electronics

MDPI

Article

Elastic Computing in the Fog on Internet of Things to Improve the Performance of Low Cost Nodes

Rafael Fayos-Jordan †, Santiago Felici-Castell *,†, Jaume Segura-Garcia †, Adolfo Pastor-Aparicio † and Jesus Lopez-Ballester †

Departament de Informàtica, ETSE, Universitat de València, Avd. de la Universitat S/N, 46100 Burjassot, Spain; rafael.fayos@uv.es (R.F.-J.); jsegura@uv.es (J.S.-G.); adolfo.pastor@uv.es (A.P.-A.); jesus.lopez-ballester@uv.es (J.L.-B.)

* Correspondence: felici@uv.es

† These authors contributed equally to this work.

Received: 27 September 2019; Accepted: 2 December 2019; Published: 6 December 2019

Abstract: The Internet of Things (IoT) is a network widely used with the purpose of connecting almost everything, everywhere to the Internet. To cope with this goal, low cost nodes are being used; otherwise, it would be very expensive to expand so fast. These networks are set up with small distributed devices (nodes) that have a power supply, processing unit, memory, sensors, and wireless communications. In the market, we can find different alternatives for these devices, such as small board computers (SBCs), e.g., Raspberry Pi (RPi)), with different features. Usually these devices run a coarse version of a Linux operating system. Nevertheless, there are many scenarios that require enhanced computational power that these nodes alone are unable to provide. In this context, we need to introduce a kind of collaboration among the devices to overcome their constraints. We based our solution in a combination of clustering techniques (building a mesh network using their wireless capabilities); at the same time we try to orchestrate the resources in order to improve their processing capabilities in an elastic computing fashion. This paradigm is called fog computing on IoT. We propose in this paper the use of cloud computing technologies, such as Linux containers, based on Docker, and a container orchestration platform (COP) to run on the top of a cluster of these nodes, but adapted to the fog computing paradigm. Notice that these technologies are open source and developed for Linux operating system. As an example, in our results we show an IoT application for soundscape monitoring as a proof of concept that it will allow us to compare different alternatives in its design and implementation; in particular, with regard to the COP selection, between Docker Swarm and Kubernetes. We conclude that using and combining these techniques, we can improve the overall computation capabilities of these IoT nodes within a fog computing paradigm.

Keywords: fog computing; Kubernetes; Docker Swarm; containers; container orchestration platforms; elastic computing

1. Introduction

The Internet of Things (IoT) has become a widely used network to integrate almost everything, everywhere to the Internet, with different applications which include environmental monitoring, etc. [1–4], and with many open challenges [5,6]. These networks consist of distributed devices that have their own power supply, processing unit, memory, sensors and wireless communications. In addition, low cost devices are being used to make cheaper deployments. There is a wide range of these devices, such as TelosB motes [7] and small board computers (SBCs), such as Raspberry Pi (RPi) [8], with different technical features. Normally, the operating systems of such a device is based on a Linux system.

However, new trends on IoT and their applications make them to require a reconfigurable sensor architecture that can span multiple scenarios, requiring storage, networking and computational resources to be efficiently used at the edge of the network [2]. Thus, there are many scenarios that require enhanced computational power that these nodes alone are unable to provide, showing constraints in terms of limited energy, bandwidth, memory size and computational capabilities. In this context, we must make these devices cooperate to overcome these constraints by using orchestration techniques. In addition, it is necessary to implement a wireless mesh network within the cluster.

In addition, we need to orchestrate resources within the cluster to manage and improve overall processing capabilities in an elastic manner. This paradigm is in the context of fog computing on IoT. It is worth mentioning, that we can find slight differences from different documentary sources for the concept of fog computing, since it is a term not consolidated yet. In this context, we assume that fog computing is the collaboration of neighboring devices in order to improve their overall performance. Additionally, we can find similar definitions, such as, "Fog computing is an architecture that uses edge devices to carry out a substantial amount of computation, storage and communication locally".

It must be stressed that fog computing is one of the research fields that is gaining more importance and relevance. It has emerged to support the requirements of IoT applications that cannot be met by today's nodes [9]. Nevertheless, as discussed before, fog computing is not mature yet and there are not many solutions and available alternatives. We can find different applications and challenges of this technology in others areas [10]; for instance, security issues in vehicular networks [11].

Thus, we propose in this paper, a lightweight virtualization approach enabling flexibility and scalability within the cluster, by using Linux containers and a container orchestration platform (COP), but adapting them to a fog computing paradigm. All these technologies have been developed for a Linux operating system and they will be described in the following sections. The novelty of our proposal and our contribution is the use and combination of these technologies in this paradigm, since these technologies (containers and COPs) were initially defined for cloud computing, where both computers and interconnection networks have nothing to do with the SBC devices and wireless mesh network that we can find in an IoT deployment in terms of CPU, memory and network speed. In this context, the use and analysis of the mentioned technologies is new and there are not (to the best of our knowledge) any studies using them before, combining IoT, low cost devices, and fog computing. In addition, in order to exploit the results for this new approach we compared the different options in the design.

With more detail, the COP will manage the resources on the top of a cluster of these nodes, in the master. Linux container is a technology that allows an application to be broken into different containers that can be run on a single node or on a cluster of multiple nodes. Docker is currently the most used commodity container framework [12]. The reason behind of that orchestration is when the number of containers and nodes are high, we need to schedule and distribute the different tasks. Some COPs are widely used for container-based cluster management in cloud computing environments (but not so extended in fog computing), such as Docker Swarm [13], Marathon-Mesos [14] and Kubernetes [15]. All theses technologies are open source and designed for Linux operating systems. That is the reason we use SBC devices' running operating systems based on Linux.

