From the Edge to the Cloud: Enabling Reliable IoT Applications

Cristian Martín, Daniel Garrido, Manuel Díaz and Bartolomé Rubio
Department of Languages and Computer Science
University of Málaga, Málaga, Spain
{cmf,dgarrido,mdr,tolo}@lcc.uma.es

***Abstract*—This paper presents a reliable architecture for the IoT considering multiple levels: edge, fog and cloud. This architecture can help to reduce latency and improve resiliency of IoT applications. The platform is based on a set of containerised components replicated at different levels. Fault tolerance mechanisms are provided by means of replication, the Apache Kafka framework and shadow devices. Apache Kafka is used to distribute messages along the multiple levels. Shadow devices include device states, and they can be used to avoid device interruptions using physical replication and state restoration. The architecture is also protocol-agnostic, allowing the use of different adaptors for the most common IoT protocols. A mission-critical use case is presented where this architecture can be applied. Finally, an evaluation has been carried out in order to test the feasibility of the fog infrastructure.**

***Index Terms*—Internet of Things, Fault Tolerance, Container Virtualisation, Edge, Fog, Apache Kafka**

I. Introduction

The Internet of Things is a disrupting paradigm of interconnected things that exchange information over the Internet [1]. This paradigm has enabled new sources of data in the Internet era, with a forecast of 500 billion of connected devices by 2030 [2]. As a consequence, paradigms such as connected cars, Industry 4.0 and smart cities are possible, and the most important, they contribute to the digitisation of services and phenomena of the physical world.

Limitations on IoT devices brought current Cloud Computing infrastructures (CC) to support data intensive applications in a scalable way. However, timing issues such as latency are not considered in those systems at the same time [3]. This can be an important problem in next years due to the explosion of the IoT, where more and more applications are expected to have stronger real-time requirements.

In this sense, the so-called Fog Computing was introduced by Cisco in 2012. It is an architecture that extends traditional CC by getting processing and data analysis (through local server and gateways) closer to the data source (IoT edge devices), thereby reducing the amount of data to be transmitted to the cloud and balancing the computing necessities between both systems. Consequently, latency and response times are reduced when an event happens. For instance, in an environmental monitoring system, regional and local data can be aggregated and processed in the closer nodes providing a fast response, especially in emergency situations such as environmental alerts. At the same time, a detailed analysis that requires intensive computation activities can be performed in the CC [4].

The edge refers to the layer between the fog and the IoT devices, typically found on gateways and servers in close proximity to the IoT. Typically, when it acts like a gateway, it provides an interconnection between the IoT devices and the Internet, as for example in LoRaWAN networks. The goal of this layer is the same than the fog, reducing the latency and bandwidth with the IoT.

Apart from a low latency, in mission-critical applications fault-tolerance guaranties are required at different levels. On the one hand, the architecture itself has to provide fault-tolerance. In that sense, algorithms and techniques such as Paxos or Virtual Synchrony have contributed throughout history by providing consistency, especially in CC. Although the use of these techniques in a fog/edge architecture is an ongoing work, the use of container platforms, like Docker, and container management platforms like Docker Swarm and Kubernetes, have enabled a lightweight virtualisation infrastructure that can be portable to resource-constrained devices (e.g., edge devices), and an infrastructure to monitor, allocate and distribute containers, respectively. Therefore, these technologies can deal with fault-tolerance in addition to enabling horizontal and vertical migrations [5]. On the other hand, mission-critical IoT applications that depend on IoT devices as data sources or actuators cannot tolerate a service disruption on these devices. For instance, a data loss can have serious consequences in structural health monitoring. Nevertheless, due to their nature and limitations, the above techniques and algorithms are far away to be applied in IoT devices. One solution could consist of a decentralised fault tolerance framework in an IoT network [6], however it would increase the already limited overhead on these networks and could replace a device not being used by any application. IoT replication is essential in these cases, and an optimised use of it is presented in this paper.

To overcome this challenge, this paper presents an architecture that takes into account the global and heterogeneous Edge/Fog/Cloud infrastructure to enable the development of reliable IoT applications. Apart from dealing with software failures, this architecture manages failures in the underlying IoT infrastructure in order to have reliable IoT applications without device interruptions. In particular, this architecture adapts the application logic upon a device disruption by recovering IoT devices states. The architecture is designed

to be IoT application agnostic, although the first protocol connector is the OMA Lightweight M2M (LWM2M) [7], a promising standard both in the industry and the academy for managing lightweight and low power devices. A container infrastructure has also been adopted to enable the deployment and management of containers and allow both vertical and horizontal migrations in heterogeneous devices. This paper presents the architecture proposal of this ongoing work, currently under development.

