Network awareness for scheduling of containerized applications in an multilevel edge/fog/cloud distributed environment*

Pablo Jaramillo, Agustín C. Caminero, Rocío Muñoz
National Distance University of Spain (UNED)
Madrid, Spain
pjaramill7@alumno.uned.es, accaminero@scc.uned.es, rmunoz@dia.uned.es

***Abstract*—The use of containers to deploy applications has suffered a huge growth in the last years. Containers allow the easy deployment of applications in such a way they can be isolated from the underlying hardware without the resource consumption and the performance degradation of traditional virtualization technologies (e.g. virtual machines). With this regard, containers can be used in a variety of scenarios, some of them harness distributed multilevel infrastructures (edge, fog and cloud). In these scenarios, the network plays an essential role in the proper execution of the applications, since it is the communication medium for all the agents in such distributed scenario and its poor performance may lead to poor quality of service of the entire system. This paper presents a new technique to perform scheduling of containerized applications on distributed multilevel infrastructures (edge, fog, and cloud). This technique considers the status of the interconnection network as a top-class resource to be considered when performing the scheduling tasks. A performance evaluation using a cluster of microcomputers Raspberry Pi has been conducted as a proof of concept of or work.**

***Index Terms*—scheduling algorithm, network, virtualization, container, distributed system, edge, fog, cloud**

I. Introduction and Motivation

During the last years, distributed heterogeneous computing resources have arisen. *Cloud computing* can be considered as the application of business models existing in traditional utilities (such as water or power supply) to computing equipment, where users consume virtualized storage or compute resources offered by a provider based on a pay-as-you-go model. The resources providers use are large datacenters with plenty of computing, memory and storage capacity. Along with this, devices at the *edge* of the network can be part of Internet of Things (IoT) deployments. These can be sensors and devices that are part of Smart City infrastructure, lifestyle gadgets like wearables, and smart appliances or smart phones. These devices have been originally designed to sense and generate data streams, and they have some spare capacity (computing, memory, and storage) which could be harnessed to execute applications [1].

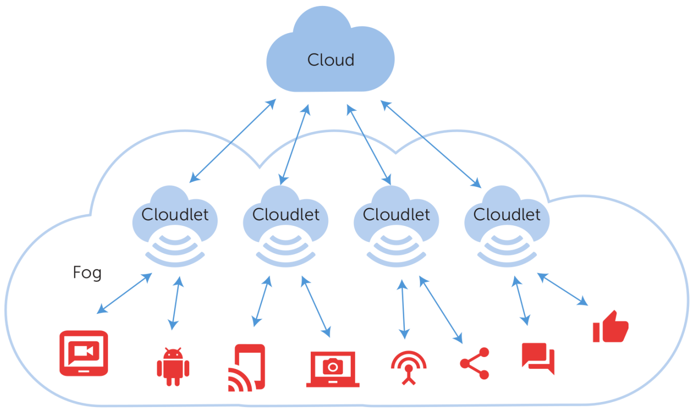

Fig. 1. Multilevel edge/fog/cloud scenario [5].

The last term to be coined is *fog computing*, which refers to computing resources that are between the edge and cloud in the network hierarchy, with compute, storage and memory capacities that fall between these layers as well [2]. These edge and fog resources provide the opportunity for low latency processing of the generated data, closer to its source, and on the wide-area network [3]. As a result, there is critical need to understand how this diverse ecosystem of edge, fog and cloud resources can be effectively used by large-scale distributed applications [4]. This multilevel scenario can be seen in Figure 1 [5], where the term *cloudlets* refers to the access points devices extended with computing and storage services to create the fog system.

In such multilevel environment, the network plays an essential role since it is the communication medium between all the actors in the system and its poor performance affects the whole system. Thus, the network should be taken into account when making all kinds of decisions in order to provide Quality of Service (QoS), one of the main ones is scheduling. The term *scheduling* refers to the process of allocating computing resources to an application and mapping constituent components of that application onto those resources, in order to meet certain QoS and resource conservation goals [6].