Finally, we will show an implementation under these constraints applied to noise pollution and soundscape monitoring. The scenario is a real example of the aforementioned new trends of the IoT, requiring high computing without loss of generality. In addition, it will be used as a proof of concept, allowing us to compare different alternatives in its implementation; in particular, with regard to the COP selection between Docker Swarm and Kubernetes. From our results, we conclude that by using and combining the proposed techniques, we can improve the overall computation capabilities of these IoT nodes.

Eventually, with this paper we will try to answer the following question: *how can I increase the overall computational power of a set of SBC devices in an IoT deployment?*

The rest of the paper is structured as follows. Section 2 shows the related work. In Section 3, we define the requirements for the proposed architecture, the clustering options to coordinate the nodes in the fog, and an analysis of containers and orchestration platforms. In Section 4 we explain the design and implementation details. In Section 5, we analyze the test bed for psycho-acoustic annoyance/noise pollution monitoring, in order to highlight the enhancements introduced compared to traditional systems, showing the results from a performance evaluation with the different alternatives. In Section 6 we discuss the results obtained, and finally, in Section 7, we conclude the paper.

2. Related Work

Regarding cloud and fog computing on IoT, we found relevant references. In [9], a unified architectural model and taxonomy was presented by comparing a large number of solutions on fog computing for the IoT, taking into account several aspects, such as communications, security, management, and cloudification. The authors analyzed the main IoT applications requirements, and using the proposed model, they were able to compare different solutions and how they are applied. For instance, in the study presented in [1], the authors explore alternative deployments for a "smart warehouse" IoT application, both based on cloud and based on fog (fog-like) computing. The goal of their exercise was to determine if a cloud-based approach is able to meet the latency requirements of the application, given that low-latency is usually considered an essential requirement for many IoT applications. The authors were able to compare the event latency performance for both cloud and fog deployments, showing as it could be expected, that latency shows better results when the application is deployed according the fog-based approach.

Another approach of fog computing on IoT is shown in [2]. Their authors focus on the use of available gateways in order to enable IoT application deployments. The authors mention that there are a number of platforms and gateway architectures that have been proposed to manage these components. However, these platforms and gateways lack horizontal integration among multiple providers, and other functionalities like load balancing and clustering. The authors state that is partly due to the strongly coupled nature of the deployed applications, and a lack of abstraction of device communication layers as well as a lock-in for communication protocols. This limitation is a major obstacle for the development of a protocol agnostic application environment that allows for single application to be migrated and to work with multiple peripheral devices with varying protocols from different local gateways. Then, the authors propose a messaging-based modular gateway platform that enables clustering of gateways and the abstraction of peripheral communication protocol details. This proposal allows applications to send and receive messages regardless of their deployment location and destination device protocol, creating a uniform development environment.

Regarding management and orchestration within fog computing, we found interesting contributions. In [16], an efficient automated resource management in cloud computing was shown to improve important tasks such as launching, terminating, and maintaining computing instances rapidly, with a minimum overhead. The authors ran a performance analysis over Kubernetes using a Petri network based performance model. The authors suggested that the proposal could be used for supporting capacity planning and designing Kubernetes-based elastic applications.

Finally, it is worth mentioning similar works focused on the evaluation of the aforementioned COPs. In particular, in [17], the authors analyzed Kubernetes as a COP in order to design an on-demand model for renting computing resources and easy-to-use elastic infrastructure in cloud computing environments. The authors considered the choice of a reactive autoscaling method to adapt this demand. Kubernetes already embeds and autoscaling method but it significantly affects both response time and resource utilization. Then the authors discuss and suggest the use of different factors that should be taken into account under different types of traffic to develop new autoscaling methods. They conclude that the default autoscaling method in Kubernetes can be improved by considering the suggested influencing factors. These factors, which should be taken into consideration to handle different workload patterns, consist of (i) a conservative constant (α), (ii) an adaptation interval or

control loop time period (CLTP), and (iii) stopping at most one container instance in each adaptation interval. It must be stressed that the authors used as a testbed, a cluster of computers with four core CPUs at 2397 MHz, with 4GB of RAM and 1 Gbps network interfaces. Additionally, using Kubernetes, in [18] the authors focused on a new feature of this COP to support a federation function of multiple Docker container clusters, called Kubernetes Federation. It allows one to increase the responsiveness and reliability of cloud computing applications by distributing and federating container clusters to multiple service areas of cloud service providers. But the management required is high and complex. Thus, the authors proposed an interesting method and a tool to automatically form and monitor Kubernetes Federation.

To conclude this section, it is worth mentioning that fog-computing on IoT is not yet a mature research line and interesting tools developed for cloud computing could be considered and adapted to this new context. Although big efforts have been made, we cannot find many solutions and available alternatives, or a detailed architecture in order to follow some steps. Thus, with this paper we try to clarify different issues bound to this technology and its deployment through a case study.

3. Analysis of the System

We require an adaptive, reconfigurable, wireless, and scalable architecture in the fog, able to perform both simple and difficult tasks for IoT applications, that can span multiple scenarios, requiring storage, networking, and computational resources to be efficiently used at the edge of the network [2]. For this purpose, in this section we will analyze the clustering options to allow cooperation among the nodes and the Linux container technology to allow load balancing distribution.

3.1. Clustering Options for Nodes in the Fog

Since the IoT nodes will be close one each other in the fog within an area range lower than 50 m, we want a wireless solution, without requiring any additional external device except the nodes themselves. In particular, we will focus on RPi. RPi nodes are well-known and commonly used as SBCs in IoT applications. They fit in almost any coarse and initial IoT deployment. Thus, without loss of generality we consider them in order to make a real cluster, as a proof of concept.

There are several ways to interconnect these nodes in a cluster, as shown in Figure 1.

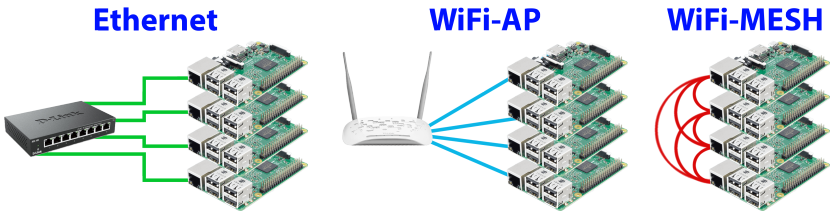


Figure 1. Different options for RPi clustering.