The rest of the article is structured as follows. Section II considers the related work. In Section III the reliable architecture is presented. Then, a target use case with this architecture is discussed in Section IV. An evaluation of the fog infrastructure is presented in V. Lastly, the conclusions and future work are discussed in Section VI.

II. Related Work

In previous work we created Appdaptivity [8], a platform to develop personalised, device-decoupled and adapted to changing context applications. Appdaptivity adapts IoT applications with the changes in the underlying infrastructure and injects IoT devices into the application logic. Although Appdaptivity dynamically manages IoT applications to adapt them to context changes, it requires multiple IoT devices working to provide fault-tolerance. For instance, in the case of temperature monitoring, Appdaptivity has to have multiple devices monitoring temperature to have fault-tolerance, e.g., through CoAP (Constrained Application Protocol) observations. In this work, in the case of a device malfunction, the architecture manages it to obtain the same behaviour from another device without having multiple devices doing the same task, which can lead to an increase in power consumption.

In CEFIoT [9] a fault-tolerant IoT architecture for edge and cloud is presented. The approach of CEFIoT is similar to the one presented in this paper, since both use Apache Kafka as publish/subscribe platform and Docker as virtualisation platform. CEFIoT also provides a platform for container management and allocation, Kubernetes, similar to Docker Swarm. However, the CEFIoT only manages fault-tolerance respecting to node failures along the architecture, but this does not adapt the architecture itself, and the corresponding IoT applications, when a data source is out of order. In this paper, we propose a fault tolerance architecture that does not only manage fault-tolerance along the architecture, but also in the data sources belonging to the system.

Eclipse Hono [10] is an Eclipse Foundation project that provides an interconnection system between IoT protocols and external applications in a uniform way. The agnostic IoT protocol feature is similar to the one presented in this architecture, and the external application connector, AMQP, is also a distributed message queue, similar to Apache Kafka. Although Kafka provides a higher throughput and availability thanks to the leader election and consumer groups among others features. However, Eclipse Hono has a different approach. It requires a more proactive approach of IoT devices sending data to the platform, whereas our architecture manages IoT devices interactions in a fault-tolerant way.

III. A Reliable architecture

The architecture presented is this paper has been designed to enable reliable IoT applications in an Edge/Fog/Cloud architecture. In this architecture, a container infrastructure has been adopted to facilitate the deployment of the software stack in the heterogeneous devices involved: edge devices, fog servers and the cloud. The software stack apart of being deployed in any node of the architecture, it is replicated and monitored to provide high availability through Docker Swarm, a container management platform.

A. Shadow devices

To prevent a direct connection between the application logic and the IoT devices, IoT applications do not directly access the final devices, but they access the virtual devices created in the architecture, also known as *shadow devices*. Shadow devices enable fault-tolerance in the underlying IoT infrastructure since multiple devices can be associated to a shadow device, and in the case of a physical device disruption, shadow devices are automatically reconfigured to obtain the same behaviour (phenomena monitoring, actuator) from another associated device. A shadow device is only reconfigured if it is used or required by an application logic. We assume that if a shadow device is not being used or required by any application at a given time, it is not needed to recover the device's state in the case of a physical device disruption. This also alleviates the physical IoT devices of communications and changes, which are both limited in these devices.

The recovery process is then transparent from the IoT applications, and application developers only need to focus on the business logic. Architecture managers through a Web UI (User Interface) or by means of a REST API can easily define shadow devices. Once a shadow device is registered in the system, a JWT (JSON Web Token) is generated and will be the one used by IoT devices for: (i) authentication in the architecture; (ii) associating with shadow devices; (iii) and enabling fault-tolerant device groups.

