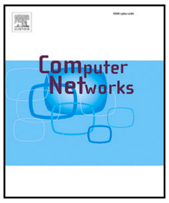
Computer Networks 209 (2022) 108868

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



WoTemu: An emulation framework for edge computing architectures based on the Web of Things

Andrés García Mangas [a,*], Francisco José Suárez Alonso [b], Daniel Fernando García Martínez [b], Fidel Díez Díaz [a]

[a] Technological Center for Information and Communication (CTIC), W3C Spain Office host, Ada Byron 39, 33203 Gijón, Spain
[b] Department of Computer Science, University of Oviedo, Spain

ARTICLE INFO

Keywords:
Web of Things
Internet of Things
Edge computing
Container orchestration

ABSTRACT

The edge computing model is an approach to Internet of Things (IoT) architectures based on the redistribution of services and infrastructure from centralized clouds to locations closer to IoT devices. The Web of Things (WoT) is another important IoT trend, currently led by the W3C, which aims at solving the IoT interoperability problem by adopting proven technologies and patterns from the Web. The design and validation of IoT deployments based on these paradigms is a complex task that involves multiple services, heterogeneous hardware and diverse communication technologies. Testing such projects in real world conditions usually requires a significant investment of resources. There are simulation tools that can assist in this process with much lower barriers of entry, however, they involve the designer making modelling assumptions that are not always representative of the real systems. This work presents an emulation tool for IoT projects based on the edge computing model that is able to seamlessly scale horizontally by leveraging container orchestration (Docker swarm mode). Furthermore, the W3C WoT model is included as a first-class citizen, enabling the designer to model all actors in the system as Things. The tool can run the real production code with minimal modifications and provides meaningful insights into the behaviour of the proposed architecture. This knowledge serves to rapidly iterate the optimization process, simplifying design issues and the detection of bottlenecks before committing to a real deployment in the field. A real-world scenario is also emulated in order to demonstrate its capabilities and validate its contribution.

1. Introduction

The concept of fog computing was first introduced by Flavio Bonomi et al. [1] as a platform which exposes services that had previously been considered native to the cloud (i.e. networking, computing, storage) in any layer between the IoT data sources and the core data centres. The distinction between *edge computing* and *fog computing* is sometimes vague. Fog computing can be considered as the prime example and even an evolution of the more general edge computing concept [2]. It implements the core idea of edge computing, that is, the redistribution of resources from the cloud to locations adjacent to the IoT end devices. However, it is not constrained to the edge and can exist in the upper layers that are closer to the cloud. In this work, we refer to *edge computing* as a more general term, which includes *fog computing*.

Adoption of the edge computing architectural model can provide solutions to transmission, computation and storage requirements [3] that the cloud computing model is unable to handle. Examples include adapting to computational loads that evolve due to the geographic mobility of end devices [4], and ensuring consistently low latency [5].

The rapid growth of the IoT has led to a high degree of heterogeneity. Many different systems and platforms can be found, providing different implementations for similar challenges [6]. This can cause interoperability problems and data silos: isolated datasets that may become costly barriers for integration into other systems or applications. The Web of Things is a reasonably recent paradigm [7] which aims to meet the IoT interoperability challenge by adopting proven patterns and technologies from the Web.

In one of the earliest works on the WoT [8], Guinard et al. proposed the integration of physical Things into the Web by exposing an HTTP API based on the REST architectural model. This work is one of the foundations of the W3C WoT, which is currently the leading implementation of the WoT paradigm. The W3C WoT reference architecture [9] revolves around the concept of the Thing—any virtual or physical

* Corresponding author.
E-mail addresses: andres.garcia@fundacionctic.org (A. García Mangas), fjsuarez@uniovi.es (F.J. Suárez Alonso), dfgarcia@uniovi.es (D.F. García Martínez), fidel.diez@fundacionctic.org (F. Díez Díaz).

https://doi.org/10.1016/j.comnet.2022.108868
Received 16 July 2021; Received in revised form 2 February 2022; Accepted 22 February 2022
Available online 2 March 2022

Table 1
Summary of related simulation and emulation tools.

Tool	Highlights	Real code	Horizontal scalability
CloudSim [18]	Widely extended cloud computing simulator.	No	*Not applicable*
ContainerCloudSim [19]	CloudSim extension that enables simulation of containers.	No	*Not applicable*
iFogSim [20]	CloudSim-based IoT fog computing simulator.	No	*Not applicable*
CloudSim Plus [21]	CloudSim fork with additional features and a focus on software engineering best practices.	No	*Not applicable*
YAFS [22]	Edge computing simulator able to dynamically react to events during the simulation.	No	*Not applicable*
Mininet [23]	Network emulator for the creation of network testbeds consisting of virtual hosts, switches and links.	Yes	No
MaxiNet [24]	Enables Mininet to run across multiple hosts and adds Docker-based virtual hosts.	Yes	Manual container placement in non-trivial cases.
Containernet [25]	Mininet fork that adds Docker-based virtual hosts.	Yes	No
EmuFog [26]	Tool to optimize edge computing architectures (in the form of MaxiNet experiments) based on a series of constraints.	Yes	Same limitations as MaxiNet
Fogbed [27]	Emulation of edge computing topologies based on Containernet and MaxiNet.	Yes	Same limitations as MaxiNet

entity that is described by a Thing Description (TD) [10]. A TD is a machine-readable document commonly serialized in JSON-LD [11] that contains Thing metadata. All functionalities exposed by the Thing are modelled in the TD as one of three types of interactions: properties, actions and events, as defined by the WoT Interaction model. High-level interactions are then mapped to specific platforms by WoT Binding Templates [12], e.g. a template could map property write operations to HTTP POST requests. Finally, the Scripting API specification [13] is an optional but important building block that defines the public programming interface exposed by compatible WoT runtimes. This API enables developers to quickly expose, consume and discover Things while abstracting from the implementation details.

Containers are a ubiquitous presence in the cloud computing space. This technology leverages Linux kernel primitives (e.g. *cgroups*) to enable isolated, portable execution of processes. Docker [14] is the most relevant container product, providing an entire stack of components including container runtimes and image repositories. It should be noted, however, that there are alternatives to individual pieces in the Docker ecosystem, for example Red Hat *crun* [15] is an alternative container runtime to *runc*. Moreover, container orchestration tools, such as Kubernetes [16] or Docker Swarm mode, enable the management of container-based applications across multiple hosts. Given the dynamic nature of edge computing and the similarities with cloud computing, container technologies can be of benefit for deployment, development and maintenance tasks [17].

1.1. Background

This section describes a set of relevant simulation and emulation tools in the context of cloud computing and the Internet of Things. Table 1 contains a summary of references to the tools. The tools have been categorized depending on their ability to run real code and scale horizontally. More specifically, horizontal scalability is the ability of a system to increase its performance and capacity by adding more nodes to the system. Vertical scaling is based on upgrading the components of each node (i.e. CPU, GPU, memory, storage). Horizontal scalability tends to be a more optimal strategy in this context, as vertical scaling has significant cost barriers in the majority of cases. Furthermore, container orchestration tools contribute significantly to achieving seamless horizontal scalability, which is one of the main rationales for using container orchestration as the foundation of WoTemu.

CloudSim [18] is a widely referenced software framework that allows for the modelling of systems based on cloud computing entities, e.g. datacenters, virtual machines, using a programmatic interface. These systems can be used to represent a wide variety of cloud computing scenarios and simulated in a repeatable fashion. ContainerCloudSim [19] is a built-in module that augments CloudSim to include the concept of application containers, enabling researchers to define strategies for container allocation on virtual machines and model cloud architectures in terms of Containers as a Service (CaaS).

CloudSim is the cornerstone for a range of simulation tools that provide extended functionality for other domains. The iFogSim project [20] is one of the most relevant examples in this group. IoT applications in iFogSim are represented as directed graphs based on the Distributed Dataflow (DDF) model [28]. Vertices in these graphs correspond to three distinct types of entities: *sensors*, which generate messages following a predefined transmission distribution (e.g. deterministic, normal); *actuators*, which act as the receiving end of messages; and *modules*, a class meant to symbolize any algorithm or stage in an IoT processing pipeline (e.g. a classification algorithm). Messages exchanged by the previous entities are modelled as edges, and are parameterized by their computation and network costs. *Application loops* represent the flow of messages in the IoT application graph and must be explicitly initialized to indicate the graph paths to monitor. *Modules* are then allocated on *fog devices*, which are described in terms of CPU resources, available memory and power usage. Users have the option of setting the *module* placement or using an automated strategy.

The authors in [21] introduce CloudSim Plus as an alternative fork of CloudSim with the aim of improving code quality, adopting software engineering best practices and including exclusive features. These efforts have provided several advantages, such as a simplified programming interface that requires less code for comparable CloudSim examples; an updated hierarchy of classes that improves extensibility and facilitates the implementation of ad hoc algorithms; and the ability to define vertical and horizontal autoscaling strategies for virtual machines.

Unlike the previous references, YAFS [22] is an edge computing simulator built from the ground up, outside of the CloudSim ecosystem. It is, however, fairly similar to iFogSim in its modelling approach—applications are represented as DDF graphs, and nodes are characterized in terms of computation, memory and cost. One of its most significant contributions is its ability to define ad-hoc algorithms that run in parallel with the simulation. This enables the designer to dynamically update the system in response to simulation events, for example to deploy new actuators.

