Characterising Resource Management Performance in Kubernetes

Víctor Medel[a], Rafael Tolosana-Calasanz[a], José Ángel Bañares[a], Unai Arronategui[a], Omer Rana[b]

[a] *Aragon Institute of Engineering Research, University of Zaragoza, Spain*
[b] *School of Computer Science & Informatics, Cardiff University, UK*

Abstract

A key challenge for supporting elastic behaviour in cloud systems is performant automated (de-)provisioning and scheduling of computing resources. Containers are rapidly replacing Virtual Machines (VMs) as the compute instance of choice in many cloud deployments, due to lower overhead of starting up and terminating containers in comparison to VMs. Performance overheads associated with deploying, terminating and maintaining a container can be significant. We analyse performance of the Kubernetes system and develop a Petri net-based model of resource management within this system. Our model is characterised using data from a Kubernetes deployment, and can be used as a basis to support capacity planning and design of scaleable applications that make use of Kubernetes.

Keywords: Performance Models, Cloud Resource Management

1. Introduction

Cloud systems enable computational resources to be adjusted on-demand and in accordance with (changing) application requirements. Applications can rent computational resources of different types: virtual machines, containers,

*Corresponding authors
Email addresses: vmedel@unizar.es (Víctor Medel), rafaelt@unizar.es (Rafael Tolosana-Calasanz), banares@unizar.es (José Ángel Bañares), unai@unizar.es (Unai Arronategui), ranaof@cardiff.ac.uk (Omer Rana)

specialist hardware (e.g. GPU or FPGA), or bare-metal resources, each having their own characteristics and cost. An effective automated control of cloud resource (de-) provisioning needs to consider [1]: (i) resource utilization, (ii) economic cost of provisioning and management, and (iii) resource management actions that can be automated. Increasingly, many cloud providers support resource provisioning on a per second or even millisecond basis, such as GCE [1], or Amazon Lambda. [2] – referred to as "serverless computing". A lambda function is provisioned through a container-based deployment, whose execution is billed at 100ms intervals. Therefore, understanding performance associated with deploying, terminating and maintaining a container that hosts that function is significant, as it affects the ability of a provider to offer finer grained charging options for users with stream analytics/ processing application requirements.

Provisioning and de-provisioning actions are subject to a number of factors [1], mainly: (i) the *overheads* associated with the action (e.g. launching a new Virtual Machine (VM) can often take minutes [2]); (ii) the actual processing time required can vary due to resource contention – leading to uncertainty for the user.

Kubernetes [3] supports a container-based deployment within Platform-as-a-Service (PaaS) clouds, focusing specifically on cluster-based systems. It can provide a cloud-native application (CNA) [4], a distributed and horizontally scalable system composed of (micro)services, with operational capabilities such as resilience and elasticity support. Kubernetes enables deployment of multiple "pods" across physical machines, enabling scale out of an application with a dynamically changing workload. Each pod can allocate multiple containers, which can make use of services (e.g. file system and I/O) associated with a pod. Any OCI compliant container runtime engine could be used, but we chose Docker as it is the most popular engine for Kubernetes.

We investigate the performance of deploying, terminating and maintaining

[1] https://cloud.google.com/

[2] https://aws.amazon.com/lambda/

containers with Kubernetes, identifying operational states that can be associated with a "pod" and container in this system. This is achieved through Reference Nets (a kind of Petri-Net (PN) [5]) based models. The models can be further annotated and configured with deterministic time, probability distributions, or functions obtained from monitoring data acquired from a Kubernetes deployment. It can also be used by an application developer / designer: (i) to evaluate how pods and containers could impact their application performance; (ii) to support capacity planning for application scale-up / scale-down.

This paper extends [6] by: (i) inclusion of additional experiments to measure performance within a larger cluster; (ii) considering the impact of variable latency/Round-Trip Time (RTT) in the communication network; (iii) analysing impact of varying numbers of containers inside a pod; (iv) analysing impact of downloading a container image at deployment time; (v) using rules to enable developers to better structure their Kubernetes deployment. The paper is structured as follows. In Section 2, we present our model and its use with real data. Section 3 presents our characterisation of the pod abstraction overhead, discussing deployment results in Section 4. Related work is presented in Section 5, with conclusions in Section 6. We present a brief description on Reference Nets and Kubernetes in Appendix A and Appendix B.

2. Kubernetes Overhead Analysis & Performance Models

The Kubernetes architecture (Appendix B) incorporates the concept of a pod, an abstraction that aggregates a set of containers with some shared resources on the same host machine. The concept of a pod plays a *key* factor in the overall performance of Kubernetes.We make use of Reference Nets (Appendix A) to model pods and containers and to conduct performance analysis.

2.1. Kubernetes Performance Model

Kubernetes supports two kinds of pods: (i)*Service Pods*: which run permanently, and can be seen as a background workload in the cluster. Two key

performance metrics are associated with them: (i.a) availability (influenced by faults and time to restart a pod/container) and (i.b) utilisation of the service (impacting response time to clients). For example, high utilisation leads to an increased response time. Several Kubernetes system services (e.g. container network or DNS) and high level services (e.g. monitoring, logging tools) are provided by Service Pods. (ii)*Job/batch Pods*: are allocated containers that execute tasks and terminate on task completion. For a Job pod, both deployment and total execution time (including restarting, if necessary) are important metrics. The restart policy of these containers can be *onFailure* or *never*.

When a pod is launched in Kubernetes, it requests resources from the Kubernetes scheduler (RAM and CPU). If enough resources are available, the scheduler selects the *best* node to deploy the pod. The requested CPU could be considered as a reservation in contingency situations. For instance, when a container is idle (e.g. it is inside a service pod and the service has low utilisation), other containers can use the CPU. With this resource model, the overall performance of the pod depends on its resource requests and on the overall workload. We could define a limit on the usage of CPU but then some other resources might remain unused.

We model the life cycle of a pod in order to estimate the impact of different scenarios on the deployment time and the performance of the applications running inside the pod. In Kubernetes, the life cycle of a pod depends on the state of the containers that are inside it. For instance, a pod has to wait until all its containers have been created. With the Reference Nets abstraction, we can provide an unambiguous hierarchical representation of the Kubernetes manager system as the System Net and the Pods (with the containers) as Token Nets. The tokens inside our Token Net are containers and tokens inside our System Net represent Pods, as illustrated in Figures 1 and 2. The models have been derived from the Kubernetes documentation[3]; specifically, from the Pod Lifecy-

[3] `https://kubernetes.io/docs`

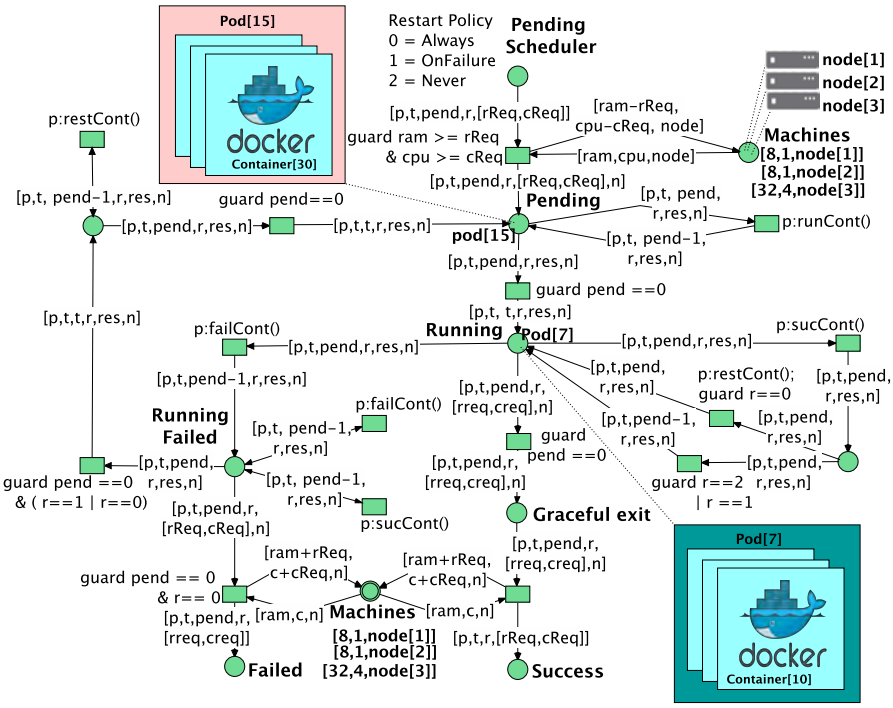

Figure 1: Model of the life cycle of pods in Kubernetes.