B. Software stack

The software stack comprises several components enclosed in Docker containers as presented in Fig. 1. The source code of the project is under development and available on GitHub[1]. These components provide as little functionality as possible, comprising highly portable and reusable components, also known as microservices. These microservices are explained below:

1) **IoT register**. This is the first component in contact with the underlying IoT infrastructure. Upon a device request with JWT authentication, this component records the IoT device or IoT devices through an IoT facilitator (e.g., an LWM2M server) in a distributed database. After that, it

[1]https://github.com/ertis-research/reliable-iot

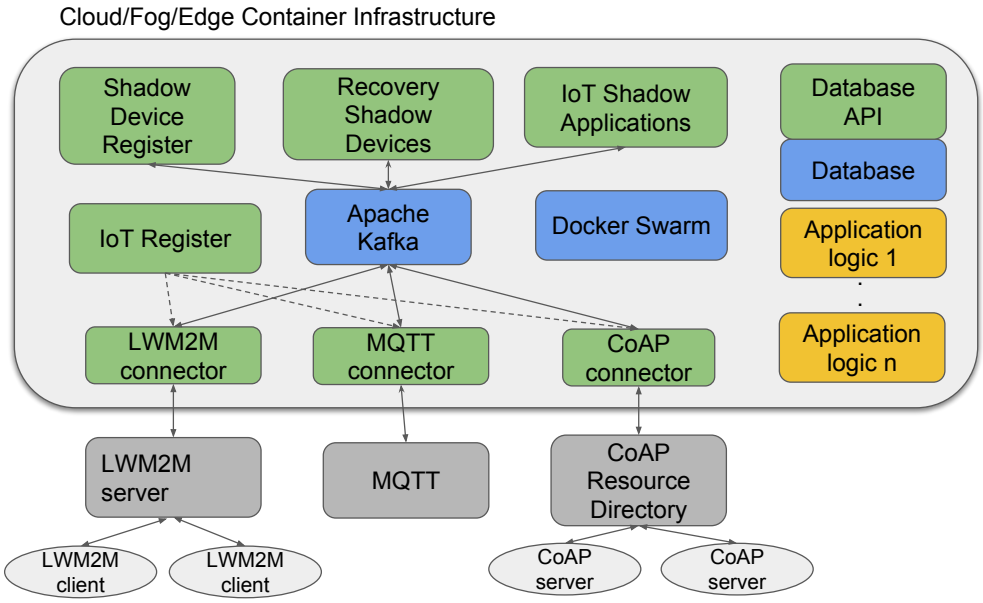

Fig. 1. An overview of the reliable architecture in a container infrastructure. Green box: new container component defined in this architecture; blue box: technology used to support this architecture; yellow box: application logic to be deployed in this architecture; grey box: IoT devices and protocols.

initiates the corresponding IoT connector for the IoT device/s to start a monitoring process for the underlying devices.

2) **IoT Connector**. This component implements the specific APIs for accessing IoT protocols. Once it has been deployed by the IoT register, it starts monitoring the device or the networks associated with the device. For instance, in the case of a LWM2M server, this component monitors all the endpoints and their resources registered in the server. Upon a resource or endpoint change, the IoT Connector is notified and sends a message through Apache Kafka to start a fault recovery. This component also waits for Kafka messages to perform operations in the IoT such as starting a device observation.
3) **Apache Kafka**. Apache Kafka is a distributed message queue that dispatches and consumes large amounts of data with a low latency [1]. In this architecture Apache Kafka serves as an interconnection system for the software stack, as shown in Fig. 2 and 3. Many distributed queue frameworks delete information after consumption. On the contrary, Apache Kafka maintains the information available over a configured period of time or until the stored date grows over specific configurable limit, enabling later data retrieving by components. This is quite helpful in the case of component failure. Finally, it makes possible the creation of consumer's groups, which distribute the data consumption among customers (application logic, software stack) and prevent ingestion bottlenecks.
4) **Recovery shadow devices**. Once a device had been disconnected (e.g., due to battery consumption) a notification is received in this component. Then, it checks if the available resources of the device (e.g., video surveillance, temperature) are being used or required by an application logic. If so, it determines a suitable device that could provide the same behaviour on the shadow device group, and sends a notification though Kafka to the corresponding IoT connector to enable this change.
5) **IoT shadow applications**. Requests by IoT applications for accessing or actuating with shadow devices are managed by this component. Upon an application request, it provides the physical device accessing through the corresponding IoT connector of the shadow device.
6) **Shadow device register**. This component serves a Web UI in which shadow devices can be defined and managed. As stated, each shadow device is associated with a JWT, which will be used by physical devices to become part of shadow devices.
7) **Database API**. The architecture needs a database to store some information such as shadow devices, JWTs and the physical devices status. This API is built upon a distributed database (such as MongoDB) to provide the unified API for accessing the database. If a database is replaced by another one, the API would not suffer changes and only should be implemented the specific API of the new database.
8) **Docker Swarm**. Docker Swarm manages the deployment and allocation of containers and enables high availability and load balancing on them though container replicas. In particular, a cluster of Docker Swarm managers with high availability will manage the infrastructure.
9) **Application logic**. Finally, the application logic represents the business logic that will make use of the