Mininet [23] is a relevant project in the Software-Defined Networks (SDN) field that is closely related to edge computing emulation. It leverages features native to Linux, such as network namespaces, to build a framework that is capable of emulating networks with hundreds of entities (e.g. virtual hosts, switches and links) in a single host. One of its main limitations – scaling to multiple hosts – is addressed by MaxiNet [24]. To this end, MaxiNet uses an ad-hoc cluster manager and the Generic Routing Encapsulation (GRE) protocol to build IP-in-IP tunnels that interconnect Mininet workers. In addition, it has support for Docker containers. However, complex scenarios tend to require manual container placement, as the automated placement algorithm is rather limited. In a related note, Docker support can also be enabled for Mininet experiments thanks to the contribution of the Containernet [25] fork.

Building upon MaxiNet, EmuFog [26] provides capabilities for efficient design of edge computing topologies. It enables users to define a set of end clients (*device nodes*) and a catalogue of edge computing devices (*fog nodes*) in terms of CPU, memory and latency cost. As input this tool uses a network topology in BRITE [29] format. It then performs an optimization process to arrive at a MaxiNet experiment comprised of a *fog node* selection from the previous catalogue that fulfils the user constraints. Although it offers support for the execution of Docker containers, it is based on MaxiNet, and thus shares its limitations.

In a similar fashion to EmuFog, Fogbed [27] leverages Containernet and MaxiNet to offer both local (i.e. Mininet) and distributed (i.e. MaxiNet) emulations of Docker-based edge computing network topologies. An interesting feature of Fogbed is its ability to dynamically update the topology of a running experiment, for example to add a new host.

All of the software utilities described above have proved to be of great help in the design process of IoT applications. For instance, the authors in [30] present an optimized remote pain monitoring IoT architecture based on the simulation results provided by iFogSim.

To date, research on design tools for edge and cloud computing architectures has mostly focused on a more theoretical and high-level view. Users are required to settle on a set of parameters and models to characterize their scenarios as accurately as possible. This entails significant difficulties and compromises during the design process. It is reasonable to argue that lowering the barrier of entry may drive the adoption of design tools in this domain. This would in turn serve to optimize resources and avoid costly deployment issues that could have been detected in an earlier stage.

To the best of our knowledge, all of the studied tools, which aim at emulating real code, struggle to some degree with horizontal scaling. Furthermore, they abstract from the full scope of application monitoring, leaving significant time-consuming responsibilities, such as identifying the flow of traffic between services, to the user.

1.2. Motivation

Considering these limitations, this work proposes an application-centric emulation framework based on container orchestration to help in the design of IoT/WoT systems that follow the edge computing architectural model. The motivation is detailed in the following list.

- To bridge the gap between developers, who may be more inclined to test IoT architectures using real production code, and the field of theoretical simulation tools.
- To provide, in addition to high-level host measurements, performance metrics that are focused on the application itself and therefore more meaningful to developers. For instance, network traffic for each application component broken down by protocol would be of great use.
- To lever the capabilities of a modern container orchestration utility, such as Docker swarm mode, for emulation of IoT architectures with seamless horizontal scalability. This approach offers a sensible compromise between the realism of hardware testbeds and the experiment scale that is achievable with theoretical simulators.
- To make the WoT paradigm a first-class citizen in the emulation scenarios by enabling users to represent all actors in the system as WoT Things.

2. Design

This section presents the architecture of experiments in WoTemu and explains the rationale behind the design choices. The main types of entities in WoTemu are identified and defined, and extended subsections for the most relevant modules in the framework are also included.

A fully functional implementation of WoTemu is publicly available in both Zenodo [31] and GitHub[1] under the MIT licence. The minimum requirements of this implementation are Python 3.6, Docker Compose 1.27.0 and Docker Engine 20.10.0.

WoTemu is based on Docker swarm mode, a container orchestration tool included in the Docker ecosystem. It should be noted that Kubernetes, which is arguably the most popular orchestration tool, was also considered. However, the following reasons led to the adoption of containers in general and swarm mode in particular as the foundation of WoTemu.

- Docker is a widely extended containerization platform that is easy to install and is readily available in multiple platforms. Swarm mode is the default built-in orchestration tool in Docker. It is also easy to configure, taking a few minutes at most to setup a cluster with multiple machines.
- Orchestration tools are characterized by straightforward horizontal scalability, which is a basic requirement to enable the emulation of experiments with a high number of entities, but which is lacking in tools in the current state of the art.
- Emulation of constrained platforms is reasonably simple using containers. Resources such as CPU and memory can be configured on a container-by-container basis.
- The life cycle of experiments is simplified by using containers. Existing resources (e.g. volumes, networks) can be simply removed from the hosts without leaving traces.
- The default *overlay* network stack implementation included in swarm mode is a great fit to represent isolated networks and connections between entities in an edge computing scenario.

Therefore, WoTemu is intrinsically linked to Docker swarm mode concepts. Throughout this paper there are references to swarm *services*, *tasks* and *networks*; these concepts are clarified below.

- A swarm *task* is linked to a specific container and is the atomic execution unit in a swarm (i.e. there is a container for each *task*). If a task fails for any reason (i.e. the container exits with a non-zero code) the swarm manager will attempt to reload it to maintain the desired swarm state.
- A swarm *service* is a template for *tasks*, describing, for example, the base image or the connected *networks*. A *service* may have multiple replicas, each one being a different *task*. Requests going into a *service* are balanced between all the replicas.
- A swarm *network* interconnects different *services*. *Services* in the same *network* are able to communicate with each other. It is important to note that a *network* is not limited to *services* in a single physical swarm node—Docker provides the overlay network driver to enable distributed networks.

Fig. 1 shows a general view of the WoTemu architectural model, including the different types of containers, which are described in more detail in the following paragraphs.

Node Node containers, also referenced to as *applications*, represent programs that must be monitored during the execution of an experiment to be analysed later. From a development standpoint, the application executed by a node takes the form of a Python

[1] https://github.com/agmangas/wotemu.

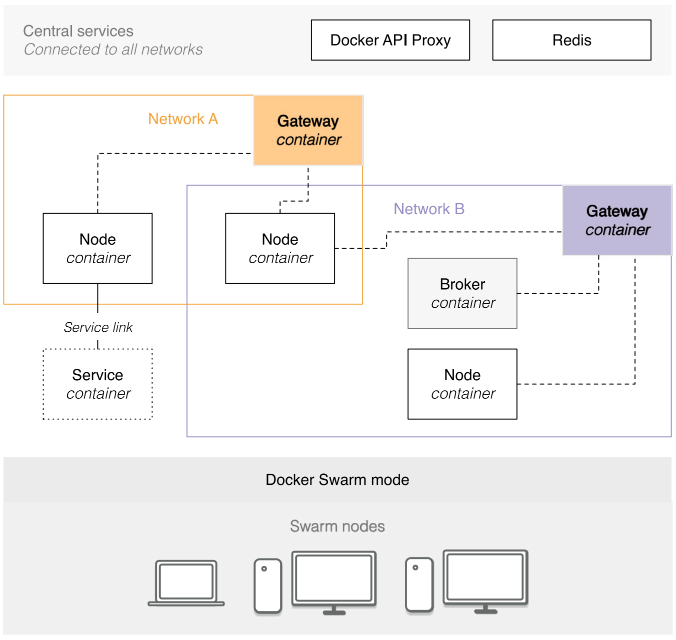


Fig. 1. Architectural model in WoTemu.

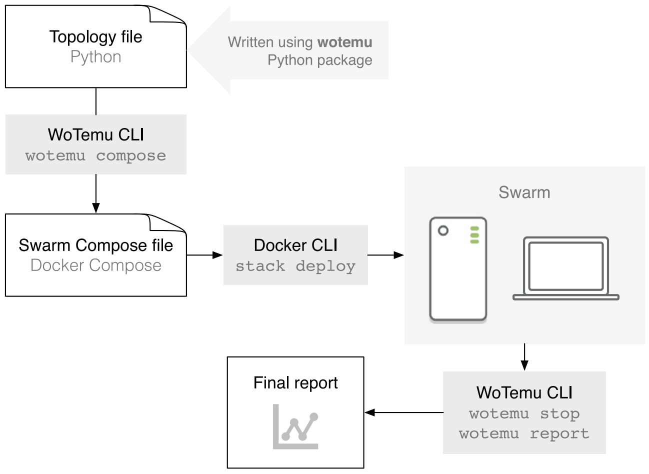


Fig. 2. Typical workflow for an experiment in WoTemu.

file exposing an asynchronous function, which takes the WoT runtime entrypoint, configuration and event loop instance as arguments. WoTemu is tailored for WoT applications, that is, programs that use the WoT runtime provided in WoTPy [32] to perform their operations in terms of exposing and consuming Things. However, WoTemu has no restrictions on what is executed in the application function, except that programs must follow the asynchronous I/O programming model of asyncio. Therefore, applications can act as emulated IoT devices, control programs for real IoT devices, processing pipelines, or any other element required in the scenario. Furthermore, WoTemu includes a series of *built-in apps*. These are configurable applications which can serve as mocks and placeholders for testing purposes.

Gateway There is one gateway container for each network in the experiment. These containers are created automatically and injected transparently into the communications of node containers, acting as middlemen to emulate real world network conditions. See Section 2.1 for further information.

Service This type of container represents any service or dependency that may be required by an application. Possible examples include databases, message queues or identity and access management. There are two main differences from other types of containers. The first is that no background monitoring processes run in service containers. Secondly, that services exist in their own isolated networks, and node containers must declare an explicit *service link* in order to communicate with them.

Broker Unlike the other Protocol Binding implementations that can operate in a self-contained fashion, the MQTT implementation requires an external MQTT broker component. Therefore, broker containers are treated as first-class citizens in WoTemu experiments, including the usage of background monitoring processes. The implementation is based on the widely-used, open-source Eclipse Mosquitto [33].