cle section [4] and from the Resource Management section[5]. Details about places and transitions needed to specify the initial marking are hidden to improve legibility. In addition, we assume that the scheduler assigns a pod to a single node arbitrarily, as long as the machine has enough resources available. If there are not enough resources in the cluster, the pod waits in **Pending Scheduling** place. This behaviour could be refined by introducing more sophisticated poli-

[4] `https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/`

[5] `https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/`

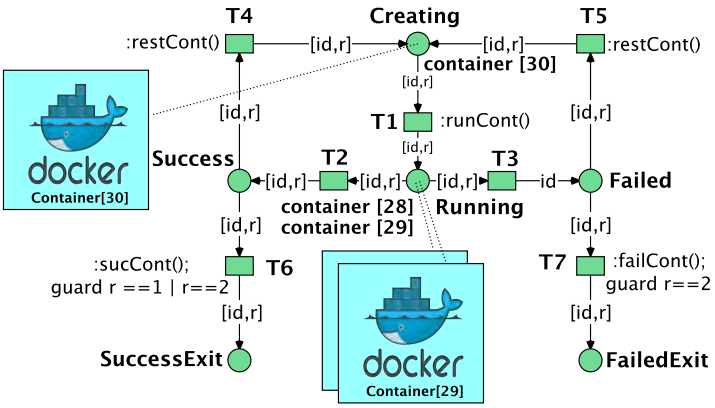

Figure 2: Model of the life cycle of containers inside a pod. r models the restart policy of a container – Always = 0, OnFailure=1, Never = 2.

cies and a rejection place for pods. Place **Machines** [6] represents the resources managed by the scheduler. For each machine, there is a tuple token with the identification of the node, the available RAM size and number of available cores. Figure 1 shows three machines ranging from 8GB to 32GB, with 1 and 4 cores. The resources assigned to a pod are only released when the pod restart policy is "never" or "onFailure" (without places and transitions needed to create the initial marking, as before).

Once the pod has been assigned to a machine, Kubernetes starts creating the containers in place **Pending Scheduler** and the pod waits in **Pending** place. The System Net model in Figure 1 synchronizes with the Token net (Figure 2) through the inscription –transition channel– *runCont*. In this way, when a container in a Pod changes to state **Running** in Figure 2, the number of pending containers in this pod is decremented in the **Pending** place of Figure 1. When all containers are running in the pod, the transition with the **guard pend**

[6] It should be noted that Place Machines appears twice: With a single circle (actual definition) and with a double circle (a duplication to simplify the model). Reference nets support double circle to simplify the model and to improve its legibility. If it were not used, several arcs would cross the model with their corresponding arc labels

Transition	Variable	Trans.	Variable
T1	Container creation	**T2**	Container execution
T3	Time to Container failure	**T4**, **T5**	Container termination time if the container is restarted
T6, **T7**	(Graceful) Container termination		

Table 1: Timed transitions in the model

==0 changes the pod state to **Running**.

While in **Running** state, the pod waits for its containers to terminate. If a container fails, the pod goes to **RunningFailed** place where it waits for the termination of all containers (with a potential restart action). If there are no failures, the pod will be in **Running** place or eventually will reach **Success** place when all containers have finished.

Figure 2 illustrates the behaviour of a container. Tokens in the net are the identifiers for each container. Their restart policy is included in the net. A created pod enters the **Running** place, and may reach the **Success** or **Failure** place. The firing of the corresponding Transitions (**T1**, **T2** and **T3**) is synchronised with the System Net. According to the restart policy, the containers might return to **Running** place or they might finish in **SuccessExit** or in **FailedExit** places. We include several timed transitions, as summarised in Table 1. By default, the firing of **T2** and **T3** is arbitrary and non-deterministic; however, with Renew, it is possible to simulate any probability distribution for them in order to simulate a failure. Additionally, it is possible to assign different random distributions for the timed transitions. In the next sections, we have made different experiments to obtain the real value of these metrics. The termination time (**T6** and **T7**) and the termination time when a container is restarted (**T4** and **T5**) does not depend on the success of the container, so both transitions are modelled with the same distribution. When a container is restarted, the total restarting time can be calcultated as **T4** – or **T5**– + **T1**.

2.2. Experiments to feed the performance model

We conducted several experiments to estimate the value of transitions in Table 1 by deploying Kubernetes on a cluster with eight physical machines –$n = 8$–, each with 32GB of RAM and 4 Intel i5-4690(3.500GHz) cores. Results are shown in the next subsections. The performance metrics of the high level model (Figure 1) is determined by the firing sequences of transitions in the token net (Figure 2). For example, if there is a pod with three containers, the **T1** transition of this pod is fired three times. The pod is waiting this time in the **Pending** place.

2.2.1. Benchmarking Starting Time

To estimate the value of Transition **T1**, we launched variable number of containers, whose image was preloaded on all machines, and measured the total deployment time. Each experiment was repeated thirty times, and we calculated the mean, the standard deviation, and the confidence intervals (for $\alpha = 0.05$). In order to calculate the confidence interval, we assumed (by the Central Limit Theorem) that the underlying distribution of the sampled mean follows a normal distribution.

In Figure 3, we show how different variables influence the deployment time. These variables are: (i) The *number of machines* available in the cluster (n). We observe that the Kubernetes scheduler launches pods sequentially, without multi-threading (red line in Figure 3), showing a linear total deployment time with the number of deployed containers. (ii) The *number of containers C inside a pod* (ρ factor). The ρ factor is calculated as follows: $\rho = \frac{\#Pods}{C}$. For instance, a ρ factor of 0.25 implies that there are 4 containers inside each pod. We can see that the time to deploy 10 pods with 4 containers in a single machine is 25.89 s., which is higher than the time to deploy 10 pods in a single container on a single machine (16.01 s.). (iii) *Cluster constraints* as the number of nodes.

