PiCasso: A Lightweight Edge Computing Platform

Adisorn Lertsinsrubtavee, Anwaar Ali, Carlos Molina-Jimenez, Arjuna Sathiaseelan and Jon Crowcroft
Computer Laboratory, University of Cambridge, UK
Email: first.last@cl.cam.ac.uk

***Abstract*—Recent trends show that deploying low cost devices with lightweight virtualisation services is an attractive alternative for supporting the computational requirements at the network edge. Examples include inherently supporting the computational needs for local applications like smart homes and applications with stringent Quality of Service (QoS) requirements which are naturally hard to satisfy by traditional cloud infrastructures or supporting multi-access edge computing requirements of network in the box type solutions. The implementation of such platform demands precise knowledge of several key system parameters, including the load that a service can tolerate and the number of service instances that a device can host. In this paper, we introduce PiCasso, a platform for lightweight service orchestration at the edges, and discuss the benchmarking results aimed at identifying the critical parameters that PiCasso needs to take into consideration.**

I. Introduction

Latest advances in lightweight OS virtualisation technologies such as Docker and Unikernels allow service providers to deploy and replicate their services on demand at the edge of the network i.e. for supporting edge/fog computing. The motivation for this approach is to improve the QoS (latency in particular), provide high level of security (provided by the strict isolation capabilities of technologies such as Unikernels) and provide privacy. An edge cloud can be built on the basis of several hardware technologies (e.g., rack of servers), however clouds built by clusters of single-board devices (e.g., Raspberry Pi, Cubie boards etc) are drawing significant attention for several reasons: these devices are cheap, consume low energy and (if optimised) have enough resources to support applications of practical interest such as those used in smart cities [1]. The aim of optimisation is to maximize the use of running devices so that we can deliver the service with as less as possible of them without compromising the necessary QoS requirements of applications.

To overcome this challenge, the service provider can benefit from flexibility of a lightweight service deployment infrastructure [2] that provides on-demand computing capacity and enables elastic service provisioning. For instance, a service image can be migrated from the service repository and automatically instantiated on the edge device after receiving a service request from the end-user. With this approach, the service provider can opportunistically aggregate idle resources from widely distributed computational devices demonstrate the huge potential of pool resources to build highly scalable, low cost and easily deployable platforms.

This paper contributes to cover the research gap. It introduces *PiCasso* – a platform for deploying QoS-sensitive services in edge clouds built of single board devices. *PiCasso* can deploy multiple instances of a given service opportunistically to ensure that it complies with service requirements. The core of *PiCasso* is the orchrestration engine that deploys services on the basis of the service specifications and the status of the resources of the hosting devices. Although PiCasso is still under development, this paper offers insights into the building of service deployment platforms (its main contribution). We demonstrate that the effort involves the execution of practical experiments to yield results to identified the parameters that the orchrestration engine needs to take into account.

II. Related Work

Several works [3]–[6] have explored the benefits of lightweight service deployment through a deployment scheme similar to ours i.e., based on microservices (e.g., Docker containers) running in low cost hardware substrate. The insight of these studies is that pushing services from centralised clouds to the edge can potentially improve end users' experience, in particular, for latency sensitive applications. However, the policies to deploy the service instances are not discussed. Our work addresses this issue, by the use of an intelligent orchestration engine that decides when/where to deploy a service instance to meet its requirements. A deployment policy that takes into account startup latencies (instantiation) is discussed in [7]. According to this work a service is deployed when the overall latency incurred is more than the life expectancy of the application itself. A limitation of this solution is that it considers only the deployment cost and overlooks the status of the resources of underlying hardware. This issue is a major concern in our work. The authors in [8] point out that the load inflicted on the hardware can significantly impact the network performance in centralised clouds. In our work, we are concerned about similar issues but in resource constrained clouds deployed at the edge.

Related to *PiCasso* are container orchestration technologies such as Docker Swarm, Mesos and Kubernetes that are used to orchestrate and manage service instances in both centralised and edge clouds [9]–[11]. Although they provide some automation and load balancing functions, they still rely on manual intervention in the administration and orchestration of the services. In our work, we aim at an intelligent orchestration engine capable of performing the managerial tasks much like Docker Swarm and other tools but in an automated fashion that is supported by the monitoring of resources.

Frameworks such as MuSIC [12] and MapCloud [13] proposed the dynamic service allocation algorithms to improve QoS while considering factors like application delay, device power consumption, user cost and user mobility patterns. Similarly, *PiCasso* aims to develop intelligent service orchestration algorithms but we consider other factors such as the current workload of underlying hardwares and characteristics of different container applications.