This work has been funded by UNED through the Independent Thinking-Jóvenes Investigadores 2019 project efFIcient scheduLing of containErs (FILE, 2019-PUNED-0007), by Ministry of Economy and Competitiveness (Ministerio de Economía y Competitividad, MINECO) through the Red Temática Española de Analítica de Aprendizaje, Spanish Network of Learning Analytics (SNOLA, RED2018-102725-T), and by the Region of Madrid through the project e-Madrid-CM (P2018/TCS-4307).

Virtualization technologies allow the package and execution of applications in isolated environments, independently from the underlying hardware resources. The most traditional virtualization technology is *hardware virtualization*, which creates *virtual machines (VM)*, which are abstractions of hardware on top of which an operating system runs. They require extensive hardware resources to run since VMs run full operating systems. A more recent way of virtualization is *operating system-level virtualization*, which creates *containers* [7]. They use the host operating system and thus they do not require one for each container (as for the VMs), so the hardware requirements to run containers are lower than for VMs. Thanks to this, containers are suitable to run on low cost computers such as Raspberry Pi.

In order to deploy containers, this can be done in a cluster of computers. In this case, there is a need to manage such cluster, and this is done by the *container orchestrator*, one of the best known being Kubernetes [8]. Among the tasks the orchestrator has to perform, one of the most important is the scheduling of the containers into the nodes of the cluster.

Considering all of this, this work proposes an scheduling algorithm in which the container orchestrator uses the status of the network, along with more information such as the memory or CPU status of the nodes, in order to choose the node of the cluster that will run a container – this is the main contribution of this work. Also, this paper presents proof-of-concept implementation of the algorithm on Kubernetes, and a performance evaluation of the algorithm, which has been conducted by deploying Kubernetes on a cluster of micro-computers Raspberry Pi [9]. Raspberry Pi has been chosen because of its low cost and its ability to run docker containers. Kubernetes has been chosen because it has been used for compute containers on Raspberry Pi-class edge devices [10], and used to deploy software on the fly on fog resources [11].

This paper is organized as follows. Section II presents the related work of this paper. Section III presents the scheduling algorithm which is the main contribution of this work. Section IV details the performance evaluation conducted to illustrate the usefulness of our work. Finally, Section **??** concludes the paper and suggests guidelines for future work.

II. Related work

The growth of the use of light virtualization technologies or containers is evident in the vast majority of areas where previously the hardware virtualization was used. Most of the times, the deployment of an application is done in a distributed computing environment, in the cloud or in a local cluster. When deploying applications in distributed environments a container orchestration tool like Kubernetes is needed. Kubernetes manages the deployment of the application introducing the *pod* concept which is the basic deployment unit in the system. A pod is capable of running one or more containers and every pod has to be deployed in a selected node of the distributed computing environment which will be selected by Kubernetes based on the resources required by the node to run smoothly and in some affinities and taints which has to be configured depending on the type of deployment. Studies have been conducted that address the effects of the deployment configuration in the performance and robustness of the application. In [12] a performance model of Kubernetes application is obtained with the application of benchmarks. It defends that for applications with more than eight containers, the bandwidth and final throughput is higher if every container is deployed in a separate pod. This suggest that deploying more pods with fewer containers in each pod, is better in terms of performance and bandwidth availability. In [13], from the same authors, the effects of the type of deployment explained above are studied and detailed depending on the nature of the application and its services. The paper makes a distinction between high CPU and I/O demanding services and high bandwidth demand. At the same time, it makes a distinction between service container, that are executing full time, and job containers, which are created and destroyed with each execution. For CPU and I / O intensive applications it is argued that if there is more containers than nodes in the cluster, the best option is to group all the containers, however, it is important to divide the pods among all possible nodes, so it is recommended to divide the number of containers by the number of nodes available in the cluster and deploy a pod on each node. If there is a smaller number of containers than nodes, a container in each pod and a pod in each node. For the second type of application, intensive use of network resources, it is recommended the same policy applied previously in the case of type containers service, while for job-type containers it is always recommended to deploy a single container per pod. This is due to the fact that the network overload in a pod increases significantly when several containers are executed, since they all share the same IP address, unique in each pod.