There are also two central services in all WoTemu experiments which are connected to all the networks in the topology:

Docker API proxy This container is an instance of the *tecnativa/docker-socket-proxy* [34] image. It serves as a gateway for other containers in the experiment to access the Docker API of the swarm through the internal overlay networks. The Docker API is a program interface to manage all entities in a Docker environment. This elevated access is required to provide containers with introspection capabilities so they can self-configure independently from the others. Section 2.1 contains further discussion on this design decision.

Redis This is an open-source, in-memory key–value data store. It is used to save historical data and measurements necessary to characterize and analyse the behaviour of the experiment. Using Redis [35] as opposed to a relational database results in lower latencies for read/write operations at the cost of an increased memory footprint. It is especially indicated for this case, as it ensures that the background monitor processes running on all containers have a small performance impact. Section 2.2 contains more information on the data that is persisted in the Redis central service.

One configuration of the entities describe above is known as a *topology*. An *experiment* is an execution of a topology. Topologies are defined in Python using the programmatic interface of WoTemu which are then converted into Docker Compose files with the WoTemu Command Line Interface (CLI). This workflow is shown in Fig. 2.

One of the main advantages of the WoTemu design is that users are not limited to built-in emulated IoT devices. WoTemu runs real code in real-time. In practice, any Python application can be integrated, which enables users to communicate with the majority of real IoT devices. For example, an application could be designed to communicate over MQTT with a real wireless sensor board, or even through a serial port in one of the swarm nodes. However, this would require defining placement constraints so the application is always deployed in the swarm node that the board is connected to.

2.1. Routing module

Real life networks, especially those used in the context of IoT deployments, must deal with packet loss, bandwidth and data transfer limitations. These issues must be factored into architecture designs to ensure an adequate implementation. The routing module in WoTemu enables users to emulate the behaviour of real networks transparently. To this end, WoTemu leverages the iproute2 [36] Linux package and the iptables [37] packet filtering tool. The traffic control utility of

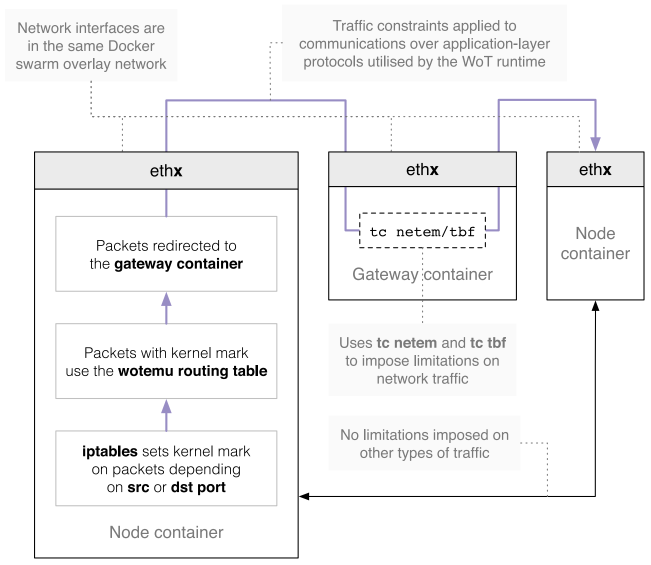

Network interfaces are in the same Docker swarm overlay network
Traffic constraints applied to communications over application-layer protocols utilised by the WoT runtime
ethx
Packets redirected to the gateway container
Packets with kernel mark use the wotemu routing table
iptables sets kernel mark on packets depending on src or dst port
Node container
ethx
tc netem/tbf
Gateway container
Uses tc netem and tc tbf to impose limitations on network traffic
No limitations imposed on other types of traffic
ethx
Node container

Fig. 3. Traffic shaping strategy in WoTemu.

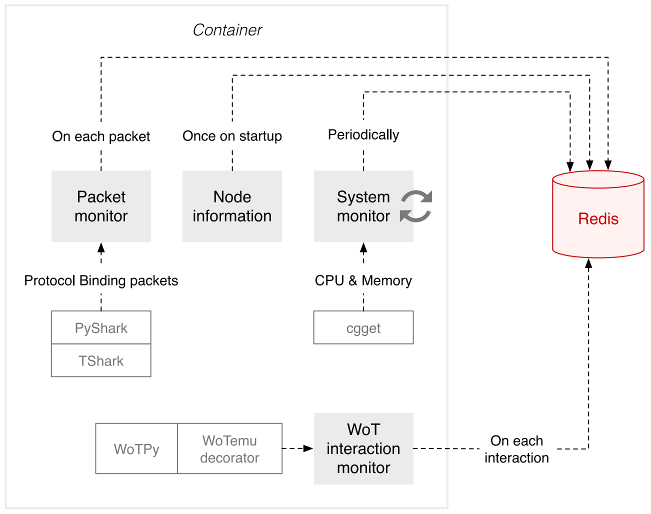


Fig. 4. Monitor processes transparently included in containers.

iproute2 (tc), and particularly the Network Emulator (NetEm) and Token Bucket Filter (TBF) modules, provide an interface to impose arbitrary restrictions on the quality of network connections.

All *node* and *broker* containers are automatically configured on startup by the WoTemu entrypoint with the appropriate routing rules to enable network emulation. These rules are detailed below.

- There is a set of iptables rules that match the ports of the application-layer protocols on the WoT runtime Protocol Binding implementations (i.e. HTTP, Websockets, CoAP and MQTT). These rules are located in the *mangle* table and appended to the OUTPUT chain to add an internal kernel *mark* to the matching packets.
- There is a rule in the routing policy database that causes all packets with the kernel *mark* stamped by iptables to use the WoTemu routing table.
- The WoTemu routing table contains entries to force all matching Protocol Binding packets or datagrams to go through the gateway container of the current Docker overlay network.

Gateway containers run a set of TC processes that shape the traffic passing through the container to emulate the latency, bandwidth and loss conditions of any given network. There is a single gateway container for each network where all traffic from the different containers in the network is aggregated. This leads to a more realistic emulation of the bandwidth of a network, as opposed to imposing bandwidth constraints in each container.

Fig. 3 shows a routing configuration where a subset of network communications are redirected through a gateway container that shapes traffic according to predefined rules.

The approach here is to ensure that each individual node is able to configure its own network stack without the intervention of a proactive centralized component. This fulfils one of the main objectives of WoTemu by increasing scalability, at the cost of decreased container isolation. The loss in isolation is because containers have access to the Docker API proxy service in order to retrieve information on the Docker swarm stack configuration. Examples of centralized information required by containers include:

- the swarm task identifier of the current container.
- WoTemu network identifiers currently attached to the container.
- the swarm task identifier of the gateway container for all WoTemu networks.
- all the currently existing container replicas of any given swarm service.

The main downside of the loss in isolation is that it leads to decreased security. It could be argued, however, that WoTemu experiments are not meant to be publicly exposed in a production environment and are ephemereal in nature. This position significantly eases concerns about security.

2.2. Monitoring module

The monitoring module provides visibility to the performance and operation of the experiment, capturing the relevant metrics during its execution. Captured data points are stored in the central Redis service to be used during the construction of the final report.

There are four distinct monitor processes that are injected into containers during the experiment. This is a transparent operation from the user's point of view, that is, the user-provided application is automatically augmented by the WoTemu entrypoint with the monitor processes. Fig. 4 shows an overview of the monitor processes, which are described in more detail in the following paragraphs.

Packet monitor Based on the TShark [38] network sniffer, this monitoring module runs a background process configured with a set of filters that match the ports of the Protocol Binding implementations. The PyShark [39] Python package is used to interface with TShark. All matching packets are pre-processed and stored in Redis to extract meaningful information related to the network footprint of the application.

Node information Runs only once on container startup and collects metadata about the current container that is necessary to identify the application during the generation of the final report. Examples of metadata captured by this process include the CPU model, extended information about network interfaces, the virtual IP addresses for the current swarm service and a snapshot of the environment variables.

System monitor Periodically retrieves the memory and CPU resource usage of the application. These metrics are read by *cgget*, a command line utility to read parameters of Linux control groups (cgroups). This is necessary to ensure that the measurements reflect the resources used by the current container instead of the system as a whole.

WoT interaction monitor The user-provided application function receives an argument that is a decorated version of the WoTPy entrypoint modified to capture and log all WoT interactions. These data points enable a higher-level, WoT-focused analysis of the behaviour of the experiment that cannot be obtained from low-level metrics such as CPU usage.

Furthermore, users can explicitly write ad-hoc *application metrics*. These are namespaced by the hostname of the container (*task*), and an arbitrary metric key provided by the user. They are then stored in Redis alongside the other data points generated by the monitor processes.

2.3. Benchmarking module

Memory and CPU constraints can be easily configured in containers. This can be leveraged to approximately emulate the behaviour of any given application in a constrained environment without the need of actually running it in a real platform. Memory limits are expressed as size, while CPU limits are expressed as a quota on the CPU Completely Fair Scheduler (CFS).

Memory limits retain their significance when applied to different host machines in the swarm cluster, regardless of the total amount of installed memory; for example, a limit of 100 MB means the same on all machines. However, the same does not hold for limits on the CPU CFS quota. The same quota can represent wildly different levels of computational power on different CPU models. There is a significant difference between full usage of a single core in a low power ARM processor and in a core in a modern desktop CPU.

WoTemu proposes a solution to the portability of processor constraints by using sysbench [40] synthetic CPU benchmark scores to represent the desired level of computational power in the container. The rationale behind using Sysbench is that it is a versatile, actively maintained, widely available and proven performance benchmark tool with multiple modules. Its popularity, especially in the context of database benchmarking [41], is reflected in the 3.9k stars and 771 forks in GitHub at the time of writing this paper.

The following list describes the strategy used in all swarm nodes to update the CPU quotas of the containers that require CPU constraints. Fig. 5 shows a diagram with a simplified view of this process.

