Commodore: Fail Safe Container Scheduling in Kubernetes

Christos Christodoulopoulos, Euripides G.M. Petrakis

Abstract  Kubernetes is a tool to facilitate deployment of multiple virtualized applications using container technology. Kubernetes scheduling mechanism orchestrates computing resources per application at runtime. However, resource allocation is static, as the maximum amount of computing resources that each application can use, must be reserved in advance. If the application requests more resources than the maximum, a fail scheduling event is generated. Although solutions to the problem of automatic scaling in Kubernetes are known to exist and automatic scaling is supported by cloud providers such as Amazon and Google, these solutions are fully proprietary and not generic (e.g. do not apply to all Kubernetes distributions). Our solution, referred to as “Commodore”, is capable of allocating (or deallocating) resources based on the actual demands of running applications. Taking advantage of the virtualization features of cloud computing, applications are deployed on worker machines (nodes) as Virtual Machines (VMs). This not only results in better utilization of computing resources (i.e. CPU, memory and network are defined virtually) but also, in enhanced software security by isolating services or applications from each other. The experimental results demonstrated that Commodore responds to the increasing (or decreasing) resource demands of each application leading to significantly faster response times compared to a non-auto scaled implementation.

1 Introduction

Scalability refers to the ability of a system to continue to function well when its size (i.e., in terms of computing resources) has to change (i.e. increase or decrease) in order to respond to increasing or decreasing workloads. A system is scalable if its performance changes uniformly (e.g. linearly) with the size of the workload. Scalability can appear as a problem of infrastructure, where the lack of computing resources can decrease the performance of the system or, as a problem of architecture when the design of the system itself does not allow it to scale.

Christos Christodoulopoulos
School of Electrical and Computer Engineering, Technical University of Crete (TUC), Chania, Greece, e-mail: chr.christodoulopoulos@gmail.com

Euripides G.M. Petrakis
School of Electrical and Computer Engineering, Technical University of Crete (TUC), Chania, Greece, e-mail: petrakis@intelligence.tuc.gr
This work focuses on the infrastructure aspect of the problem. If an application is experiencing an increased workload, the addition of computing resources (e.g. more CPU or memory) and a reconfiguration of the run-time environment may solve the problem (vertical scaling). If this is not good enough to guarantee uniform change of performance, more instances of the application can be spawned in the same or different physical or virtual machines (horizontal scaling). Cloud computing along with virtualization technology facilitate such operations by providing an abstraction layer that decouples the hardware from the environment that runs on top of it and by supporting features like process isolation and resource customization [5, 2].

Containers provide the benefits of cloud virtualization in a lightweight manner. Solutions to the problem of autoscaling of resources in virtualized environments are known to exist [4, 1].

Kubernetes, Container orchestration and management system that automates deployment, scaling and management of containerized applications. Each application can run in many copies in clusters of nodes (each node is typically deployed as a VM). Each node can run more than one instance of the application. The maximum number of nodes is specified in advance. Inside a node, an application is deployed in clusters of containers referred to as “pods”; each pod is deployed as a separate Docker environment running one or more containers. Kubernetes resumes responsibility for scheduling and managing each application and its replicas inside each cluster. Kubernetes can also be configured to balance the traffic across clusters. If the system experiences a heavy workload and a derating of performance is observed, the solution is to increase the number of pods (horizontal scaling).

An alternative would be to support vertical scaling (in the example of [1] or live container migration between machines (e.g. VMs) when there is no enough resources on a machine [1]. However, this solution is rather static as it is constraint by the allocated resource capacity of the running deployment and, it does not show how new VMs can be allocated or deallocated according to workload.

Kubernetes can attempt to schedule an additional instance of the application in an existing node. However, if the maximum number of nodes is reached, Kubernetes fails to allocate an extra node and emits a fail event. Commodore [3] is a solution to the problem of infrastructure scalability on a virtualized cloud environment managed by Kubernetes. It adds infrastructure scalability features to Kubernetes, by supporting addition or removal of computing resources (as VMs) depending on cluster state. In Kubernetes, a pod is defined by specifying the maximum amount of resources (i.e. CPU and RAM) it is allowed to consume. Typically, Kubernetes scheduler decides where to put the new pod based on this amount of resources and places the pod in a node only if the capacity of the node is less that the resource re-

1 https://www.docker.com/resources/what-container
2 https://kubernetes.io
quests of the pod (even though the actual resource usage within the node can be low). Although simple and fast, this strategy is not always optimal (i.e. may result in underutilization of resources).