III. PiCasso

PiCasso[1] is a platform for service deployment at the edge of the network. Its architecture is shown in Fig. 1. The current implementation is developed in Python. It assumes a network provider-centric model where the provider is in full control of the communication infrastructure. *PiCasso* has the following components:

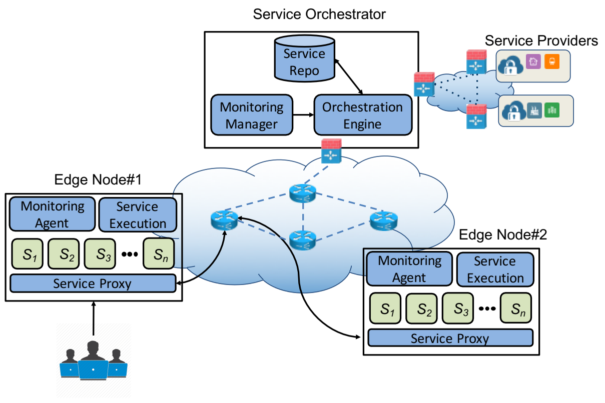


Fig. 1: PiCasso's architecture.

A. Edge node

An edge node is a single board computer such as a Raspberry Pi (RPi) or any device with storage, CPU and software facilities for hosting (upon request of the service orchestrator) the execution of microservices. Each edge node is provided with *Monitoring Agent*, *Service Execution* and *Service Proxy* functionalities.

1) Monitoring Agent is responsible for measuring the current status of resources and the current demand imposed on the services. The monitored data is formatted as json objects. The code below shows the actual data collected from an edge node, called "SEG_1".

```
{"softResources":{"OS": "Linux"},
"hardResources": {"mem": "1 GB", "disk": "16 GB", "cpu":
                             "A 1.2GHz quad-core ARMv8 CPU"},
"resourceUsage": {"cpuLoad": "0.04","memUsage": "14",
                 "cpuUsage": "24.31"},
"PiID": "SEG_1",
"PiIP": "192.0.0.2",
"containers": [{"status": "Up 3 hours", "port_host": "8002",
              "memUsage":"2015232", "name": "/adisorn",
              "port_container": "80", "cpuUsage": "58986155"
              "image":"hypriot/rpi-busybox-httpd:latest",
              "id": "dbef34542d2649a1d97521ef042c4b59ec7b"}]}
```

The *resourceUsage* key contains the values of current CPU load, memory usage and CPU usage. Also the object informs that SEG_1 is currently running a single container that has been running for 3 hours, has the name */adisorn*, has used 2015232 bytes of memory and 24.31% of CPU. This information is regularly measured and reported to the *Monitoring Manager* where the orchestration engine uses it for deciding on deployments.

2) Service Execution allows the edge node to instantiate containers automatically. It provides an API that allows edge nodes to receive *Docker images* and *json obj with deployment description* from the *Service Orchestrator*. The following json object is a deployment descriptor to instantiate a *httpd-busybox* web server.

```
deploy_descriptor={
 'imageName': 'hypriot/rpi-busybox-
              httpd:latest',
 'port_host': '80'
 'port_container': '8083'}
```

Without loss of generality, further parameters can be added to the json object to enable the deployment of more sophisticated services.

3) Service Proxy is an intermediary that seeks the service instance for clients. A user' s request is intercepted at the local edge node and forwarded to a running container that serves a requested service. If the service is not available in the local edge node, the user's request will be forwarded to the closet edge node hosting the requested service. To improve the performance and reliability, a particular service can be deployed in several containers. These containers (replica) can be deployed either in the same edge node or multiple edge nodes across the network edge. The *Service Proxy* also includes load balancing function where it distributes the users' requests across multiple containers. We intend to use HAProxy[2] to develop the service proxy and integrate it with the edge node.

B. Service Orchestrator

The service orchestrator is a central entity of PiCasso which is responsible for making an informative decision to deploy the services. The design of service orchestrator is shown in Fig. 2.

1) Orchestration Engine (OE) implements the logic for deployment of instances of services to meet specific QoS requirements. OE has access to an algorithm repository that can execute to make decisions on deployment of instances of services. For instance, an edge node can rapidly become exhausted when a number of user requests are placed against a particular service at the same time. This can significantly result in large response time perceived by the end-users. To mitigate this problem, the orchestration engine could potentially make a decision to deploy another copy of the corresponding container to balance the computation load. In this paper, we consider two replication strategies for orchestration engine. The first strategy refers to *local service replication* where a copy of the corresponding container is replicated inside the same edge node as the original container. The second strategy refers to *remote service replication* where the orchestration engine replicates a container to another remote edge node which has enough resources to handle the deployment of additional service instances. We aim to evaluate the impact of these two strategies and identify critical parameters to design effective algorithms for orchestration engine.

