Network awareness for scheduling of dockerized applications in an edge/fog/cloud environment*

Pablo Jaramillo Dominguez, Agustin C. Caminero Herraez, Rocio Munñoz Mansilla
National Distance University of Spain (UNED)
Madrid, Spain
pjaramill7@alumno.uned.es, accaminero@scc.uned.es, rmunoz@dia.uned.es

***Abstract*—More and more cases of use of light virtualization and distributed computing. Both technologies offer great advantages in the development and deployment of software applications. Combined with cloud computing, they constitute a tool with great potential and foreseeable expansion in the near future. During this work, a distributed computing cluster based on light virtualization is deployed and a container scheduling algorithm is developed, finally applying a performance evaluation of the algorithm developed for the extraction of conclusions.**

***Index Terms*—virtualization, container, distributed computing, cluster, scheduling algorithm**

I. Introduction

During the last years, distributed computing has taken a key role in the research and development of information technologies. More and more proyects are being developed using this paradigm due to the advantages it offers compared to centralized or monolithic computing. This type of paradigm, makes use of container and container orquestration technologies to provide high availability, reliability and resilience for the applications that run on top of them. One of the key advantages of this technologies is the independency of the hardware platform, so they support all type of CPU architectures and configurations, including low-cost computers like the Raspberry Pi. With the use of this paradigm, new problems arise; process planification inside the cluster is a key factor to assure a good performance of the system and maintain the quality of service. The aim of this study is, combinig these ideas, deploy a Kubernetes clÃžster over low-cost Raspberry Pi computers and perform an evaluation on the influence of the planification algorithm on the capabilities of the containers deployed on the cluster.

II. State Of Art

The growth of the use of light virtualization technologises or containers is evident in the vast majority of areas where previously the heavy 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 orquestration 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 [5] a performance model of Kubernetes application is obtained with the application of benchmarks. It defends that for applications with more than eight containers, the bandwith 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 bandwith availability. In [6], 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 dinstinction between high CPU and I/O demanding services and high bandwith 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

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).

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. 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 etehernet switch. A Kubernetes cluster has been deployed over this hardware infraestructure 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. Prometheus Operator[10] 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. iPerf tool [12] 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 bandwith 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 planification algorithm which will select the best node for a certain pod taking all this data in account.

IV. Container Planification 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. 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.

Algorithm 1 Custom Scheduler

```
1:  function CALCULATEPRIORITIES
2:      for int i = 0; i < 4; i++ do
3:          if nodeMetrics[i].cpu < bestCpu then
4:              bestCpu ← nodeMetrics[i].cpu[i]
5:              bestCpuNode ← nodeMetrics[i].name
6:          if nodeMetrics[i].memory < bestMemory then
7:              bestMemory ← nodeMetrics[i].memory[i]
8:              bestMemoryNode ← nodeMetrics[i].name
9:          if nodeMetrics[i].recPackets < bestRecPackets then
10:             bestRecPackets ← nodeMetrics[i].recPackets[i]
11:             bestRecPacketsNode ← nodeMetrics[i].name
12:         if nodeMetrics[i].sentPackets < bestSentPackets then
13:             bestSentPackets ← nodeMetrics[i].sentPackets[i]
14:             bestSentPacketsNode ← nodeMetrics[i].name
15:         if nodeMetrics[i].bandwidth > bestBandwidth then
16:             bestBandwidth ← nodeMetrics[i].bandwidth[i]
17:             bestBandwidthNode ← nodeMetrics[i].name
18:         if nodeMetrics[i].diskIO < bestDiskIO then
19:             bestDiskIO ← nodeMetrics[i].diskIO[i]
20:             bestDiskIONode ← nodeMetrics[i].name
21:
22:     nodePriorities[bestCpuNode] += 3
23:     nodePriorities[bestMemoryNode] += 2
24:     nodePriorities[bestRecPacketsNode] += 1
25:     nodePriorities[bestSentPacketsNode] += 1
26:     nodePriorities[bestBandwidthNode] += 2
27:     nodePriorities[bestDiskIONode] += 1
28:
29:     return nodePriorities
```

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 planification 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 bandwith 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 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.

V. Performance Evaluation

Performance is really important for an scheduling algorithm.

References

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.