As an example, consider a two node Kubernetes cluster with an application deployed in it as illustrated in Fig. 1. The nodes are identical in terms of computational resources (i.e. each can run up to two pods). The application is deployed in 3 replicas (pods) running in these two nodes (i.e., one copy of the application in Node 1 and two copies of the same application in Node 2). Each pod consumes half of the computational resources of each node. Now consider the following scenario:

- The application experiences heavy load slowing down response times (i.e. the pods can not cope with the increased workload).
- The system observes the derating and decides to increase the number of pods from 3 to 5 in order to reduce the load per running application (pod).
- The system instructs Kubernetes to schedule two pods in the existing nodes.
- Kubernetes tries to schedule two new pods.
- Only one additional pod can be scheduled in Node 1 (Fig. 1). No more resources are available and the fifth pod can’t be scheduled.
- Kubernetes emits a fail event and can not take any further action.

![Fig. 1 A Kubernetes deployment with two nodes and three instances of an application (pods).](image)

Kubernetes is lacking an optimal mechanism for providing the additional resources (see Sec. 2 for technical detail). This is the problem Commodore is dealing with that is, the problem of infrastructure scalability in a virtualized environment managed by Kubernetes. Commodore service is capable of automatic provisioning or de-provisioning of computational resources auto-
matically in a Kubernetes cluster: New work nodes can be added to deal with increasing workloads or, un-used work nodes can be freed. Continuing our previous scenario but with Commodore service in place:

- Commodore received the schedule fail event emitted by Kubernetes.
- Commodore assumes that a smaller node (i.e. a VM with half the resources of the existing VMs) can be used to accommodate the fifth pod.
- Commodore contacts Kubernetes about the needed resources.
- Commodore inquires the infrastructure provider (i.e. the Nova service\(^3\) of Openstack in our implementation) for a VM with the computational resources that Kubernetes needs.
- The cloud provider creates a new VM.
- The new VM is initialized and bind to the Kubernetes cluster as Node 3 (as illustrated in Fig. 2).
- Kubernetes schedules the fifth pod of the application to run in Node 3.

2 Background and Contributions of the Work

In the following, existing solutions to the problem of scaling in Kubernetes environments are discussed highlighting the contributions of Commodore.

Azure Kubernetes Service (AKS\(^4\) (formerly known as Azure Container Service) is the container management service by Microsoft Azure. It allows quick deployment of a production ready Kubernetes (DC/OS2 or Docker

\(^3\) [https://docs.openstack.org/nova/latest/](https://docs.openstack.org/nova/latest/)

Swarm) cluster. It runs on a standard VM infrastructure using a container optimized Linux image (a stripped-down version of Linux kernel) as host operating system. In order to deploy a Kubernetes cluster on AKS, a user must use Azure CLI (Command Line Interface), which is a shell specific for Azure operations. A cluster can be created by specifying the number of VMs needed. In terms of scaling, worker nodes can be added or removed from the cluster using Azure CLI. Autoscaling is not supported.

Amazon Elastic Container Service (ECS) by Amazon Web Services (AWS) allows management of Docker containers on a cluster of Amazon Elastic Compute Cloud (Amazon EC2) instances (VMs). It can be used to launch and stop containerized applications by making API calls, allows monitoring the state of the cluster from a centralized service and integrates with many familiar AWS features like Elastic Load Balancers, CloudTrail, CloudWatch etc. The cluster setup is a single step process in which the number and flavor of EC2 instances needed for the cluster is specified. The rest of the setup process, as well as the management of those instances is handled by the ECS service. In terms of autoscaling, ECS provides a mechanism which lets a user configure policies on how scaling operations take place. A policy consists of a set of rules and a set of actions. Rules typically refer to thresholds defined upon utilization metrics and actions refer to scaling operations. An example policy would be the following: If CPU utilization in all cluster nodes is above x%, create a new node with y amount of computing resources and add it to the cluster. ECS, autoscaling operations are driven by measurements of the actual computing resources consumed. In addition to scaling, ECS can move applications (tasks) across the cluster of nodes in order to achieve better utilization, meaning that ECS’s scheduler operates based on the actual run-time utilization of each application.