[1]https://github.com/AdL1398/PiCasso.git

[2]http://www.haproxy.org

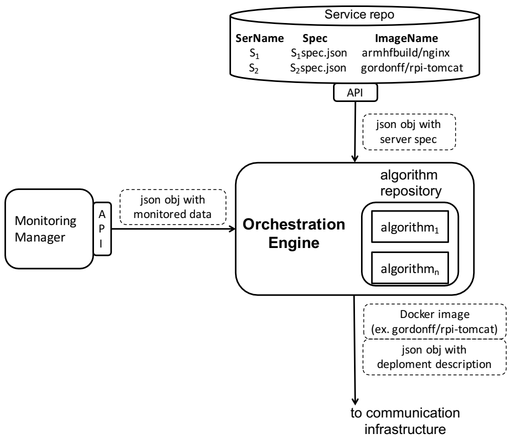

Service repo
SerName Spec ImageName
S1 S1spec.json armhfbuild/nginx
S2 S2spec.json gordonff/rpi-tomcat
API
json obj with server spec
Monitoring Manager
API
json obj with monitored data
Orchestration Engine
algorithm repository
algorithm1
algorithmn
Docker image (ex. gordonff/rpi-tomcat)
json obj with deploment description
to communication infrastructure

Fig. 2: Functions and interfaces of the service orchestrator

2) Monitoring Manager is responsible for placing pull requests against the *Monitor Agent* deployed in each edge node to collect information about the current status of their resources and the current demand imposed on the service. As shown in Fig. 2, the monitoring manager provides an API to orchestration engine to retrieve a json object with monitored data that includes information of all edge nodes.

3) Service Repo is a repository where dockerized compressed images (s_i) of the services are stored augmented with specification about their QoS requirements. In addition to service specification, we opted for pragmatic and simpler approach and hence decided to use json notation. An example of a service specification of service S_1 contains the following information:

```
S1_spec=  {'par':{
         'serviceName':  'S1',
         'imageName':    'armhfbuild/nginx',
         'imageSize':    '368',
         'maxUsers':     '50',
         'startUpTime':  '5'
         }
  'QoS':{
         'responseTime':  '5',
         'availability': '99.99',
         'numUsers":     '100'
        }}
```

The *par* key contains a set of (key, value) pairs related to the features of the service. For example, ($imageSize$, 368) indicates that the size of the image is 368 MB. Likewise, the pair ($maxUsers$, 50) indicates that an instance of the service can handle up to 50 concurrent users. Finally, ($startUpTime$, 5) indicates that it takes 5 seconds to start up an instance of the server. The *QoS* key contains a set of (key, value) pairs that stipulate the QoS requirements. In this order, S_1 is expected to support at least 100 concurrent users, be available 99.99% of the time and respond to request within 5 seconds.

IV. Performance Evaluation

PiCasso can opportunistically deploy one or more instances of a given service opportunistically to ensure its service requirements. When the orchestration engine of *PiCasso* observes that demand for a given service increases, it decides to create additional instances of the service, or alternatively, to handle the demand with the existing instances. The decision needs to be taken on the basis of the QoS benefits and cost of the deployment of additional instances. We have observed that these parameters depend on the current status of the hosting hardware, the current status of the existing instances and the particularities of the service. To clarify what parameters are needed by the orchestration engine to make decisions, we have conducted a series of experiments as follows.

A. Configuration

1) Edge nodes: To deploy service instances we use a set of RPi-3. They run the Hypriot OS Version 1.2.0[3], a customized Rasbian integrated with Docker daemon [14], [15]. We use default mode of docker engine that allows containers to compete for resources from underlying hardware.

2) Services: We use web services (perhaps the dominant application in today's Internet) as use cases to demonstrate how the particularities of the application impact service performance. We have selected four of the most popular web servers in docker hub[4] (See Table I).

Image name	Size
hypriot/rpi-nano-httpd	88kB
hypriot/rpi-busybox-httpd	2.16MB
armhfbuild/nginx	368 MB
gordonff/rpi-tomcat	251 MB

TABLE I: Docker web service base image and size

In accordance with the amount of bytes involved in the response, we regard *hypriot/rpi-nano-httpd*, *hypriot/rpi-busybox-httpd* and *armhfbuild/nginx* as lightweight web servers. The three of them deliver a single html document that consists of html text of 300 bytes with a link to a local jpeg image of 80 kB. We deliberately use a small html document to reduce the memory consumed by the document and leave it entirely at the disposition of the Docker containers. On this account, *gordonff/rpi-tomcat* is a heavy weight web server since its front page consists of multiple objects (e.g., photos, external links, java scripts). The payload size is about 750 kB.