In terms of speed and performance, the wired option is the first and the most common interconnection method, using Ethernet interfaces and getting transmission speeds around 100 Mbps. But this requires adding network cables and an Ethernet switch. In this case, scalability would depend on this switch. Alternatively, using wireless interfaces, the nodes can connect by an access point (AP). This would give us a speed of approximately 40 Mbps in real scenarios [19]. But the limitations, apart from speed, are given by the AP itself, that can act as a bottleneck and requires a lot of energy. Besides, there are not direct connections among the nodes, since all traffic must necessarily go through the AP.

Thus, the last alternative shown in Figure 1 is a wireless mesh network. Since SBCs are equipped with WiFi interfaces, this option is feasible and even more interesting than the previous ones. WiFi mesh networks represent a simple and smart option for node clustering. However there is a constraint in these topologies due to the short communication range of these nodes, but in our case and in the fog we will not consider this a problem.

3.2. Linux Containers and Orchestration

The Linux container technology is a lightweight virtualization technology at the operating system level. Containers are an excellent option and can be booted up in few seconds; they are an efficient use of hardware resources. In this scenario, an application consists of numerous containers that can be run in one node or in a cluster of nodes. Thus, it is important for an orchestrator to keep track of containers belong to the same application and to deal with network connection. The orchestrator can manage hundreds or thousands of containers in a cluster.

Docker is an open source project to automate the deployment of applications within containers. Additionally, Docker is currently the most used commodity container framework; thus, we used this type of container.

When the number of containers in a cluster is high, new tools are needed for orchestration. Some orchestrators are widely used for container-based cluster management, such as Docker Swarm [13] and Kubernetes [15]. They are called container orchestration platforms (COPs). Thus, COPs are used to orchestrate the execution of containers in a cluster. The user describes the container and the COP selects which of the physical hosts or nodes are going to perform the execution of the container. It must be noticed that one could use the interfaces provided by a COP to directly deploy containers on a set of computing resources. Nevertheless, this approach would be disruptive since usage patterns would change. Next, we describe the most important available COPs and their features.

Kubernetes [15] is the most prominent open source orchestration system for Docker containers supported by Google and later donated to the Cloud Native Computing Foundation. It performs the scheduling in a computing cluster (Kubernetes cluster) made up with different nodes and actively manages workloads. In Kubernetes, there is a master, called Kubernetes master, to manage and orchestrate the cluster resources. It provides interesting features, such as reliable container restarting, load balancing, autoscaling, and self-healing. The scheduling in Kubernetes is based on *Pods*, that are groups of containers deployed and scheduled together. These *Pods* can be distributed among different nodes. Additionally, one single node can run several *Pods*. They form the atomic unit of scheduling as opposed to single containers in other systems. Containers within a *Pod* share an IP address and different labels can be used to identify each group of containers. These labeling features allow Kubernetes to work on a heterogeneous cluster, where different nodes are specialized to run specific *Pods*.

Docker Swarm [13], or simply Swarm, represents the native clustering approach proposed by Docker, and takes advantages of the standard Docker API. Swarm manages a set of resources to distribute Docker workloads, managing internal overlay networks within the containers. That way, a container-based virtual cluster can be easily created on top of virtual or physical resources. The architecture of Swarm consists of hosts running a Swarm agent (working nodes) and one host running a Swarm manager. The Swarm agent will run several Docker stacks and each one will accept containers. The concept of Docker stacks is equivalent in this paper to the *Pod* in Kubernetes. The manager is responsible for the orchestration and scheduling of containers on the agent nodes. While the connection between Swarm manager and agent is established by opening the port for the Docker daemon, the Swarm manager can access all existing containers and Docker images in the agent nodes. Additionally, it must be noticed that Swarm can be run in a high-availability mode.

4. Design and Implementation

In this section we describe the design and implementation of the cluster to perform elastic computing in the fog.

4.1. Hardware, Operating System, and Network Configuration of the Nodes

We used four RPi version 3B [8] as the hardware base of the cluster, with its respective power supplies and microSD cards. RPi version 3B uses an ARM Cortex-A53 @ 1.2GHz and 1GB LPDDR2 of

RAM. Figure 2 shows an example of a cluster using four RPis. In this figure, we see also that the cluster is performing a soundscape monitoring and this information is visualized on a map. This scenario is described later in Section 5. In the cluster, the assigned names for the nodes in the cluster are RPiC1 as master, and as slaves—RPiC2, RPiC3, and RPiC4. We used the default operating system provided by the manufacturer, called Raspbian [20]. Raspbian was developed by Raspberry Pi and based on a Linux Debian distribution. We used version 9, named "Stretch".

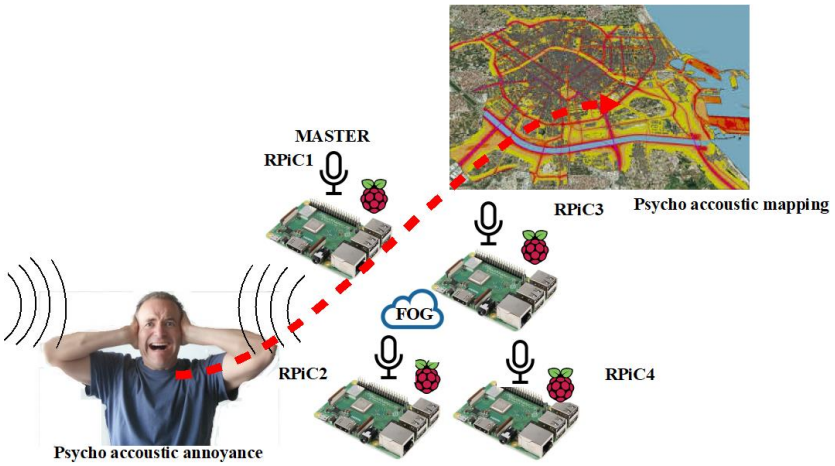


Figure 2. Soundscape monitoring scenario using a fog computing approach based on a cluster of Raspberry Pi 3B within a wireless mesh network, one acting as master (RPiC1, with external access) and the others as slaves (RPiC2-4). The information we gathered is visualized on a map.