1. The WoTemu entrypoint first checks the existence of an environment variable in the container indicating the target CPU performance score. If the variable is defined, the entrypoint uses the WoTemu CLI to update the CPU quota to match the desired computational power.
2. The first container in the swarm node that needs to update the CPU quota creates a new key in Redis and then runs a series of CPU benchmarks. This key is namespaced by the swarm node ID and is the same for all containers in the swarm node. Thus, all containers in the swarm node that reach at this step later are aware that a CPU benchmark is already in process.
3. The CPU benchmarking process consists in running multiple sysbench CPU benchmarks consecutively. Sysbench processes are executed inside ephemereal containers that are constrained with increasing levels of CPU quotas (up to 100% usage of a single core). The aim here is to obtain the coefficients of a fitted polynomial to characterize the performance of a core in the current CPU.
4. The rest of containers in the swarm node are simply waiting for the CPU performance coefficients to be set in Redis. This ensures that the CPU is not overloaded with multiple parallel benchmarks.
5. When the process ends, all containers update their own CPU quotas by solving the CPU performance polynomial for the target performance score. The target score is initially defined in the swarm service as an environment variable.

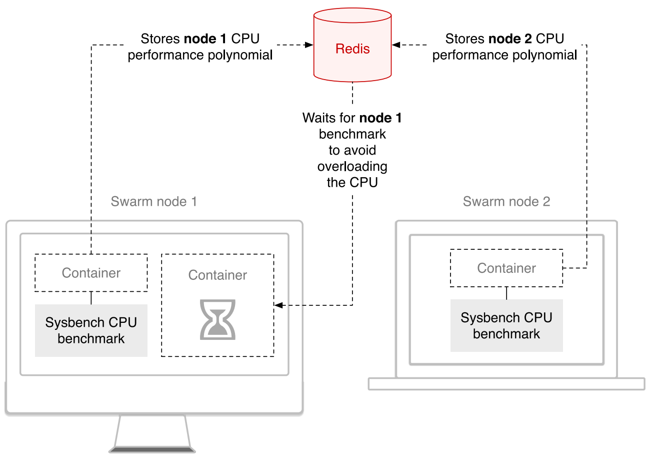

Stores node 1 CPU performance polynomial
Redis
Stores node 2 CPU performance polynomial
Waits for node 1 benchmark to avoid overloading the CPU
Swarm node 1
Swarm node 2
Container
Container
Container
Sysbench CPU benchmark
Sysbench CPU benchmark

Fig. 5. Processor benchmarking stage in WoTemu.

Although it would be preferable in terms of computational resources to reuse the performance polynomial for multiple experiments, WoTemu places more importance on being stateless and leaving minimal traces on swarm nodes after an experiment.

2.4. Reporting module

As discussed in Section 2.2, many performance metrics and data items are collected and stored in Redis during an experiment. This is, however, an intermediary state, as the format in which data is kept in Redis cannot be read by human users.

The reporting module transforms captured metrics into user-friendly representations. WoTemu supports two distinct report formats:

- Reports can be rendered as static webpages. To this end, the reporting module relies heavily on different charts provided by the Plotly Python Open Source Graphing Library [42]. An example of a WoTemu static webpage report is available online for demonstration purposes.[2]
- The user may need to apply additional processing or use the output of WoTemu as an input for another tool or script. For these cases, WoTemu supports rendering the report as a machine-readable JSON [43] document.

The following list provides a description of views and charts included in a WoTemu report.

Service traffic Total volume of network traffic grouped by *task* and *service* for data coming into the *service* and data coming out of the *service*. It is especially useful to detect at a glance the *services* with the highest activity from a network traffic standpoint.

Network traffic Timeseries of the evolution of network traffic grouped by *network*.

Resource usage rankings Distribution of the memory and CPU usage samples captured during the experiment for all *tasks*. This provides insight into the variance of the resource consumption of *tasks*.

Timeline of tasks Timeline of the creation and stop of all the *tasks* in the experiment. This is a convenient way of reviewing the frequency and occurrence of *task* failures.

[2] https://agmangas.github.io/demo-wotemu-report/.

Task resource usage Timeseries of the evolution of memory and CPU usage for a given *task*. Two different baselines are used in this case for comparison: the host machine (i.e. swarm node) and any resource constraints imposed on the container.

Task data transfer Timeseries of the volume of network traffic in a *task* grouped by network interface or protocol. Protocol names are identified by the TShark capturing process in the packet monitor.

Task interactions A WoT-focused view that provides insight into the performance of WoT applications deployed in node containers. This includes the distribution of latencies for interaction *request* verbs, the total count of successful and failed interaction verbs and the timeline of event verb occurrences.

For further clarity, the following list describes the structure of a WoTemu output report serialized in JSON format. Each element of the list represents a key in the JSON document; key names containing dots represent nested objects. In addition, as the reporting module makes heavy use of Pandas [44], some of the values in the document are JSON-serialized Pandas DataFrames (DF):

`app_metrics` An array of ad-hoc *application metrics* as defined by the user. The array items are objects that contain the metric key, the task where the metric was generated and an array of data points.

`service_traffic.inbound` DF that contains the total volume of network traffic grouped by *source task* and *destination service*. This allows the traffic coming into any given *service* to be analysed.

`service_traffic.outbound` DF that contains the total volume of network traffic grouped by *source service* and *destination task*. This allows the traffic coming out of any given *service* to be analysed.

`snapshot` DF that contains the status of all swarm *tasks* for all *services* at the time the emulation process is stopped. This includes the service ID, the most recent log entries, the date of creation and the container ID.

`tasks.<task>.info` An object with miscellaneous information about the given *task*, including network interfaces, memory limits, virtual IP addresses for each network and environment variables.

`tasks.<task>.interaction` DF that contains the time series of WoT interactions for both consumed and exposed WoT Things for the given *task*. This includes the payload of WoT events and the latency of consumed WoT actions.

`tasks.<task>.packet` DF that contains the detailed time series of network packets sent or received by the WoT runtime protocol binding implementations for the given *task*. This includes the packet size, the swarm network name and the protocol.

`tasks.<task>.system` DF that contains the time series of system utilization metrics (i.e. CPU and memory) for the given *task*.

A sample of a WoTemu output report in JSON format is available online [45] for further details.

3. Experimental results

3.1. Introduction

This section presents a real-world edge computing scenario and demonstrates how WoTemu topologies can be utilized to analyse the behaviour of IoT architecture proposals. The scenario in question is an approximation of the *Intelligent Surveillance* application discussed in iFogSim [20,46]. The rationale for selecting this particular scenario was the following:

- It highlights the differences between WoTemu and comparable simulation tools.
- It is complex enough and serves to demonstrate the flexibility of WoTemu topologies. The experiment integrates different services and techniques such as deep learning algorithms and NoSQL data stores.

The *Intelligent Surveillance* scenario is approximately emulated by a set of smart cameras with motion detection capabilities. In the event of motion detection, video streams from the cameras are forwarded to intelligent object detection modules. The output of the object detection modules is then used to update the Pan Tilt Zoom (PTZ) camera configuration. Finally, there is an interface for the users to check the activity of the system.

A large part of the value of WoTemu lies in the insight obtained from quickly iterating with intermediate versions of the topology. To illustrate this idea, Section 3.2 presents an initial architecture proposal that evolves to a more optimal version based on edge computing in Section 3.3.

The source code of the framework implementation and the experiment described in this section is publicly available [31] and the dataset containing the full experimental results is also published online [45]. All experiments were run on a swarm with 2 nodes with the following specifications:

- Intel Core i7-6770HQ CPU @ 2.60 GHz (4 cores, 8 threads),
- 32 GB of DDR4 2133 MHz (2 × 16 GB in dual channel configuration).
- 480 GB SATA3 SSD.

3.2. Cloud topology

A topology intended to fulfil the requirements of the *Intelligent Surveillance* scenario is shown in Fig. 6. The diagram includes the different nodes, the connections between them, the networks attached to each node and the typical flow of communications. This topology is based on a simple cloud computing approach where the sensor layer is directly connected to the cloud layer.

There are two sets of *cameras* that represent two geographical locations where surveillance cameras are deployed. All *cameras* are connected to a *detector* in the cloud through a MQTT broker. The *detector*, in turn, is monitored by a *cloud* node in the same LAN that maintains a history of the *detector* state. The *cloud* node is the only stateful element in the topology. Finally, a set of *user* nodes communicate with the cloud services.

The four types of node applications used in this topology are further described below:

Camera These nodes are instances of the *camera* built-in app, an application based on OpenCV [47] that aims at emulating a simple video camera. To this end, the application reads a test H.264-encoded local video file [48] in an infinite loop, performing naive motion detection in the process. Movement in the video is detected when the motion score goes over a threshold. The motion score is the mean of the differential of the most recent frames. Whenever motion is detected, the corresponding frame is converted to JPG, encoded in Base64 and emitted as an event interaction.

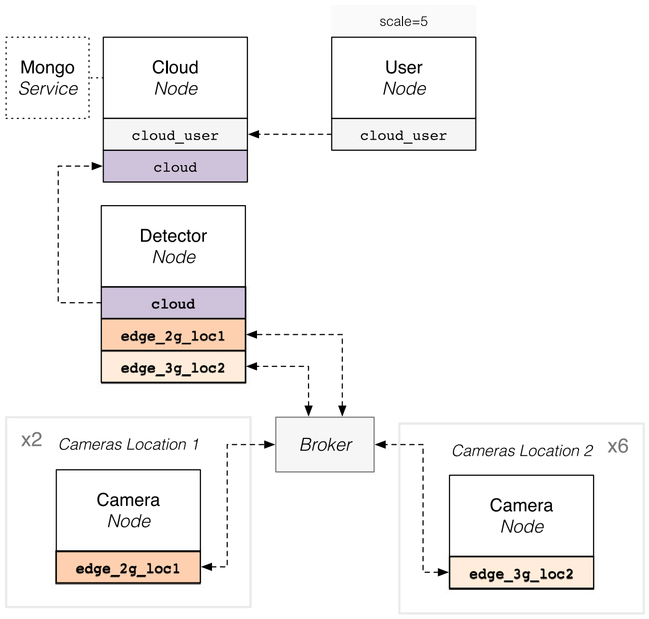


Fig. 6. Surveillance application topology (*cloud* version).