B. Impact of the Number of Concurrent Users

In the first experiment, we evaluated the response time of containers running web servers exposed to various numbers of concurrent HTTP requests. We used the four web servers shown in Table I and configured each of them with the necessary libraries to serve a single web page. We hosted each web server in its own container, deployed the four resulting containers in four RPis (one each) and exposed each web server to a total of 10000 HTTP requests in each individual experiment. To generate the HTTP requests, we run the Apache Benchmarking (ab) tool[5] in a test machine (lenovo E560: Intel Core i5-6200U 2.3GHz, 8GB RAM, Ubuntu 14.04). We created linux shells in the test machine. In each shell, we run

[3]https://blog.hypriot.com/downloads/

[4]https://hub.docker.com

[5]https://httpd.apache.org/docs/2.4/programs/ab.html

an instance of the ab tool and configured it to create a number of concurrently active users: each user generated a number of sequential HTTP requests, i.e., a user placed a request, waited for the arrival of the corresponding response and proceeded to generated the next one.

Under this configuration, we conducted individual response time stress tests on each container varying the number of users from 5 to 1000. In this order, in a five concurrent users experiment, each user generates 10000/5=2000 requests sequentially, in a 100 concurrent users experiment, each user generates 10000/100=100 sequential requests only, and so on. It is worth emphasizing that the number of concurrent users determines the number of concurrent HTTP requests received by the container. For instance, with 1000 concurrent users, the container receives and handles 1000 requests concurrently.

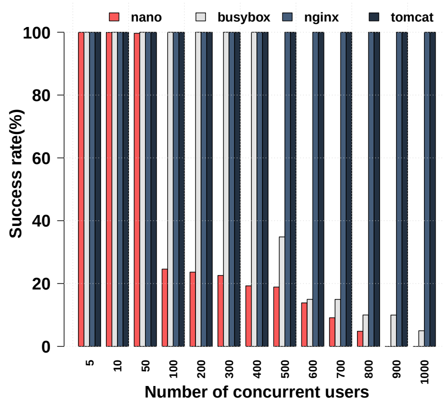


Fig. 3: Average success rate of HTTP requests

We measured the success rate at the test machine by comparing the number of HTTP requests sent by the ab tool and the number of responses received. Fig. 3 shows the average success rate of HTTP request under different concurrent user levels. We repeated each experiment for 30 times. First we examine the nano container - from five concurrent users onwards, it exhibits a success rate of 99.98% as it fails to respond to some of the 10000 requests. The success rate significantly decreases to 24.59% when the number of concurrent users is increased to 100. Busybox success rate falls to 34.85 % when the number of concurrent user level increases to 500 users. As for the nginx and tomcat containers, both can serve up to 1000 concurrent users with a success rate of 100%.

Interestingly, each container can serve different level of concurrent users with different success rate. This parameter needs to be taken into account by the orchestration engine which might decide to deploy additional instances of the containers to avoid compromising the QoS. The creation of additional instances is reasonable for lightweight containers, for example the nano container consumes only a few kB of memory of the underlying hardware. In previous work, we have proven that a single RPi can run more than 2400 instances of the nano container simultaneously [16].

C. *Impact of Local Container Replication*

In some situations it is convenient to deploy additional instances of a service, say to share the load of existing ones. The three experiments that we discuss in this section were performed to investigate how many replicas an *Edge Node* can handle without exhausting its resources (see Fig. 4). This parameter is crucial for orchestration engine. We use the ab tool to create clients that independently send a number of HTTP requests against web services. The client sends a request to retrieve a web page, waits for the response, sends another request, and so on.

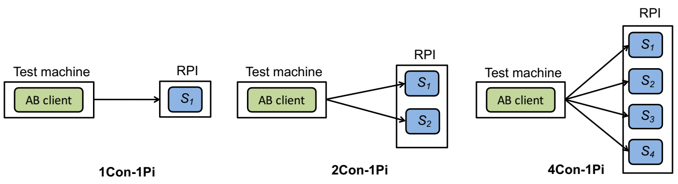
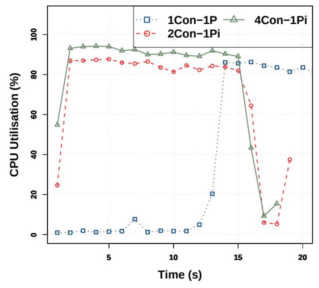


Fig. 4: Experiment settings for local container replication

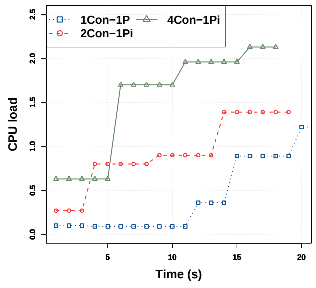
In the 1Con-1Pi experiment, we instantiated a single container in a RPi and a linux shell in the test machine. In the shell, we configured the ab tool to simulate 100 clients instructed to send 100 HTTP sequential requests, to retrieve an html document from the container. By the end of the experiment, the container would have processed 100x100= 10000 requests in total.

