Computers and Electrical Engineering 68 (2018) 286–297

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Characterising resource management performance in Kubernetes☆

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

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

ARTICLE INFO

Keywords:
Performance models
Container lifecycle
Cloud resource management
Petri nets
Kubernetes

ABSTRACT

A key challenge for supporting elastic behaviour in cloud systems is to achieve a good performance in automated (de-)provisioning and scheduling of computing resources. One of the key aspects that can be significant is the overheads associated with deploying, terminating and maintaining resources. Therefore, due to their lower start up and termination overhead, containers are rapidly replacing Virtual Machines (VMs) in many cloud deployments, as the computation instance of choice. In this paper, we analyse the performance of Kubernetes achieved through a Petri net-based performance model. Kubernetes is a container management system for a distributed cluster environment. Our model can be characterised using data from a Kubernetes deployment, and can be exploited for supporting capacity planning and designing Kubernetes-based elastic applications.

1. Introduction

Cloud systems enable computational resources to be acquired (and released) on-demand and in accordance with (changing) application requirements. Users can rent computational resources of different types: virtual machines (VMs), containers, 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) the resource management actions that can be automated. Increasingly, many cloud providers support resource provisioning (and billing) on a per second or even per millisecond basis, such as GCE,[1] or Amazon Lambda[2] – referred to as "serverless computing". 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 VM can often take minutes [2]); and (ii) the actual processing time required can vary due to resource contention – leading to uncertainty for the user.

Kubernetes [3] is a system that enables 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. From an architectural

☆ Reviews processed and recommended for publication to the Editor-in-Chief by Associate Editor Dr. L. Bittencourt.

* Corresponding author.
E-mail addresses: vmedel@unizar.es (V. Medel), rafaelt@unizar.es (R. Tolosana-Calasanz), banares@unizar.es (J.Á. Bañares), unai@unizar.es (U. Arronategui), ranaof@cardiff.ac.uk (O.F. Rana).

[1] https://cloud.google.com/.
[2] https://aws.amazon.com/lambda/.

https://doi.org/10.1016/j.compeleceng.2018.03.041
Received 30 May 2017; Received in revised form 23 March 2018; Accepted 23 March 2018

point of view, Kubernetes introduces the *pod* concept, a group of one or more containers (e.g. Docker, or any OCI compliant container system) with shared storage and network. In this paper, we investigate deploying, terminating and maintaining performance of (Docker) containers with Kubernetes, identifying operational states that arise with the associated *pod*–container. 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; or (ii) to support capacity planning for application scale-up / scale-down.

This paper extends [6] by: (i) the inclusion of additional experiments in a larger cluster; (ii) considering the impact of variable latency/Round-Trip Time (RTT) in the communication network; (iii) analysing the impact of varying the number of containers inside a pod; (iv) analysing the impact of downloading a container image at deployment time; (v) using rules to assist developers to better structure their Kubernetes deployment. The paper is organized as follows. In Section 2, we describe our model. Section 3 shows our pod abstraction overhead characterization. We discuss the deployment results in Section 4 and related work in Section 5. The conclusions are outlined in Section 6.

2. Kubernetes overhead analysis & performance models

The Kubernetes architecture[3] incorporates the concept of a pod, an abstraction that aggregates a set of containers with some shared resources at the same host machine. It plays a *key* factor in the overall performance of Kubernetes. We make use of Reference Nets[3] to model pods and containers and to conduct performance analysis. Reference Nets can be interpreted by Renew [4] [7], a Java-based Reference net interpreter and a graphical modelling tool.

2.1. Kubernetes performance model

Kubernetes supports two kinds of pods: (i) *Service Pods*: They are 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 the 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*: They are 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 (RAM and CPU) to the Kubernetes scheduler. If enough resources are available, the scheduler selects the *best* node for deployment. 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 CPU usage limit, but then some resources might remain unused.

We model a pod's life cycle in order to estimate the impact of different scenarios on the deployment time and the performance of the applications running inside a pod. In Kubernetes, a pod's life cycle depends on the state of the containers that are inside it. For instance, a pod has to wait until all its containers are 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 the Token Nets. The tokens inside our Token Net represent containers and the tokens inside our System Net represent Pods, as illustrated in Figs. 1 and 2. The models were derived from the Kubernetes documentation [5]; specifically, from the Pod Lifecycle section[6] and from the Resource Management section.[7] 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 policies and a rejection place for pods. **Machines** place [8] 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. Fig. 1 shows three machines ranging from 8GB to 32GB, with 1 to 4 cores. The resources assigned to a pod are only released when the pod restart policy is "never" or "onFailure". Once the pod has been assigned to a machine, Kubernetes starts creating the containers – it is in **Pending Scheduler** place – while the pod waits in its **Pending** place. Both nets are synchronized through the inscription *runCont*. In this way, when a container in a Pod changes to **Running** place in Fig. 2, the number of pending containers in this pod is decremented in the **Pending** place of Fig. 1. When all containers are running in the pod, the

[3] We present a brief description on Reference Nets and Kubernetes in two Appendices, which are available at http://cos2mos.unizar.es/COS2MOS/papers/COMPELECENG_2017_1250_Appendix.pdf.

[4] http://www.renew.de.

[5] https://kubernetes.io/docs.

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

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

[8] It should be noted that **Machines** place appears twice: One 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