These studies are very useful when making decisions about the deployment of an application on infrastructures with container orchestration. Despite all this, more precise adjustments may be required in the process of assignment of a node to each pod. If the application is based on a large number of job-type containers that are created and destroyed proportionally with the load of work, it may be the case that the default Kubernetes planner is not enough. This scheduler takes into account resources such as CPU and memory, but it does not take into account the network congestion at the time of assignment or the overhead of the interfaces During this work, a Kubernetes and Raspberry Pi based testing environment will be deployed and a container planning algorithm based on resource metrics per node and network congestion will be developed.

III. Container scheduling Algorithm Development

The Kubernetes scheduling algorithm, from now on, scheduler, has the only mission of scheduling the just created pods to a cluster node. It is a process executing with the rest of components in the master node. When a new pod is created, an event is created ant the Kubernetes scheduler is triggered to start the assignation of the pod. It has three main execution phases: filtering, node ranking and node selection. During the filtering phase, the scheduler discards the nodes that are not suitable to run the pod due to restrictions and taints, that are applicable to each pod and node. During the second phase, the scheduler populates a list containing all the suitable nodes for the pod execution, applying a collection of priority functions to every node, trying to divide the pods among all the available nodes and giving more importance to the ones with less workload. During the last execution phase, the node with higher priority is selected, and in case of existing more than one, it a random one is choosen. The scheduler is written in Golang, just like the rest of the Kubernetes project. The scheduler performance is a key part of the system, and in some cases it become the bottleneck of the cluster. For this work, the developed algorithm will only implement the last two phases of the original scheduling algorithm, avoiding the filtering phase as it is out of scope in the work.

The scheduler that has been developed makes use of the metrics that Prometheus exports on each node through the node exporters. This means that requests to know the status of each node will be made to each of the node exporters since it is desired to know the status at that same moment. If they were made to the Prometheus Server, to obtain the measurements you would have to make the requests to the exporter node, so they will be made directly from the scheduler saving an extra jump when passing through the server. These metrics obtained from each export node are used to calculate the priority of the candidate nodes to be assigned to the pod in planning. During the planning process, five metrics exported by Prometheus are taken into account: current frequency of each core of the node's CPU, percentage of memory occupied, read and write operations on disk and the number of packets sent and received by the user interface. network in use. The ideal node for the assignment with the pod is the one that uses the least amount of all these resources at the time of selection. After this, the iPerf metrics are collected and stored into the scheduler container, to feed the scheduling algoruthm among the Prometheus metrics.

For the node ranking the scheduling algorithm will compare each node metrics, giving, for every metric, a pondered score to the node with best metrics. Each metric score has a different value, being CPU, memory and bandwidth the ones with highest weight. The priority computation function implements this logic, and is executed for every new pod in the cluster. The pseudocode for this logic can be seen in the figure below.

The algorithm makes use of the metrics exported by Prometheus and iperf, being nodeMetrics an array with the metrics for each node, cpu, mem, sentPck, recPck, bwidth and diskIO the metrics taken into account for each node in the priority computation function. For every node, each metric is compared with the same metric for the rest of the nodes, and the best metric is choosen (in the case of cpu, memory, packets sent and received and disk usage metrics, less is better and for bandwith metric more is better). The node with the best metric for each category is stored in bestCpuN , bestMemN, bestRecPckN, bestSentPckN, bestBwidthN and bestDiskION, so, at the end of the computation, scores are asigned to each of the nodes, returning an array with the nodes ordered by priority of scheduling.