In the 2Con-1Pi experiment, we instantiated two containers in the RPi and created two linux shells in the test machine. In the first shell, we configured the ab tool to simulate 50 clients instructed to send 100 http sequential requests each, to retrieve an html document from the S_1 container. The second shell was configured similarly but targeted the S_2 container. Like in the previous experiment, by the end, the RPi would have processes 10000 requests in total (5000 by each container). In the 4Con-1Pi experiment, we instantiated four containers in the single RPi and created four linux shells in the test machine. In the first shell, we configured the ab tool to create 25 clients instructed to send 100 HTTP sequential requests each to retrieve an html document from the S_1 container. Consequently, S_1 received 2500 requests in total. The second, third and fourth shells were configured similarly but targeted the S_2, S_3 and S_4 containers, respectively. Due to the clients' concurrency in each shell, at any time, each container has to process 25 concurrent requests. Like in the two previous experiments, by the end, RPi would have received 10000 requests in total (2500 by each container).

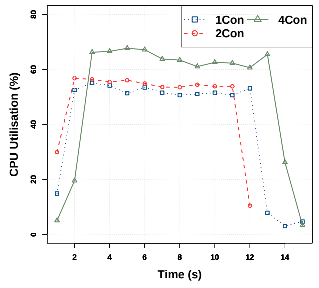
As shown in Fig. 5, we conducted the 1Con-1Pi, 2Con-1Pi and 4Con-1Pi independently with the busybox, nginx and tomcat containers with aim of measuring how the resources of the RPi are impacted by the local replications of containers. We left out nano container because its inadequacy to support large numbers of concurrent clients renders it unsuitable for these experiments. The results demonstrate that the CPU utilization, CPU load and memory usage of the RPi increase significantly when the number of containers increase. This is because the RPi allocates independent resources (for example, memory buffers and CPU cycles) to each container to handle the communication with the clients. As a result, the creation of an additional container replicates the consumption of RPi resources. These results need to be taken into consideration by the orchestration engine. For instance, in Fig. 6a, the CPU utilization of busybox container in the 1Con-1Pi experiment exhibits a sharp increase after 10s. This is a sign of exhaustion of the container and important parameter that the orchestration engine needs to take into consideration. It might decide to deploy an additional instance of the container before the QoS is compromised.