For simplicity and scalability within the whole architecture, we relied on the wireless mesh infrastructure placing the nodes within a range lower than 50 m. In this scenario, a multihop routing protocol is required to interconnect the different nodes within the network and/or the cluster.

There are many multihop routing protocols. From all of them, based on our experience and our requirements, better approach to mobile ad hoc networking (BATMAN) [21] is very reliable, stable, well-known, and has little in the way of an overhead.

BATMAN is a proactive routing protocol based on distance-vector algorithm, that builds a loop free network. BATMAN does not try to estimate the whole path to each destination, but only the best next-hop to a given destination, avoiding unnecessary routing-information exchanges among nodes. If the origin and destination nodes are close, it creates direct routes. This is especially relevant in a cluster, since it allows direct communication among the nodes, without any additional intermediate node (relying nodes), speeding up the transmission of packets and minimizing errors. It must be stressed that BATMAN works in Layer 2; thus, the whole multihop network is a LAN and each node is identified by its own MAC address, simplifying the nodal configuration.

Finally, to allow Internet connection for the whole system, the master acts as a router (default gateway) for the nodes. The master must use another interface for this purpose, such as an Ethernet card, a second wireless card, or any telco adapter.

4.2. Cluster Configuration to Manage Containers and Their Orchestration

In order to perform the load distribution of the different tasks for an IoT application within the cluster, we use Docker containers. Docker works by downloading images from a repository (with authentication), that at the same time is in a container too, customizing and executing them in the system. This local repository is created at the master node where the slaves have access and all the images that the slaves will run at the master's request are available.

For the COP selection, as discussed before for the orchestration, we focus on Kubernetes and Swarm as COPs, since they are common.

Regarding Kubernetes configuration, first we will start up the node that will act as orchestrator and we will specify the address where the Kubernetes service will be available for the slaves, as well as the IP range where the *Pods* will be executed. In addition, we have to share a token between the master and the slaves in order to authenticate the process. It must be stressed that we will not enable the autoscaling, since the number of *Pods* will be assigned before hand. It does not mean that Kubernetes will not adapt to the node status but it will not autoscale using its default mechanism. In this case, we will schedule from the master the load distribution by using port forwarding to the slaves. Additionally, we have not included the default monitoring process embedded in Kubernetes because in the RPi, this process increases in excess the CPU use, around a 60% extra.

Regarding Swarm configuration, things are easier compared to Kubernetes since it is the native orchestrator of Docker containers and it is embedded in it. Following the same steps that in Kubernetes, we publish the address at the master where the service is available for the slaves. This was to provide a token to bind the slaves.

5. Test Bed for Soundscape Monitoring and Its Performance Analysis

As a proof of concept and in order to highlight the advantages of the proposed fog computing scheme, in this section we analyze a deployment that requires high computational costs applied to psycho-acoustic annoyance (PA) monitoring, or soundscape monitoring, as shown in Figure 2. We used this scenario as a real example to performance elastic computing on a cluster of RPi, without loss of generality in our results. This IoT application for soundscape monitoring is an example of the new trends on IoT, as mentioned before. Thus, in this case we see the challenges imposed by these low cost devices (RPi) when requiring enhanced computational power that these devices alone are unable to provide, and that they need to collaborate in a fog computing paradigm.

Soundscape monitoring is characterized by requiring tough signal processing algorithms. These algorithms are explained and analyzed in [22,23]. Behind this monitoring process, there are several rules and standards such as Environmental Noise Directives (ENDs) 2002/49/EC and ISO 12913 (soundscape) [24,25]. In particular, END 2002/49/EC requires main cities (with more than 250,000 inhabitants) to gather real data on noise exposure in order to produce local action plans and to provide accurate real time mappings of noise pollution levels.

The evaluation of this annoyance (or PA) is mainly based on the work by Zwicker and Fastl [26], defining a set of parameters such as loudness (L), sharpness (S), fluctuation strength (F), and roughness (R) that will let us measure PA. In other words, PA is given by a function knowing L, S, F, and R, and we need all of them before calculating PA. The flow diagram of these algorithms is shown in Figure 3. Due to the complexity of their implementation, we cannot perform this monitoring process with conventional hardware (such as RPi) in a simple IoT deployment, and less so in real time. We must highlight that we refer to real time as when the time required to process an audio chunk is shorter than the chunk itself, which is by default 1 second.

It must be stressed in these scenarios that due to stringent laws related to personal privacy, it is essential that PA calculations must be performed in real-time and as close as possible to the source within the communication range, because the data are not allowed be to sent the recorded audio out.

First, we carry out the performance analysis in two steps. First, we analyze parallelization and granularity issues of the containers, and secondly, we analyze its throughput.