Algorithm 1 Network Aware Scheduler

```
1:  function CALCULATEPRIORITIES
2:      for int i = 0; i < numNodes; i++ do
3:          if $nodeMetrics[i].cpu < bestCpu$ then
4:              $bestCpu \leftarrow nodeMetrics[i].cpu$
5:              $bestCpuN \leftarrow nodeMetrics[i].name$
6:          if $nodeMetrics[i].mem < bestMem$ then
7:              $bestMem \leftarrow nodeMetrics[i].mem$
8:              $bestMemN \leftarrow nodeMetrics[i].name$
9:          if $nodeMetrics[i].recPck < bestRecPck$ then
10:             $bestRecPck \leftarrow nodeMetrics[i].recPck$
11:             $bestRecPckN \leftarrow nodeMetrics[i].name$
12:         if $nodeMetrics[i].sentPck < bestSentPck$ then
13:             $bestSentPck \leftarrow nodeMetrics[i].sentPck$
14:             $bestSentPckN \leftarrow nodeMetrics[i].name$
15:         if $nodeMetrics[i].bwidth > bestBwidth$ then
16:             $bestBwidth \leftarrow nodeMetrics[i].bwidth$
17:             $bestBwidthN \leftarrow nodeMetrics[i].name$
18:         if $nodeMetrics[i].diskIO < bestDiskIO$ then
19:             $bestDiskIO \leftarrow nodeMetrics[i].diskIO$
20:             $bestDiskION \leftarrow nodeMetrics[i].name$
21:
22:     $nodePriorities[bestCpuN] += 3$
23:     $nodePriorities[bestMemN] += 2$
24:     $nodePriorities[bestRecPckN] += 1$
25:     $nodePriorities[bestSentPckN] += 1$
26:     $nodePriorities[bestBwithN] += 2$
27:     $nodePriorities[bestDiskION] += 1$
28:
29:     return $nodePriorities$
```

The scheduler code is compiled and packaged in a container and deployed in a pod, and has to be allowed with a role based access account to schedule and allocate new pods. The docker image is available in DockerHub [14].

IV. Performance Evaluation

In order to illustrate the proposed algorithm, a performance evaluation has been conducted. This is detailed in this section, where the experiments setup is explained in the fist place, followed by an analysis of the results obtained.

A. Work Environment Preparation

For the testing environment, four Raspberry Pi 3 Model B and one AMD64 Intel Core i7 architecture computer have been used. They have been connected using a five ports ethernet switch as shown in figure 2.

A Kubernetes cluster has been deployed over this hardware infrestructure configuring the AMD computer as the master node and the Raspberries as worker nodes, and tainting the master node to avoid the execution of pods on it.

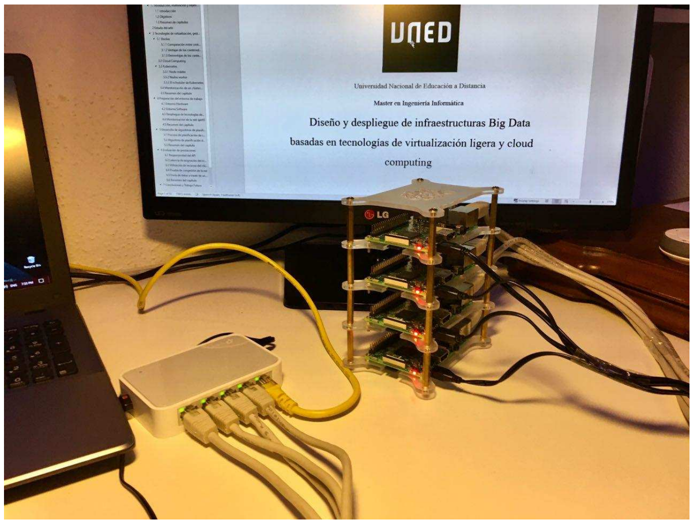
Fig. 2. Raspberry Pi Cluster built

Prometheus [?] has also been installed in the cluster, so we are able to monitor the cluster resources usage, CPU and memory occupation, pods distribution and network metrics.

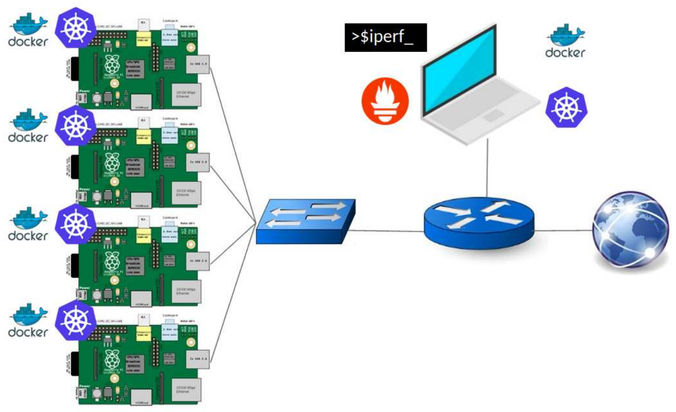

Fig. 3. Cluster hardware and software distribution

iPerf tool [?] which allows network monitoring between two or more nodes has also been installed and configured in the cluster, to collect information and metrics about the available bandwidth between each node, packet loss and delay. With this two monitoring tools we are able to know the state of the cluster and feed the container scheduling algorithm which will select the best node for a certain pod taking all this data in account.