Google Kubernetes Engine (GKE) uses a proprietary Kubernetes distribution and runs on top of Google Compute Engine (GCE). The cluster setup process, is carried out in steps during which, a developer can specify (a) the size of the cluster in terms of VMs, (b) the flavor and the deployment location of these VMs, (c) the operating system image for the VMs, (d) the Kubernetes version to be deployed. GKE provides logging and monitoring tools (Stackdriver) which can be enabled during the setup process. Autoscaling decisions are based on CPU utilization, HTTP utilization (i.e. rate of requests), load balancing policies or Stackdriver Monitoring metrics but does not apply an elaborate rule-based auto-scaling policy similar to Amazon ECS.

Cluster Autoscaler (CA) is a Kubernetes plug-in (it is part of the official Kubernetes project) that enables infrastructure scalability. Autoscaling

5 https://aws.amazon.com/ecs/
6 https://cloud.google.com/kubernetes-engine/
7 https://cloud.google.com/compute/docs/autoscaler/
8 https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler
decisions are based on the amount of resources that pods request, not the amount of resources they are actually using (as Amazon ECS and Google GKE do). CA automatically resizes a Kubernetes cluster by adding new nodes to the cluster if there are not enough computing resources available; conversely, if a node is under-utilized and its pods can run on other nodes, CA moves the pods and deletes the node. However, CA’s autoscaling is rather static: The minimum and maximum size for each node pool in a cluster must be specified in advance and CA makes rescaling decisions within these boundaries. On the one hand, it is reasonable not to allow an application to scale without limits; on the other hand, this policy is not optimal and results (often) in under-utilization of resources since CA reserves the maximum amount of resources in advance.

In regards to autoscaling, Amazon ECS and Google GKE are optimal or almost optimal respectively. Both CA and Commodore rely on non-optimal strategies: Autoscaling decisions and requests for new pods are based on the amount of resources that pods request, rather than on actual utilization metrics (e.g. CPU, RAM). As explained in Sec. 1, this policy, although simple and fast, may result in under-utilization of resources. CA, Google GKE and Amazon ECS, may move pods from one node to another in order to achieve better resource utilization. Commodore does not support this functionality (i.e. moving pods among nodes is left as future work). Notice that, Amazon ECS and Google GKE are fully proprietary solutions running on Amazon’s and Google’s platform respectively. Commodore and CA can operate on any Kubernetes infrastructure. Finally, unlike CA which makes scaling decisions within limits (i.e. the maximum size of the node pool must be specified in advance so that the maximum amount of computational resources are reserved), Commodore is fully dynamic (i.e. scales-up or down within limits or, if desired, as long as it is needed) but without reserving any resources in advance.

3 Architecture

Commodore is implemented in Openstack. Application services are deployed in worker machines (nodes) which are realized as Virtual Machines (VMs). This allows for definition of virtualized resources including network, CPU and memory leading to better utilization of physical resources while ensuring software security by isolating applications from each other. Commodore runs in cooperation with the Kubernetes master node (alternatively it can be implemented within the master node). It makes periodic checks to the cluster state, by requesting the latest events emitted by Kubernetes scheduler. Commodore acts when there are scheduling events that fail (i.e. events emitted by the scheduler when it fails to deploy one or more pods in the cluster). Then, a scale-up operation is initiated following the steps described in Sec. 1. If there are no failed scheduling events, Commodore checks if a scale-down operation can take place: Inquires Kubernetes API server regarding what pods are deployed on each node. If there are nodes
Commodore comprises of many interconnected components (or services) shown in Fig. 3, the most important of them being: The Database, the Collector Service, the Infrastructure manager, the Kubernetes manager, the Rule Engine and the User Interface. For implementation details of all Commodore services please refer to [3].

Commodore produces large amounts of usage data (logs) that account for utilization scores from the cluster, events from the Kubernetes API server, as well as history data regarding scaling operations. All data is stored in an Apache Cassandra (NoSQL) database, which suits best the nature of semi-structured log data and for operations allowed on such data (ACID transactions are not mandatory).

Collector (Fig. 4) collects utilization scores, stores them in the database and exposes an API for accessing the database. It consists of two components, a daemon process and a Web server. The daemon is a script that runs on each cluster node. It collects utilization scores from the node at regular time intervals (i.e. every second) including CPU, disk and memory usage and reports them to the server. It is implemented as a Web application that exposes a set of REST services that can be invoked in order to submit and retrieve utilization metrics. It also performs a moving average aggregation on the incoming data, which compute the average utilization of each node per minute. This aggregated score is a more accurate representation of the actual resource utilization within the node since instant resource usage metrics change sharply over time. These scores are used only for visualization. Utilization scores may potentially power a more advanced autoscaling

---

9 [http://cassandra.apache.org](http://cassandra.apache.org)
mechanism based on actual resource utilization metrics (similar to Amazon ECS). This is left as future work (see Sec. 5).

