future internet

MDPI

Article

A Sidecar Object for the Optimized Communication Between Edge and Cloud in Internet of Things Applications

Stefano Busanelli [1], Simone Cirani [1], Lorenzo Melegari [1], Marco Picone [1,*], Mirco Rosa [2] and Luca Veltri [2]

[1] Caligoo Srl, Via Don Minzoni, 112, 42043 Taneto di Gattatico (RE), Italy
[2] Department of Engineering and Architecture, University of Parma, Parco Area delle Scienze, 181/A, 43124 Parma, Italy
* Correspondence: marco.picone@caligoo.com

Received: 31 May 2019; Accepted: 30 June 2019; Published: 5 July 2019

Abstract: The internet of things (IoT) is one of the most disrupting revolutions that is characterizing the technology ecosystem. In the near future, the IoT will have a significant impact on people's lives and on the design and developments of new paradigms and architectures coping with a completely new set of challenges and service categories. The IoT can be described as an ecosystem where a massive number of constrained devices (denoted as smart objects) will be deployed and connected to cooperate for multiple purposes, such a data collection, actuation, and interaction with people. In order to meet the specific requirements, IoT services may be deployed leveraging a hybrid architecture that will involve services deployed on the edge and the cloud. In this context, one of the challenges is to create an infrastructure of objects and microservices operating between both the edge and in the cloud that can be easily updated and extended with new features and functionalities without the need of updating or re-deploying smart objects. This work introduces a new concept for extending smart objects' support for cloud services, denoted as a sidecar object. A sidecar object serves the purpose of being deployed as additional component of a preexisting object without interfering with the mechanisms and behaviors that have already been implemented. In particular, the sidecar object implementation developed in this work focuses on the communication with existing IoT cloud services (namely, AWS IoT and Google Cloud IoT) to provide a transparent and seamless synchronization of data, states, and commands between the object on the edge and the cloud. The proposed sidecar object implementation has been extensively evaluated through a detailed set of tests, in order to analyze the performances and behaviors in real- world scenarios.

Keywords: internet of things; edge computing; cloud computing; sidecar pattern

1. Introduction

Among the multitude of emerging trends in the IT industry, the internet of things (IoT) is one of the most important and certainly one that promises to have a significant impact on people's lives. The design and developments of new paradigms and architectures coping with this completely new set of challenges is imperative. The technological progress and the imprinting of research over the last few decades has made it possible to develop devices that meet the strict requirements imposed by the internet of things evolution (where size, energy consumption, and costs are key), making them increasingly smaller yet powerful. Along with the evolution of hardware, new communication protocols, software stacks, and services have been designed, in order to allow an ever-increasing number of devices to communicate with each other in an immediate and efficient way. This evolution,

which took place in an extremely short timespan, poses the problem of integrating the existing objects and infrastructures with new paradigms and implementations: as it is easy to guess, this problem is not trivial and requires to be handled in a completely separate way.

New IoT architectures and the associated design thinking is quickly evolving and moving to the full exploitation of the computation, memory, sensing, and actuation resources. Furthermore, researchers are trying to face and overcome the problems associated heterogeneity of IoT applications and systems. In this context, device virtualization solutions and platforms may definitely play a key role in enabling the desired tradeoff between flexibility and performance. At the same time, fundamental properties, such as fault tolerance and system availability, can be achieved by splitting the application logic across different layers and decomposing complex business logics into microservices [1,2]. The objective is to create a more scalable and natively redundant solutions, with a minimal impact on the single-device performance, but with high potential in the whole system. With microservices-oriented architectures, the operative functionalities are broken down into small, modular, independently deployable and loosely-coupled microelements. This reduces the integration and evolution complexity faced with traditional monolithic architecture, and increases the dynamic and opportunistic deployment of new application and services according to new requirements and needs of the system itself.

The approach presented in this paper follows this new technological trend by introducing an innovative and scalable methodology to extend smart objects functionalities without modifying the original device/object. This result is feasible through the design and deployment of one or multiple IoT sidecar objects. In the context of microservices oriented IoT applications this solution will allow to easily introduce new IoT behaviors and services making existing smart objects capable of increasing their capabilities, without requiring any direct changes to their original implementation.

The sidecar pattern is a widely established architectural paradigm for several applications. However, its adoption in IoT application (in particular if designed following the edge computing vision [3]) is a new concept that may be convenient to achieve faster and efficient evolution of systems. The IoT sidecar object paradigm is a novel definition to let existing smart objects enhance and extend their behavior by providing them a sidecar. A sidecar is a separate piece of software dedicated to handling specific issues, which are not directly related to the smart object and should be managed outside the object's scope, according to the principle of separation of concerns. It is deployed as additional component of a pre-existing smart object without interfering with the mechanisms and behaviors already implemented.

The use case detailed in this paper concerns to the extension of a smart object's behavior for the seamless integration with cloud services, such as AWS IoT and Google Cloud IoT. This paper covers all the details related to its design and implementation.

The rest of the paper is organized as follows. In Section 2, we present an overview of related works on sidecar pattern and smart object integration is provided. Section 3 provides a detailed description of the proposed architecture. Section 4 presents an extensive set of experimental results from the implemented testbed. Finally, in Section 5, we draw our conclusions.

2. Related Work

Over the last few years, the IoT has become an extremely hyped research area. A vast community of makers and developers started working both on personal and business projects, ranging from hardware prototyping (e.g., based on Arduino and Rapsberry Pi among others) to connected home automation, such as connected lights, smart plugs, domotic systems for smart homes, to smart building management systems. This huge momentum in the evolution of the IoT represents a great opportunity from a business perspective: of course, all of these systems require software for managing, monitoring, and controlling connected devices, as well as for serving a presentation layer for the end user. The simplest way to develop and deploy such systems is, of course, to leverage the cloud, as it represents an always-available and globally-reachable backend infrastructure that offers a number

of advantages for scalable distributed systems. Originally, IoT product manufacturers followed a do-it-yourself (DIY) approach and developed their own cloud-based management platforms to control their devices. However, this approach proved to be ineffective as the costs and efforts for building and managing these platforms was too high. Researchers [4,5] started moving IoT application logics and services to the cloud, and at the same time several cloud players have therefore developed new products and services to solve this issue and provide managed IoT platforms that IoT product and software developers could leverage. This is the case of Amazon'a AWS IoT Core [6], Google's GCP Cloud IoT [7] or Microsoft Azure [8], which are designed to connect devices to the cloud in an easy and secure way. The advantages of using these services are mainly related to: (i) support for scalability; (ii) a managed platform that relieves developers from maintaining and monitoring tasks; (iii) native intergration with other services, such as messaging platforms and serverless execution. From an architecture point of view, cloud IoT platforms are designed around the following principles:

- communication between devices and the cloud occurs using pub/sub communication protocols (typically Message Queuing Telemetry Transport—MQTT [9]);
- devices send telemetry and receive control messages using the pub/sub interface;
- devices embed a vendor-provided software development kit (SDK), which takes care of wrapping the platform's functions, such as authentication and messaging, in a developer-friendly set of programming interfaces;
- security is implemented using a public key infrastracture where devices use a client certificate generated by the cloud platform to authenticate against the platform's services; policies can be associated with certificates to determine the access permissions to resources;
- the state of each device is mirrored on the cloud using a "digital twin", which can be accessed using a HTTP-based interfaces [10] by external client applications;
- the cloud platform provides management Application Program Interfaces (APIs) for on-boarding and management of devices.

Typically, in order to let an existing smart object connect to the cloud platform, the developer must embed the SDK into the object's software stack, create the device's certificate, provide the certificate into the object, and then use the SDK's programming interfaces to send and receive messages. Of course, while these operations may be trivial in most cases, it is not always possible to operate directly on the smart object and change its implementation to embed an SDK. This represent a huge limitation from a development point of view and at the same time also a design restriction because since from the beginning the device should be deployed to support one or more target cloud services. We believe that this approach can be overtaken with the presented work, in particular for application scenarios where the business and the application logics can be distributed among multiple microservices and nodes.

An example of an alternative approach is provided in [11], where the authors propose a standard-based cloud IoT platform, leveraging the Constrained Application Protocol (CoAP) [12] and MQTT [9]. The work is based on the concept of IoT hub, which is a node places at the border of a constrained network, implementing the functions of service discovery; border router; HTTP/CoAP and CoAP/CoAP proxy; cache; and resource directory. The paper extends the availability of IoT Hubs by creating replicas in the cloud in order to mirror all of its functions and let applications residing outside the constrained network to access the resources managed by the IoT Hub.

The cloud of things (CoT) refers to the interaction between IoT and the cloud [13]. In [14], an architecture for integrating cloud/IoT is proposed, based on a network element, denoted as "Smart Gateway", which is intended to act as intermediary between heterogeneous networks and the cloud.