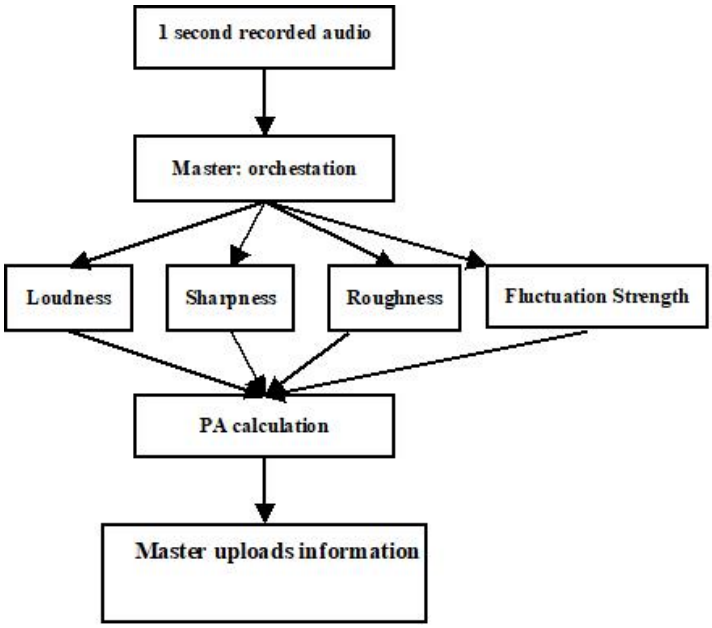


Figure 3. Flow diagram to measure and calculate psycho-acoustic annoyance (PA) on the proposed novel, fog computing architecture based on L, S, R, and F.

5.1. Analyzing Parallelization and Granularity

Thus, first we characterize the processing time of the acoustic parameters. We created as many containers as psycho-acoustic parameters to allow their parallelization and to speed up the PA calculation. In this case, our aim was to increase the granularity of the containers (to make them small and one per parameter) in order to distribute them easily and lightly, among the different slaves in the cluster, to reduce the processing time.

In this scenario, there is a client node (external node) that records audio chunks and sends them at a certain rate to the system under test (SUT). We used 100 audio chunks (of daily sounds of one second) randomly selected from 60,150 chunks, recorded beforehand using a USB microphone. This process was repeated several times till we achieved a confidence interval of 95%. The client timestamped each audio chunk, and then it sent it to the SUT by WiFi. Once each audio chunk was processed, the SUT sent the results back to the client. Once the client received them, it measured the total computation time, as the elapsed time between the audio chunk was sent (and time stamped) and the results were received. The SUT ran a REST-API and for a comparison; we used different approaches to compute these parameters: (a) in a computer as a base line, (b) in a single RPi, and (c) in a cluster of four RPis using the proposed fog computing.

In the case of the computer, we ran the parameters on one computer using both Matlab and C++/Python. The computer was an i7-7700HQ at 3.5 GHz and 16GB DDR4 of RAM with eight cores.

In the case of single node (RPi) and fog computing (cluster of RPi), we used RPi models 3B and 3B+, using only C++/Python code. In the single node (RPi) approach, only one RPi was computing all the parameters. This is the worst case, the slowest approach. In the cluster (fog approach), we had four RPi3B, one as master and three as slaves. In this case, we were running the parameters (in different containers), based on Swarm. All of the RPis were close one each other. The master in the cluster was in charge of orchestrating the different slaves. RPi 3B used an ARM Cortex-A53 at 1.2 GHz and 1 GB LPDDR2 of RAM and RPi 3B+ used an ARM Cortex-A53 at 1.4 GHz and 1 GB LPDDR2 of RAM, all of them with four cores. It must be noted that with RPis, the main program was based on Python, and we used C++ in order to implement a Python library that performed all the tough processing from each psycho-acoustic parameter in an efficient way by using the linear algebra library called Armadillo [27]. This code has not been programmed using threads.

In Table 1, we summarize the computational times for L, S, F, R, and PA on average per each second of recorded audio, using different devices and programming languages and specifying these approaches, specifying both the computer, single devices (one RPi), and a cluster of four RPis. This table only shows average values in seconds and the standard deviation is at least three orders of magnitude smaller; the highest is given by RPi working as a single node. The computation time is nearly constant for all of them independently of the audio chunk (all of them of one second). The communication times are included in each approach. In addition, as it could be expected, we saw that the computer outperformed any combination with RPis. Additionally, we saw the effect of parallelization in the cluster, where the total time was approximately the longest time of the parameters (R is the toughest), compared with the total time required on a single RPi. In the cluster, because each psycho acoustic parameter (L, S, F, and R) can be processed independently, we could parallelize them, making the final computation of the PA (the parameter that includes all of them) faster. In particular, we saw that when using a single and isolated RPi, the results (total time for PA estimation) were the worst ones, 1.479 and 1.406 seconds for RPi3B and RPi3B+ respectively. These times are far from a real time execution, which should be less than one second (the duration of the audio chunk). However, when using fog computing in the cluster, we took advantage of their parallelization and those times were reduced to 0.875 and 0.824 seconds for RPi3B and RPi3B+ respectively. These results initially validate our approach.

Table 1. Time comparison in seconds to calculate the pyscho-acoustic parameters (loudness (L), sharpness (S), fluctuation strength (F), roughness (R), and PA (total time)), among different devices and programming languages using different types of approach: a computer (base line), a single node (RPi), and a cluster with four RPis (fog computing approach).

	Type	**L**	**S**	**R**	**F**	**PA (Total)**
Matlab	*Computer*	0.058	0.000	0.288	0.404	0.699
C++/Python	*Computer*	0.003	0.000	0.128	0.235	0.238
RPi3B	*Single node*	0.018	0.000	0.849	0.742	1.479
RPi3B+	*Single node*	0.017	0.000	0.794	0.694	1.406
RPi3B	*Cluster*	0.022	0.000	0.853	0.754	0.875
RPi3B+	*Cluster*	0.021	0.000	0.802	0.703	0.824

5.2. Performance Evaluation of the Cluster in the Fog

Now, we will focus on the cluster and its capabilities. We performed the same calculations as before, but embedding all the parameters within the same container, in order to measure the total throughput of the whole cluster, comparing Swarm and Kubernetes, with the aim of measuring its performance. We will refer to the concept of Docker stacks from Swarm as *Pod* as in Kubernetes, since both are equivalent.