The total provisioning time (T_t) is calculated as: $T_t = T_d + T_{down}$ Where T_d is the time to deploy pods and containers on physical machines and T_{down} is the time to download the needed container image into the involved machines. Our previous experiments show that T_d has a linear behaviour. Therefore, T_d

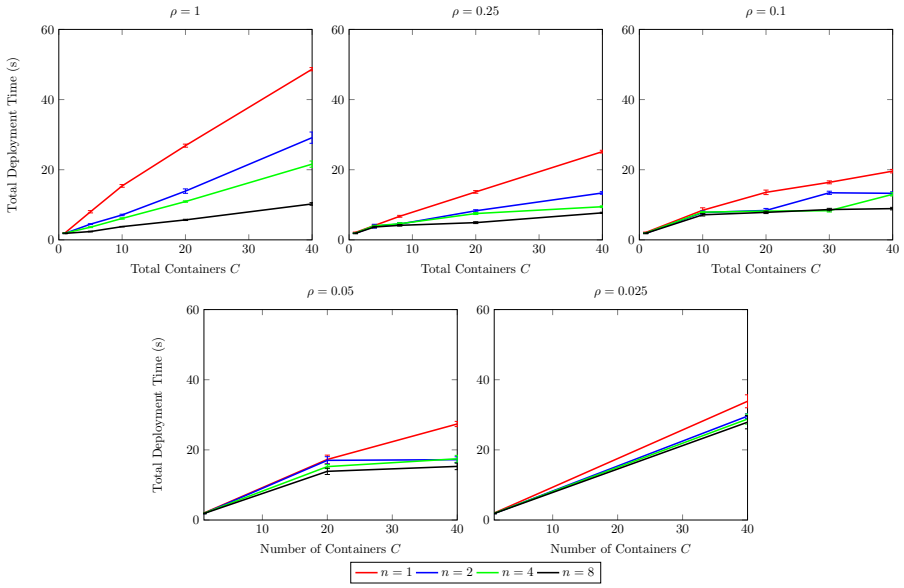

Figure 3: Total deployment time (T_d) vs. Number of deployed containers (C). Each graph shows: mean time, confidence interval for the mean for a varying number of machines in cluster, n. The results are grouped by the number of containers inside a pod, $\frac{1}{\rho}$

depends on the number of deployed containers C and the number of deployed pods, $\#Pods$: $T_d = \frac{C\ T_c(\rho,n,C)}{\min\{\#Pods,n\}}$. As the Kubernetes scheduler manages the pod as the minimal schedulable unit, the maximum number of pods deployed in parallel in a cluster is given by $\min\{\#Pods, n\}$. T_c is a function which returns the time to create a single container. This value depends on how the deployment is structured – ρ and C parameters – and the number of machines in the cluster (n). In Figure 4, we show different values for T_c obtained experimentally. We can see that for large C values, and as ρ approaches 0, T_c becomes constant. Therefore, we can write: $\lim_{\rho\to 0, n\to\infty, C\to\infty} T_c(\rho, n, C) = t_c$. Under these assumptions, the T_c value could be considered as a constant – attached to **T1** transition.

In order to characterise the impact of container image download time, we repeated the previous experiments without preloading the image. The image is

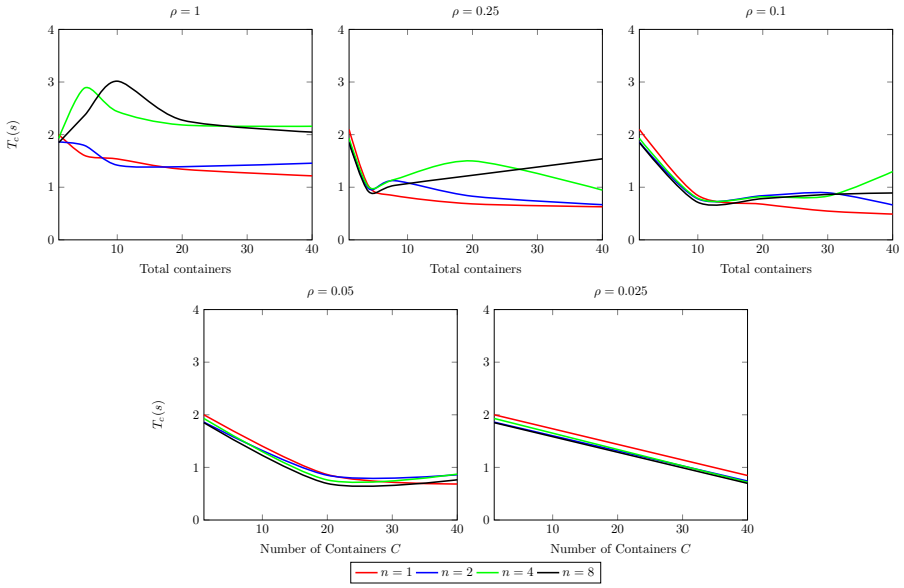

Figure 4: Time to create a *single* container (T_c function) vs. Number of deployed containers (C). Each graph shows: mean time, confidence interval for the mean for a varying number of machines in cluster, n. The results are grouped by the number of containers inside a pod, $\frac{1}{\rho}$

downloaded from a machine located inside the cluster connected directly to the same switch. The results are shown in Figure 5. The results from the homogeneous latency scenario are better than the ones from variable latency scenario – as ρ tends to zero, the latency impact on the results decreases. We can see that as the number of machines increase, the total deployment time also increases, because the image needs to be downloaded by all the machines in the cluster, and all the machines are connected to the same server. When the number of deployed pods is greater than the number of machines, the deployment time remains stable, so we can conclude that Kubernetes only downloads the image once per machine.

Several variables related to the cluster architecture impact the deployment time, such as parameters of the physical machines and the network topology. To assess the network topology impact, we repeated the experiments in a cluster

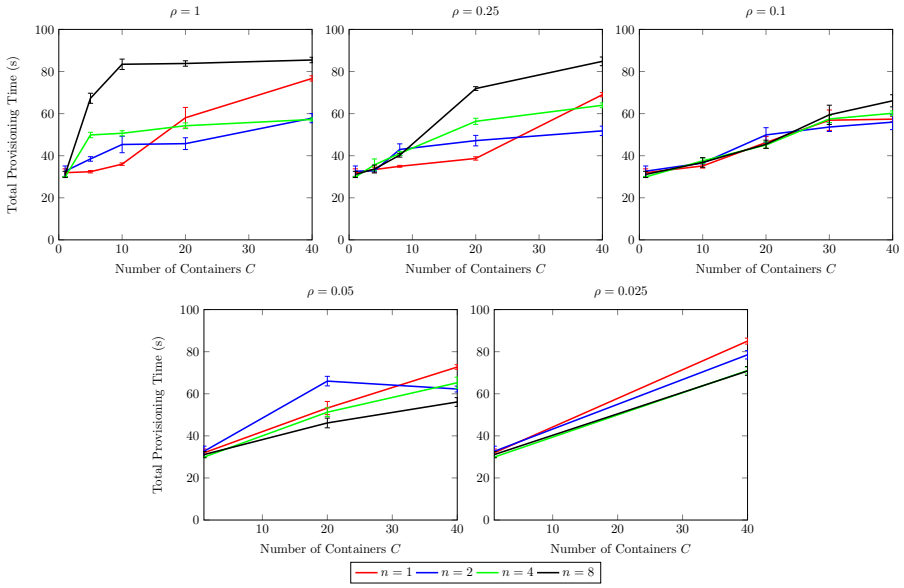

Figure 5: T_t vs. C. Each graph shows: mean time, confidence interval for the mean for a varying number of machines in cluster, n. The results are grouped by the number of containers inside a pod, $\frac{1}{\rho}$. The container image (1.225 GB) is not present in the machines.

with heterogeneous latency. We simulated that half of the machines are in a different network area, so their Round Trip Time (RTT) is about 100ms. The RTT for the rest of the machines is 0.25ms. Table 2 depicts the results for $\rho = 1$ and $n = 8$. The results for other values of ρ and n are quite similar. We can see that the latency has not a significant impact on T_d – and neither on T_c. As in T_t is included the time to download the container image, this value is higher. However, the size of the image mitigates the latency impact.