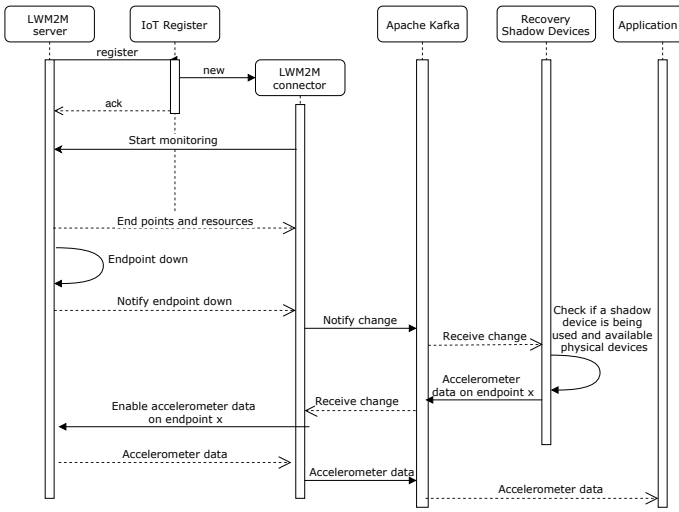

Fig. 2. Sequence diagram of a LWM2M server registration and fault recovery process after an endpoint's disconnection.

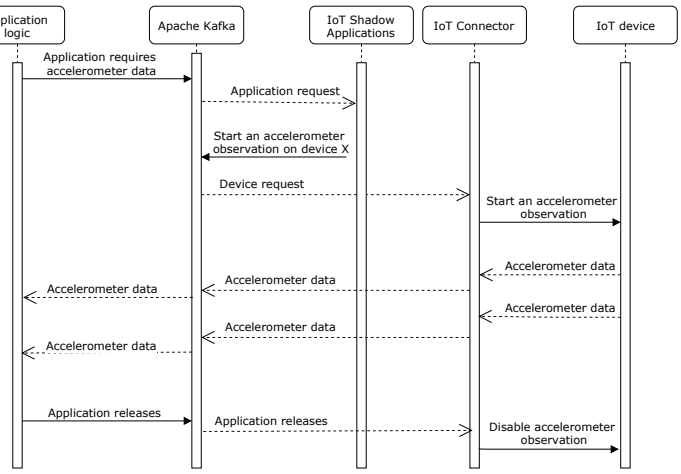

Fig. 3. Sequence diagram when an application logic is deployed and requires accelerometer data.

architecture to compose the final applications. The application logic can also be executed as microservices such as a distributed neural network where each layer or neuron represent a microservice. Apache Kafka Streams, a library that processes information directly in Apache Kafka, it is intended to be used as processing engine.

C. Architecture interactions

The register of a LWM2M server and a recovery process once one of its endpoints fails are shown in Fig. 2. Firstly, the LWM2M server sends a HTTP request with JWT authentication for registering itself in the system. Once the request is received and validated, the IoT Register deploys the corresponding IoT Connector for the registered IoT device, a LWM2M connector in this case. From that moment on, the LWM2M connector starts monitoring the LWM2M server, its registered endpoints and their resources. Any change in the registered devices in the LWM2M server will be detected by this latter, and by the IoT connector. Later on, an endpoint that is providing accelerometer data to a structural health monitoring application fails. The LWM2M connector detects the endpoint disruption and sends a message to be managed by the Recovery Shadow Devices component. Once this component had checked a shadow device is being compromised, it sends a notification to obtain accelerometer data again on an available device in the shadow device. This last notification is managed by the LWM2M connector, which starts an accelerometer observation on the selected endpoint and the application logic starts receiving accelerometer data again.