In this dynamic context an additional and fundamental element is represented by the edge/fog computing [3] vision. It brings a new approach to internet access networks by making computation, storage, and networking resources available at the edge of access networks. This improves the performance, by minimizing latency and availability, since resources are accessible even if internet access is not available [15]. Fog-based solutions aim at introducing an intermediate architectural layer where resources and applications are made available in the proximity of end devices, thus

avoiding continuous access to the cloud. Edge-based access networks are based on the presence of highly specialized nodes, denoted as Fog Nodes, able to run distributed applications at the edge of the network. Local resources are kept synchronized by multiple clones of the same machine, thus achieving a high level of reliability and load balancing. Smart management of the activation/deactivation of replicas and choice of the most appropriate fog node to run the clone allows to optimize the usage of CPU and memory available on the infrastructure, according to the specific real-time resources requirements by running applications. Major cloud IoT platforms have embraced the edge computing paradigm and introduced their very own solutions (e.g., AWS Greengrass [16] and Google Cloud IoT Edge [17]) to extend the reach of services that were targeting cloud-only deployments also to scenarios where strict requirements, such as low-latency and unstable connection conditions, require computation resources to be close to the smart objects. In order to support this approach, connected objects still have to embed a dedicated SDK, thus resulting in the same complexity that was highlighted above for the integration with cloud platforms.

Furthermore, lightweight virtualization approaches are growing faster and represent a new technological enablers of a distributed virtualization infrastructure supporting heterogeneous internet of things devices. In [2,18] authors investigated the opportunity to exploit nodes' resources through the use of docker containers. Container-based service application are independent from a specific technology or programming language consequently provide the unique possibility to develop once and deploy everywhere. In this context, container migration became also strategic for the IoT researchers in order to provide strategic advantages, in terms of resource efficiency and performance, over traditional hypervisor-based virtualization [19–21]. This approach enables the creation of lightweight containerized applications suitable for IoT services where the application logic is separated in different layers and the business login obtained through the cooperation of multiple microservices [1,2]. This combination of edge computing together with microservices and container-based solutions will represent the new technological asset for the internet of things. The next generation of applications will be scalable, dynamic and interoperable by design allowing the creation of a real cyber-physical world where everything can be discovered and interconnected.

On top of this new vision, the concept of physical object virtualization [22,23] is gaining a lot of attention in order to separate the physical and the digital worlds through the creation of virtual replicas of real objects or devices. For example, the authors of [24] propose the use virtual IoT devices on the edge for local data processing, management of physical devices, and quick actuation. The cloud is envisioned only as a way to access the edge infrastructure remotely but without the introduction of gateway and/or objects replicas and the related synchronization procedures. With our work, we would like to combine and extend this vision introducing the possibility for a smart object (real or virtual) to evolve and add news features and functionalities without modifying its core or being aware of that change.

This IoT evolution creates a context where IoT application's components can be deployed into a separate process/microservice, in order to preserve the isolation of the main application and comply with the principle of separation of concerns. The sidecar pattern [25] has been introduced to augment and improve an application's container, without interfering with the application or affecting its logic. A sidecar container adds functionality to an application container in simple, non-intrusive way. Possible add-on functions that can be implemented using this pattern are (i) logging; (ii) monitoring; (iii) health checks. The sidecar pattern is the basis for the deployment of a service mesh, such as Istio [26]. In this work, we propose to adopt a similar concept to easily extend the functions of a smart object in order to let it connect to a cloud IoT platform without requiring any change to its logic.

3. Architecture

In this section are discussed and explained in detail all the design choices made for the definition of a sidecar object dedicated to the transparent and seamless synchronization of smart objects data, states and commands with one or more cloud or remote services. As a general definition, a sidecar

object is an entity that can be attached to one or more existing objects (real or virtual) with the purpose of mirroring states and capabilities of that object(s) to one or more corresponding entities. From a different perspective, it can also be seen as an extension of the existing object(s) (hence the name sidecar) that provides as additional functionality the ability to interface seamlessly with other entities. The choice to designing and use a sidecar object, in contrast with traditional approaches of integrating new functionalities directly on the original entity, has several advantages such as:

- The original entity must not be necessarily aware of the presence of the sidecar object, as the latter runs within its isolated environment and relies on the preexisting mechanisms provided by the former: this allows to attach the sidecar object without influencing the mechanisms already implemented;
- Potentially, more than one sidecar object could be attached to an existing entity, thanks to its isolation properties, allowing the realization of modularized solutions;
- In case of errors or failures occurring inside the sidecar object, the main entity can prosecute its tasks regularly and the overall system behavior will degrade gracefully.

Figure 1 depicts a common IoT use case where a sidecar object (running for example on the same node of the object or on a dedicated external infrastructure) subscribes to a shared MQTT broker listening on the topics of interest. It has the responsibility of processing the incoming messages from the associated object and forwarding the output data to the cloud services as needed. It is important to underline that this approach does not require any modification and change to the preexisting communication mechanisms and active technologies. The smart object keeps the same Pub/Sub approach communicating fro example trough an MQTT broker without knowing the presence of the sidecar entity and its responsability.

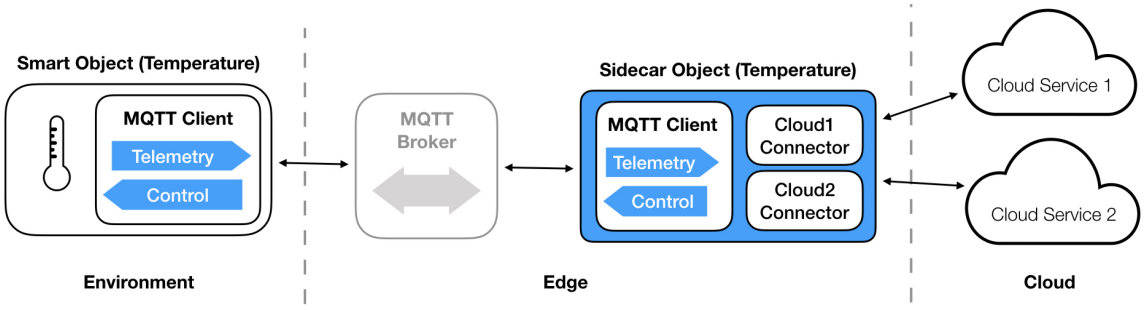


Figure 1. Application scenario involving a smart objects with enhanced capabilities through the activation of a sidecar objects for data synchronization with the cloud.

Figure 2 shows the global architecture of the sidecar object. Main components are the following:

- Smart object module: this module deals with all the interactions between sidecar and smart object, using several specialized sub-modules (as it will be explained in the following sections). There may be more than one, but usually te communication is handled by only one smart object module.
- Cloud module(s): those components handle the communication between sidecar object and cloud platforms; there can be more than one depending on the specifications, and typically an single cloud service interacts only with one of them at a time. Like the smart object module, cloud modules contains sub-modules designed specifically to achieve atomic tasks (such as connection, message publishing, etc.).
- Sidecar processors: those intermediate modules have the important role of "bridges" between the smart object module and cloud modules; each sidecar processor can handle the data message flow only in one direction (smart object to cloud or vice versa). They may or may not perform some operation on messages depending on the use case, but it is important to note that the real processing must be implemented in specific modules that will be presented in the following sections. It is directly managed by the sidecar object orchestrator.

- Sidecar orchestrator: this component is in charge for all the higher level operations regarding the coordination and synchronization of the elements described in the previous points. It never acts directly on the data flow but only sends and receives control messages to and from sub-components; it is also responsible for the initialization (gathering all the necessary startup information from ad hoc configuration files provided externally) and shutdown procedures of the sidecar object.

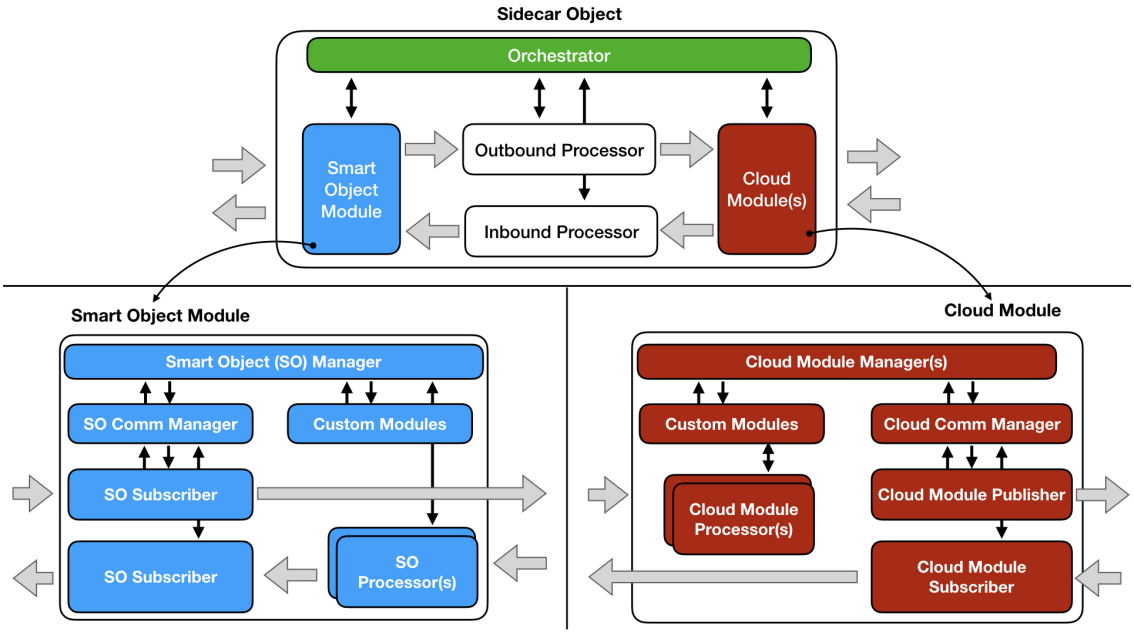


Figure 2. Overall smart object module internal architecture with the detail of smart object communication modules and cloud communications modules.