2.2.2. Benchmarking Termination Time

A Pod is expected to be terminated at some time. If it is a service, and consequently it has to be running all the time, the termination may be due to a failure and the pod has to be restarted. This philosophy also applied to containers, as we have discussed previously in the Container Net model (transitions

	Homogeneous RTT		Heterogeneous RTT	
C	T_d	T_t	T_d	T_t
1	1.85	31.09	1.94	33.26
5	2.37	67.28	2.66	66.79
10	3.77	83.44	3.87	82.66
20	5.69	83.81	5.56	86.93
40	10.24	85.47	10.21	91.02

Table 2: T_d and T_t values from a Kubernetes cluster with homogeneous RTT (0.25ms) and from a Kubernetes cluster with heterogeneous RTT. $\rho = 1$ and $n = 8$. The container image is 1.225 GB. Results are in seconds.

T4, **T5**, **T6**, **T7**). We consider **T3** to depend on the application, and it represents the failure rate (or the time between failures). When a pod terminates, Kubernetes waits a grace period (which by default is 30 seconds) until it kills any associated container and data structures.

As far as we have tested, the only variable which affects termination time of a pod is the number of containers in that pod. This occurs because when a pod finishes, all its containers have to be finished; however, in a normal scenario, pods finish – or restart – asynchronously. Therefore, the overhead caused by finalising several pods on several machines is negligible. As ρ approaches zero, the mean time to stop a single container inside a pod remains constant. How these times are aggregated and synchronised depends on the scenario, and the specific performance metrics can derived from the complete Petri Net Model.

To associate the corresponding metric for transitions, we perform the experiments shown in Table 3 in the same cluster as in the previous section. We present the results for **T4** and **T6** which correspond to a succesful scenario. Without taking into account the time to detect the failure, the behaviour of transitions **T5** and **T7** is similar: (i) *Transitions **T4, T5***: These transitions measure the time to stop a container when it is going to be restarted. We have deployed pods with a variable number of containers to measure the time. Results are shown in column "**T4** per Container" in Table 3. When we decrease ρ, the mean time to terminate a container remains constant. Additionally, the highes measured mean time is $\sim$ 0.3s and 80% of sampled times are $<$ 0.22s.

C	ρ	**T4 (T5) per Container**	**T6 (T7) Graceful termination**	**T6 (T7) per Container**
1	1	0.01	30	0
10	0.1	0.11	30.99	0.10
20	0.05	0.12	32.8	0.14
40	0.025	0.15	34.69	0.11
60	0.016	0.16	37.04	0.11

Table 3: **T4** and **T6** experimental results. Results are in seconds.

(ii) *Transitions **T6, T7***: These transitions model the normal behaviour of Kubernetes. On successful completion, Kubernetes waits for the grace period and deletes all data structures associated with a container. We measured these variables in columns "**T6** Graceful termination" and "**T6** per container" in Table 3. For these experiments, we set the grace period to 30s (the default). We can observe that stopping time remains constant for more than 10 containers in a pod (Column "**T6** per container") and for low values is negligible. It is interesting to note that the time to stop a container is higher when the container is going to be restarted. This overhead is about 10ms.

3. Overhead Analysis of Pod Abstraction

The pod abstraction allows several containers to be grouped together sharing different resources. However, how resources are shared between containers in the same pod and which is the impact on the performance of a container is not easy to determine. In this section, we analyse this performance change based on how the deployment is structured (e.g. the number of containers inside each pod), using the the total execution time as a metric. We performed several experiments to measure the overhead induced by the pod abstraction. The aim is to measure how transition **T2** is affected by the deployment configuration.

Let us consider the following scenarios: (i) *Scenario 1*: One pod is deployed and all the containers are inside that pod – $\rho = 1/c$. (ii) *Scenario 2*: Several pods are deployed and there is exactly one container inside each pod – $\rho = 1$.

The total number of containers deployed is given by C and all pods are deployed on the same machine. The machine has 12 Intel Xeon E5-2620 (2.00GHz) cores and 32GB of RAM. Each experiment, one for each scenario, was repeated thirty times, so that we can consider that the probability distribution of both means follows a normal distribution (by the Central Limit Theorem). We present the mean execution time (μ_i) and the standard deviation (σ_i). To compare both scenarios, we propose the following statistical hypothesis test:

$$\begin{cases} H_0 : \mu_1 - \mu_2 = 0 \\ H_1 : \mu_1 - \mu_2 \neq 0 \end{cases}$$

As we assume that both means follow a normal distribution and they have the same variance, we can use the Student t test [7].

As there are several resources shared between containers, we can expect different behaviour for each one. In the following subsections, we accomplished a hyphothesis test for an application with high CPU usage – Pov-Ray 3.7–, high I/O usage – IOzone benchmark – and high network usage –netperf.

3.1. CPU intensive application

We used the multi-threaded pov-ray 3.7 application as a benchmark to measure the overhead of pods for CPU intensive use. Kubernetes inherits from Docker the CPU quota reservation. This contingency mechanism allows a container to reserve a maximum quota of CPU. However, the quota is only used when there is contingency in the machine, otherwise, all available CPU is used. The comparison between Scenarios 1 and 2 is presented in Table 4. We can see that when the number of containers increases, the null hypothesis should be rejected. Additionally, when H_0 is rejected, Scenario 1 is faster than Scenario 2. The overhead caused by having one container inside each pod is about 0.01%.

3.2. I/O intensive application

We used IOzone as a representative benchmark of an I/O application. Table 5 depicts the results (in seconds) for the execution of the iozone benchmark

	Scenario 1		Scenario 2		
C	μ_1	σ_1	μ_2	σ_2	$\mu_1 - \mu_2 = 0$?
1	123.47	0.43	123.38	0.39	Yes
4	473.65	0.96	475.15	0.62	No
8	946.90	0.72	946.63	0.69	Yes
12	1417.76	1.67	1420.40	1.35	No
20	2370.21	1.16	2374.36	3.89	Yes

Table 4: Pov-ray experiment. Comparison between the execution time (s) for Scenario 1 (μ_1) and for Scenario 2 (μ_2) and hypothesis testing.

	Scenario 1		Scenario 2		
C	μ_1	σ_1	μ_2	σ_2	$\mu_1 - \mu_2 = 0$?
1	23.52	0.82	23.19	0.64	Yes
4	60.85	1.45	65.02	1.25	No
8	85.98	2.25	91.36	2.24	No
12	108.54	4.14	91.36	3.40	No
20	153.51	6.47	170.99	5.28	No

Table 5: IOzone experiment (iozone -a -i 0 -i 1 -g 4M). Comparison between the execution time (s) for Scenario 1 (μ_1) and for Scenario 2 (μ_2) and hypothesis testing.

– the benchmark was executed as follows: *iozone -a -i 0 -i 1 -g 4M*. If we compare both scenarios, we can conclude that there is enough statistical evidence to accept H_0. As the number of pods in a machine increases, the caused overhead is higher. The conclusion of these experiments is that it is better to group all the containers in the same pod.

3.3. Network intensive application

The network infrastructure of a machine is shared by all the containers inside a pod. All the containers in a pod share the port space and the pod has only one IP address. Sharing the access to the network by several containers might cause an overhead on the container performance. To measure the overhead, we deployed an `iperf` server inside a pod and several clients with the previous scenario configuration. All test measure the network bandwidth for a 30 second interval of TCP traffic.

