtime occurred. The tests performed with the scenarios previously defined in an environment configured with elasticity are presented below.

1. - Ascending scenario - the scenario that reached 10,000 users by firing requests simultaneously was configured. The test started with one user, and ten new users (spawn rate) were added per second. The result of the execution of this test scenario can be seen in Figure 4. The graphs of CPU Utilization and Total instances indicate that as the percentage of CPU utilization increased, new instances were added according to the expansion policy. Furthermore, as the CPU usage dropped, the reduction policy was implemented, reducing the number of instances by one unit.
2 - Descending scenario - the scenario that started with 10,000 users firing requests simultaneously was configured. This scenario gradually reduced the number of users during the test period until there were no more users. As shown in Figure 5, as the number of users was reduced, the CPU usage was also freed up, and, consequently, the number of instances was reduced by the reduction policy. At the end of the test, the auto-scaling group contained only one EC2 instance.

3 - Ripple scenario - where the number of requests was gradually increased and reduced during the test period, always starting from 1 user, increasing up to 5,000 users and reducing again to 1 user. Figure 6 shows the variation of loads in a wavy shape, where it is possible to observe the increase and reduction of CPU and, in parallel, the increase and reduction in the number of instances.
As requests increased, the response time also increased, and the Auto Scaling Group added new instances. The creation of a new instance takes about 300 seconds. With that, some periods had requests without sufficient resources to attend them. Then, from the total number of requests made, an average of 1.67% of these failed was calculated due to packet losses in the interval in which the EC2 instances were created. Even with the increase in the number of requests, there was also an increase in latency. With 200 requests per second, the average latency was 42 milliseconds, and with 500 requests per second, the latency increased to 91 milliseconds, disregarding the failed requests.

The final result of this work demonstrates that the model provides efficient treatment of data generated in IoT environments, which use the MQTT and CoAP application layer protocols. However, the cloud environment could be improved using a proactive elasticity approach to predict the need for resource allocation to better meet requests.

6. CONCLUSION

This work aimed to present a proposal to allow messages from IoT devices that use the MQTT and CoAP protocols to be processed and made available through a cloud API so that the portability of the information collected by the model is provided. The existing technologies to be developed in this context have a high level of complexity, and the integration of these technologies requires several validation tests. The analysis and evaluation of these are possible through real implementations, mathematical models, and simulators.

Given the specificities, this work carried out the implementation and control of the MQTT and CoAP protocols, which have been mentioned in studies and widely used in IoT systems, whether in devices with limitations, with low power consumption and in general-purpose devices, with higher processing power and higher energy consumption. The microservice model supports the addition of new services to the architecture without compromising other functions, scalability of the solution, and a high level of resilience.

Concerning research, it is noted that much is being done to boost the implementation of IoT applications. As future work, the knowledge obtained through the development of this work can be considerably expanded through the implementation of other application layer protocols that have been used in IoT devices, the scalability of the scenarios where the experiments were carried out can be further explored a security policy was developed for accessing information.
ACKNOWLEDGEMENT

The authors would like to thank to the following Brazilian agencies: CNPq, CAPES and FAPERGS.

REFERENCES