We used the same testbed as before, with one client node recording audio chunks and the cluster (the SUT) processing all of them. The cluster was set up with four RPi 3B, one node acting as a master and three slaves. We embedded all the parameters within the same container and the master distributed this container among the slaves and their different *Pods*.

In the testbed, the external client timestamped each audio chunk and sent it to the master by WiFi on the mesh network at a certain rate (λ or audio chunks per second). The master distributed the audio chunks among the slaves by WiFi too. The slaves were running a REST-API and received requests from the master. The slaves kept the audio chunks assigned temporarily, if they were busy (while working with other audio chunks), in a FIFO queue implemented internally in the REST-API implementation. If the audio chunks had to wait for processing, the queuing time depended on the number of pending audio chunks. Each slave processed their assigned audio chunks one by one and sent the results (acoustic parameters and PA) back to the master, every time an audio chunk was processed. The master forwarded them to the client. Once the client received each result, it measured

the total computation time, as before. It must be stressed that each computation time per audio chunk includes: the sending time from the client to the master and from the master to the slave.

As seen in Table 1 in a RPi3B, each audio chunk takes an average of 1479 ms to calculate PA, with a standard deviation of 1.405×10^{-4} (it is near a deterministic time). If we consider 1479 ms, as the average service time, equal to $\frac{1}{\mu}$, the service rate (μ) is 0.676 chunks/second. Besides, we will call the number of available *Pods* c. The number of *Pods* is defined by configuration before hand at each slave in the cluster. In this case, we can model this approach as $D/D/c$ according to the Kendall notation. That is, a deterministic arrival process (since audio chunks are recorded one per second) and a deterministic time service with c cores (number of *Pods*). Because each RPi had four cores, while in the cluster we had one, two, or three slaves, we got four, eight, or 12 cores, maximum. Each core can run a *Pod*. Thus, with one slave c is in the range $[1, ..., 4]$; with two slaves c is within $[2, ..., 8]$; and with three slaves c is within $[3, ..., 12]$.

In particular, we evaluated the performance of the cluster with and without congestion.

On one hand, if $\lambda \geq c \cdot \mu$, we have congestion. In this case, each slave running a REST-API will keep the audio chunks in RAM memory. The slaves have memory enough to keep the audio chunks in RAM. The global queue will grow as $\lambda - c \cdot \mu$ and it will be distributed among the slaves. We test λ in steps of 0.1 chunks/second from values greater than $c \cdot \mu$, with a maximum of 10 chunks/second. In all the scenarios, we sent the same workload, 100 audio chunks at the given rate several times till we met the confidence interval. Notice that we always create congestion at each slave once its cores are busy with their first audio chunk. The idea is to observe how the system behaves under congestion. Additionally, it must be stressed that the client is configured without time outs in the application (REST-API) because it would produce duplicated chunks. Then, the cluster is behaving as a conservative system.

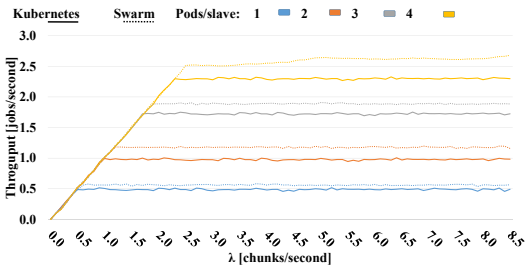
On the other hand, without congestion, the input workload would be lower than the output workload processed, avoiding any queue. In the scenario without congestion, $\lambda < c \cdot \mu$, we tested λ from 0.1 chunks/second till $c \cdot \mu$. Figure 4a–c shows the input workload (λ in audio chunks per second) compared to throughput (audio chunks processed (or jobs) per second), comparing Kubernetes to Swarm with *Pods*/slave and different slaves ((Figure 4a) one slave, (Figure 4b) two slaves, and (Figure 4c) three slaves). In this case, in all the scenarios when the total number of *Pods* were busy, as could be expected, the scenario became congested, although the systems showed a constant throughput since each node kept their tasks in RAM in a FIFO fashion. It must be noticed that Swarm shows a greater throughput according to the previous results, around 10% more, and this enhancement increases with the number of *Pods*.

In addition, we compare in Figure 5a–c, the average resource uses in terms of percentage of CPU, percentage of RAM memory, and CPU temperature, respectively.

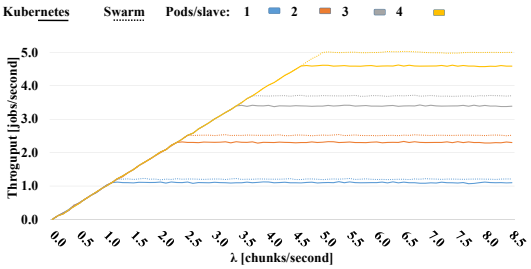
In terms of percentage of CPU (Figure 5a) both COPs have a similar behavior, increasing with the number of *Pods*. They show a minimum of 35% approximately, adding 20% approximately when each core is activated. In theory, because the RPi has four cores, every time a core is completely busy the CPU will increase 25%. From these results, Swarm shows a slightly greater use of CPU, because the cores have less idle time providing higher utilization. The maximum was reached by Swarm with a 95% (with four cores working) and Kuberentes had 10% less. Notice that we should take into account the initial resource use without any service (by default), with both COPs.In this case, Kubernetes used in the master, 17% CPU and 74% RAM, and in the slaves, 6% CPU and 33% RAM. In the same conditions, Swarm used in the master, 0.7% CPU and 22.48% RAM, and in the slaves, 0.53% CPU and 16.16% RAM.

In terms of percentage of RAM memory (Figure 5b), things are different because Kubernetes usually uses 15% approximately more memory. In part, this is due to a lower throughput, as we saw in Figure 4a–c, and then it has to keep the audio chunks in memory in the meantime.