The smart object module (left side of the schemes) is composed of several sub-modules:

- Smart object manager: this component is on top of the smart object module managing chain, and supervises the behavior of the smart object communication manager (see below) and other case-specific components that vary based on the implementation. It never acts directly on the data flow but only exchanges service messages with sub-components and the sidecar orchestrator (which has full control over it).
- Case-specific components (custom modules): those sub-modules are optional, and may or may not be present depending on the use case; they never handle data messages, and communicate directly only with the smart object manager. An example of case-specific components are connection operators, often used inside modules to manage connections with external actors.
- Smart object communication manager: this element manages all the sub-modules in charge of communicating with the connected smart object, like publishers and Subscribers. Like the other managers, it never handles data messages but only communicates with adjacent modules using control messages.
- Smart object subscriber: this "operational" component has the task of subscribing to all the smart object topics specified in the sidecar object configuration, forwarding the incoming messages to the designated sidecar processor performing little or no processing on the aforesaid messages. It never has sub-modules, and is directly coordinated by the smart object communication manager using control messages.
- Smart object processor(s): those components collect all the data messages coming from sidecar processors, eventually perform an additional processing (defined by specific policies), and then forward the result to the smart object publisher; in most cases there is one smart object processor

for each topic of interest. Like other operational modules it communicates only with the manager that coordinates its behavior, which in this case is the smart object manager.

- Smart object publisher: this module publishes all the messages coming from the smart object processors to the smart object, and its implementation strictly depends on the scenario it is operating in. It also has the fundamental function of handling all the mechanisms inherent to the recovery of data messages in case of errors or faults (as will be detailed in the following sections); it is directly coordinated by the smart object communication manager.

The structure of a typical cloud module (right side of the schemes) is shown in Figure 2; as stated before, a sidecar object can have more than one cloud module, and in this section it is described the general structure that must be common for all of them. The structure is specular to the one seen for the smart object module:

- Cloud module manager: this component is on top of the cloud module managing chain, and supervises the behavior of the cloud communication manager (see below) and other case-specific components that vary based on the specific implementation. It never acts directly on the data flow but only exchanges control messages with sub-components and the sidecar orchestrator.
- Case-specific components (custom modules): those sub-modules are optional, and may or may not be present depending on the use case; they never handle data messages, and communicate directly only with the cloud module manager. An example of case-specific components are registry operators, often used to manage the synchronization of the sidecar object with the corresponding data structures provided by cloud services.
- Cloud communication manager: this element manages all the sub-modules in charge of communicating with the specified cloud service. Like the other managers, it never handles data messages but only communicates with adjacent modules using control messages.
- Cloud module processor(s): those components collect all the data messages coming from sidecar processors, eventually perform an additional processing (defined by specific policies), and then forward the result to one of the cloud module publishers; in most cases there is one loud module processor for each topic of interest. Like other operational modules it communicates only with the manager that coordinates its behavior, which in this case is the cloud module manager.
- Cloud module publishers: those modules publish all the messages coming from the cloud module processors to the cloud services, and their implementation strictly depends on the scenario they are operating in. They also have the fundamental function of handling all the mechanisms inherent to the recovery of data messages in case of errors or faults (as will be detailed in the following sections); they are directly coordinated by the cloud comm manager.
- Cloud subscriber: this "operational" component has the task of subscribing to all the cloud topics specified in the sidecar object configuration, forwarding the incoming messages to the designated sidecar processor performing little or no processing on the aforesaid messages. It never has sub-modules, and is directly coordinated by the cloud comm manager using control messages.

Operational Phases

The designed sidecar object is mainly characterized by three Operational Phases (Startup, Runtime, Fault Handling/Recovery and Shutdown) described in this subsection. Figure 3 shows the Sequence Diagram of the typical startup procedure performed by the sidecar object.

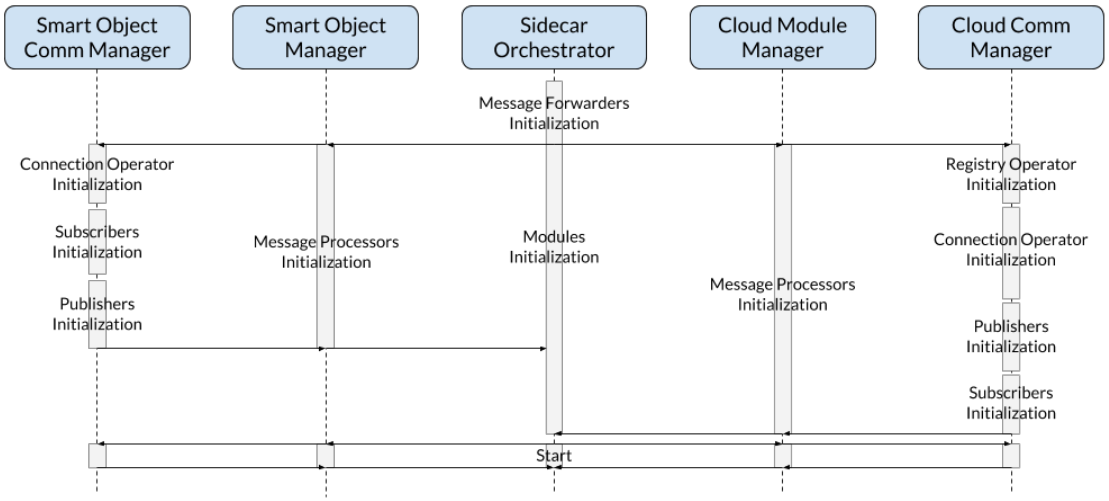


Figure 3. Startup sequence diagram.

The first operation executed by the sidecar orchestrator is the initialization of the sidecar processors. In this specific case and scheme they are called message forwarders, since they usually have the only task of sending messages from one module to another without performing particular operations. Then, the Orchestrator sends a initialization command to the smart object manager and all the cloud module managers, that in turn trigger their specific initialization procedures: for example, subscribers will register to the topics of interests, publisher will prepare themselves to send messages, and so on. In the scheme are reported examples of sub-modules that can be initialized, but as stated before this sequence could be different depending on the use case (it will fully be explained in the Implementation chapter). It is important to note that the Orchestrator forwards the initialization messages simultaneously: this is possible thanks to the choice of making all the modules separated and independent, allowing to initialize them simultaneously with a potential time saving in real-use scenarios (especially with time-consuming operations like remote connections). After that all the modules have notified the sidecar orchestrator that initialization operations are terminated, a start command is propagated hierarchically: as can be seen from the sequence diagram, this operation takes much less time than the previous one, as all the components are already initialized. After the orchestrator has collected all the start completed notifications coming from modules and sub-modules, the sidecar object is ready to begin the data message handling.

The second phase denoted as "runtime" is shown in Figure 4. First of all, it is important to specify that this diagram depicts the forwarding of a message from the smart object to the cloud, but the exact same mechanism applies for messages coming from the cloud and directed to the smart object. The first step is obviously the reception of the message by the Subscriber, that has registered to the topics of interest in the initialization phase; then, the message is sent to the appropriate sidecar processor (in this particular case a simple forwarder), that in turn forwards it to the specific Message Processor of the "destination" module. In this case the processor, without performing any particular operation, simply passes the message to the publisher, that takes care of sending the message to the designed destination. An important constraint that must be applied during implementation concerns the queuing mechanism: each module in charge of handling data messages must have its own separate queue with fair order policies and synchronization measures, in order to ensure the correct elaboration of each single message avoiding concurrency and fairness problems. In this example we exclude the occurrence of faults during the message handling, whose relative cases will be specifically analyzed in one of the following sections.

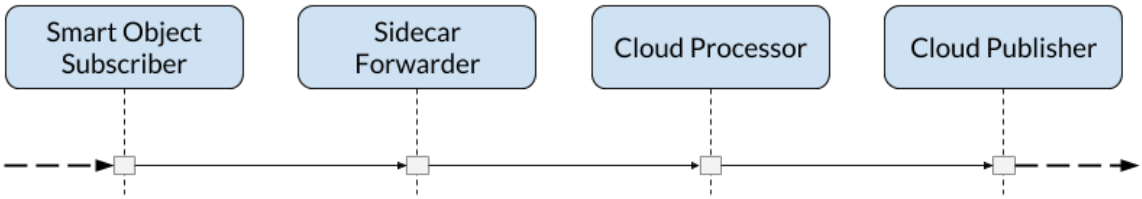


Figure 4. Data messages handling in the case of simple forwarding.

Figure 5 depicts a more complex data message handling during the Runtime phase. It is easy to see that the operation chain is the same seen in the previous case until the message reaches the module processor, but in this case the behavior of the latter is to send only one message every three received; the publisher then receives only the messages that must be effectively sent to the designed destination.