The first experiment schedules all the containers at the same machine. In

	Scenario 1			Scenario 2			
C	μ_1	σ_1	$\sum \frac{BW_i}{C}$	μ_2	σ_2	$\sum \frac{BW_i}{C}$	H_0?
1	1.88	0.06	1.88	1.90	0.04	1.90	Yes
4	8.61	0.21	2.15	8.82	0.05	2.20	Yes
8	15.53	0.12	1.94	16.26	0.20	2.03	No
12	14.99	0.21	1.25	16.42	0.38	1.37	No
20	15.10	0.19	0.75	18.32	0.91	0.91	No

Table 6: Hypothesis test for Network bandwidth (GB) for C iperf Clients. Iperf server & client are on the same machine.

	Scenario 1			Scenario 2			
C	μ_1	σ_1	$\sum \frac{BW_i}{C}$	μ_2	σ_2	$\sum \frac{BW_i}{C}$	H_0?
1	108.26	0.04	108.26	108.26	0.04	108.26	Yes
4	110.53	0.39	27.63	110.17	0.84	27.54	Yes
8	113.81	0.47	14.23	115.64	0.51	14.46	No
12	117.42	0.92	9.78	117.53	4.01	9.79	Yes
20	124.74	1.22	6.24	126.52	2.09	6.33	No

Table 7: Hypothesis test for Network bandwidth (MB) for C iperf Clients. Iperf Clients are on a different physical machine from the server one.

a real scenario, this situation might arise when the scheduler groups pods / containers together with high network traffic among them. In Table 6, we show the average bandwidth per container and the hypothesis test. When the number of containers inside the pod is above four, there is enough statistical evidence to reject H_0. The best results are achieved when each pod has an isolate container (Scenario 2).

We repeated the experiments with the iperf server placed on a machine and the clients scheduled in another machine. Table 7 shows the results, which are similar to the previous ones. The bandwidth values from Scenario 2 are higher than the values from Scenario 1. From these experiments, we can conclude that deploying several pods with a few coupled containers is better than a single pod with a large number of containers.

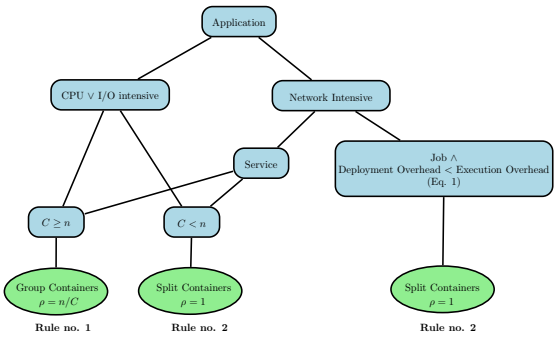

Figure 6: Decision-tree to choose the best ρ parameter. C is the number of containers to deploy and n is the number of machines in the cluster.

4. Discussion

We demonstrated that the deployment of an application on a specific type of infrastructure can impact its overall execution performance. We used results from our experiments to derive rules which try to improve performance. Since we assume that, by default, the Kubernetes scheduler assigns a pod to a host machine, then ρ is the parameter which has the highest impact on performance. We assume that the Kubernetes nodes are homogeneous and that all the containers can be distributed across physical nodes, improving the performance of the application – i.e. there is no coupling between containers, and this is considered as a design restrictions. Figure 6 summarises the rules to choose the best ρ from our experiments

If an application is CPU or I/O intensive it is better to group all the containers together –from experiments in Tables 4 and 5. However, we want to distribute the pods among as many machines as possible. As the pod is the minimal schedulable unit, the ρ parameter has to reflect this situation. If the number of containers is greater than the number of machines, ρ should be equal to $\frac{n}{c}$ (**Rule no. 1**) – we have n pods with $\frac{c}{n}$ containers in each pod. This rule tries to minimise the impact of T_d – which decreases for low values of ρ. If the number of containers to deploy is less than the number of machines, then $\rho = 1$ (**Rule no. 2.**) – we deploy a pod with a container in each machine.

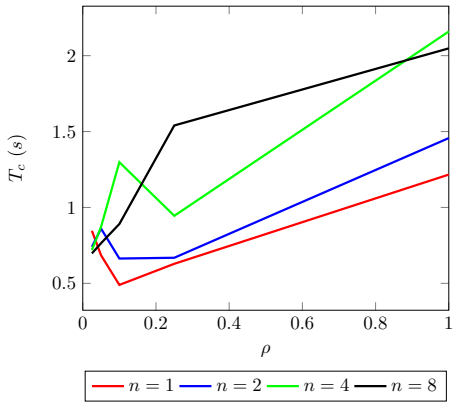

Figure 7: Function $T_t(\rho)$ for different values of n. The number of deployed containers is assumed to tend to infinity

The ρ choice is different if we consider an application which makes a high use of the network. If it is a service pod and there are few failures in the scenario –equivalently, T_d is negligible – the best choice is to set $\rho = 1$. The reason is that regardless of the machine where a pod is scheduled, the effective bandwidth is higher when there is only one container inside a pod (Table 6 and Table 7). However, if T_d is relevant, we can calculate the total time T_t (deployment time T_d plus execution time T_e) as a function of ρ: $T_t(\rho) = T_d(\rho) + \alpha(\rho)T_e = \frac{c}{n}T_c(\rho) + \alpha(\rho)T_e$; where $\alpha(\rho)$ is the overhead caused by the pod abstraction (Section 3) and it can be calculated as $\frac{\mu_1}{\mu_2}$, where μ_2 corresponds to a scenario with $\frac{1}{\rho}$ containers per pod. In general, as μ_2 is expected to be greater than μ_1, then $\alpha > 1$. Additionally, $\alpha(1) = 1$. Figure 7 depicts an example of $T_c(\rho)$, calculated when $C \to \infty$, obtained from Figure 4.

It is a complex task to minimise the function $T_t(\rho)$. As a simplification, we can assume that $\alpha(\rho)$ remains constant and when n tends to infinity, the mean time to create a container also remains constant. In our experiments, for low values of ρ (Table 7), its value is about 1.01. Assuming that the major improvement in the execution time is achieved by executing tasks in parallel, we can compare the situation where $\rho = 1$ versus $\rho = \frac{n}{c}$. The first one will be faster than the second one when:

$$T_t(1) < T_t(n/c) \implies \frac{c}{n}T_c(1) + T_e < \frac{c}{n}T_c(n/c) + \alpha T_e \implies$$
$$T_c(1) - T_c(n/c) < \frac{n(\alpha - 1)}{c}T_e \quad (1)$$

If Equation 1 is satisfied, then the **Rule no. 2** is applied. Otherwise, **Rule no. 1** will be more suitable.

These rules are based on the experiments in Section 3. There are other container technologies – such as Linux LXC or Core OS rocket – which can be abstracted in a similar way. The use of a particular technology does not have an impact on our model, as many of these container framework will also share similar lifecycle states. However, the performance values may vary depending on the use of a particular container framework/ technology. In Section 5, we provide a comparison of the performance of different technologies. Additionally, there are different container management systems as Docker Swarm, Apache Mesos. Although these other platforms do not have the pod abstraction and mechanism, our models and results could be relevant to them in scenarios where $\rho = 1$.

In our work, we have proposed a methodology to feed the model and to analyse the overhead of pod abstraction. This methodology should be applied to different configurations to generalise our approach. For instance, all our experiments have been carried out within a private cloud and Kubernetes has been deployed over a *bare metal* system. This configuration allows us to avoid the additional overhead caused by the execution of Kubernetes inside VMs. On the other hand, the Google Cloud Engine platform gives the possibility of running a Kubernetes cluster; however, the containers are run over VMs, which may have an impact on the performance and the hypothesis tests may change. Additionally, the underlying service architecture is different. For example, as the storage service is accesses through the cloud, the I/O intensive application will have a different behaviour, and the overhead caused by pod abstraction may not be negligible.