B. Experiments and results

For the performance evaluation, the official benchmarking tools of the Kubernetes project have been used. This tool kit include different test suites in order to evaluate the performance and reliability of the cluster. For the performance of the Raspberry cluster and the planification algorithm the suite clusterloader2 [?] has been used. This suite creates a variable number of deployments, composed by a variable number of pods, which contain a Docker container with a Debian image inside. Each container starts running and executes the command pause indefinitely. This suite is especially suitable for the performance evaluation of the project since it applies a considerable workload on the cluster and the developed scheduler will be in charge of assigning the pods to the most suitable nodes based on the parameters exported by Prometheus and iperf deployment.

The clusterloader2 suite, after creating all the deployments specified in its configuration, waits for all the containers to be in the Running state and then begins to eliminate the deployments. Throughout this process, it measures the response times of the API, the use of resources by the cluster, the latency times of the pods and the API responsiveness. Thanks to this data, the behavior of the cluster with the developed scheduler can be analyzed. Three different tests have been performed with each of the planning algorithms, varying in each one the total number of pods deployed in the cluster. For the first test, ninety pods have been configured to be planned by the algorithm, for the second one, one hundred ten and for the third one hundred and thirty.

1) API responsiveness: The API responsivity measures the latencies between the execution of an operation on the Kubernetes API server and the resolution of the response. It is an indicator to take into account during the performance evaluation since the API server works together between the master node and the slave nodes to evaluate and resolve the requests, so the response times offer a good approximation to know the performance of the cluster during the execution of the suite.

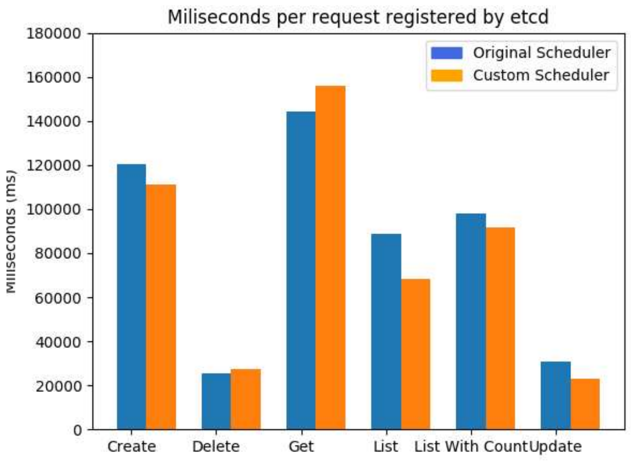

Fig. 4. API responsivness, 90 pods

Figures 3 show the average response times in milliseconds for each of the operations available on etcd during the executions with both algorithms and 90 pods. The results with bigger deployments are similar so they are not shown on this short paper. They are divided between the operations of creation, deletion, obtaining, listing, listing with counting and updating. As can be seen, the most demanding operations for the cluster are those of creation and get, due to the high number of pods housed in the cluster at the same time. It can also be seen that these are quite high response times, this is due to the nature of the cluster, being composed of low-cost computers such as Raspberry Pi, its resources are limited. They do not have high speed storage and the CPU has limitations when executing all the requests that the suite requires. This data allows to know the limitations of the cluster when analyzing possible applications of this in other areas. If the response times of both algorithms are compared, the similarity between the data obtained can be seen. This indicates that the developed scheduler has come into operation and has made an assignment of the nodes in a manner similar to that made by the original scheduler, without overloading any node, so the data collected is very similar to each other.

2) Scheduler latencies: The executed suite also offers the latency data between the state transitions of each pod, that is, the time that passes from being in one state to the next. For this evaluation, the latency between the Pending and Scheduled states has been taken into account.