![Diagram of Commodore Collector](image)

Fig. 4 Commodore Collector.

The User Interface (Web application) allows cluster administrators to monitor the actual cluster load without the need of extra tools. It is responsible for monitoring a) the health status of all services, b) the number, flavor and state of the Kubernetes nodes, c) the actual utilization of resources within each node and d) the pods that deployed within each node. It provides also an API interface and it is invoked to get the statistics and metrics needed for monitoring and visualization purpose.

**Algorithm 1 Rule Engine - Timeout Loop.**

```
1: procedure DECIDE
2: decision, flavor ← scaleUp()
3: if decision = 'up' then
4:   infrastructureManager.addNode(flavor)
5: end if
6: sleep(longTime)
7: else
8:   decision,nodeList ← scaleDown()
9:   if decision = 'down' then
10:      for node in nodeList do
11:         infrastructureManager.deleteNode(node)
12:      end for
13:   end if
14: end if
```

Infrastructure Manager manages the cluster and allows CRUD operations to be executed at infrastructure level like addition, resize or deletion of cluster nodes (VMs). It exposes a set of RESTful Web services which can be invoked from other services. and implements the following operations: a) listing available flavors, b) listing nodes and pods, c) getting details of a specific node, d) resizing and deleting nodes from the cluster, e) adding nodes,
Listing events. The Infrastructure Manager communicates with the cloud provider (i.e. OpenStack) and Kubernetes Scheduler.

Algorithm 2 Rule Engine - Scale Up.

1: procedure SCALEUP
2: decision ← ‘neutral’
3: events ← kubernetesManager.getFailedSchedulingEvents()
4: if events is empty then return decision, null
5: pods ← kubernetesManager.getPods(events)
6: if pods is empty then return decision, null
7: defaultCpu ← 0.1
8: defaultMemory ← 128
9: reqCpu ← 0.02
10: reqMemory ← 0.0
11: for pod in pods do
12: containers ← pod.getContainers()
13: for container in containers do
14: if container requires cpu then
15: reqCpu ← reqCpu + container.getCpu()
16: else
17: reqCpu ← reqCpu + defaultCpu
18: if container requires memory then
19: reqMemory ← reqMemory + container.getMemory()
20: else
21: reqMemory ← reqMemory + defaultMemory
22: flavors ← infrastructureManager.getFlavors()
23: flavor ← findBestFit(flavors, reqCpu, reqMemory)
24: if flavor is empty then return decision, null
25: decision ← ‘up’ return decision, flavor

The Rule Engine implements infrastructure scalability functionality. It is responsible for managing the number and the flavor of the worker nodes (VMs) in use. It receives information about the cluster state from Infrastructure Manager and decides whether or not an infrastructure change (i.e. scale-up or down) is necessary. If a change is mandatory, Rule Engine initiates a sequence of operations for the purpose of adjusting the cluster (addition or deletion of machines). The operation is triggered periodically by a timeout value which is set at run-time. Algorithm 1 illustrates the basic logic that implements the service. Initially, the method checks if a scale up operation is needed by calling the “scaleUp()” method (Algorithm 2). This method returns a decision and a VM flavor. If the decision is “up”, it means that Kubernetes scheduler has failed to deploy a number of pods on the existing cluster nodes (that information is retrieved from Kubernetes Manager). scaleUp() computes also the resources that are needed in order to successfully deploy the needed pods (i.e. by summating the requested computing resources from all pods). Based on this score, the VM flavor that best matches this score is returned. Then, a new node with the specific flavor is
created by invoking the “addNode()” function of Infrastructure Manager; finally the thread sleeps. The time that the thread sleeps depends on the time the cloud (i.e. OpenStack in our case) needs to start a new VM and initialize it.

Algorithm 3 Rule Engine - Scale Down.