This behavior is a good example of isolation between modules: the publisher has no knowledge of how messages are processed in the previous steps, but is only focused on its main task of sending messages.

It is also important to note that the way in which the processor acts on the incoming messages is ruled by specific custom update policies, that can be defined at will during implementation following case-specific criteria.

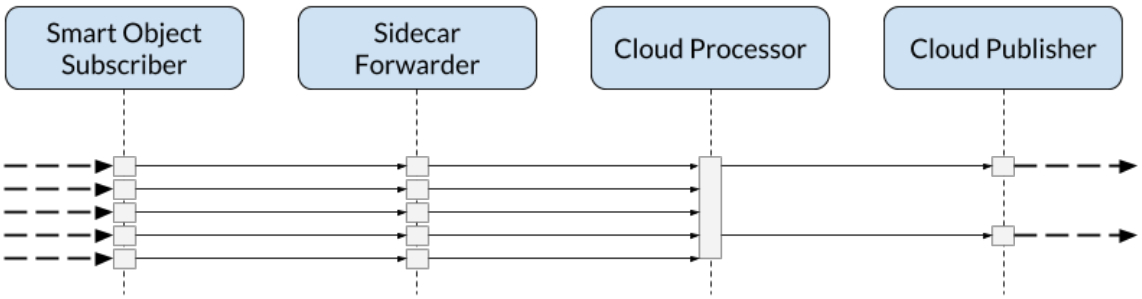


Figure 5. Data messages handling with intermediate processing.

In Figure 6 is reported the sequence diagram of the standard fault handling and recovery mechanism: this procedure can be extended to comply with specific requirements, but the base structure must always be the same.

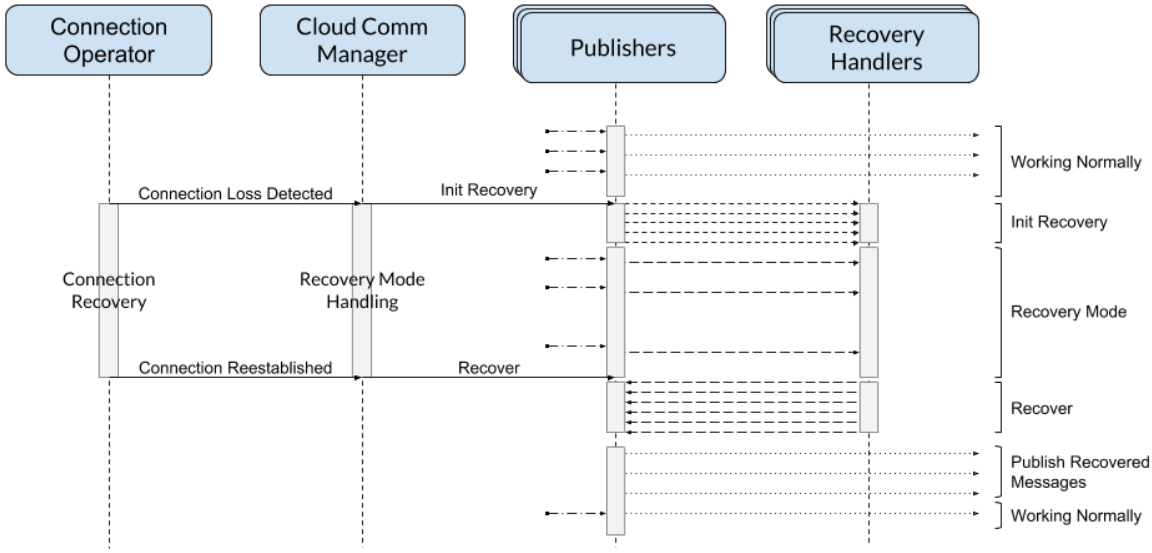


Figure 6. Fault handling and recovery mechanism.

For this example, we introduce the connection operator: this component, often present in practical implementations, has the main task of monitoring the network connection, eventually trying to recover the connectivity in case of faults. It is important to point out that faults can be detected by any of the modules present in the sidecar object, that must notify the event to their specific Comm Manager in any case: for example, if a Publisher throws an error generated by a message sending failure, it must notify that to its particular comm manager, that reacts accordingly. Back to our example, as soon as the connection operator detects a fault it sends a control message to the comm manager, that begins the recovery operations: publishers are then notified, and in turn they begin the recovery handling sorting all the messages currently waiting to be sent to topic-specific recovery handlers (that should be instantiated only when needed). After that the recovery initialization has been completed, the module enters the recovery mode and all the future messages coming from processors flow in the specific recovery handlers until the fault is resolved. When connection operator restores the connectivity, it immediately notifies the comm manager that in turn propagates the notifications to all the publishers: at this point the recovery handlers pass back the messages to publishers, following topic-specific recovery policies that regulates the reintroduction of the messages in terms of time, number, or even content depending on the use case implementation. After the recovery handlers have emptied, the module exits from recovery mode and message publishing operations returns to normal, eventually having in queue messages coming from the recovery handling process. It is worth noting that faults may also occur during the startup phase: in this case the mechanisms seen in this section do not apply, an specific procedures are executed depending on the use case (e.g., try to reinitialize the entire sidecar object).

4. Experimental Results

This section is entirely dedicated to an exhaustive set of experimental test and evaluation of the implemented functionality of a sidecar object for the integration and synchronization with Google Cloud IoT Platform. The environment used to perform the tests on the sidecar had the following characteristics: (i) Dell XPS 15 9560 equipped with Intel Core i7-7700HQ CPU 2.80 GHz x 8 RAM 16 GB Ubuntu 18.04.2 64-bit and (ii) WiFi connectivity with high-speed internet access (largely enough to meet the maximum data requirements of a single sidecar object).

In the following test it has also been used a Mosquitto MQTT Broker, running in a local docker container, and a custom Java program that emulated the behaviors of a real smart object in terms of data generation. The section explains minutely all the tests performed (considering and averaging through multiple runs for each configuration) on the sidecar object, analyzing and motivating in-depth the obtained results. The metrics that will used are summarized in Table 1.

Table 1. Experimental evaluation's metrics.

Metrics	Dimension	Description
Startup Time	[ms]	Time in milliseconds required by the sidecar object and its internal components to startup
Message pass-through delay	[ms]	The pass-through delay as the time elapsed between the arrival of the message inside the SmartObjectSubscriber and the execution of the asynchronous publish by the GoogleCloud-TelemetryPublisher
Message Ack Delay	[ms]	The elapsed time between the asynchronous publish performed by the TelemetryPublisher and the server acknowledgment of that specific publish
Queues Sizes	Adimensional	Outbound and Recovery Queues sizes, for the Telemetry Message Recovery tests
Heap Memory Size	[MBytes]	Allocated memory for the Java Heap
Heap Memory Usage	[MBytes]	User memory of the allocated Java Heap
CPU Usage Percentage	Adimensional	Percentage of used CPU by the object

4.1. Initialization and Start Times

The first set of experimental tests measures the required time to initialize and start the sidecar object and all its inner components associated to five full processing chains (corresponding to five MQTT topics) that has to be synchronized with the cloud.

Figure 7a shows the initialization times of the higher level components of the sidecar object. The `GoogleCloudModuleManager` is the component that needs more time to be initialized since the communications between sidecar object and cloud is performed through the internet connectivity instead of a local communication. The five `SidecarForwarders` on the contrary have a limited influence on the overall startup time, as proof that the structures adopted during implementation add an extremely low overhead to startup times. It is important to note that the sum of the startup times showed in the graph is not equal to the total sidecar object startup time, as the initialization of `GoogleCloudModuleManager` and `SmartObjectModuleManager` (as explained in the Implementation section) is performed on separated threads running in parallel; the average startup time of the entire sidecar object resulting from the tests is 1229.8 ms, with a standard deviation of 178 ms. In Figure 7b are reported the results relative to the starting times of the same components.