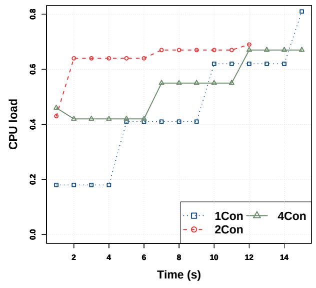
(a) CPU utilization busybox



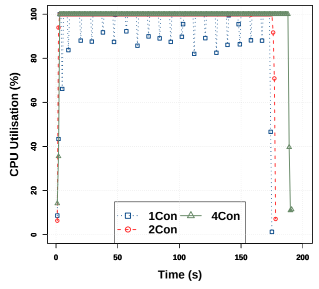
(b) CPU load busybox



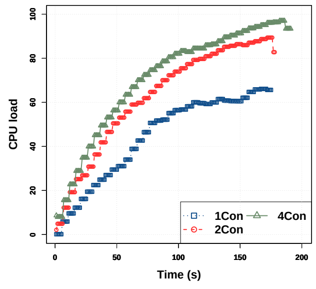
(c) CPU utilization nginx



(d) CPU load nginx



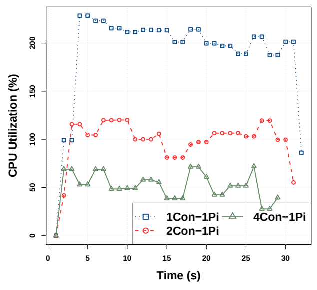
(e) CPU utilization tomcat



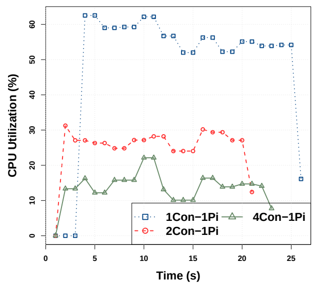
(f) CPU load tomcat

Fig. 5: Measuring CPU utilization and CPU load on RPi

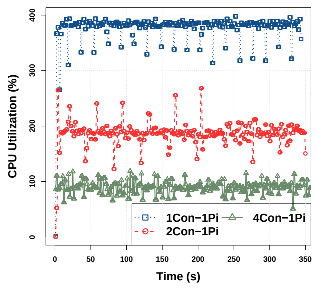
In practice, there will be several containers running in the same physical host sharing the common pool of resources. Because of this, the information about the status of the RPi's resources is not sufficient to program the orchestration engine. In addition to that, the orchestration engine needs to be aware of the status of resources consumed by each container. The results shown in Fig. 6 which compare the CPU utilization of three containers with different configurations, support our argument. The CPU utilization of all three containers fall to around 50% and 75% when two (2Con-1Pi) and four (4Con-1Pi) containers are deployed in the RPi. The three plots shown that each service exhibits different level of exhaustion. The CPU utilisation inflicted on a single nginx container is only about 60%; this finding indicates that a single instance of nginx is sufficient to handle the load (10000 requests from 100 users). With the same load as nginx, the busybox requires four containers to keep CPU utilization under 50%. The tomcat containers exhibit instability over all experiments that drove CPU utilization to the extremes (over 100%).



(a) busybox



(b) nginx



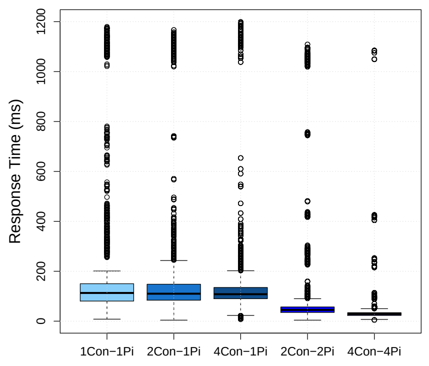
(c) tomcat

Fig. 6: Measuring CPU utilization of containers

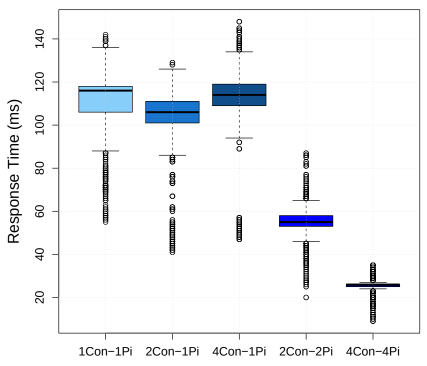
D. *Local vs Remote Container Replications*

We conducted experiments to understand how exhausted resources impact the response time perceived by clients. In addition to the three configurations mentioned in the previous section (local replication), we include other two configurations (2Con-2Pi and 4Con-4Pi) aimed at showing the impact of container replication on alternative RPis (remote replication).

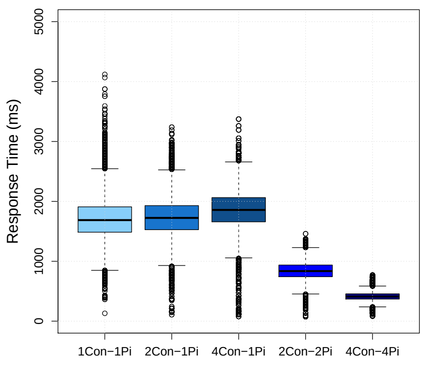
In the experiment of 2Con-2Pi, we instantiated two containers (S_1 and S_2) on two RPis - one container each. In the test machine, we created a linux shell where we used the ab tool to create 50 clients operating concurrently. Each of them sent 100 sequential requests to S_1. Thus at any time, S_1 had 50 requests to process. We associated S_2 to another shell with similar configuration. The 4Con-4Pi experiment is similar but aimed at reducing the level of client concurrency. We created four containers on four RPis. Each container was exposed only to 25 concurrent clients instructed to generated 100 sequential requests. To setup the experiments we connect four RPis and the test machine with a L2 switch via Ethernet cable. The average round trip time between test machine and each RPi is 5ms.