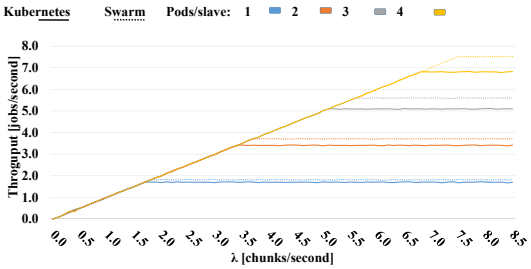
Finally, for the CPU temperature (Figure 5c), according to the CPU use, Swarm gets between 2 and 3 degrees higher than Kubernetes in all the combinations.



(**a**) 1 slave

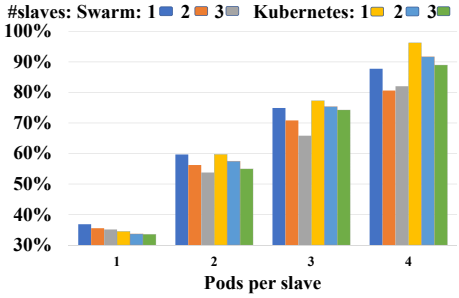


(**b**) 2 slaves

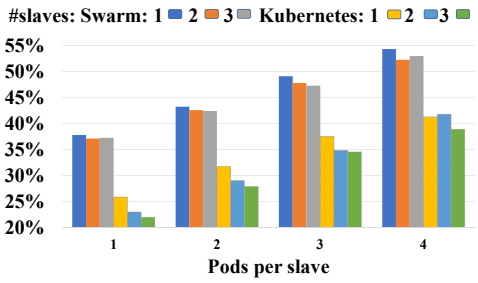


(**c**) 3 slaves

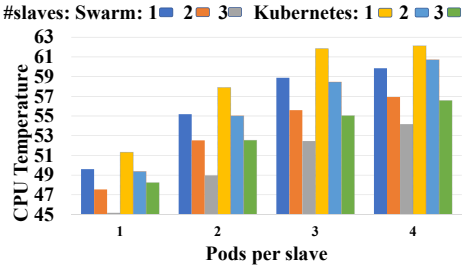
Figure 4. Throughput comparing Kubernetes vs Swarm with different slaves ((**a**) 1, (**b**) 2 and (**c**) 3) and *Pods*/slave.



(**a**) % CPU



(**b**) % RAM



(**c**) Temperature

Figure 5. Resource use comparing Kubernetes to Swarm with different slaves and *Pods*/slaves: (**a**) percentage of CPU, (**b**) percentage of RAM memory, and (**c**) temperature.

6. Discussion

On one hand, as we can see from Section 5.1, by reducing the granularity of the containers we decrease the processing time, as could be expected, in the proposed fog computing architecture. But in this case, because the different running containers require sharing the same audio chunk, it increases the input data overhead that makes it inefficient in terms of throughput. The bottleneck is given by the mesh network and the communication processes within the cluster. For that reason, on the other hand in Section 5.2, we also evaluated an approach where one container included all the different PA parameters. In this scenario, the overhead was reduced, allowing a higher throughput, and the limits were imposed by the nodes and the number of cores. It must be noted that in this case, the cluster has one master and three slaves based on RPi 3B (with four cores each), that will limit the number of *Pods* running efficiently at each slave.

Additionally, we compared Swarm to Kubernetes. We have seen that Swarm always outperforms Kubernetes on RPi with the different metrics used (memory and CPU use, throughput, etc.). Swarm is more efficient (10% approximately) and faster on RPi3B nodes in all the scenarios and for the different evaluated metrics.

7. Conclusions

Nowadays IoT is requiring a flexible and scalable network design but built on low cost nodes to access and connect almost everything at everywhere. In particular in this paper, we focused on nodes such as RPi. But with the new trends, the IoT is facing many scenarios that require high computation capabilities beyond the possibilities of these nodes alone. For this goal, we proposed an architecture leveraging fog computing, based on Linux containers and an orchestration platform, to run on the top of a cluster of these nodes in order to cooperate and schedule different tasks in an efficient way.

As a proof of concept, we showed a scenario that requires high computing requirements, such as soundscape monitoring, and compared different alternatives in its implementation. With these results, we conclude that using and combining clustering techniques, Linux containers, and an orchestrator, we can improve the overall computation capabilities of these IoT nodes. We have used Linux Docker containers and compared two different COP as orchestrators, such as Docker Swarm and Kubernetes. The experimental results showed the improved performance in terms of execution time and throughput in a cluster of four RPis. We have seen that Docker Swarm always outperforms Kubernetes in this scenario.

Thus, finally, an educated answer for "how can I increase the overall computational power of a set of SBC devices in an IoT deployment?" is: We can overcome the constraints imposed by single SBC devices using a cluster of interconnected nodes in a wireless mesh networks, since these devices have wireless capabilities. In addition, in order to improve the overall computational power, the use of Linux Docker containers add flexibility, adaptability, and responsiveness. However, it is necessary to orchestrate the resources among the nodes in the cluster by using a COP; in particular, Docker Swarm performs best.

Author Contributions: conceptualization, methodology, and validation by R.F.-J. and S.F.-C.; software, A.P.-A.; resources, J.S.-G.; writing—original draft preparation, S.F.-C.; writing—review and editing, J.L.-B.

Funding: This work was supported by the Ministry of Economy under the project Urbauramon BIA2016-76957-C3-1-R and a grant by University of Valencia UV-INV-EPDI19-995284.