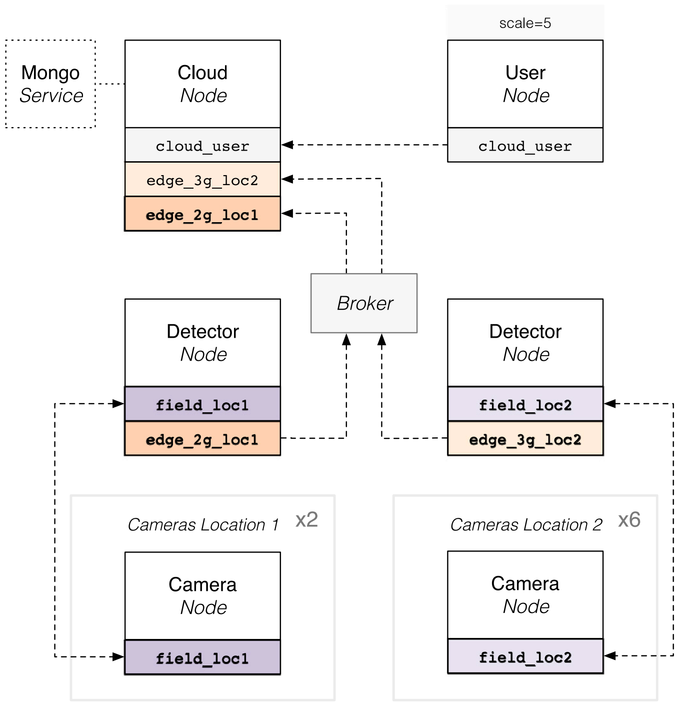


Fig. 7. Surveillance application topology (*edge* version).

Detector A *detector* subscribes to the JPG frame events of a set of cameras and then processes all received frames to perform face recognition. This process is based on the face-recognition Python package [49], which is publicly available in the Python Package Index. The face-recognition package in turn leverages the deep learning capabilities of dlib [50]. Two read-only properties are exposed by a *detector*. First, the *latestDetections* property contains the most recent in-memory video frame and detection result for each camera. Second, the *cameras* property contains the configuration, including the *camera* URLs. The PTZ adjustment action of a *camera* is invoked by the *detector* on each positive detection (i.e. a human face was detected in the video stream) as long as the previous invocation for said *camera* has already completed. The *detector* application is carefully designed to optimize resource usage: video frames are stored in a buffer queue in memory and processed asynchronously. The processing order of the frames is determined by their priority, which is a function of the age of the frame (in seconds) and the number of frames that have already been processed for that *camera*.

Table 2
Details of the networks in the application topologies.

Network name	Constraint profile	Latency	Jitter	Bandwidth
edge_2g_loc1	GPRS	700 ms	100 ms	50 kb
edge_3g_loc2	REGULAR_3G	300 ms	150 ms	1500 kb
field_loc1	WIFI	25 ms	5 ms	50 mb
field_loc2	WIFI	25 ms	5 ms	50 mb
cloud_user	CABLE	5 ms	5 ms	100 mb
cloud	CABLE	5 ms	5 ms	100 mb

Table 3
Node resource limits in the application topologies.

Node application	Memory	CPU score
Camera	256 MB	200
Detector (only in *edge* version)	1 GB	600

Cloud There is a single *cloud* node in the topology. The *cloud* node is an instance of the *historian* built-in app. Connected to a MongoDB service, it acts as an aggregator, data store and HTTP API for clients to retrieve the detection data from the different locations. A *historian* reads the properties of arbitrary WoT Things passed as arguments, writing the collected samples in MongoDB. In the particular instance of the *cloud* topology, the *cloud* node reads a single *detector* node that is located in the same LAN.

User The user nodes, which check the video frames and face detection results, represent the clients of the architecture. The *cloud* node serves as an entry point for these clients, exposing data in an aggregated fashion through HTTP. *User* nodes are instances of the *caller* built-in app, which continuously invokes the actions of an arbitrary WoT Thing passed as argument. The invocations follow a Poisson process in an attempt to model the behaviour of a group of real users. The request rate λ (1/s) of this process can be passed as an argument. However, it cannot grow indefinitely, as there are limitations imposed by using a single core. To overcome this limitation, the *user* service can be easily scaled up to generate a higher load by changing the number of replicas. In this particular experiment, both λ and the number of replicas are set to 5, which generates a significant volume of requests (approximately 25 requests per second).

It is important to highlight that although this particular experiment only uses built-in apps, that is, applications that are already included in WoTemu for convenience, any Python program that conforms to the asyncio asynchronous I/O programming model can be integrated into a WoTemu topology.

Table 2 details all the networks, including available bandwidth, latency and jitter. Networks `edge_2g_loc1` and `edge_2g_loc2` represent a cellular network backhaul, which is a common occurrence in IoT sensor deployments. Networks `cloud` and `cloud_user` are effectively unconstrained, with the former representing the LAN interconnecting cloud resources, and the latter the WAN connection between users and cloud resources.

Generally, devices in the sensors layer tend to be more constrained in terms of resources (i.e. memory, computation, storage) than devices in the edge layer, which, in turn, tend to be less capable than devices

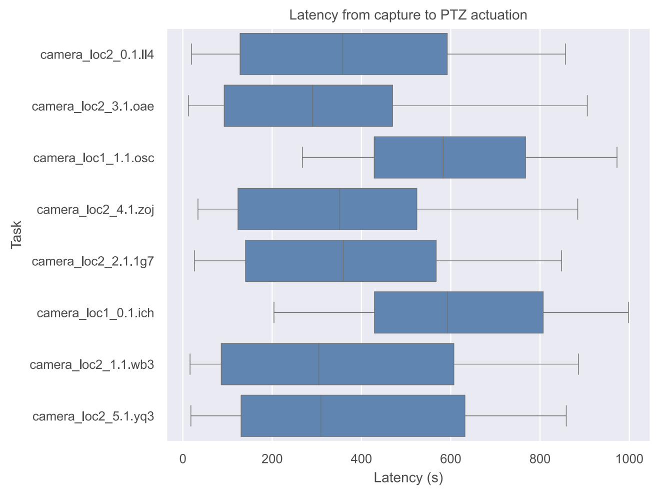


Fig. 8. PTZ latency (*cloud* version).

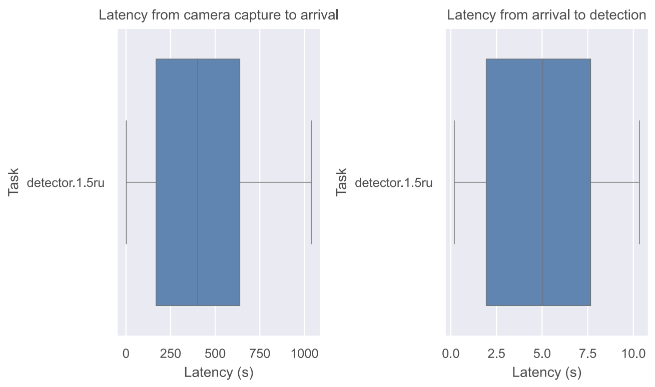


Fig. 9. Detection latency (*cloud* version).

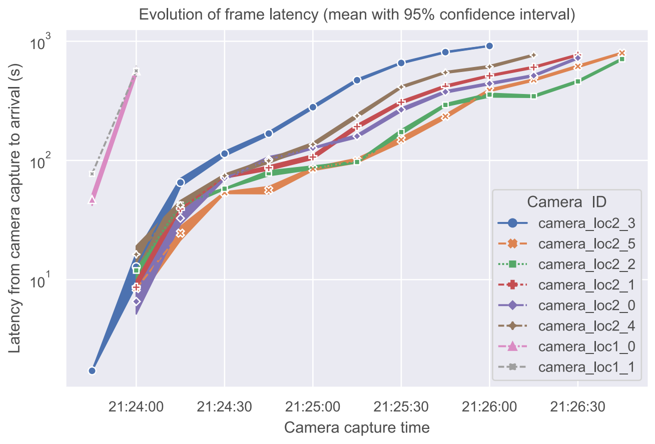


Fig. 10. Evolution of frame latency in the *detector* (*cloud* version).

in the cloud layer. This is a consequence of multiple factors, such as the energy constraints that are usually present in the sensors layer, and the number of devices in each layer. To incorporate these differences into the topology, Table 3 describes the memory and CPU resource constraints imposed on nodes. To represent a processor with very low capabilities, similar to those that could be found in a low-cost camera, the target CPU score of the *camera* nodes is set to 200. For comparison, the value for one thread in a fourth generation mobile Intel Core i7 is approximately 1000.

Please note that Tables 2 and 3 include additional elements, which are explained in Section 3.3.

Two ad-hoc application metrics (see Section 2.2) are defined for the *Intelligent Surveillance* scenario. These ad-hoc metrics are in addition to the performance metrics that are automatically captured by the monitor processes.

Detection latency Detection latency originates in the *detector* nodes and is divided into two metrics. The first one is the time in seconds from the moment a video frame is captured in a *camera* to the moment the frame is enqueued in the *detector*. The second one is the time from the moment the frame enters the queue to the moment the face detection process for that frame is completed.