On the other hand, the sequence diagram when a structural health monitoring application requires accelerometer data of shadow devices is shown in Fig. 3. First of all, the application expresses its interest in obtaining accelerometer data. Once the request has been received by the IoT Shadow Applications component, a request is sent to the corresponding IoT connector for monitoring accelerometer information. Finally, the application starts receiving accelerometer information until the application is released, and the IoT device stops monitoring.

IV. Use case: structural health monitoring with distributed convolutional neural networks

This architecture is intended to reduce the latency and bandwidth in the IoT communications, but also reducing the response latency by moving the computation closer to where it is produced. This would reduce the response time in situations where it is critical a fast response, such as structural health monitoring. Moreover, all devices in this scenario will be monitored and recover in case of device failures, thereby maintaining the service continuity of the application logic. In this application scenario both are required, a low latency and high availability.

Convolutional neural networks have been widely used for applications such as image and video recognition and image classification, typically in large processing systems like CC. Currently, these networks are being deployed in edge/fog/cloud architectures for various purposes. It has been demonstrated that having many layers can increase the recognition robustness [11], these layers are typically in a fog/edge/cloud architecture; but also this architecture would reduce the execution latency [12]. Therefore, by having multiple layers in each one of the parts of the architecture: edge, fog, cloud; the processing time would be reduced, and results could be more accurate.

The use case intended to be deployed in this architecture is structural health monitoring [13], a mission-critical application that continuously monitors the behaviour and condition of critical infrastructures, such as nuclear power stations, bridges and transportation networks. In this scenario, it is required fault-tolerance, a low latency and continuously device monitoring without interrupting the service continuity, otherwise a dangerous situation could not be detected on time. In these cases are where the architecture presented in this paper should be deployed. The data analysis could be carried out

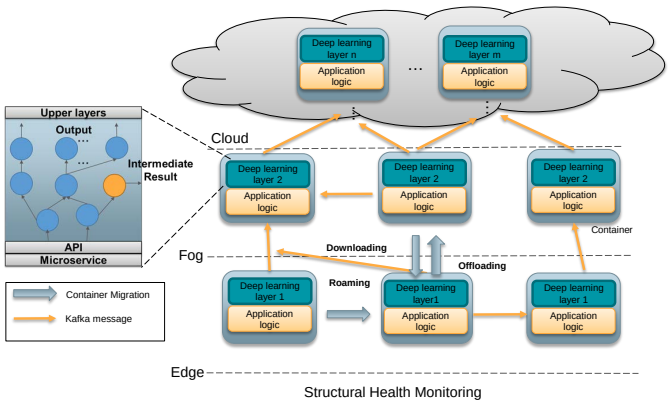

Fig. 4. Structural health monitoring with distributed convolutional neural networks and intermediate results.

through distribute convolutional neural networks allocated in containers along the architecture, in order to reduce the actuation time and prevent dangerous situations. Each layer of the architecture can provide intermediary results as shown in Fig. 4, such as preliminary and critical detection on the edge, intermediary processing on the fog, and long-term analysis on the cloud. Therefore, the application logic could act as soon as possible with this architecture when a damage has been detected in the infrastructure.

V. Container Infrastructure Validation

To validate the feasibility of the container infrastructure, we have performed an evaluation of Docker containers in a cluster of fog devices. Our container infrastructure is built over a cluster composed by 12 Raspberry PI 3 Model B+. Raspberry PI 3 Model B+ is system on chip (SoC) device equipped with a quad-core 64-bit ARM Cortex A53 working at 1.2 GHz and a 1GB of RAM. Raspberry PI is a representative device on the edge and represents a low-cost and scalable solution for a fog infrastructure. The connection between them is provided by means of a 10/100 Cisco Catalyst 2960 switch. The Raspberries have been equipped with HypriotOS, which is an Operating System especially suitable to be used with Docker in the Raspberry PI since it is installed by default and other features that help in the configuration tasks are included.

The proof of concept that we have carried out in our infrastructure have consisted of stress tests over Docker containers. A very simple web server has been deployed using Python 3.7 and Flask, a microframework for Python applications. The web server has been containerised using Docker Swarm. This way, we have deployed several services (i.e. several microservices) exposing the web server to the external world with a different number of replicas. Docker Swarm is responsible for the management and deployment of the replicas over the Raspberry PI cluster. In this test, we have chosen to do not tune Docker with any special option to see how it works with the default options. Docker Swarm also provides a default load-balancer that distributes new requests when they are directed to the web server. The configuration of the service with replicas is as follows. First of all, we choose an arbitrary node as master of the swarm. A key is automatically generated and used in the worker nodes to join to the swarm. From the outside world, the request can be directed to any node of the swarm. It is the responsibility of the Docker load-balancer to choose the most suitable worker node. In our configuration we have created services including 1, 2, 4, 8, 12 and 48 replicas to see how the behaviour of our infrastructure is in terms of latency and throughput.