Conflicts of Interest: The authors declare no conflict of interest

References

1. Gomes, M.; Pardal, M.L. Cloud vs. Fog: Assessment of alternative deployments for a latency-sensitive IoT application. *Procedia Comput. Sci.* **2018**, *130*, 488–495. [CrossRef]
2. Verba, N.; Chao, K.M.; James, A.; Goldsmith, D.; Fei, X.; Stan, S.D. Platform as a service gateway for the Fog of Things. *Adv. Eng. Inform.* **2017**, *33*, 243–257. [CrossRef]

3. Pastor-Aparicio, A.; Segura-Garcia, J.; Lopez-Ballester, J.; Felici-Castell, S.; Garcia-Pineda, M.; Pérez-Solano, J.J. Psycho-Acoustic Annoyance Implementation with Wireless Acoustic Sensor Networks for Monitoring in Smart Cities. *IEEE Internet Things J.* **2019**. [CrossRef]
4. Segura-Garcia, J.; Perez-Solano, J.J.; Cobos, M.; Navarro, E.; Felici-Castell, S.; Soriano, A.; Montes, F. Spatial Statistical Analysis of Urban Noise Data from a WASN Gathered by an IoT System: Application to a Small City. *Appl. Sci.* **2016**, *6*, 380. [CrossRef]
5. Pérez-Solano, J.J.; Felici-Castell, S. Improving time synchronization in Wireless Sensor Networks using Bayesian Inference. *J. Netw. Comput. Appl.* **2017**, *82*, 47–55. [CrossRef]
6. Cobos, M.; Perez-Solano, J.J.; Felici-Castell, S.; Segura, J.; Navarro, J.M. Cumulative-Sum-Based Localization of Sound Events in Low-Cost Wireless Acoustic Sensor Networks. *IEEE/ACM Trans. Audio Speech Lang. Process.* **2014**, *22*, 1792–1802. [CrossRef]
7. Polastre, J.; Szewczyk, R.; Culler, D. Telos: Enabling Ultra-low Power Wireless Research. In Proceedings of the 4th International Symposium on Information Processing in Sensor Networks, Los Angeles, CA, USA, 24–27 April 2005; IEEE Press: Piscataway, NJ, USA, 2005; pp. 364–369.
8. Raspberry Pi 3B+. 2018. Available online: https://www.raspberrypi.org/documentation/ (accessed on 2 January 2019).
9. Bellavista, P.; Berrocal, J.; Corradi, A.; Das, S.K.; Foschini, L.; Zanni, A. A survey on fog computing for the Internet of Things. *Pervasive Mob. Comput.* **2019**, *52*, 71–99. [CrossRef]
10. Mukherjee, M.; Shu, L.; Wang, D. Survey of Fog Computing: Fundamental, Network Applications, and Research Challenges. *IEEE Commun. Surv. Tutor.* **2018**, *20*, 1826–1857. [CrossRef]
11. Erskine, S.K.; Elleithy, K.M. Secure Intelligent Vehicular Network Using Fog Computing. *Electronics* **2019**, *8*. [CrossRef]
12. Docker Containers. 2018. Available online: https://docs.docker.com/engine/examples (accessed on 2 March 2019).
13. Docker Swarm. 2017. Available online: https://docs.docker.com/swarm (accessed on 11 November 2018).
14. Apache Mesos. 2018. Available online: http://mesos.apache.org (accessed on 11 January 2019).
15. Kubernetes (K8s). 2018. Available online: https://kubernetes.io (accessed on 11 October 2018).
16. Medel, V.; Tolosana-Calasanz, R.; Ángel Bañares, J.; Arronategui, U.; Rana, O.F. Characterising resource management performance in Kubernetes. *Comput. Electr. Eng.* **2018**, *68*, 286–297. [CrossRef]
17. Taherizadeh, S.; Grobelnik, M. Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications. *Adv. Eng. Softw.* **2020**, *140*, 102734. [CrossRef]
18. Kim, D.; Muhammad, H.; Kim, E.; Helal, S.; Lee, C. TOSCA-Based and Federation-Aware Cloud Orchestration for Kubernetes Container Platform. *Appl. Sci.* **2019**, *9*, 191. [CrossRef]
19. Strazdins, G.; Elsts, A.; Nesenbergs, K.; Selavo, L. Wireless Sensor Network Operating System Design Rules Based on Real-World Deployment Survey. *J. Sens. Actuator Netw.* **2013**, *2*, 509–556. [CrossRef]
20. Foundation, R.P. Download Raspbian for Raspberry Pi, 2019. Available online: https://www.raspberrypi.org/downloads/raspbian/ (accessed on 3 December 2019).
21. Neumann, A.; Aichele, C.; Lindner, M.; Wunderlich, S. Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.). IETF Draft. Available online: https://tools.ietf.org/html/draft-wunderlich-openmesh-manet-routing-00 (accessed on 3 December 2019).
22. Noriega-Linares, J.E.; Rodriguez-Mayol, A.; Cobos, M.; Segura-García, J.; Felici-Castell, S.; Navarro, J.M. A Wireless Acoustic Array System for Binaural Loudness Evaluation in Cities. *IEEE Sens. J.* **2017**, *17*, 7043–7052. [CrossRef]
23. Pastor-Aparicio, A.; Lopez-Ballester, J.; Segura-Garcia, J.; Felici-Castell, S.; Cobos, M.; Fayos-Jordan, R.; Perez-Solano, J. Zwicker's annoyance model implementation in a WASN node. In Proceedings of the INTER-NOISE and NOISE-CON Congress and Conference Proceedings, Inter-Noise 2019, Madrid, Spain, 16–19 June 2019; Volume 1, pp. 1–11.
24. ISO. *Acoustics—Soundscape—Part 1: Definition and Conceptual Framework*; ISO 12913-1:2014; International Organization for Standardization: Geneva, Switzerland, 2014.
25. ISO. *Acoustics—Soundscape—Part 2: Data Collection and Reporting Requirements*; ISO 12913-2:2018; International Organization for Standardization: Geneva, Switzerland, 2018.

26. Fastl, H.; Zwicker, E. *Psychoacoustics: Facts and Models*; Springer Series in Information Sciences; Springer-Verlag: Berlin/Heidelberg, Germany, 2007; Volume 22.
27. Sanderson, C.; Curtin, R. A User-Friendly Hybrid Sparse Matrix Class in C++. *Lect. Notes Comput. Sci. (LNCS)* **2018**, *10931*, 422–430.