YACR : Yet Another Container Runtime

Youhan Wu, Zhaocheng Huang
Rice University

1 Introduction

Container-based virtualization is really popular in recent years. Thanks to the advent of Docker, container technology has subverted the traditional systems development life cycle and refactored the way people build and deploy software. While Docker is way more lightweight than other virtualization technologies, it is still not efficient enough in a high load environment, as well as in a resource limited platform such as IoT devices [1].

Docker introduces an underlying container engine to help start, terminate processes and multiplex system resources, which utilizes Linux namespaces [2], as well as a lot of system ABIs, such as read/write, exec, fork, etc.. However, many existing container runtimes like runc, gVisor are written in Go, which introduces overhead to Linux system call and resources utilization [3] [4]. From this perspective of view, Go might not be the best programming language for this task. It doesn't have a complete support for the fork/exec model of computing. Besides, the container sill suffer from a performance cost due to language runtime. There do exist some alternative container runtime written in system languages with relatively low overhead. In 2017, Red Hat introduced crun to replace runc. However, C is not recognized as a memory safe language, it can expose too many vulnerabilities due to its inherent defect. One simple memory-related vulnerability may cause huge economic losses to container users.

We want to implement an OCI-compliant runtime in Rust to explore the potential of being faster and more memory-saving than other existing container runtimes. Our goal is to demonstrate that such a runtime can be more efficient and safer in general cases, and has an edge in resource limited environments, like single-board computers.

Hopefully, if everything is successful, we will perform a benchmark test against runc, crun and gvisor. More specifically, to compare them from performance, memory utilization and security aspects.

The anticipated output includes a runnable OCI-compliant runtime, a set of benchmark testing programs, and a final report.

2 Background

Docker is one of the most popular technologies for containerization. Instead of boosting the whole system environment in VMs, Docker utilizes the Linux kernel's resource isolation and constraint features to share a kernel across containers. Docker can create isolated environment for containers in different levels (e.g., network, PIDs, UIDs, IPC) with support of Linux namespaces [5]. Besides, it can control resource allocation and limitation among containers using cgroups.

Rust is a programming language that emphasizes performance and memory safety. Rust aims to be as efficient as C++. It doesn't perform garbage collection, and it has features like zero-cost abstractions and inline expansion to optimize the compiled code. Rust is designed to be memory safe. It does not permit null pointers, dangling pointers, or data races, and safety and validity of the underlying pointers is checked at compile time. Rust has been growing fast in recent years, and has been widely used in many large projects. It is reported that Rust is incorporated into the forthcoming 6.1 Linux kernel, which is an important milestone for Rust, and also a good signal for our project.

We believe with official support from system level, and the brilliant features from the language level, a new

container runtime written in Rust will be a better alternative to existing runtimes.

3 Methods and Plans

In order to build a OCI-compliant runtime, we need to study the Runtime Specification first to see what a minimal runtime should include. So that we can swap the Docker's runtime from runc to yacr, and then furthermore perform a benchmark testing against other runtime implementations.

Apart from the specs, we would dive into the Docker's source code, to see how exactly a container is created and started, and how the system resources are isolated and allocated. We will try to port the core components to our Rust implementation.

If everything can be delivered on time, we will perform a benchmark testing over the some of popular implementations and our yarc version. The expected evaluation criteria includes performance (e.g. wall time for container startup), resource utilization (e.g. memory usage, cpu usage), and safety [6] (e.g. fuzzing, perform an attack exploiting CVE samples).

4 Milestones

10/13 Project Proposal

10/16 Study OCI runtime-spec

10/23 Study Docker Architecture

10/30 Warm up Rust lang

11/01 Project Midterm Report

11/06 Port part of core components

11/08 Project Midterm Meetings

11/13 Port the rest of components

11/20 Write benchmark testing programs

12/01 Project Presentations

12/03 Project Final Report

References

[1] Antonio Celesti, Davide Mulfari, Maria Fazio, Massimo Villari, and Antonio Puliafito. Exploring container virtualization in iot clouds. In *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–6, 2016.

[2] Karl Matthias and Sean P Kane. *Docker: Up & Running: Shipping Reliable Containers in Production*. " O'Reilly Media, Inc.", 2015.

[3] Ethan G Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The true cost of containing: A {gVisor} case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[4] Emiliano Casalicchio and Vanessa Perciballi. Measuring docker performance: What a mess!!! In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 11–16, 2017.

[5] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[6] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. Container security: Issues, challenges, and the road ahead. *IEEE Access*, 7:52976–52996, 2019.