Efficient Microservices with Elastic Containers

Greg Cusack*
Maziyar Nazari*
gregory.cusack@colorado.edu
maziyar.nazari@colorado.edu
University of Colorado Boulder
Boulder, Colorado

Sepideh Goodarzy
Prerit Oberai
sepideh.goodarzy@colorado.edu
prerit.oberai@colorado.edu
University of Colorado Boulder
Boulder, Colorado

Eric Rozner
Eric Keller
Richard Han
eric.rozner@colorado.edu
eric.keller@colorado.edu
rhan@colorado.edu
University of Colorado Boulder
Boulder, Colorado

ABSTRACT

Containers are a popular mechanism used among application developers when deploying their systems on cloud platforms. Both developers and cloud providers are constantly looking to simplify container management, provisioning, and monitoring. In this paper, we present a container management layer that sits beside a container orchestrator that runs, what we call, Elastic Containers. Each elastic container contains multiple subcontainers that are connected to a centralized Global Cloud Manager (GCM). The GCM gathers subcontainer resource utilization information directly from inside each kernel running the subcontainers. The GCM then tries to efficiently and optimally distribute resources between the application subcontainers residing on a distributed environment.

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

Cloud Computing, Microservices, Compute Resource Efficiency

1 INTRODUCTION

The worldwide cloud computing revenue generation is expected to grow by almost 55% over the next 3.5 years [6]. As a result, cloud service providers and application developers are constantly looking for ways to run, manage, and scale cloud applications as efficiently as possible [4, 5]. Enter containerized applications – also known as microservices. Microservices are applications broken down into their fundamental components (or tasks). Each task is run inside a container (e.g. Docker container) as its own "service" and communicates with other containers over the network. Microservices provide a number of advantages over traditional, monolithic applications including, independent upgrade cycles, fine-grained resource control, and high elasticity.

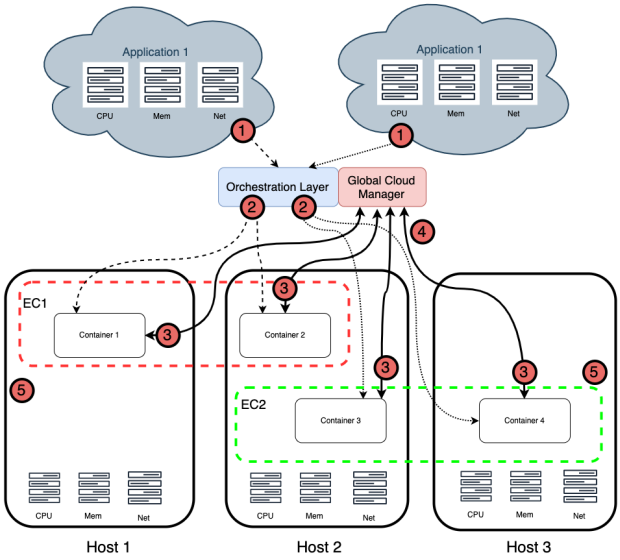

Figure 1: Operation of two Elastic Containers with specified CPU, memory, and network limits. 1) Application is deployed in the cloud. 2) Orchestration layer deploys containers on different hosts. 3) Containers register with Global Cloud Manager (GCM). 4/5) GCM monitors and allocates CPU/memory/network limits.

As microservices grow in popularity, weâĂŹre also seeing the containerization of virtual network functions (VNFs). VNFs have emerged as the de facto deployment method in modern day network applications as VNFs are more cost effective, flexible, easier to scale, and migrate than traditional network functions running on specialized hardware. We're beginning to see the containerization of VNFs, opening the door for container orchestration systems (e.g. Kubernetes [2]) to automate the deployment, scaling, management, and failure handling of network function clusters. In fact, Arpit Joshipura, the general manager for networking at The Linux Foundation, "...expects Kubernetes to evolve into a management platform for managing and deploying [cloud-native network functions]... [8]." The containerization of VNFs and its integration with Kubernetes is promising as the resulting system will provide all the benefits of microservices with the benefits of VNFs.

However, while the combination of Kubernetes and containerized NFVs provide promise for an elastic and efficient cloud network

*Both authors contributed equally to this research.

CoNEXT '19 Companion, December 9–12,2019, Orlando, FL, USA

ACM ISBN 978-1-4503-7006-6/19/12.
https://doi.org/10.1145/3360468.3368180

infrastructure, a few issues arise. First, container management solutions require resource allocation on a per-container level. As a result, each container must be statically allocated CPU, memory, and network resources prior to deployment, resulting in situations of resource over and under allocation. The second issue is the inability for containers within an application to dynamically share resources based on current demand. For example, let's say we have an application with two containers C_A and C_B, each allocated one core. If C_A is under high CPU load and C_B is relatively idle, C_A cannot use some of C_B's available CPU bandwidth even though C_B isn't using it [1]. As a result, the overall application throughput is less than it could be.

In this work we address these two shortcomings of all microservices through the lens of containerized network functions. We introduce Elastic Containers (ECs), an early state system that allows microservices to be deployed such that they can dynamically share resources among containers running within the same application to reduce resource wastage, maximize efficiency, and improve scalability.