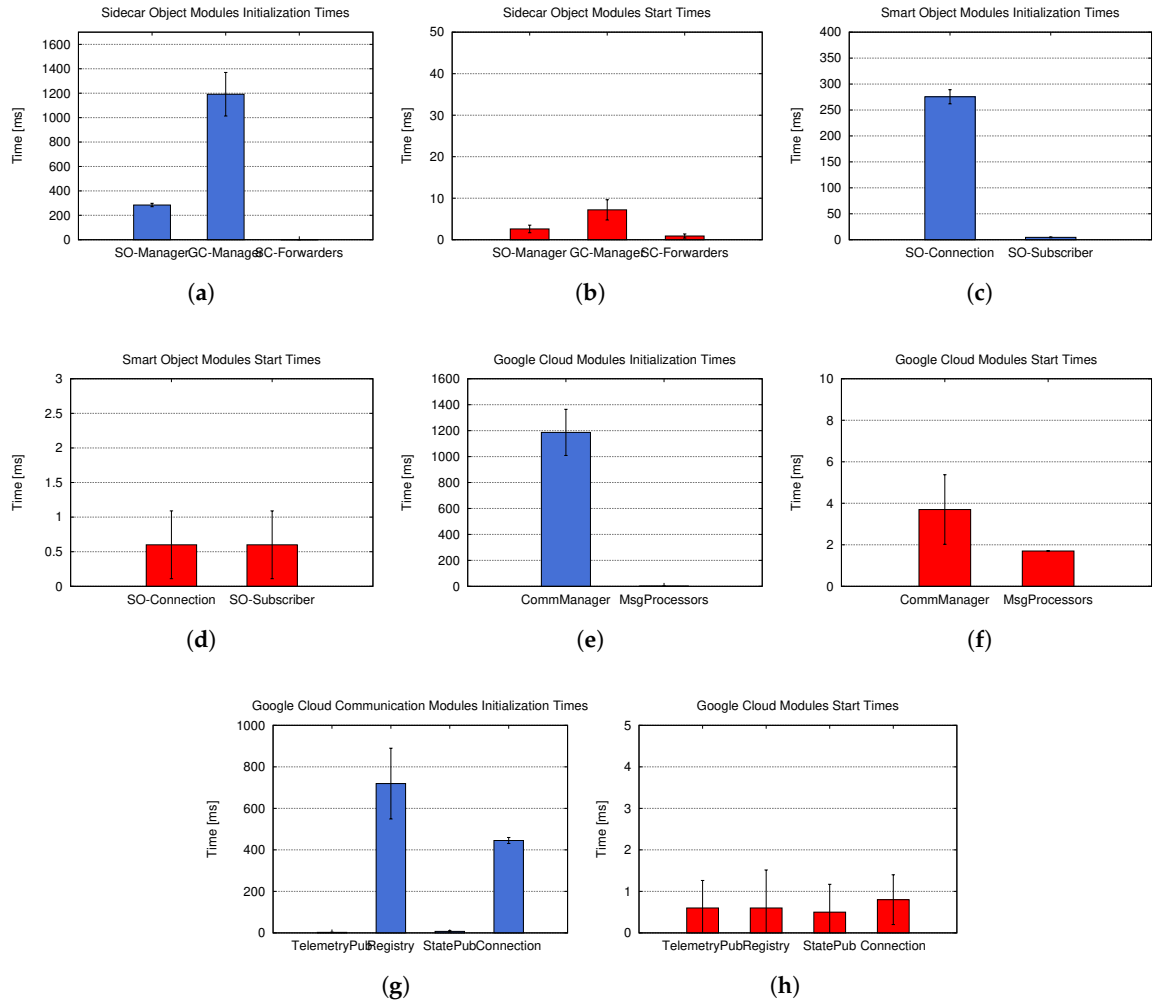


Figure 7. (**a**) Sidecar object modules initialization times; (**b**) sidecar object modules start times; (**c**) smart object modules initialization times; (**d**) smart object modules start times; (**e**) Google Cloud Modules initialization times; (**f**) Google Cloud Modules start times; (**g**) Google Cloud Communication Modules initialization times and (**h**) smart object modules start times.

As expected, starting times were significantly lower than the startup times, since they are associated only the startup of Java Threads for each component. In Figure 7c are shown the results related to the initialization times of the inner modules of the smart object module.

The `SmartObjectConnectionOperator` took much more time to be initialized compared with the `SmartObjectSubscriber`, since it had to setup the secure connection through the use of cryptographic keys. Figure 7d depicts start times of the same components.

Considering that both `SmartObjectConnectionOperator` and `SmartObjectSubscriber` ran on a single thread, the startup times were extremely low (∼1 ms) Figure 7e shows the initialization times of the `GoogleCloudModule` submodules. Presented results show that, as expected, the communication manager had the most relevant impact on the overall timing due to the remote connection with the cloud services. On the other hand the initialization of internal and local components (such the five `MessageProcessors` in this case) is barely perceptible.

For completeness, in Figure 7f are reported the start times relative to the last components analyzed. As for the other cases analyzed, starting times are much lower of initialization times, as consequence of the implementation choices made. The last (and probably more significant) test results are related to the inner components of the `GoogleCloudCommManager`: in Figure 7g are detailed the initialization times of such components. As expected, `RegistryOperator` and `ConnectionOperator` are the most expensive in terms of initialization. It is also important to note that those two modules run sequentially, as the `RegistryOperator` must be necessarily executed before the MQTT connection with the Google Cloud API. Figure 7h confirms that the starting times of the components inside the Google Cloud Communication Module are in line with the results registered for previous submodules.

4.2. Telemetry Messages Pass-Through and Ack Delays

This set of tests has the goal of better understand and analyze how the sidecar objects behaves sending telemetry messages and data under different conditions and configurations. The first set of tests has been focused on the evaluation of the performance as function of the message rate. Evaluated rates follow the throughput limits imposed by the Google Cloud Platform on the maximum throughput allowed for a single device. The experimental setup configuration takes into account: (i) one full message processing chain (corresponding to one MQTT topic from smart object to the cloud); (ii) 1000 message sent for each run; (iii) a messages payload size of 1 KB and 5 KB and (iv) multiple message rates (5, 10, 30, 50, 70 and 100 messages per second). Source messages have been sent by a custom Java program emulating a real smart object at fixed rate with the target specified payload size. Figure 8a shows the pass-through delays emerging from the tests described above.

These results show that the sidecar object behaves well even with higher message rates without performance degradations. The payload size does not have relevant influence on the results. Figure 8b reports the results relative to ack delays under the same conditions, the delays are significantly higher compared to the previous ones since the ack delay measures the round-trip time of a single telemetry message transmitted through the internet. Furthermore, the most interesting aspect emerging is that even at high rates the cloud maintains a constant responsiveness without signs of performance degradations, proving how both sidecar object and Google Cloud platform are able to support high-frequency message exchange. The second set of tests focuses on the ack delay when the throughput limit imposed by the Google Cloud Platform is exceeded: such tests are performed using a wide range of throughput values.

Figure 8c shows the ack delays as a function of different Throughputs. As expected the graph highlights good performances and low delays with several message throughputs within the threshold of Google Cloud platform (512 KB/s). Exceeding this limit we have and expected increasing of the delay and in order to better understand the Google Cloud Platform behavior an additional set of tests has been performed by using using throughput values of 480, 800, 1200 and 1600 KB/s. Figure 8d shows the ack delays emerging from the tests using parameters previously mentioned. The sidecar object performs well with the 480 KB/s throughput (that is near the limit of 512 KB/s), with ack delays

aligning with the ones registered in the the previous tests. Furthermore, the obtained results with the three values exceeding the limit show how the Google Cloud Platform behaves when a device exceeds the maximum throughput rate. Incoming messages are not discarded, but queued internally and elaborated as soon as possible returning the ack message only when the message has been effectively ingested by the platform.

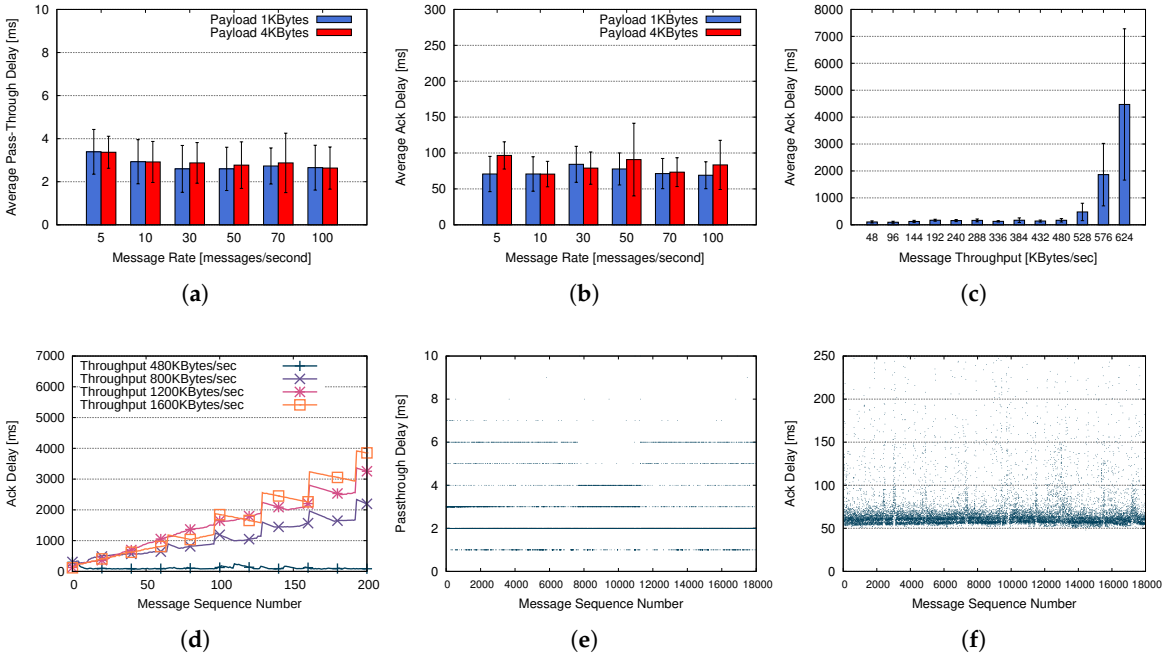


Figure 8. (**a**) Telemetry messages pass-through delay average; (**b**) telemetry messages ack delay average; (**c**) telemetry messages ack delay average; (**d**) telemetry messages ack delay; (**e**) telemetry messages pass-through delays; (**f**) telemetry messages ack delays.

