

Characterising Resource Management Performance in Kubernetes. Appendices.

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*

Appendix A. Background: Reference nets

A Petri Net [1] 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) [2]. 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

*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)

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 ¹ [2], 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 evalu-

¹<http://www.renew.de>

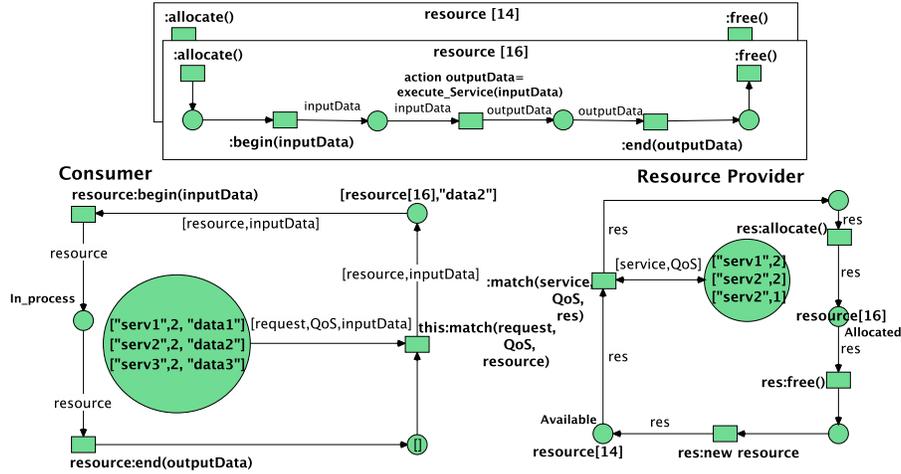


Figure A.1: A Reference net model that represents a **Service Consumer** and a **Resource Provider**. References to **Resources** net instances are managed by the consumer to provide data and collect results, and providers to manage resources.

ated 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.

In order to illustrate the main concepts of Reference Nets, Figure A.1 depicts a Reference net Model 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.1 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 [3] 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 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 be-

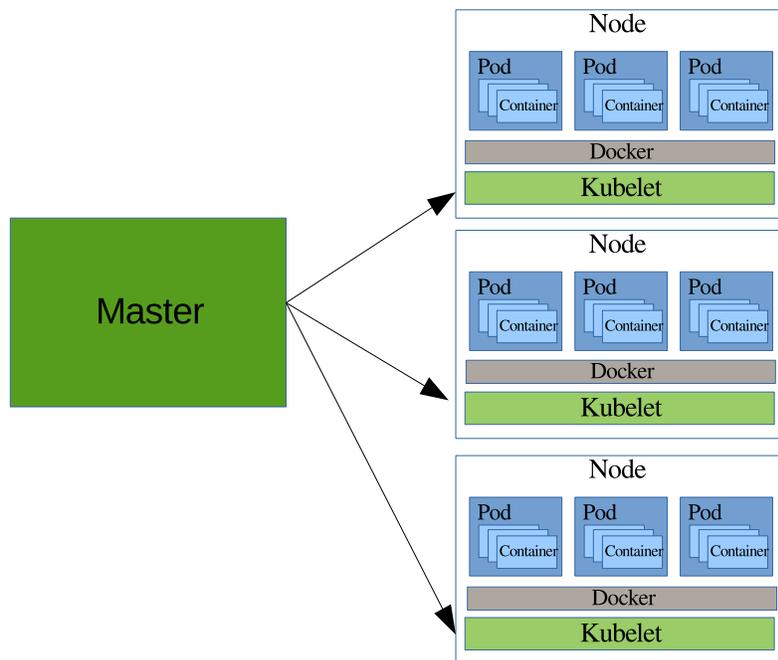


Figure B.2: Kubernetes architecture.

tween 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.2. 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 expose 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.

References

- [1] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.
- [2] 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.
- [3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016.