PTZ latency This application metric originates in the *camera* nodes and contains the time between the timestamp in which a video frame is captured, and the timestamp when the PTZ adjustment invocation for said video frame is received. Note that a video frame must be processed by a *detector* before the PTZ adjustment action is invoked, and that not all video frames result in a PTZ adjustment action invocation. Moreover, the previously described detection latency is fully contained within the PTZ latency.

In the following figures, task names contain an apparently arbitrary alphanumeric suffix, separated by a dot. This is part of an alphanumeric ID that is automatically assigned by Docker swarm mode to uniquely identify each task. The number before said alphanumeric ID is the *replica number* (indexed from 1). Only the most representative and interesting figures are shown here for clarity. Section 2.4 gives a more detailed view of all the reports produced by a WoTemu experiment.

The PTZ latency and detection latency application metrics for the *cloud* scenario are shown in Figs. 8 and 9 respectively. The PTZ latency distribution is in the order of hundreds of seconds, which is totally inadequate. This is mostly due to network congestion caused by all *cameras* accessing the edge network concurrently to publish the video frame events. The low bandwidth, high packet delay and jitter (see Table 2) exhibited by the networks degrade the quality of the connection and result in issues such as excessive TCP retransmissions.

Latency issues are exacerbated by the message queueing feature of the MQTT binding. That is, messages are enqueued in the broker and delivered sequentially to the *detector*, which causes delays to accumulate.

Fig. 10 shows how the latencies of video frames from the *cameras* grow to the point where all frames are rejected by their respective *detectors* due to an excessive delay. This effect is significantly more pronounced for the first set of *cameras* due to the 2G network constraints imposed on the edge network `edge_2g_loc1`.

3.3. *Edge topology*

The results discussed in Section 3.2 prove that an arguably naive cloud computing approach is unfeasible for the requirements of this instance of the *Intelligent Surveillance* scenario. In this case, adopting the edge computing paradigm is a viable solution, as discussed below.

Fig. 7 shows the *edge* version of the topology, an alternative proposal that leverages the edge computing paradigm. The following list describes in more detail the differences with respect to the *cloud* version:

- Unlike the *cloud* version, a *detector* is locally placed at the edge of each set of *cameras* in an attempt to optimize latency and data transfer volume. That is, *cameras* are not directly connected to the WAN and are in the same LAN as their *detector*.

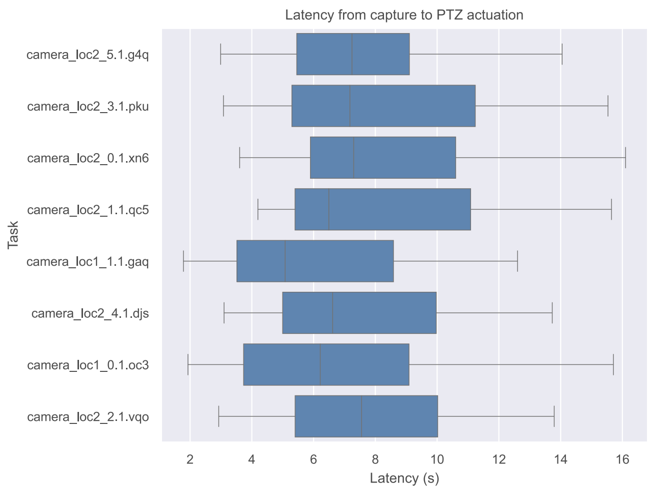


Fig. 11. PTZ latency (*edge* version).

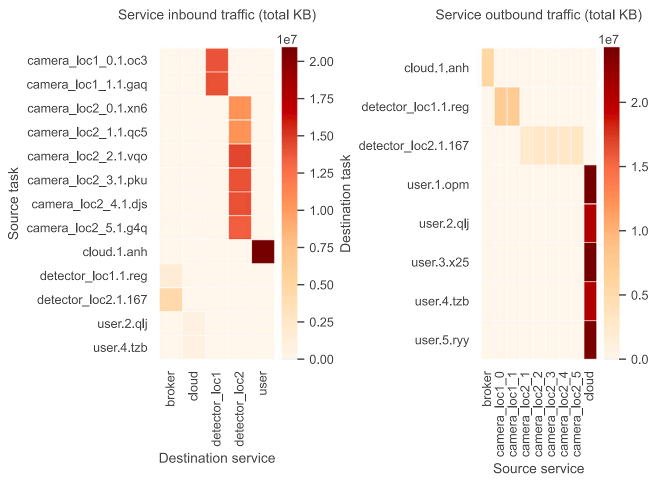


Fig. 13. Traffic for the most significant services (*edge* version).

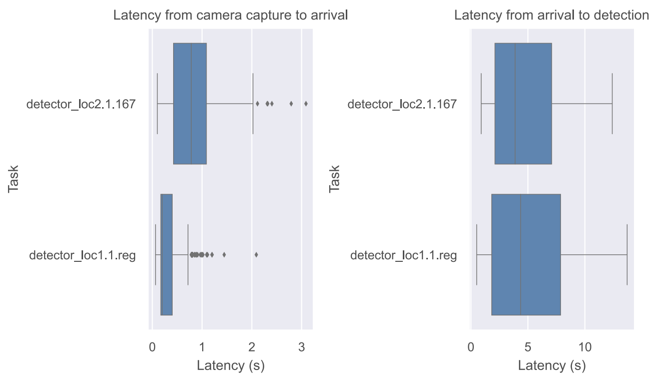


Fig. 12. Detection latency (*edge* version).

- The *cloud* node reads two *detector* nodes through a MQTT broker over mobile connections, as opposed to being connected to a single *detector* in the same LAN.
- Resource limits are imposed on *detector* nodes (see Table 3). A benchmark score of 600 is an approximation of the results reported by the ARM processor in a Raspberry Pi Model 3B, which is a popular single-board computer that may be commonly found in the edge layer. The *camera* constraints are the same in the *edge* and *cloud* versions.
- Networks `field_loc1` and `field_loc2` only appear in the *edge* version, acting as a local WLAN connection between the edge *detector* and the *cameras* (see Table 2).
- There are no differences in the behaviour and configuration of the node applications (i.e. *camera*, *detector*, *cloud* and *user*). Therefore, the detection latency and PTZ latency application metrics have the same interpretation as before. See Section 3.2 for a detailed description of applications and metrics.

The edge computing paradigm proves to be a good fit for the *Intelligent Surveillance* scenario, as demonstrated by the observed PTZ latency (Fig. 11) and detection latency (Fig. 12). Median PTZ latency is under 8 s for all *cameras*, which is a reasonable performance, especially when considering the limited resources of the *camera* and *detector* nodes. Placing the *detector* nodes at the edge of the *camera* locations contributes to achieving a median latency from video frame capture to arrival in under 1 s. On the other hand, as expected, latency from arrival to detection shows no differences with respect to the *cloud* version of the topology, this is a purely local process.

Fig. 13 shows a heatmap of the network traffic between tasks and services in both inbound and outbound directions. Section 2 shows details of the differences between Docker swarm tasks and services. A service is a template from which one or more tasks are created, and each task is a container. In this case, for example, the *user* service includes all five *user* tasks. A subset of only the most significant services and tasks are presented for clarity.

As could be expected, a majority of the network traffic occurs from *cameras* to *detectors*, and bidirectionally between *users* and *cloud*, while traffic from and to the MQTT broker is markedly lower.

Fig. 14 shows the distribution of CPU and memory usage samples for all containers. The X axis of the memory subplot shows the percentage of use over the container memory limit rather than the absolute memory. This is because the former provides more information about how close the container is to the point where it runs out of memory. The memory limit of unconstrained containers (*broker*, *user* and *cloud*) is the size of the entire host memory pool (32 GB), whereas constrained containers (*detector* and *camera*) have a predefined hard upper limit (see Table 3). The interpretation of the X axis of the CPU subplot varies depending on whether the container is constrained or unconstrained. Unconstrained containers are allowed to use more than one core, and as such can go over 100%—a value of 100% represents full usage of one core. On the other hand, constrained containers are limited to a specific quota in one core, thus, 100% represents full usage of the allocated CPU quota (see Table 3).

Memory constraints imposed on *camera* nodes are adequate for this topology, although these nodes are close to the limit and would benefit from a larger memory pool. This conclusion also applies to the *detector* node in the second location, unlike the *detector* in the first location, which has the capacity to scale up to handle more *cameras*. The *cloud* node, although unconstrained, shows a reasonably small memory footprint (the median memory consumption is approximately 1.3 GB, that is, 4% of a total of 32 GB). The *detector* nodes are CPU-bound. Furthermore, the logs indicate that a significant percentage of the video frames are dropped due to the internal buffer queue being full (i.e. the *detector* is not able to keep up with the rate at which frames arrive). Therefore, *detectors* would require a processor with a higher capacity to achieve optimal performance. The *cloud* node is also CPU-bound, as could be expected from a HTTP server handling a significant volume of requests.