The last tests associate to telemetry performance evaluation have been performed executing a longer run (30 min) in order to evaluate check the sidecar object behavior over time. The experiment is associate with: (i) 1 full message processing chain (from smart object to the cloud); (ii) 18,000 messages sent; (iii) 10 messages per second and (iv) message payload size of 4 KB. Figure 8e,f show the results related to pass-through and ack delays. Both graphs show that the results of the previous tests (obtained from shorter runs) are confirmed, as the sidecar object maintains the same performances even with significantly longer execution times.

4.3. State Messages Pass-Through Delay and Queue Sizes

This additional group of tests and analysis has been designed in order to analyze the behavior of the sidecar object related to the publishing of state messages coming from the smart object at different rates. The test configuration takes into account: (i) 1 full message processing chain (from smart object to the cloud); (ii) 300 messages sent and (iii) a message payload size of 4 KB. Messages have been sent at fixed rates of 0.7, 0.8, 0.9, 0.95, 1.0, 1.1, 1.2 and 1.5 messages per second. Obtained results are shown in Figure 10a,b.

As described in the implementation chapter, the Google Cloud Platform applies a limit of one state message per second for each device. If this limit is exceeded the connection is interrupted giving a proper "error message", and it must be reestablished. During our tests no disconnection has been registered: this proofs that the sidecar object has never exceeded the limit imposed by the platform. Graphs show that until messages arrive from the smart object with an acceptable rate, pass-through delays remain low and stable and queue sizes close to zero. With message rates equal or greater than

one per second, pass-through delays and queue sizes rise linearly: this is the expected behavior as the sidecar object keeps in memory the received State messages publishing them at the aforementioned rate of one per second, in order to avoid the disconnection from the Google Cloud Platform. It is worth mentioning that the slight increase of pass-through time and queue size at the limit rate (1 message per second) is caused by the publishing overhead, as this is a synchronous operation.

4.4. Telemetry Message Recovery

This experimental evaluation aims to analyze the sidecar object's behavior in case of connection faults occurring at runtime using different types of recovery policies. The experiment configuration is the following: (i) 1 full message processing chain (from smart object to the cloud); (ii) 600 messages sent; (iii) Payload message size of 4 KB and (iv) a keep-alive time of 14 s (keep-alive is the maximum time from the last interaction with the server, after which the MQTT client defines the connection lost). Messages are sent at the fixed rate (600 messages @ 5 msg/s) with the following schedule (depicted in Figure 9):

- `0:00`: all systems running normally with internet connectivity;
- `0:30`: the testing machine is disconnected from the WiFi network and internet;
- `1:30`: the testing machine is reconnected to the same WiFi network and to internet;
- `2:00`: the test ends.

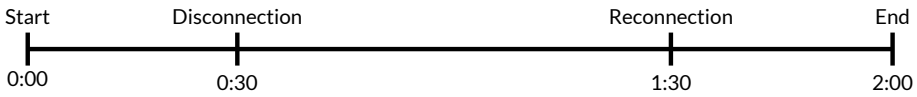


Figure 9. Recovery tests timeline.

This test typology is repeated for each of the three Recovery Policy described in the Implementation chapter. The first case analyzed concerns to the `NoRecovery` policy, that discards all the messages received by the `Publisher` while in `RECOVERY MODE`. The results are shown in Figure 10a. It can be observed that at 0:30, in concurrence with the network disconnection, the outbound queue size starts to increase. Around 1:00 the sidecar object detects the connection loss and applies the `NoRecovery` policy. All the outbound messages are discarded, and any new incoming message is consequently discarded and lost. Around 1:40 connection is reestablished and new messages are published normally.

The second analyzed case is the one relative to the `AllRecovery` policy, that keeps in memory all the messages received during `RECOVERY MODE` and restores them on top of the publishing queue during the `RECOVERING` phase. The results are shown in Figure 10b–d. Obtained results show that at 0:30, in concurrence with the network disconnection, the outbound queue size started to increase. Around 1:00 the sidecar object detected the connection loss and applied the `AllRecovery` policy: all the outbound messages were transferred to the `RecoveryHandler` queue, and any new message coming from the smart object was forwarded to the same queue. At around 1:45 connection was reestablished: messages contained in the recovery queue were sent back to the publisher, which took around 3 s to handle and recover all the collected messages before returning to the normal behavior.

The third case analyzed is the one relative to the `OneEveryNRecovery` policy, which keeps in memory only one message every N received during `RECOVERY MODE` (for this test, `N = 5`) while discarding the others. The results are shown in Figure 10e–g. It can be observed that at 0:30, in concurrence with the network disconnection, the Outbound Queue size starts to increase. Around 1:00, the sidecar object detected the connection loss and applies the `OneEveryNRecovery` policy: only one message every 5 is transferred to the `RecoveryHandler` queue, and any new message coming from the smart object follow the same procedure. At around 1:45, connection was reestablished: messages contained in the recovery queue were sent back to the publisher, which this time took only 0.5 s to deal with the restored messages before returning to the normal behavior.

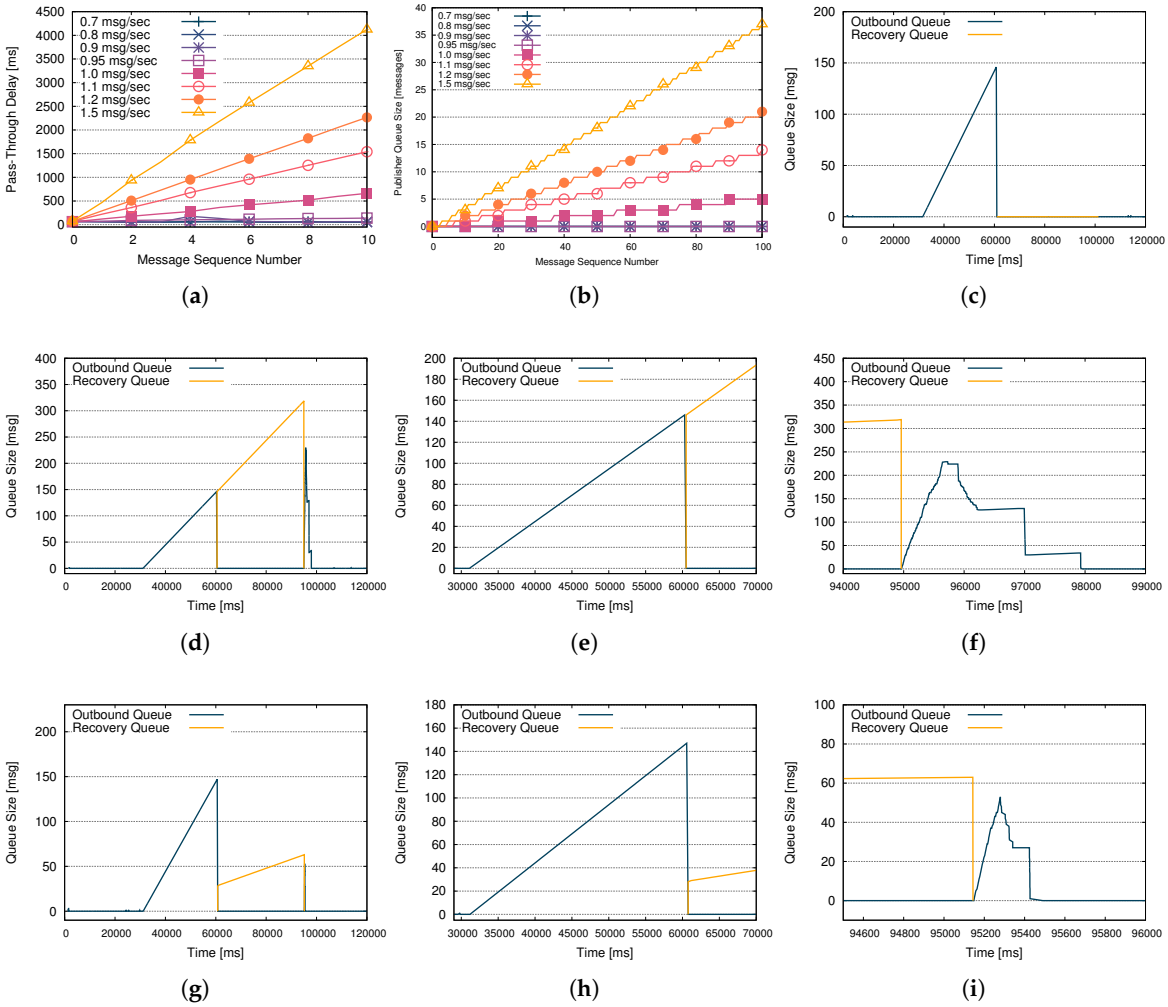


Figure 10. (**a**) State messages pass-through delays; (**b**) publisher queue sizes; (**c**) queue sizes adopting the `NoRecovery` Policy; (**d**) queue sizes adopting the `AllRecovery` Policy; (**e**) queue sizes adopting the `AllRecovery` policy during the disconnection event; (**f**) queue sizes adopting the `AllRecovery` policy during the reconnection event; (**g**) Queue sizes adopting the `OneEveryNRecovery` policy; (**h**) queue sizes adopting the `OneEveryNRecovery` policy during the disconnection event; (**i**) queue sizes adopting the `AllRecovery` policy during the reconnection event.