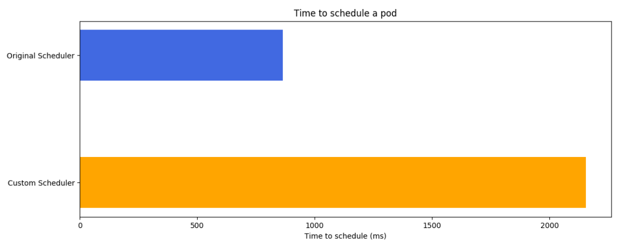

Fig. 5. Schedule time per pod

Figure 4 shows how the original Kubernetes planning algorithm is faster in pod assignment than the developed algorithm. This measurement is the same in all three executions. This has been foreseen from the beginning and it is because the algorithm developed is not optimized for use in production. The project scheduler makes several requests to Prometheus node exporters and has to load and parse the JSON file with the information about the bandwidth at each execution, that is to say every time a pod is assigned. The original scheduler obtains the CPU and memory metrics directly from the kubelet component, which is deployed on each node. This supposes a greater latency for the developed scheduler and although it has been developed in the Golang language, which offers greater performance than Python, it is an important point of improvement.

3) Network Congestion Evaluation: To test the efficiency of the algorithm in the detection and selection of nodes in the event of a network congestion, a high demand for network resources has been simulated in three of the worker nodes, limiting their available bandwidth to a minimum by means of use of the netem tool [**?**], belonging to the Linux Foundation. Once configured, the clusterloader2 suite has been re-executed

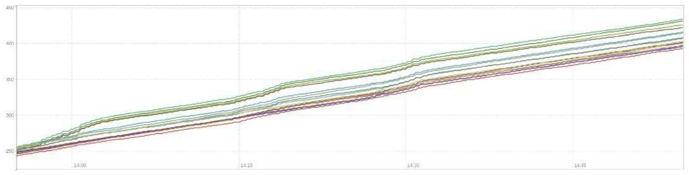

Fig. 6. CPU active time, original scheduler

with both algorithms, obtaining the following results. Figures 5 and 6 show the CPU usage time of each of the cores of the node processors. Each line represents the usage time of each processor core of the worker nodes.

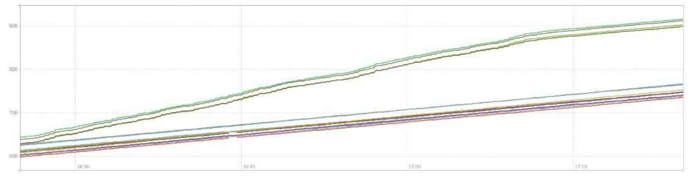

Fig. 7. CPU active time, Network Aware Scheduler

At the top, the execution with the original Kubernetes algorithm shows a similar use of the CPU of each node, indicating that the pods have been assigned in the usual way, without taking into account that there is a network congestion. In Figure 6 the CPU usage timelines that have increased belong to the unmodified worker node. The algorithm developed, thanks to the constant monitoring of the network implemented with netperf, has been able to detect the low bandwidth available by the three modified worker nodes and has assigned the majority of the workload to the remaining worker node. In this way it is proved that the developed planning algorithm is able to adapt to unfavorable situations of the network and act accordingly.

4) Sending data through an unfavorable network: The last test carried out to evaluate the favorable use cases for the developed scheduler has consisted of sending data through a network in an unfavorable situation, similar to the previous case, but in this case the clusterloader2 suite has not been executed, but a benchmark developed specifically for work. Three of the worker nodes have been modified by limiting their bandwidth to a minimum, leaving only one worker node in normal state. The benchmark developed has consisted of configuring a deployment of a variable number of containers with the image of iperf and programming each of them to perform a network test consisting of sending data. When creating the cluster deployment, the pods are assigned to the node that the running planning algorithm orders. Two tests have been carried out, the first with five containers and the second with ten containers, each sending 100 megabytes. Both tests have been performed on both algorithms. The results are presented below.

Figures 7 and 8 show the total execution time of the tests performed, which is equal to the allocation time of the scheduler added to the total delivery time of all scheduled