It can be seen in Fig. 15 that *detectors* struggle to serve read requests for their properties (see Section 3.1 for a detailed description of the properties). This is a result of the combination of multiple factors, including the size of the video frames contained in the *latestDetections* property, the performance of the cellular (i.e. 2G and 3G) networks

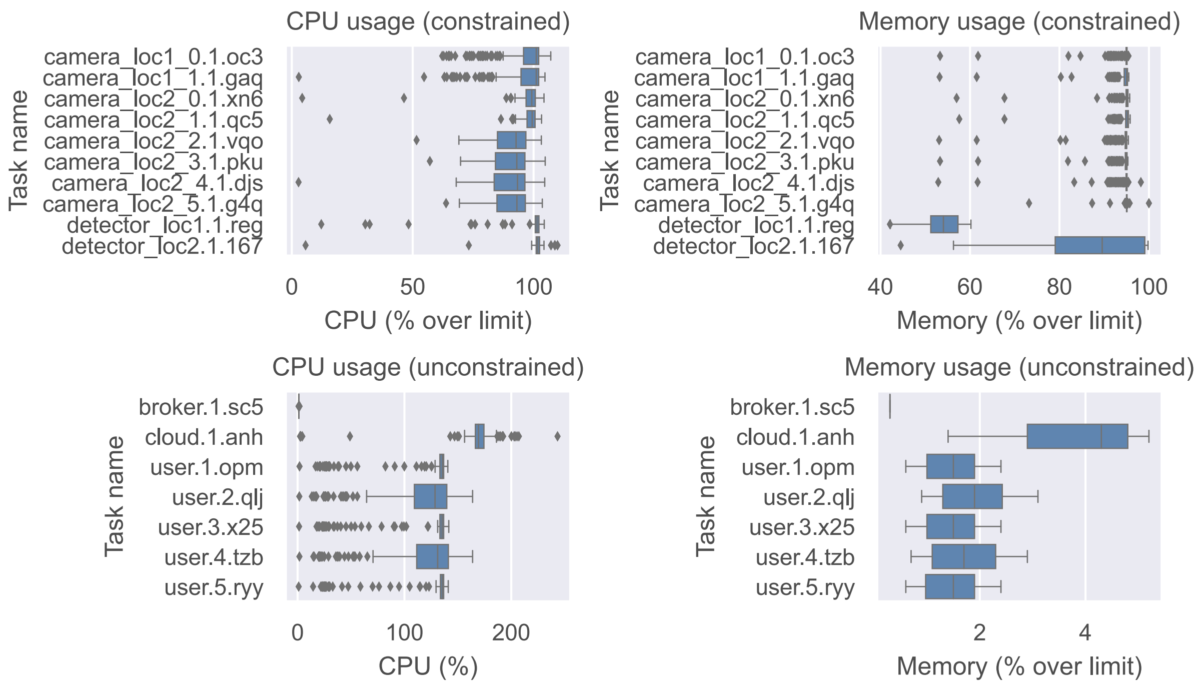


Fig. 14. CPU and memory usage distributions (*edge* version).

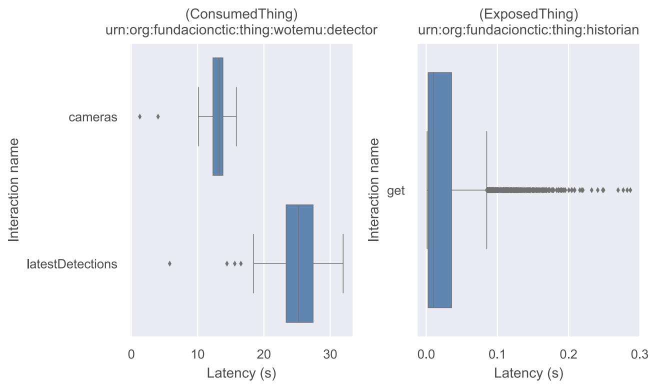


Fig. 15. Interaction latencies for task *cloud* (*edge* version).

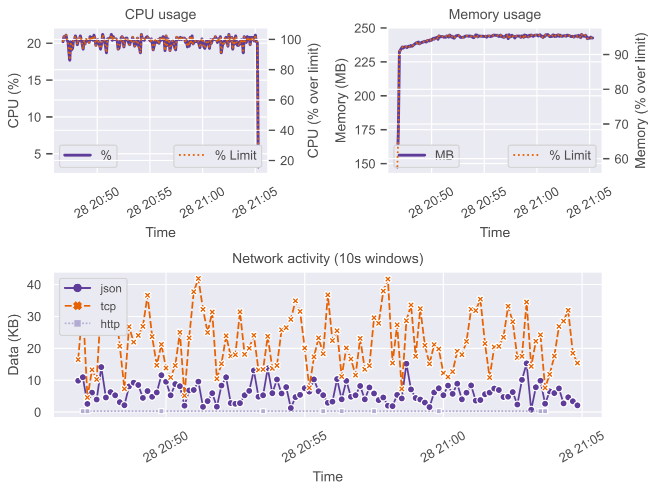


Fig. 16. Performance of a *camera* node (*edge* version).

that interconnect the *detectors* with the *cloud*, latencies inherent to the MQTT binding and the resource constraints imposed on *detectors*. The experiment shows that the *edge* topology would be adequate for cases where a reasonably high average latency for video frames with face detection data is acceptable for clients. Otherwise, more resources (i.e. network, computation) would be required by *detectors*.

Finally, Fig. 16 shows a representative example of performance metrics in a *camera* node. All the other *cameras* have similar behaviour. These nodes operate close to the limit, as could be expected, due to the severe resource constraints imposed on them. Respecting these constraints is necessary to keep costs to a reasonable limit. It is important to note that, although the majority of the network traffic in the *camera* nodes is transported over the HTTP binding, TShark reports most of the packets as *json* (the serialization format of the body in HTTP requests) or *tcp* (the underlying HTTP transport protocol) rather than *http*.

4. Conclusions

Emulation tools are of great importance for IoT architectures. These systems are characterized by a large number of interconnected nodes with multiple communication flows, especially in the context of the edge computing paradigm. Therefore, validating the performance of an architecture design is a complex but important issue. Adoption of an emulation tool such as WoTemu leads to a decrease in deployment and testing costs. Unlike other tools, WoTemu scalably runs real application code with minimal modifications. It enables users to test real applications and avoid costly errors by quickly detecting flaws in proposed designs before committing to a deployment in the real world.

WoTemu leverages the repeatability and horizontal scalability capabilities of Docker swarm mode, a modern container orchestration tool, to provide a solution for this validation issue. Furthermore, WoTemu gives detailed insight into the behaviour of IoT applications in multiple domains, including the application layer, network layer and hardware infrastructure. Finally, the W3C WoT specifications, which are a proven solution for the IoT interoperability problem, are considered as first class citizens to ensure a future-proof approach.

A complex real world scenario was used to validate a full-featured experimental implementation. In this scenario, WoTemu was instrumental in detecting bottlenecks, obtaining meaningful application performance metrics, optimizing hardware resources and discarding problematic designs.

Future work will focus on optimizing resource usage for very large experiments, that is, experiments with hundreds of containers or of considerable duration (in the order of several hours or even days). This is likely to entail the evaluation and adoption of Redis cluster in the data storage layer, and also updates to the reporting module to avoid processing large amounts of data in a naive non-distributed way. Another interesting addition would be to include a recommendations module to provide optimization suggestions for the topology, such as automatically detecting bottlenecks in network links.

CRediT authorship contribution statement

Andrés García Mangas: Conceptualization, Software, Investigation, Writing – original draft, Visualization. **Francisco José Suárez Alonso:** Writing – review & editing, Supervision. **Daniel Fernando García Martínez:** Writing – review & editing. **Fidel Díez Díaz:** Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This research has been partially funded by the Spanish National Plan of Research, Development and Innovation under the project OCAS (RTI2018-094849-B-I00); and the Spanish Ministry of Science and Innovation through the Centre for the Development of Industrial Technology (CDTI) under the project CEL.IA (CER-20211022).

References

[1] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, in: MCC '12, ACM, New York, NY, USA, 2012, pp. 13–16, http://dx.doi.org/10.1145/2342509.2342513.

[2] M. De Donno, K. Tange, N. Dragoni, Foundations and evolution of modern computing paradigms: cloud, iot, edge, and fog, IEEE Access 7 (2019) 150936–150948, http://dx.doi.org/10.1109/ACCESS.2019.2947652.

[3] W. Yu, F. Liang, X. He, W.G. Hatcher, C. Lu, J. Lin, X. Yang, A survey on the edge computing for the internet of things, IEEE Access 6 (2018) 6900–6919, http://dx.doi.org/10.1109/ACCESS.2017.2778504.

[4] C. Aguzzi, L. Gigli, L. Sciullo, A. Trotta, M. Di Felice, From cloud to edge: seamless software migration at the era of the web of things, IEEE Access 8 (2020) 228118–228135, http://dx.doi.org/10.1109/ACCESS.2020.3045632.

[5] P. O'Donovan, C. Gallagher, K. Bruton, D.T. O'Sullivan, A fog computing industrial cyber-physical system for embedded low-latency machine learning industry 4.0 applications, Manuf. Lett. 15 (2018) 139–142, http://dx.doi.org/10.1016/j.mfglet.2018.01.005.

[6] J. Mineraud, O. Mazhelis, X. Su, S. Tarkoma, A gap analysis of internet-of-things platforms, Comput. Commun. 89–90 (2016) 5–16, http://dx.doi.org/10.1016/j.comcom.2016.03.015.

[7] D. Raggett, The web of things: challenges and opportunities, Computer 48 (5) (2015) 26–32, http://dx.doi.org/10.1109/MC.2015.149.

[8] D. Guinard, V. Trifa, E. Wilde, A resource oriented architecture for the web of things, in: 2010 Internet of Things (IOT), 2010, pp. 1–8, http://dx.doi.org/10.1109/IOT.2010.5678452.

[9] M. Kovatsch, R. Matsukura, M. Lagally, T. Kawaguchi, K. Toumura, K. Kajimoto, Web of Things (WoT) Architecture, W3C Recommendation, W3C, 2020, URL https://www.w3.org/TR/2020/REC-wot-architecture-20200409/.

[10] S. Käbisch, T. Kamiya, M. McCool, V. Charpenay, M. Kovatsch, Web of Things (WoT) Thing Description, W3C Recommendation, W3C, 2020, URL https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/.