(a) Response time of busybox web server

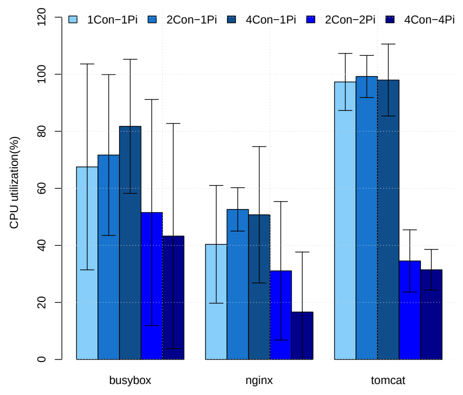


(b) Response time of nginx web server

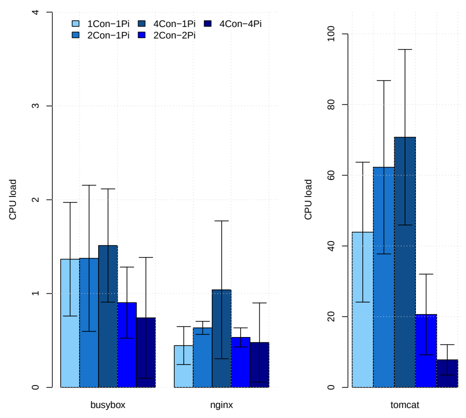


(c) Response time of tomcat web server

Fig. 7: Impact of container replication on response time with different configurations



(a) CPU usage of Raspberry Pi



(b) CPU load of Raspberry Pi

Fig. 8: Measuring hardware resources under stressing loads

Fig. 7 shows the average response time of web containers with different service deployment configurations. The computation load of 2Con-1Pi and 4Con-1Pi are theoretically reduced by 50% and 75% compared to a single container case (1Con-1Pi) as a number of concurrent users placing the request against each container is divided to 50 and 25 respectively. However, in both configurations, the end-users cannot achieve better response time. In case of busybox and nginx, the end-users achieve almost similar results for all three configurations. As for the tomcat, the average response time is slightly increased when more containers are replicated in the same RPi. On the other hand, applying the remote replication strategy (2Con-2Pi and 4Con4-Pi) significantly improves the performance of response time. For instance, in case of busybox container, the average response time is improved up to 55.01% (2Con-2Pi) and 77.25% (4Con-4Pi) compared to 1Con-1Pi case. Similar tendency is also occurred in nginx and tomcat.

The implication behind these results is related to resource exhaustion of the RPi. The measurements of CPU utilisation and CPU load are presented in Fig. 8. The deployment of two and four containers in a single RPi (2Con-1Pi and 4Con-1Pi) cause higher CPU utilization and CPU load than the deployment of a single container (1Con-1Pi). However, when the containers are deployed in another RPi (2Con-2Pi and 4Con-4Pi), the CPU usage gradually decreases.

The experiment with the tomcat container is an extreme case of resource exhaustion where the CPU is fully utilized and CPU load increases up to 70. Such a load exhausts the CPU of the RPi and severely affect its average response time which reaches up to 1847 ms (4Con-1Pi). The orchestration engine needs to be aware of these parameters and remedy the situation, for example, by deploying an additional instance of the container in another RPi to take the excessive load.

E. Container Replication Cost

Deployment of additional instances can help to improve service performance but it also introduces some extra costs.

We have identified two types of costs including network traffic cost and instantiation cost. The network traffic cost is the traffic generated by the transfer of the service image from the *Service Repo* to the *Edge Node*. It is mainly dependent on the size of service image and the bandwidth of network link. The instantiation cost is the time that edge node takes to have a newly deployed instance ready for serving. In our experiments, we measured the booting time of the docker engine on a RPi and the time it takes to create a container. Fig. 9 shows the service instantiation cost of the four web server containers. It takes about 3.37 s to boot up a docker engine in a RPi 3. This cost is zero when the docker engine is already running in the RPi. As for the containers, it takes about 1.42, 1.46, 1.44 and 1.48 s to instantiate the nano, busybox, nginx and tomcat web server containers, respectively.