5. Related Work

Work	Model	Virtualisation infrastructure	Experimental framework
[8]	Experimental approach	VMs	Hyper-V, KVM, vSphere, Xen
[9]	Experimental approach	VMs	KVM, Xen, vBox
[10, 11]	Experimental approach	Containers and VMs	Docker, KVM, Xen, LXC
[12]	Experimental approach	Containers and VMs	LXC, OpenVZ, VServer, Xen
[13]	Experimental approach	Containers	LXC
[14, 15]	Experimental approach	Containers	Docker, KVM
[16]	Experimental approach	Containers	Docker, Weave
[17, 18]	Continuous Markov Chains (Exponential PDF)	Containers over VMs	Docker Swarm over Amazon EC2
[19]	Nets within Nets (any PDF)	VMs	Simulations
Our work	Nets within Nets (any PDF) Experimental approach	Containers	Kubernetes over bare metal

Table 8: Summary of related work with the kind of model they proposed, the assumptions of the model, the virtualisation infrastructure that they used and the experimental framework.

We summarised the most important work in performance evaluation of cloud environments in Table 8. Most of it focused on performance comparison of virtual machines rather than on containers as the unit of computation. A set of workloads were developed to get the usage of memory, CPU, networking and storage [20, 15, 14]. However, all these works are based on experimental results that do not have an analytical model which supports reasoning about performance decisions.

To the best of our knowledge, no work has tackled the container performance topic from a rigorous analytical perspective. Even in more traditional cloud technologies, few works are based on formal models [17]. An analytical model, based on Continuous Time Markov Chains, is used in [18] to study the provisioning performance of microservices. This model is validated with experiments deployed in a specific microservice platform, which is based on the execution of Docker containers, with Docker Swarm providing cluster management support and Amazon EC2 as the virtual machine backend. However, the proposed analytical model assumes that the rate of workload generation follows a Poisson distribution – the time between arrivals has an exponential proba-

bility density function (PDF) – which may yield to non-realistic scenarios. In our work, we can link transitions on the Petri Net model with any PDF or with functions obtained from real application benchmarking, so more realistic scenarios can be modelled.

In [19], the researchers propose an iterative and step-wise refinement methodology that begins with the modelling of functional requirements. Afterwards, they model control and data flow together with the specification of the computational resources available. As a result the performance of the distributed system can be formally analysed by considering all the aspects that can affect performance.

All these models need temporal information to be fed. Previous works, mainly in the context of execution of traditional VMs in Clouds, were focused on obtaining this kind of information [8, 9]. Several works provide a performance comparison between virtual machines and containers [15, 10, 11]. These works show a better performance and resource usage in containers. Scalability performance in Kubernetes was studied in [21], where the results show that containers perform 22x times faster than VMs for the provisioning action.

The container platform evaluation, as Docker and LXC in [12], shows performance issues being improved and this approach being considered for High Performance Computing (HPC). The researchers conclude that the performance is near-native for both technologies. In [13], the authors present several variables (e.g. Linux kernel version) which have an impact on the performance of containers. However, both works focus on the execution time as a performance metric and they do not consider other measures, which can be significant (e.g. deployment time). They also assume that the image is pre-load in the machines and they do not provide a full methodology to use their results.

As we have described in our work, Kubernetes introduce a new abstraction, the pod. To the best of our knowledge, the performance degradation of this abstraction has not been studied by literature. Analysis of nested containers is the closest research field. In [14, 16], network performance degradation was observed in some configurations because of a deployment based on full nested

containers. This degradation is caused by the usage of network virtualization technologies – Linux Bridge or OpenvSwicht – twice or by the usage of Software Defined Networks and encryptation. However, the Kubernetes pod abstraction gives a common space port to all containers – and therefore the same IP address to all services – so the performance degradation may be different.

Finally, several performance metrics of Kubernetes [7] were reported by the Kubernetes team. In a cluster of 100 nodes, their results show a 99th percentile pod startup time below three seconds. This value is consistent with our results. Their experiments show how a Kubernetes cluster behaves when the scale of the deployment is increased for several simple metrics – pod start time, end-to-end response time and response time of different API operations –; however, how the deployment is structured and the impact of grouping containers inside a pod is not analysed.

6. Conclusion and Future Work

In general terms, an efficacious automated resource management in cloud computing requires to launch, terminate and maintain computing instances rapidly, with a minimum overhead. In this paper, we proposed a Petri Net-based performance model for Kubernetes. It allows us to analyse deployment and termination overheads of containers in Kubernetes, as well as understanding the performance of different configurations of a Kubernetes pod (i.e. the influence of the number of containers per pod). We conducted our analysis in a Kubernetes cluster of 8 machines. Our model can be exploited as a basis to improve two activities: (i) capacity planning and resource management; (ii) application design, specifically how an application may be structured in terms of pods and containers. From our experiments, we can see that a single container can be deployed in a time interval than ranges from less than a second to up to 3 seconds, depending on the circumstances, i.e. the number of pods

[7] http://blog.kubernetes.io/2016/03/1000-nodes-and-beyond-updates-to-Kubernetes-performance-and-scalability-in-12.html

per container, the number of containers deployed simultaneously, the network latency, or the number of host machines. In contrast, the termination time is typically in the order of a tenth of a second. Moreover, we also provide a set of rules that assist in allocating the number of containers per pod that provides the best performance. These rules consider a number of characteristics of the application, such as the usage of CPU or network.

As future work, we expect to study the resource contention phenomenon in containers, which appears when multiple processes (i.e. containers) compete for the same computational resource, and as a result performance degrades. We plan to exploit our model with different purposes: (i) to estimate the overhead introduced by resource contention in containers, (ii) to improve the Kubernetes scheduler so that it is aware of resource contention problems and (iii) to undertake various *what-if* scenarios to investigate the behaviour of different resource management policies.

Acknowledgments

This work was supported by: The Industry and Innovation department of the Aragonese Government and EU Social Funds (COSMOS group, ref. T93) and the Spanish Ministry of Economy (Programa I+D+i Estatal de Investigación, Desarrollo e innovación Orientada a los Retos de la Sociedad – TIN2013-40809-R). V. Medel was the recipient of a fellowship from the Spanish Ministry of Economy.

References

[1] R. Tolosana-Calasanz, J. D. Montes, L. F. Bittencourt, O. F. Rana, and M. Parashar, "Capacity management for streaming applications over cloud infrastructures with micro billing models," in *Proceedings of the 9th Intl. Conf. on Utility and Cloud Computing, UCC 2016, Shanghai, China, December 6-9, 2016*, 2016, pp. 251–256.

[2] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A performance analysis of ec2 cloud computing services for scientific computing," in *1st Intl. Conf. on Cloud Computing (CloudComp), Munich, Germany*, 2009.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016.

[4] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017.

[5] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[6] V. Medel, O. Rana, J. a. Bañares, and U. Arronategui, "Adaptive application scheduling under interference in kubernetes," in *Proceedings of the 9th Intl. Conf. on Utility and Cloud Computing*, ser. UCC '16. New York, NY, USA: ACM, 2016, pp. 426–427.

[7] Student, "The probable error of a mean," *Biometrika*, pp. 1–25, 1908.

[8] J. Hwang, S. Zeng, F. y Wu, and T. Wood, "A component-based performance comparison of four hypervisors," in *2013 IFIP/IEEE Intl. Symposium on Integrated Network Management (IM 2013)*. IEEE, 2013, pp. 269–276.