2 ELASTIC CONTAINERS

To address both of the issues associated with microservices and Kubernetes deployments, we propose Elastic Containers (ECs). An EC is like any other containerized application or microservice. However, each container (we call these subcontainers in the context of an EC) within the application communicates with a "Global Cloud Manager (GCM)" that has a global view of all subcontainers The GCM keeps track of and distributes compute resources to the EC's subcontainers based on current application demands. Figure 1 outlines the architecture of an EC system.

ECs abstract away per-container resource provisioning and management as seen in traditional microservice applications. ECs allow their subcontainers to consume the resources they require based on current demand up to a global resource threshold set by the application deployer. Instead of enforcing per-container limits on compute resources, we set per-application limits on compute resources. Per-application limits are set using three numbers, the total number of cores, the total amount of memory, and the total network bandwidth an entire application is allowed to use. The subcontainers simply just "run" and request resources as needed from the GCM.

The GCM responds to resource requests from the subcontainers taking into account current CPU, memory, and network loads and the remaining compute resources available to the application in one of three ways. One, the GCM will fulfill the subcontainer request and tell the subcontainer they can use more resources. Two, allow the container to use a subset of the requested resources Three, tell the container that it may not have any more resources. In the third case, all of the per-application resources have been consumed at the current time. However, as time goes on and loads on each individual subcontainer change, subcontainers that couldn't get the requested resources a minute ago may be able to now.

3 IMPLEMENTATION

We attacked the problem of CPU runtime and memory allocation first, with network resource allocation to come next. To implement CPU runtime allocation, we learn from the Linux kernel's completely fair scheduler (CFS) and expand a subset of the CFS to run on a multi-server and multi-container scale[1]. What is important here is that locally, subcontainers consume runtime from the in-host scheduler in chunks. Once they have used up their allocated amount, the container is throttled. Each container's available runtime is refilled on a periodic basis. For ECs, instead of refilling a container's runtime locally, we add a kernel hook such that the subcontainer makes a request to GCM for more runtime whenever it needs it. This allows subcontainers to acquire more runtime when they run out instead of waiting to get refilled by the CFS periodically. By having subcontainers reach out directly to the GCM, subcontainers can acquire more runtime in times of heavy load. Our initial measurements show an average latency increase of 0.3% over the unmodified CFS scheduler each time a subcontainer requests more runtime from the GCM.

When it comes to dynamic memory allocation for an elastic container, we deploy each subcontainer with an initial maximum memory limit that is enforced by the kernel [2]. Generally, if a process asks for memory exceeding the container's limit, the kernel will check if it can reclaim unused pages, if not, the kernel calls the OOM-killer to kill processes exceeding the container's memory limit. We added a hook in the kernel before the OOM-killer call that makes a request to the GCM to see if there is memory available from our per-application memory allocation. If there is memory, the GCM returns a new maximum memory limit to the requesting subcontainer, effectively increasing the amount of memory the subcontainer can use. If there is no memory left, the GCM will attempt to reclaim unused memory from the other subcontainers in the same EC. The GCM sends a request down to an elastic container agent running on each host. The agent acts on behalf of the GCM to reclaim available pages and sends any reclaimed pages back to the GCM. If no memory is reclaimed, the OOM-killer is called on the need container as expected. Preliminary micro-benchmarks show an increased memory allocation latency of 13.8% when requesting memory from the GCM instead of the local machine.

4 CONCLUSION

We have set up an initial system that allows microservices to be deployed and run as members of an EC. Each subcontainer is able to dynamically request resources in an "as needed" fashion from a central manager. When subcontainer loads are not uniform, subcontainers with higher demands are able to use up more of the per-application compute resources than they otherwise would be able to in an typical microservice environment. We expect ECs will enable a higher resource efficiency when container loads are mismatched within a containerized application.

5 ACKNOWLEDGEMENTS

This work was supported by the NSF and VMware grant 1700527 (SDI-CSCS).

[1] We refer readers to [7] for details on the Linux kernel scheduler

[2] We refer readers to [3] for details on the Linux kernel's memory management system

REFERENCES

[1] 2004. CGROUPS. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt. (2004).

[2] 2014. Kubernetes. https://kubernetes.io/. (2014).

[3] 2019. Memory Management. https://www.kernel.org/doc/html/latest/admin-guide/mm/index.html. (2019).

[4] Marcelo Abranches, Sepideh Goodarzy, Maziyar Nazari, Shivakant Mishra, and Eric Keller. 2019. Shimmy: Shared Memory Channels for High Performance Inter-Container Communication. In *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*.

[5] Zaid Al-Ali, Sepideh Goodarzy, Ethan Hunter, Sangtae Ha, Richard Han, Eric Keller, and Eric Rozner. 2018. Making Serverless Computing More Serverless. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 456–459.

[6] Katie Costello. Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.5 Percent in 2019. :www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g. (????).

[7] Paul Turner, Bharata B Rao, and Nikhil Rao. 2010. CPU bandwidth control for CFS. In *Proceedings of the Linux Symposium*. 245–254. http://www.linuxsymposium.org/LS_2010_Proceedings_Draft.pdf

[8] Mike Vizard. 2018. The Linux Foundation to Drive Shift to Container Network Functions. https://containerjournal.com/topics/container-networking/the-linux-foundation-to-drive-shift-to-container-network-functions/. (2018).