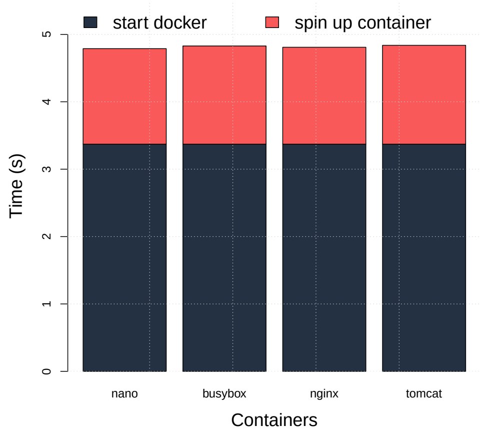


Fig. 9: Service replication cost

V. Conclusion

In this paper, we presented PiCasso, a platform for lightweight service orchestration at the edges. We also presented a series of benchmarking results that helped us to identify the critical parameters of our usecase service (web service), containers and underlying hardware. We identified how many users a service can support, how containers consume RPi resources and how many of them a RPi can support simultaneously, before showing signs of exhaustion. These parameters will be used by the orchestration engine to make appropriate service orchestration decisions - whether to replicate the services within the same device or in another device. Take as an example the benchmarking of nginx, the results show that a single container can respond to users' requests within 140 ms (see Fig. 7b). To reduce the response time, the orchestration engine can apply a *local replication* strategy and instantiate another replica of the service with the one in existence. However, this strategy is not suitable for delay sensitive services because, as shown in Fig 5, the instantiation of additional replicas of the service in the same edge node results in longer response time. In this situation, a *remote replication* strategy is a better alternative, provided that the service can bare the costs as explained in Section IV-E. The remote replication would be useful if the cost of container replication is less than the delay requirements. Understanding the system behaviour in terms of the performance is crucial to develop intelligent orchestration algorithms.

As part of future work we intend to develop intelligent service orchestration algorithms that can dynamically select a suitable service deployment/replication strategy (e.g., local, remote, etc.). The selection could be triggered automatically while balancing the tradeoffs of multiple parameters such as hardware status (e.g., work load, number of user requests), network conditions (e.g., network topology, link latency among edge nodes) and service requirement (e.g., number of users, maximum latency).

Acknowledgements

The research leading to these results has received funding from the European Union?s (EU) Horizon 2020 research and innovation programme under grant agreement No.645124 (Action full title: Universal, mobile-centric and opportunistic communications architecture, Action Acronym: UMOBILE)

References

[1] C. Perera, Y. Qin, J. C. Estrella, S. Reiff-Marganiec, and A. V. Vasilakos, "Fog computing for sustainable smart cities: A survey," *ACM Comput. Survey.*, vol. 50, no. 3, pp. 32:1–32:43, Jun. 2017.
[2] M. Bjorkqvist, L. Y. Chen, and W. Binder, "Opportunistic service provisioning in the cloud," in *IEEE Cloud Computing*, 2012.
[3] H. Chang, A. Hari, S. Mukherjee, and T. Lakshman, "Bringing the cloud to the edge," in *IEEE INFOCOM Workshop on Mobile Cloud Computing*, 2014.
[4] D. Amendola, N. Cordeschi, and E. Baccarelli, "Bandwidth Management VMs Live Migration in Wireless Fog Computing for 5G Networks," in *IEEE Cloudnet*, 2016.
[5] C. H. Benet, K. A. Noghani, and A. J. Kassler, "Minimizing Live VM Migration Downtime Using OpenFlow Based Resiliency Mechanisms," in *IEEE Cloudnet*, 2016.
[6] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, "A container-based edge cloud paas architecture based on raspberry pi clusters," in *IEEE Future Internet of Things and Cloud Workshops (FiCloudW)*, 2016.
[7] Y. Elkhatib, B. Porter, H. B. Ribeiro, M. F. Zhani, J. Qadir, and E. Rivire, "On using micro-clouds to deliver the fog," *IEEE Internet Computing*, vol. 21, no. 2, pp. 8–15, Mar 2017.
[8] R. Shea, F. Wang, H. Wang, and J. Liu, "A deep investigation into network performance in virtual machine based cloud environments," in *IEEE INFOCOM*, 2014.
[9] S. Julian, M. Shuey, and S. Cook, "Containers in research: Initial experiences with lightweight infrastructure," in *ACM XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, 2016.
[10] P. Heidari, Y. Lemieux, and A. Shami, "Qos assurance with light virtualization-a survey," in *IEEE CloudCom*, 2016.
[11] A. Tosatto, P. Ruiu, and A. Attanasio, "Container-based orchestration in cloud: state of the art and challenges," in *Complex, Intelligent, and Software Intensive Systems (CISIS)*, 2015.
[12] M. R. Rahimi, N. Venkatasubramanian, and A. V. Vasilakos, "MuSIC: Mobility-Aware Optimal Service Allocation in Mobile Cloud Computing," in *IEEE Sixth International Conference on Cloud Computing*, 2013.
[13] M. R. Rahimi, N. Venkatasubramanian, S. Mehrotra, and A. V. Vasilakos, "Mapcloud: Mobile applications on an elastic and scalable 2-tier cloud architecture," in *IEEE Fifth International Conference on Utility and Cloud Computing*, 2012.
[14] R. Rosen, "Linux containers and the future cloud," *Linux Journal*, vol. 2014, no. 240, Apr. 2014.
[15] Docker Inc., "Docker," https://www.docker.com, visited in Jan 2017.
[16] A. Sathiaseelan, A. Lertsinsrubtavee, A. Jagan, P. Baskaran, and J. Crowcroft, "Cloudrone: Micro clouds in the sky," in *ACM MobiSys - DroNet workshop*, 2016.