[9] A. J. Younge, R. Henschel, J. T. Brown, G. Von Laszewski, J. Qiu, and G. C. Fox, "Analysis of virtualization technologies for high performance computing environments," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 9–16.

[10] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *Cloud Engineering (IC2E), 2015 IEEE Intl. Conf. on*. IEEE, 2015, pp. 386–393.

[11] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," *Advanced Science and Technology Letters*, vol. 66, no. 105-111, p. 2, 2014.

[12] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro Intl. Conf. on*. IEEE, 2013, pp. 233–240.

[13] C. Ruiz, E. Jeanvoine, and L. Nussbaum, "Performance evaluation of containers for hpc," in *European Conference on Parallel Processing*. Springer, 2015, pp. 813–824.

[14] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *14th IEEE Intl. Symposium on Network Computing and Applications, NCA 2015, Cambridge, MA, USA, September 28-30, 2015*, 2015, pp. 27–34.

[15] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE Intl. Symposium on*, March 2015, pp. 171–172.

[16] N. Kratzke and P.-C. Quint, "How to operate container clusters more efficiently? some insights concerning containers, software-defined-networks, and their sometimes counterintuitive impact on network performance," in *International Journal on Advances in Networks and Services*, vol. 8, 2015, pp. 203–214.

[17] H. Khazaei, J. Misic, and V. Misic, "Performance analysis of cloud computing centers using m/g/m/m+r queuing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 5, pp. 936–943, 2012.

[18] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, "Efficiency analysis of provisioning microservices," in *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on*. IEEE, 2016, pp. 261–268.

[19] A. Merino, R. Tolosana-Calasanz, J. Á. Bañares, and J. M. Colom, "A specification language for performance and economical analysis of short term data intensive energy management services," in *Economics of Grids, Clouds, Systems, and Services - 12th Intl. Conf., GECON 2015, Cluj-Napoca, Romania, September 15-17, 2015*, ser. LNCS, vol. 9512, 2015, pp. 147–163.

[20] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 275–287, mar 2007.

[21] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in.* IEEE, 2015, pp. 342–346.

[22] O. Kummer, F. Wienberg, M. Duvigneau, J. Schumacher, M. Köhler, D. Moldt, H. Rölke, and R. Valk, "An extensible editor and simulation engine for petri nets: Renew," in *Intl. Conf. on Application and Theory of Petri Nets.* Springer, 2004, pp. 484–493.

Appendix A. Background: Reference nets

A Petri Net [5] is one of several formal models for the description and the analysis of distributed, parallel or concurrent computing systems. It can be seen as a bipartite graph, where nodes can be of two types, either places or transitions; and arcs connect a transition with a place or viceversa. A place is often represented by a circle, whereas a transition by a rectangle. Besides, places can contain a (discrete) number of tokens (a token is typically represented as *[]*). In order to model a system, all these constituents can represent the dynamics of a system in a number of different ways. For instance, a transition can model a system action, and a place can represent a state, arcs can represent that each transition has a certain number of input and output places, modelling pre-conditions and postconditions of the system action. Tokens move from one pre-condition state to a post-condition state, when the involved transition is fired. In this way, it can be used to capture the evolution of system semantics. In this paper, we are making use of a particular type of Petri nets called Reference Nets. Reference Nets belong to the class of High-Level Petri nets (HLPN) [22]. An HLPN is a Petri net whose tokens are represented by data structures or even objects. The pre-conditions of an HLPN can be labeled by expressions that identify states defined by the value of tokens. Besides, post-conditions can be labeled by expressions that define state changes by the modification of token values. In this way, HLPNs provide a more concise representation than ordinary Petri nets. In essence, Reference nets extend High-Level Petri nets with some characteristics that support the construction of hierarchical models, by allowing a token to be a net itself, creating hierarchies of nets. The nets forming part of such hierarchies can communicate by means of synchronous channels. Synchronous channels can be seen as a sophisticated way of message passing communication, but with a richer semantics based in the unification mechanism. A synchronous channel engages two transitions (typically from different nets) that, by means of the channel, fire (synchronize) simultaneously. The channel can also accommodate variables and it has two main roles: The

uplink or callee role, at the subordinated instance net, which servers requests. The *downlink* or caller role, at the parent net, which makes use of the channel to both synchronize and call the subordinate instance.

Channel communication and hierarchies provide the way to describe finely grained complex behaviors in the form of nested Petri nets. The usefulness of net instances lies in the fact that they can represent behaviors that can be moved to different execution environments. The inscription language of Reference nets have also been extended to include tuples, which can be used for representing a group of related values or variables in a single token. A net instance can be influenced by the net that holds it, called System net, and such influence is accomplished by means of the synchronous channels mechanism.

The execution of a HLPN specification requires to find bindings, i.e. mapping of variables used in are expressions to specific values.

Reference Nets can be interpreted by Renew [8] [22], a Java-based Reference net interpreter and graphical modeling tool. The way Renew binds variables to values is by unification, the same mechanism used by logic programming languages such as Prolog. In HLPNs, when a transition fires, then its expression is evaluated and tokens are moved according to the result. Furthermore, Reference Nets incorporate some characteristics from object-oriented languages: Reference Nets support the creation of net instances dynamically, and transitions also support the inclusion of inscriptions, including Java inscriptions. Therefore, Reference Nets also support the creation of Java objects, and Java method invocations inside a net. They also incorporate the assignment operator "=", and it can be used to define (assign) the value of variables.

Reference Nets can hold two kinds of tokens: Valued tokens and tokens which correspond to a reference to a PN instance. By default, an arc will transport a black token, denoted by `[]`. In case an inscription is added to an arc, that inscription will be evaluated and the result will determine which kind of token is moved.

[8] `http://www.renew.de`

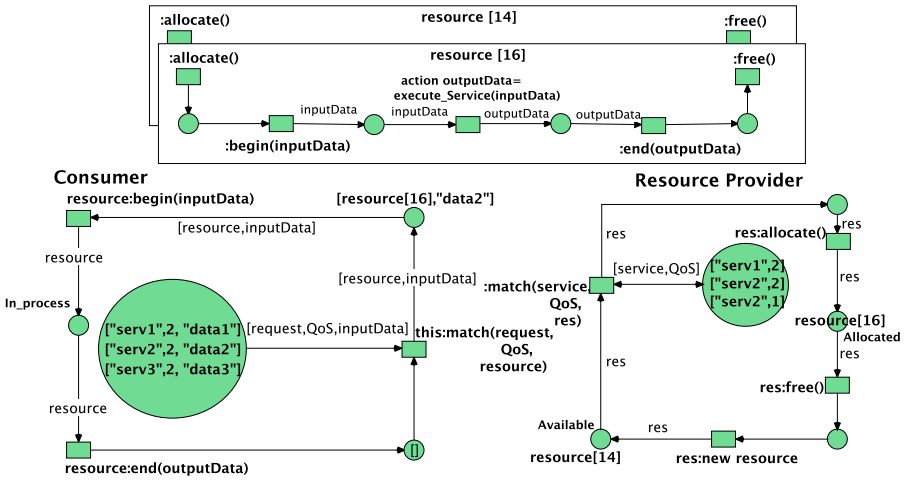

Figure A.8: A Reference net model that represents a `Service Consumer` and a `Resource Provider`. References to `Resources` net instances are managed by the consumer to provide data and collect results, and providers to manage resources.