On the client side, we have used a simple PC (Intel Core 7, 3.60 GHz, 8 GB RAM, Windows 10) simulating simultaneous invocations. To do this, we have used bombardier, a HTTP(S) free benchmarking tool. It can be easily configured to use different concurrent connections and number of requests.

Figures 5 and 6 show the result of our tests. In Fig. 5 we can see the average latency for 10,000 requests under different configurations. The test has been carried out using 10, 250, 500, 750 and 1000 concurrent connections. For every number of connections, we have measured the average latency time in our services that included (as it has been mentioned) 1, 2, 4, 8, 12 and 48 replicas respectively. Additionally, we have also measured the average latency time when the web server is executed in a native way (i.e. Docker is not used). The results are coherent. The worst case happens when a Docker service with only one replica is used. In this case, we are not getting any benefit from the use of Docker. Moreover, the overhead of Docker has to be added when compared with the native option. The benefit of using several replicas can be seen from the very beginning. The greater the number of replicas, a lower average latency is obtained. The two most significant points in the figure can be observed when 500 and 1000 connections are used. In these cases, when the number of replicas is greater than 1, we have obtained the same results. We only have obtained a worst result in the case of 2 replicas and 1000 connections, but in any case, this case is better when compared with the native and 1-replica configurations. As a conclusion, we can say that no additional benefit is obtained when a high number of requests are performed at the same time and several replicas are used.

Fig. 6 shows the throughput (number of requests per second) with the same configuration described below. In this case, the worst results are obtained again when only one replica is used, followed by the native option. In general, the throughput is lower when using 250 connections. After that, the throughput is progressively increased obtaining the best results with only 8 replicas. If we combine the results of the two figures, it seems that there are not any potential benefits of using more than 8 replicas. Nevertheless, additional tests are required to discard other potential bottlenecks.

VI. Conclusions and Future Work

IoT applications do not only comprise software, but also hardware. The latter is generally very limited in IoT devices, and unlike microprocessors, the Moore's law brings more and more limited and low-power consumption IoT devices, which are enabling a wide variety of opportunities. Managing

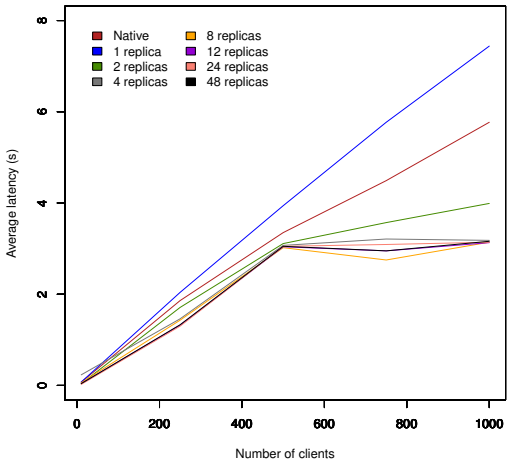

Fig. 5. Average response time (seconds) of a web server in native hardware and different replicas.

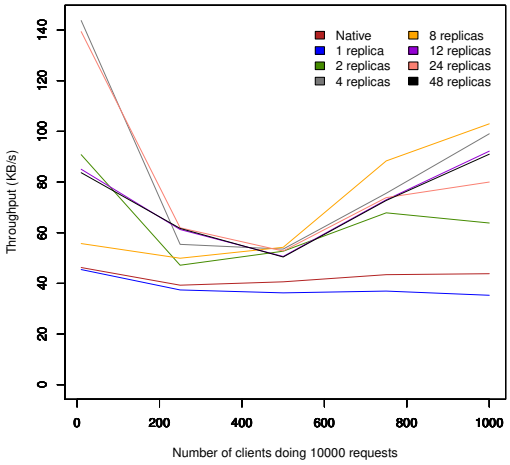

Fig. 6. Average throughput (KB/s) of a web server in native hardware and different replicas.

such a mesh of IoT devices without a proper control could lead to service disruptions, something to elude in mission-critical applications. In this paper, a reliable architecture for deploying and managing fault-tolerant IoT applications is defined. The architecture manages IoT device failures along the infrastructure, adapting the IoT applications to have connected and stable data sources and actuators. Therefore, applications developers only need to focus on the application logic, whereas the physical infrastructure is managed by the architecture. Due to the unique possibilities offered by the multiple IoT protocols, the architecture is application agnostic with respect to the IoT protocol. Finally, the architecture has been designed to be deployed in a fog/edge/cloud architecture, enabling vertical (offloading) and horizontal (roaming) migrations to optimise the latency and processing time of the final applications.