Starting from the performed and presented evaluation emerges how the Recovery Policy significantly influences the behavior of the sidecar object, highlighting the importance of choosing the right policy depending on the needs specific of each use case and application scenario.

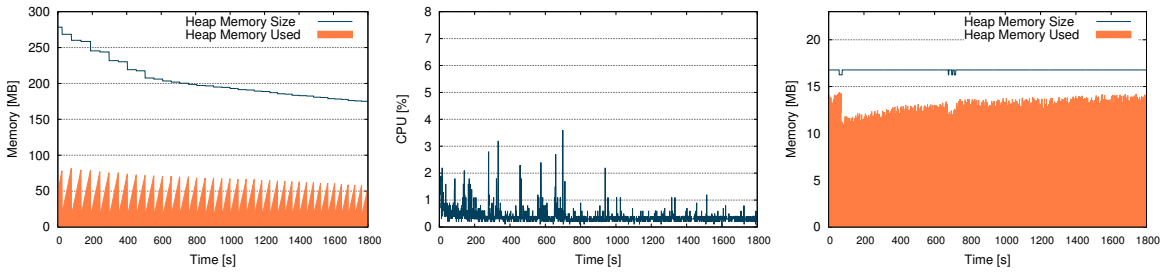
4.5. Resources Usage

The last set of tests has been designed to analyze the resources usage of the sidecar object during runtime. The used configuration takes into account the following parameters: (i) six full message processing chains (five telemetry topics and one state topic, from smart object to the cloud); (ii) delay between telemetry messages (for each topic): random in the range 100–1000 ms; (iii) delay between state messages: random in the range 500–5000 ms; (iv) message payload of 4 KB and (v) test duration of 30 min. Memory and CPU usage data were collected using the Java profiling tool VisualVM.

The first execution has been performed setting the maximum heap memory size for the sidecar object at 256 MB (using the -Xmx256m option), that is largely enough for the correct functioning of the system. Figure 11a,b show the results related to memory and CPU usage. From the memory graph emerges that the 256 MB limit configured in this test is overabundant, as the JVM automatically

reduces the max heap size in order to optimize the resource usage; it also important to note how the garbage collector shrinks the memory used by the sidecar object intervening periodically. Regarding the CPU usage, most of the measurements were ∼0.4%: those extremely low values were due to the high computational power available on the testing machine.

For the second execution the Heap Memory size limit has been set to only 16 MB (using the -Xmx16m option): the results are shown in Figure 11c. In this case, memory available is sufficient to guarantee the correct functioning of the system: the maximum heap is reduced only for extremely short times, and the Java garbage collector intervenes at a much higher rate that the previous case. Despite the "constrained" configuration, the measured behavior does not highlight any problem or error proving that the sidecar object is able to remain operational.



(**a**) (**b**) (**c**)

Figure 11. (**a**) Memory usage with the -Xmx256m option (**b**) CPU usage (**c**) memory usage with the -Xmx16m option.

4.6. Constrained Device Evaluation

In order to evaluate the performance and the behaviors of the sidecar object in constrained scenarios, a specific set of tests has been performed using a Raspberry Pi 3B+ board. The single board computer was equipped with: (i) Broadcom BCM2837B0 quad-core A53 (ARMv8) 64-bit @ 1.4 GHz; (ii) 1 GB LPDDR2 SDRAM; (iii) a 2.4 GHz and 5GHz 802.11b/g/n/ac Wi-Fi network interface and (iv) Raspbian 9 (Stretch) as the operating system. The board has been connected to a Wi-Fi network with high-speed internet access, largely enough to meet the maximum data requirements of a single sidecar object. The sidecar object instance has been executed on a JVM running on the Raspberry Pi board, while an external device connected to the same network run a Mosquitto MQTT Broker (running in a docker container) and a custom Java program that faithfully reproduces the behaviors of a real smart object. In order to minimize the footprint of testing tools on the Raspberry Pi, part of the data generated during the sessions has been collected remotely using VisualVM. This set of tests has the goal of better understand and analyze how the sidecar objects behaves sending telemetry messages in a constrained scenario.

The test configuration takes into account: (i) 1 full message processing chain (from smart object to the cloud); (ii) 18,000 messages sent; (iii) 10 messages per second and (iv) a payload size of 4 KB. Messages have been sent by a custom Java program at fixed rate with the defined payload size for 30 min (18,000 messages @10 msg/s). Figure 12a,b show the results related to pass-through and ack delays, while Figure 12c provides a combined view of the data obtained.

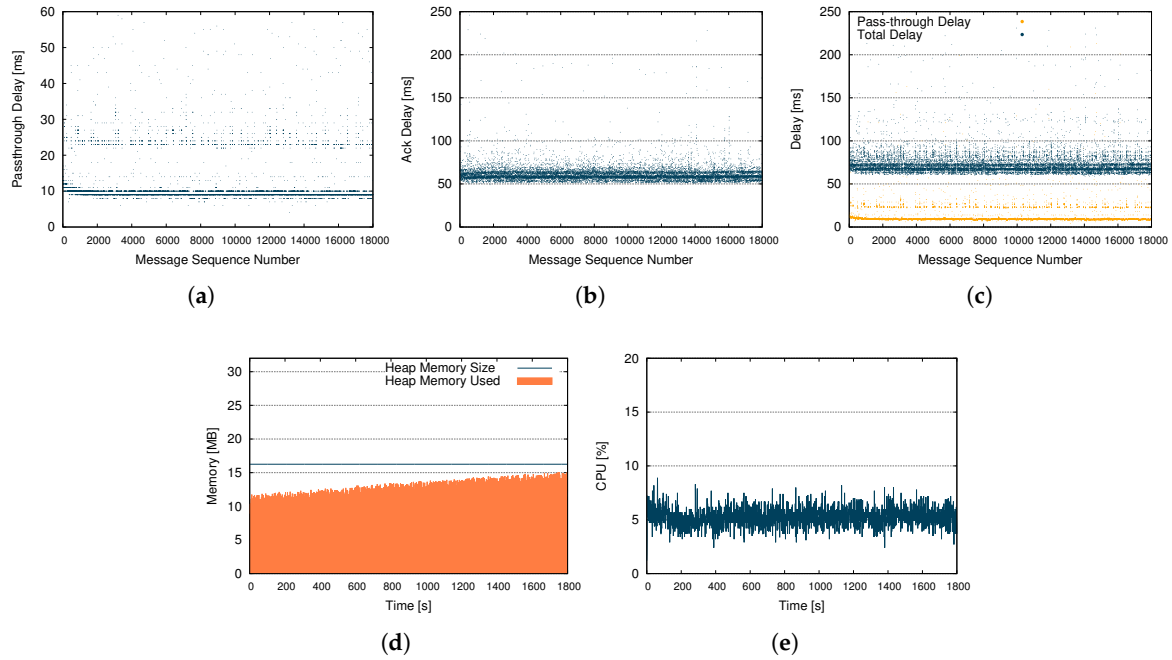


Figure 12. Raspberry Pi experimental results: (**a**) telemetry messages pass-through delays; (**b**) telemetry messages ack delays; (**c**) combined view of telemetry messages delays; (**d**) memory usage with the -Xmx16m option; (**e**) CPU usage.

It is significant to compare the results obtained with the ones coming from from the tests performed previous configuration. Regarding the Pass- Through Delay, it has been registered a slight performance degradation both in terms of absolute value and variance: such behavior was however expected, and it is safe to attribute the performance degradation to the limited power of the hardware involved. The same consideration is still valid also for the Ack Delay, the results obtained are totally comparable. The small difference can be justified by the performance of the network interface used on the Raspberry Pi that is limited if compared with the one used for the previous tests.

The second test configuration considers: (i) six full message processing chains (five telemetry topics and one state topic, from smart object to the cloud); (ii) delay between telemetry messages (for each topic): random in the range 100–1000 ms; (iii) delay between state messages: random in the range 500–5000 ms and (iv) a payload size of 4 KB. Messages have been sent by a custom Java program at fixed rate with the defined payload size for 30 min. Memory and CPU metrics have been collected remotely using the Java profiling tool VisualVM running on a separated PC. The execution has been performed setting the maximum heap memory size for the sidecar object at only 16 MB using the `-Xmx16m` option. Figure 12d,e show the results related to memory and CPU usage. As expected, the CPU usage is proportionally more affected by the execution of the sidecar object, but at the same time the results obtained with a constrained hardware with a bounded configuration confirm that the footprint of a sidecar object instance is definitely limited and multiple sidecars can be executed at the same time. Regarding the memory usage, the Java garbage collector frequently acts to cope with imposed constraints, optimizing the memory utilization gradually over time without ever exceeding limits.