1: procedure SCALEDOWN
2:   decision ← ‘neutral’
3:   nodes ← infrastructureManager.getNodes()
4:   for node in nodes do
5:     if node.canBeDeleted() then
6:       if kubernetesManager.isNodeRegistered(node) then
7:         pods ← kubernetesManager.getPods(node)
8:         if pods is empty then
9:           nodesToDelete.add(node)
10:      if nodesToDelete is empty then return decision, null
11:     decision ← ‘down’ return decision, nodesToDelete

If scale-up is not necessary the method checks if scale-down can be applied by invoking “scaleDown()” method (Algorithm 3). The service returns a decision and a “nodeList”. If the decision is “down”, it means that there are nodes in the cluster with no pods deployed on them. Those nodes are added in “nodeList”. By iterating through the “nodeList” the “deleteNode()” function of Infrastructure Manager in invoked to remove these nodes from the cluster. Finally, regardless of whether a scale-down operation can be applied or not, the thread sleeps. The time that the thread sleeps is user defined (i.e. one minute in our case), since node deletion is an instant process and there is no need for the whole process to run more frequently.

4 Experiments

The experimental infrastructure comprises of three VMs deployed in OpenStack, one running Commodore, one running Kubernetes master node and one running a worker node with 1 CPU and 2 GB RAM (small flavor). Therefore, the cluster starts with one worker node. Each node can run up to two pods or two copies of an application (i.e, each pod is scheduled to run on 1/2CPU and 1 GB RAM). We consider an application that computes all prime numbers from 1 through $10^6$ and runs on 1 pod. To run an instance of the application we issue an HTTP request. We measure the response time for an increasing number of requests up to N=500, from which 4 run concurrently. The average response time (over N requests) for an application running on 1 pod is 5,000ms. Response times are obtained using Apache Bench\footnote{10 https://httpd.apache.org/docs/2.4/programs/ab.html}.
Next, we instructed Kubernetes to schedule the application on 2 pods. The response time dropped to 2,600ms. Kubernetes balanced the load between the two pods. Commodore, did not perform any scaling operation (i.e. the scheduler deployed the new pod on the same node successfully, since there is still 1 pod available). Next, Kubernetes scaled-up to 4 pods. Kubernetes had no available resources and emitted 2 fail events. Commodore received the fail events and computed the needed resources. In order to deploy 2 more pods, Commodore started a new small flavor VM. The response time dropped to 1,700ms.

Next, the application scaled-up to 8 pods. Kubernetes could not deploy the 4 new pods and emitted 4 fail events. Commodore received the events, computed the requested resources and started a new medium flavor VM (i.e. 2 CPU cores and 4 GB of RAM) for the new pods. The response time dropped to 1,500ms. It is worth noticing that, response time did not improve as much as before (e.g. from 2 to 4 pods). Lastly, we scaled the application up to 14 pods. Kubernetes could not schedule the pods on the available nodes and emitted fail events. Commodore started two new VMs (one medium and one small flavor VM). The response time is 1,400ms.

The response time scales almost linearly until 4 pods. From this point onward, scaling-up the infrastructure did not yield the same benefits as before (i.e. resources are idle). Figure. 5 shows the response time as a function of the number of running instances of the application. Notice that creation of VMs may take several seconds, however, latest versions of OpenStack reduced the VM creation process to 2-3 seconds. An alternative would be to allow spawning VMs in advance.

Fig. 5 Response time.
5 Conclusions and Future Work

Commodore offers infrastructure scalability features to Kubernetes by allocating or de-allocating resources based on cluster load. Compared to Cluster Autoscaler (CA), Commodore is fully dynamic (i.e. scales-up or down without restrictions). Autoscaling solutions by Google and Amazon, are fully proprietary (Amazon) and are bound to specific platform (Google, Amazon).

Incorporating an elaborate decision-making mechanism for triggering the scaling of resources is an important direction for future research. Commodore collects utilization scores from the cluster nodes. Although the current implementation does not use these scores for autoscaling, implementation of such an autoscaling policy is rather straightforward. We envision a mechanism based on machine learning considering actual computation usage statistics received from running pods and implementing autoscaling strategies such as [4]. This feature would potentially power an additional service for applying pod-level operations such as altering the requested resources of the deployed pods, changing the number of pods based on actual utilization and also, taking decisions on the optimal placement of workload (by moving pods) across nodes.

To better support our claims of efficiency of the solution it is necessary to test Commodore in a more realistic experimental set-up (e.g. a database holding large amounts of real or synthetic data) and stress the application with high user concurrency and high incoming data streams.

References