In order to illustrate the main concepts of Reference Nets, Figure A.8 depicts a Reference net Nodel that represents a `Resource Consumer` and a `Resource Provider` with allocating capacity of two resources. Services are required with a QoS and data to be processed when `Consumer` invokes the `match` channel. Communication happens when unification of variables is possible. In the state represented in the figure, the transition labelled with the downlink `this:match(request, Qos, inputData)` channel in the `Consumer` can synchronise with transitions labelled with uplinks `:match(service, QoS,Res)` in the `Resource Provider`. Figure A.8 shows one allocated resource providing service, and one available resource. There are two possible bindings with `data1` and `data2`, but there is only one available resource, and therefore transitions with downlink and uplink will be synchronously fired with one of the two possible bindings.

Appendix B. Kubernetes

Kubernetes is an open source platform that abstracts and automates the deployment of applications across a number of distributed, computational resources. Kubernetes makes use of container technologies in order to manage and provide computational resources. In this section, we will briefly analyse container technologies and Kubernetes in particular.

Appendix B.1. Container approach

VMs have been one of the first and most important cloud computational resources. A VM is a piece of software that emulates a hardware computing system and typically multiple VMs share the same hardware to be executed. The emulation is accomplished by a hypervisor. Hypervisors are responsible for dividing the hardware of the host physical machine, so that it can be used by the OS inside each VM. Therefore, applications that run inside a VM can accomplish calls to their own OS inside the VM, and then their virtual kernel executes instructions on the physical CPU of the host machine by means of the hypervisor.

One of the most important benefits of using VMs is the full isolation they achieve: VMs on the same host physical machine share the same hardware, but they are completely isolated. Nevertheless, VM utilization can sometimes be difficult to achieve, e.g. when the applications to be run do not consume all the resources of a VM. Developers can therefore can try to map multiple applications onto the same VM. In this case, applications would not be isolated. *Containers*, on the other hand, represent a way to solve that isolation problem by improving utilization. A container can be seen as a set of processes where an application is executed in isolation. Multiple containers typically coexist on the same host machine, and each container in it uses the resources that the application on it consumes. Nevertheless, the degree of isolation achieved by VMs is still higher than the one achieved by containers, but containers have much less overhead. The reason for it is that all containers deployed in the same host machine share

the same OS kernel, and therefore virtualization is not required. Furthermore, while a VM needs to boot up first before an application can be executed on it, a container is a group of processes whose execution can be initiated almost immediately. The isolation of containers is obtained by two Linux mechanisms: Linux namespaces and Linux Control Groups, which isolate the view of the system and limit the amount of computational resources, respectively.

Although container technologies were developed many years ago, they became popular with the emergence of Docker. Docker containers became popular due to the fact that (i) they also help deploy library dependencies with the application and (ii) the high portability of containers on different platforms. A Docker container is a Docker image in execution. A Docker image contains an application and all the libraries required for its execution. In turn, Docker images are stored at the Docker registry, which also facilitates the availability of such images.

Appendix B.2. Kubernetes Architecture

Kubernetes is a system that allows developers to deploy and manage containerized applications. It is based on a master-slave architecture, with a particular emphasis on supporting a cluster of machines where containers are executed. Applications are submitted to the master and Kubernetes, in turn, deploys them automatically across the worker nodes in containers. The communication between Kubernetes master & slave nodes is realised through the *kubelet* service. This service must be executed on each machine in the Kubernetes cluster. The node which acts as master can also carry out the role of a slave during execution. The basic component architecture of Kubernetes is shown in Figure B.9. As Kubernetes often works with Docker containers, the *docker daemon* should be running on every machine in the cluster. In addition, Kubernetes makes use of the *etcd* project to have a key-value distributed storage system, in order to coordinate resources and to share configuration data of the cluster. The master node runs also an API server, implemented with a RESTful interface, which gives an entry point to the cluster. The API service is used as a proxy to ex-

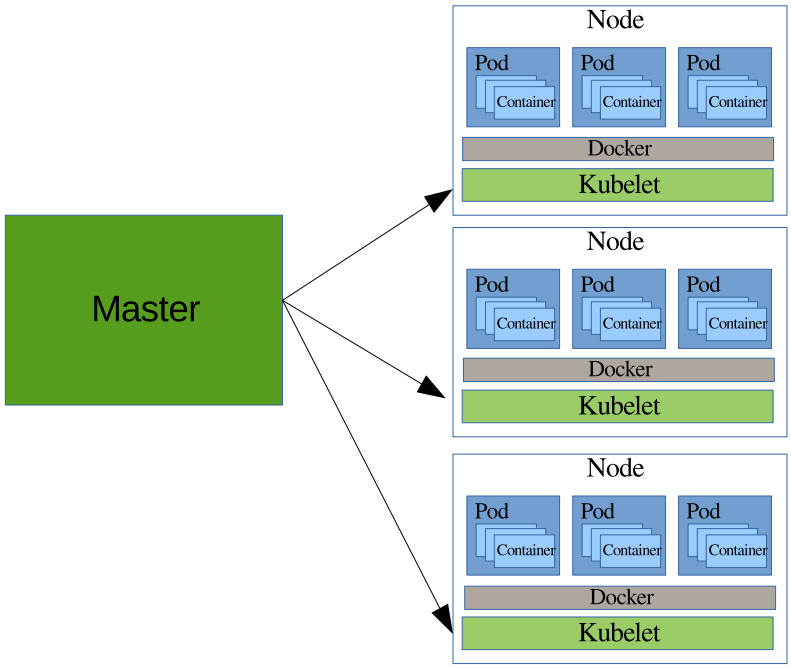

Figure B.9: Kubernetes architecture.

pose the services which are executed inside the cluster to external applications/ users.

In order to run an application, it has to be wrapped on one or more container images, then push (submit) these images to a container service registry, and then post a description of the application in the form of an application descriptor to the API server. Then, Kubernetes will automatically retrieve the container images when the application is launched. Instead of deploying containers individually, Kubernetes deploys *pods*. A pod is an abstraction of a set of containers tightly coupled with some shared resources (the network interface and the storage system). Any OCI compliant container runtime engine could be used to execute containers in pods. With this abstraction, Kubernetes adds persistence to the deployment of single containers. It is important to note two aspects of a pod: (i) a pod is scheduled to execute on one machine, with all containers inside the pod being deployed on the same machine; (ii) a pod has

a local IP address inside the cluster network, and all containers inside the pod share the same port space. Therefore, each pod has a unique IP address in a flat shared networking space that allows bidirectional IP communications with all other pods and physical computers in the cluster. The main implication of this is that two services which listen on the same port by default cannot be deployed inside a pod.

A pod can be replicated along several machines for scalability and fault tolerance purposes. When a service or a set of services are deployed over several machines, we can consider: (1) the *functional level* or application level involves exposing dependencies between the deployed services. Different services need to be coordinated in order to provide a high level functionality. An example of this kind of relationship is the deployment of a stream processing infrastructure (e.g. Apache Kafka, Storm, Zookeeper and HDFS for persistence) or the GuestBook example provided by Kubernetes, composed of a PHP frontend and a Redis master-slave system. (2) the *operational level* or deployment level involves mapping services to physical machines, VMs, pods or containers. It is platform dependant and must involve isolation between resources. Kubernetes primarily focuses on the operational/ deployment level. A pod implements a service, and some coordination between different pods is achieved through the key-value distributed store provided by *etcd*. Services running in others pods can be discovered through a DNS. This approach imposes some restrictions to Kubernetes. For instance, in the Guestbook example, Kubernetes' scheduler cannot ensure that the three pods are deployed rightly, because Kubernetes does not manage the application level.