An evaluation has been carried out over a fog infrastructure with interesting results where a higher number of replicas do not provide better results. Additional tests are required, but in case no other reasons were detected, we could have to tune the default parameters of Docker or to modify the architecture of our cluster.

As future work, enabling fault-tolerance and enriched processing directly on the IoT devices and an exhaustive evaluation in a mission-critical application are a must. Lastly, we are currently evaluating Apache Kafka mirroring (MirrorMaker) for having multiple cluster replicas, enabling horizontal migrations between them and having roaming devices.

ACKNOWLEDGMENT

This work is funded by the Spanish projects RTC-2016-5164-8 ("KAMIC: Development of a Self-Installable Kit for Structural Monitoring of Critical Infrastructures") and RT2018-099777-B-100 ("rFOG: Improving latency and reliability of offloaded computation to the FOG for critical services").

REFERENCES

[1] M. Díaz, C. Martín, and B. Rubio, "State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing," *Journal of Network and Computer Applications*, vol. 67, pp. 99–117, 2016.

[2] "Internet of things at a glance," Available online: https://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/at-a-glance-c45-731471.pdf, (accessed on 28 October 2018).

[3] M. García-Valls, A. Dubey, and V. Botti, "Introducing the new paradigm of social dispersed computing: Applications, technologies and challenges," *Journal of Systems Architecture*, p. In press, 2018.

[4] S. Yi, C. Li, and Q. Li, "A survey of fog computing: concepts, applications and issues," in *Proceedings of the 2015 workshop on mobile big data, Hangzhou, China, June 22-25,*. ACM, 2015, pp. 37–42.

[5] C. Dupont, R. Giaffreda, and L. Capra, "Edge computing in iot context: Horizontal and vertical linux container migration," in *2017 Global Internet of Things Summit (GIoTS),Geneva, Switzerland, 6-9 June*. IEEE, 2017, pp. 1–4.

[6] P. H. Su, C.-S. Shih, J. Y.-J. Hsu, K.-J. Lin, and Y.-C. Wang, "Decentralized fault tolerance mechanism for intelligent iot/m2m middleware," in *2014 IEEE World Forum on Internet of Things (WF-IoT), Seoul, Korea, 6-8 March*. IEEE, 2014, pp. 45–50.

[7] "Lightweight m2m (lwm2m)," Available online: https://www.omaspecworks.org/what-is-oma-specworks/iot/lightweight-m2m-lwm2m/, (accessed on 22 March 2019).

[8] C. Martín, J. Hoebeke, J. Rossey, M. Díaz, B. Rubio, and F. Van den Abeele, "Appdaptivity: An internet of things device-decoupled system for portable applications in changing contexts," *Sensors*, vol. 18, no. 5, p. 1345, 2018.

[9] A. Javed, K. Heljanko, A. Buda, and K. Främling, "Cefiot: A fault-tolerant iot architecture for edge and cloud," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT), Singapore, 05-08 February*. IEEE, 2018, pp. 813–818.

[10] "Eclipse hono," Available online: https://www.eclipse.org/hono, (accessed on 21 March 2019).

[11] S. R. Kheradpisheh, M. Ganjtabesh, S. J. Thorpe, and T. Masquelier, "Stdp-based spiking deep convolutional neural networks for object recognition," *Neural Networks*, vol. 99, pp. 56–67, 2018.

[12] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu, "Jalad: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution," *arXiv preprint arXiv:1812.10027*, 2018.

[13] L. Alonso, J. Barbarán, J. Chen, M. Díaz, L. Llopis, and B. Rubio, "Middleware and communication technologies for structural health monitoring of critical infrastructures: A survey," *Computer Standards & Interfaces*, vol. 56, pp. 83–100, 2018.