[11] D. Longley, P.-A. Champin, G. Kellogg, JSON-LD 1.1, W3C Recommendation, W3C, 2020.

[12] M. Koster, E. Korkan, Web of Things (WoT) Binding Templates, Tech. rep., W3C, 2020, URL https://www.w3.org/TR/2020/NOTE-wot-binding-templates-20200130/.

[13] Z. Kis, D. Peintner, C. Aguzzi, J. Hund, K. Nimura, Web of Things (WoT) Scripting API, W3C note, W3C, 2020, URL https://www.w3.org/TR/2020/NOTE-wot-scripting-api-20201124/.

[14] D. Merkel, Docker: lightweight linux containers for consistent development and deployment, Linux J. 2014 (239) (2014).

[15] The crun development team, Crun, Red Hat Inc., 2022, URL https://github.com/containers/crun.

[16] The Kubernetes Authors, Kubernetes (k8s), 2021, URL https://kubernetes.io/.

[17] B.I. Ismail, E. Mostajeran Goortani, M.B. Ab Karim, W. Ming Tat, S. Setapa, J.Y. Luke, O. Hong Hoe, Evaluation of docker as edge computing platform, in: 2015 IEEE Conference on Open Systems (ICOS), 2015, pp. 130–135, http://dx.doi.org/10.1109/ICOS.2015.7377291.

[18] R.N. Calheiros, R. Ranjan, A. Beloglazov, C.A.F.D. Rose, R. Buyya, Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Softw.: Pract. Exp. 41 (1) (2011) 23–50, http://dx.doi.org/10.1002/spe.995.

[19] S.F. Piraghaj, A.V. Dastjerdi, R.N. Calheiros, R. Buyya, Containercloudsim: an environment for modeling and simulation of containers in cloud data centers, Softw.: Pract. Exp. 47 (4) (2017) 505–521, http://dx.doi.org/10.1002/spe.2422.

[20] H. Gupta, A. Vahid Dastjerdi, S.K. Ghosh, R. Buyya, Ifogsim: a toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments, Softw.: Pract. Exp. 47 (9) (2017) 1275–1296, http://dx.doi.org/10.1002/spe.2509.

[21] M.C.S. Filho, R.L. Oliveira, C.C. Monteiro, P.R.M. Inácio, M.M. Freire, Cloudsim plus: a cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness, in: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), 2017, pp. 400–406, http://dx.doi.org/10.23919/INM.2017.7987304.

[22] I. Lera, C. Guerrero, C. Juiz, Yafs: a simulator for iot scenarios in fog computing, IEEE Access 7 (2019) 91745–91758, http://dx.doi.org/10.1109/ACCESS.2019.2927895.

[23] B. Lantz, B. Heller, N. McKeown, A network in a laptop: rapid prototyping for software-defined networks, in: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, 2010, p. 19, http://dx.doi.org/10.1145/1868447.1868466.

[24] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M.H. Zahraee, H. Karl, Maxinet: distributed emulation of software-defined networks, in: Proceedings of the 2014 IFIP Networking Conference (Networking 2014), 2014, pp. 1–9, http://dx.doi.org/10.1109/IFIPNetworking.2014.6857078.

[25] M. Peuster, H. Karl, S. van Rossem, Medicine: rapid prototyping of production-ready network services in multi-pop environments, in: 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), IEEE, Palo Alto, CA, 2016, pp. 148–153, http://dx.doi.org/10.1109/NFV-SDN.2016.7919490.

[26] R. Mayer, L. Graser, H. Gupta, E. Saurez, U. Ramachandran, Emufog: extensible and scalable emulation of large-scale fog computing infrastructures, in: 2017 IEEE Fog World Congress (FWC), 2017, pp. 1–6, http://dx.doi.org/10.1109/FWC.2017.8368525.

[27] A. Coutinho, F. Greve, C. Prazeres, J. Cardoso, Fogbed: a rapid-prototyping emulation environment for fog computing, in: 2018 IEEE International Conference on Communications (ICC), 2018, pp. 1–7, http://dx.doi.org/10.1109/ICC.2018.8423003.

[28] N.K. Giang, M. Blackstock, R. Lea, V.C. Leung, Developing iot applications in the fog: a distributed dataflow approach, in: 2015 5th International Conference on the Internet of Things (IOT), IEEE, Seoul, South Korea, 2015, pp. 155–162, http://dx.doi.org/10.1109/IOT.2015.7356560.

[29] A. Medina, A. Lakhina, I. Matta, J. Byers, Brite: an approach to universal topology generation, in: MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001, pp. 346–353.

[30] S.R. Hassan, I. Ahmad, S. Ahmad, A. Alfaify, M. Shafiq, Remote pain monitoring using fog computing for e-healthcare: an efficient architecture, Sensors 20 (22) (2020) 6574, http://dx.doi.org/10.3390/s20226574.

[31] A. García Mangas, WoTemu: Emulator for Edge Computing Applications in Python, Zenodo, 2021, http://dx.doi.org/10.5281/ZENODO.4769358.

[32] A. García Mangas, F.J. Suárez Alonso, Wotpy: a framework for web of things applications, Comput. Commun. 147 (2019) 235–251, http://dx.doi.org/10.1016/j.comcom.2019.09.004.

[33] R.A. Light, Mosquitto: server and client implementation of the mqtt protocol, J. Open Source Softw. 2 (13) (2017) 265, http://dx.doi.org/10.21105/joss.00265.

[34] J. Llopis, J. Marques, Docker Socket Proxy, Tecnativa, 2021, URL https://github.com/Tecnativa/docker-socket-proxy.

[35] Redis Labs, Redis, 2021, URL https://redis.io/.

[36] A. Kuznetsov, S. Hemminger, Iproute2, 2021, URL https://wiki.linuxfoundation.org/networking/iproute2.

[37] R. Russell, Netfilter Core Team, Iptables, 2021, URL https://www.netfilter.org/projects/iptables/index.html.

[38] Wireshark Foundation, Tshark, 2021, URL https://www.wireshark.org/.
[39] D. Green, Pyshark, 2021, URL https://github.com/KimiNewt/pyshark.
[40] A. Kopytov, Sysbench, 2020, URL https://github.com/akopytov/sysbench.
[41] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and linux containers, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, Philadelphia, PA, USA, 2015, pp. 171–172, http://dx.doi.org/10.1109/ISPASS.2015.7095802.
[42] Plotly, Plotly python open source graphing library, 2021, URL https://plotly.com/.
[43] T. Bray, The JavaScript Object Notation (JSON) Data Interchange Format, STD 90, RFC Editor / RFC Editor, 2017.
[44] J. Reback, W. McKinney, jbrockmendel, J.V. den Bossche, T. Augspurger, P. Cloud, S. Hawkins, gfyoung, Sinhrks, M. Roeschke, A. Klein, T. Petersen, J. Tratner, C. She, W. Ayd, S. Naveh, patrick, M. Garcia, J. Schendel, A. Hayden, D. Saxton, V. Jancauskas, M. Gorelli, R. Shadrach, A. McMaster, P. Battiston, S. Seabold, K. Dong, chris-b1, h-vetinari, Pandas-Dev/Pandas: Pandas 1.2.4, Zenodo, 2021, http://dx.doi.org/10.5281/zenodo.4681666.
[45] A. García Mangas, Wotemu experimental results, 2021, http://dx.doi.org/10.5281/ZENODO.4782152.
[46] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, B. Koldehofe, Mobile fog: a programming model for large-scale applications on the internet of things, in: Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing, in: MCC '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 15–20, http://dx.doi.org/10.1145/2491266.2491270.
[47] G. Bradski, The opencv library, Dr. Dobb's J. Softw. Tools (2000).
[48] W. Lun, C. Contaoi, Intel iot developer kit sample videos, 2018, URL https://github.com/intel-iot-devkit/sample-videos.
[49] A. Geitgey, Face recognition, 2020, URL https://github.com/ageitgey/face_recognition.
[50] D.E. King, Dlib-ml: a machine learning toolkit, J. Mach. Learn. Res. 10 (2009) 1755–1758.

Andrés García Mangas received his degree in Telecommunications Engineering from the University of Oviedo in 2010 and is now pursuing a Ph.D. degree in Computer Science there. Currently, he is the Tech Lead of the Web of Things team at CTIC, where he works as a software engineer with a strong focus on IoT, full-stack Web and cloud computing. He has participated in projects from a variety of fields such as industrial automation, smart energy, cybersecurity or semantic interoperability.

Francisco J. Suárez is a full professor in the Department of Computer Science and Engineering at the University of Oviedo, Spain, where he received his Ph.D. degree in 1998. His current research is focused on wireless sensors networks and edge/fog/cloud architectures for IoT. During the last 5 years, he has led several projects in that field in collaboration with industrial partners as head of the IoT Systems and Services research team.

Daniel F. García is a full professor in the Department of Computer Science and Engineering at the University of Oviedo, Spain, where he leads the area of computer engineering. His current research interest is in the area of self-adaptive computer systems, including cloud computing, IoT systems and services, and computer vision systems. During the last 25 years, he has led many research projects and co-authored 102 papers in journals and more than 150 communications in conferences and workshops. He is a member of the IEEE Computer Society.

Fidel Díez Díaz Director of R&D, Ph.D. candidate in Biomedicine "Performance evaluation of univariate, multivariate and Machine Learning methodologies in the analysis of genomic variation", Master in Project Management and B.Sc. in Computer Engineering. He has 15 years' experience in innovation management and transference, R&D projects and innovation coordination duties and quality research monitoring according to the UNE 166.002 standard. Currently leading the R&D Department at CTIC. To date, he has participated in over 30 multi sectoral projects at regional, national and international level. He also has published several scientific papers both on Computer Engineering and Evolutionary Algorithms on cancer.