5. Conclusions and Future Works

In this paper we have introduced the concept of the IoT sidecar object, an innovative and scalable methodology to extend smart objects functionalities without modifying the original device/object. Furthermore we presented the design and development of one of its implementation dedicated to the transparent and seamless synchronization of smart object's data, states and commands with one or more cloud or remote services. An extensive set of experimental analysis has been performed in

order to properly evaluate the behaviors and the performance of the proposed solution (integrated with Google Cloud IoT) with several setups and configurations. The design and the modular structure allowed the sidecar object to efficiently operate without the need of being aware of the context in which it is used and deployed. An external orchestrator can coordinate the activation of one or multiple sidecar objects according to the current context without affecting the deployment and the IoT application's designed behavior. Presented tests show, through the use of multiple metrics, that the proposed solution performs well in terms delays, overhead and consumed resources and that can be consequently deployed in several distributed IoT application scenarios.

In the context of microservices oriented IoT applications we strongly believe that this solution will allow to easily introduce new IoT behaviors and services making existing smart objects and devices capable of increasing their functionalities, without requiring any direct changes to their original implementation. This approach follows the new IoT technology trends associated to microservices and IoT edge computing applications where the operative functionalities are broken down into small, modular, independently deployable and loosely-coupled microelements. Target applications scenarios involve (but are not limited to): (i) digital twin [27,28] creation and management where the responsibility to create and update the clone is delegated to the sidecar object; (ii) an edge oriented serverless and lambda functions [29,30] execution of management where the complexity and intelligence is outside the smart object and (iii) IoT container orchestration and support for IoT microservice mesh (denoted as Service Mesh) [31] where the sidecar acts as inbound service for the smart object handling protocols translation, access control and additional security features.

Furthermore additional and interesting developments are related to: (i) the implementation of additional integration with cloud modules in order to support new IoT cloud services (e.g., AWS IoT and Azure IoT) for smart object synchronization; (ii) the performance evaluation of the sidecar object using different mobile internet connectivity in order to understand the behavior and the potential limitation and new requirements related to mobility and data caching; and (iii) to the investigation of innovative sidecar object orchestration patterns.

Author Contributions: M.P. and S.C. conceived and designed the idea, the architecture and the experiments; M.R. with the supervision of M.P and S.C. developed the main Software modules involved in the solution. M.R. performed all the experiments and testing of the components; M.P., S.C., S.B., L.M., L.V and M.R. analyzed the data and the obtained resutls; M.P. and S.C. with the supervision and coordination of L.V. wrote the paper.

Funding: This research received no external funding.

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

References

1. Butzin, B.; Golatowski, F.; Timmermann, D. Microservices approach for the internet of things. In Proceedings of the 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), Berlin, Germany, 6–9 September 2016; pp. 1–6. [CrossRef]
2. Morabito, R. Virtualization on Internet of Things Edge Devices with Container Technologies: A Performance Evaluation. *IEEE Access* **2017**, *5*, 8835–8850. [CrossRef]
3. Bonomi, F.; Milito, R.; Zhu, J.; Addepalli, S. Fog Computing and Its Role in the Internet of Things. Available online: http://conferences.sigcomm.org/sigcomm/2012/paper/mcc/p13.pdf (accessed on 4 July 2019).
4. Kovatsch, M.; Mayer, S.; Ostermaier, B. Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things. In Proceedings of the 2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, Palermo, Italy, 4–6 July 2012; pp. 751–756. [CrossRef]
5. Benazzouz, Y.; Munilla, C.; Günalp, O.; Gallissot, M.; Gürgen, L. Sharing user IoT devices in the cloud. In Proceedings of the 2014 IEEE World Forum on Internet of Things (WF-IoT), Seoul, Korea, 6–8 March 2014; pp. 373–374. [CrossRef]
6. Amazon. Amazon AWS IoT. Available online: https://aws.amazon.com/iot-core/ (accessed on 4 July 2019).

7. Google. Google Cloud IoT Core. Available online: https://cloud.google.com/iot-core/ (accessed on 4 July 2019).
8. Microsoft. Microsoft Azure—Cloud Platform. Available online: http://azure.microsoft.com/it-it/ (accessed on 4 July 2019).
9. MQTT Version 3.1.1. 2014. Available online: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html (accessed on 4 July 2019).
10. Fielding, R.; Gettys, J.; Mogul, J.; Frystyk, H.; Masinter, L.; Leach, P.; Berners-Lee, T. *Hypertext Transfer 619 Protocol—HTTP/1.1*; RFC 2616; IETF: Fremont, CA, USA, 1999.
11. Cirani, S.; Ferrari, G.; Mancin, M.; Picone, M. Virtual Replication of IoT Hubs in the Cloud: A Flexible Approach to Smart Object Management. *J. Sens. Actuator Netw.* **2018**, *7*, 16. [CrossRef]
12. Shelby, Z.; Hartke, K.; Bormann, C. *The Constrained Application Protocol (CoAP)*; RFC 7252; IETF: Fremont, CA, USA, 2014.
13. Aazam, M.; Khan, I.; Alsaffar, A.; Huh, E.N. Cloud of Things: Integrating Internet of Things and cloud computing and the issues involved. In Proceedings of the 2014 11th International Bhurban Conference on Applied Sciences and Technology (IBCAST), Islamabad, Pakistan, 14–18 January 2014; pp. 414–419. [CrossRef]
14. Aazam, M.; Hung, P.P.; Huh, E.N. Smart gateway based communication for cloud of things. In Proceedings of the 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, 21–24 April 2014; pp. 1–6. [CrossRef]
15. Yannuzzi, M.; Milito, R.; Serral-Gracià, R.; Montero, D.; Nemirovsky, M. Key ingredients in an IoT recipe: Fog Computing, Cloud computing, and more Fog Computing. In Proceedings of the 2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), Athens, Greece, 1–3 December 2014; pp. 325–329. [CrossRef]
16. Amazon. Amazon AWS IoT Greengrass. Available online: https://aws.amazon.com/greengrass/ (accessed on 4 July 2019).
17. Google. Google Cloud IoT Edge. Available online: https://cloud.google.com/blog/products/gcp/bringing-intelligence-edge-cloud-iot (accessed on 4 July 2019).
18. Renner, T.; Meldau, M.; Kliem, A. Towards Container-Based Resource Management for the Internet of Things. In Proceedings of the 2016 International Conference on Software Networking (ICSN), Jeju Island, Korea, 23–26 May 2016; pp. 1–5. [CrossRef]
19. Morabito, R.; Farris, I.; Iera, A.; Taleb, T. Evaluating Performance of Containerized IoT Services for Clustered Devices at the Network Edge. *IEEE Internet Things J.* **2017**, *4*, 1019–1030. [CrossRef]
20. Nider, J.; Rapoport, M. Cross-ISA Container Migration. In Proceedings of the 9th ACM International on Systems and Storage Conference, SYSTOR '16, Haifa, Israel, 6–8 June 2016; ACM: New York, NY, USA, 2016; p. 24:1. [CrossRef]
21. Machen, A.; Wang, S.; Leung, K.K.; Ko, B.J.; Salonidis, T. Live Service Migration in Mobile Edge Clouds. *Wirel. Commun.* **2018**, *25*, 140–147. [CrossRef]
22. Nitti, M.; Pilloni, V.; Colistra, G.; Atzori, L. The Virtual Object as a Major Element of the Internet of Things: A Survey. *IEEE Commun. Surv. Tutor.* **2016**, *18*, 1228–1240. [CrossRef]
23. Guan, Y.; Vasquez, J.C.; Guerrero, J.M.; Samovich, N.; Vanya, S.; Oravec, V.; García-Castro, R.; Serena, F.; Poveda-Villalón, M.; Radojicic, C.; et al. An open virtual neighbourhood network to connect IoT infrastructures and smart objects—Vicinity: IoT enables interoperability as a service. In Proceedings of the 2017 Global Internet of Things Summit (GIoTS), Geneva, Switzerland, 6–9 June 2017; pp. 1–6. [CrossRef]
24. Datta, S.K.; Bonnet, C. An edge computing architecture integrating virtual IoT devices. In Proceedings of the 2017 IEEE 6th Global Conference on Consumer Electronics (GCCE), Nagoya, Japan, 24–27 October 2017; pp. 1–3. [CrossRef]
25. Burns, B. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*, 1st ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2018.
26. Moyer, F. *Comprehensive Container-Based Service Monitoring with Kubernetes and Istio*; USENIX Association: Berkeley, CA, USA, 2018.
27. Canedo, A. Industrial IoT lifecycle via digital twins. In Proceedings of the 2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Pittsburgh, PA, USA, 1–7 October 2016; p. 1.

28. Song, E.Y.; Burns, M.; Pandey, A.; Roth, T. IEEE 1451 Smart Sensor Digital Twin Federation for IoT/CPS Research. In Proceedings of the 2019 IEEE Sensors Applications Symposium (SAS), Sophia Antipolis, France, 11–13 March 2019; pp. 1–6. [CrossRef]
29. McGrath, G.; Short, J.; Ennis, S.; Judson, B.; Brenner, P. Cloud Event Programming Paradigms: Applications and Analysis. In Proceedings of the 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 27 June–2 July 2016; pp. 400–406. [CrossRef]
30. Lynn, T.; Rosati, P.; Lejeune, A.; Emeakaroha, V. A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms. In Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Hong Kong, China, 11–14 December 2017; pp. 162–169. [CrossRef]
31. Khan, A. Key Characteristics of a Container Orchestration Platform to Enable a Modern Application. *IEEE Cloud Comput.* **2017**, *4*, 42–48. [CrossRef]