Chameleon Cloud Tutorial

National Science Foundation

Program Solicitation # NSF 13-602

CISE Research Infrastructure: Mid-Scale Infrastructure - NSFCloud (CRI: NSFCloud)

Docker - CGroups and Namespaces

Objectives

This document is meant to be used as an informative means to demonstrate what kernel features Docker is taking advantage of to offer an overall better and more efficient administration and security amongst its containers.

Introduction

Docker, being one of the leaders in the container-based world, often takes advantage of several features belonging to the Linux kernel as a means to better its service. In particular, Docker's use of control groups (cgroups) and namespaces and how each play a role in resource management and security cannot be overlooked. In order to understand what Docker provides through these features, one must first understand what they accomplish individually.

Namespaces

To start off with, namespaces are a very convenient feature of Linux that serves to provide a somewhat virtualized resource that acts as a go between for processes within the group and the actual resources. By sectioning off these processes into their own groups, they are unable to see the resources being used by separate groups while still remaining visible to those processes within the same group. Using this allows users to segment things such as: Networking, PIDs, Users, Mounts, and more. Essentially, the process will have no knowledge of processes outside of its group and would have no need to know about them, providing an isolated bundle of processes.

When applying this knowledge towards the implementation with Docker, the advantages then become clear in the form of containers and linked-containers being permitted to run on the same host, but oblivious about any other running containers. Docker, by default, will separate a newly created container into its own namespace to distance containers from one another. This is done across most of the namespaces listed before and allows the container to act as a solo application on the host.

Like many other Linux features, namespaces establish themselves using a virtual filesystem in the fashion of **/proc/$PID/** (where $PID is the pid of the process). Within this directory, Linux already displays intimate information about processes, but with namespaces, there is

additional information available. Primarily through the **ns** directory (namespaces) which holds file handles pointing to the corresponding namespace. Additionally, within other directories such as **mounts** and **net**, even more information about the current namespaces are available. Given that one of the core goals of a container was to replace the need for virtual machines, namespaces provide an opportunity to bridge the gap through process isolation. Thus giving the appearance of each as having their own virtual machine.

This usage of namespaces plays hand-in-hand with cgroups and their many uses.

CGroups

When speaking about cgroups, the first idea that should come to mind is of resource management. Because where cgroups excels is in an area similar to namespaces, by separating and grouping processes into whatever organizations the user prefers and inflicting global (group-wide) rules upon them. Cgroups provide a kernel-level tool for resource management and accounting that is greatly relied upon by Docker and many other modern tools. Beginning with the organization, processes are organized into cgroups based first and foremost off of inheritance. That is, child processes will inerhit their initial group based off the parent group they belong to.

NOTE: Only certain properties are inherited from parent cgroups that can also be modified upon child process creation.

Cgroups are used to divide up the resources of a system into separate catagories that can be described in terms of limits and shares among the host resources. The first and most prevalent example is the sharing of the CPU amongst different processes. Initially, one might think to allocate a certain number of cores to system processes and leave the rest for userspace processes (assuming the system contains enough cores). From there, the user processes may spawn a webserver which would be placed in its own cgroup along with its children so that they only use a certain percentage of the CPU time, RAM, I/O, Network, etc. These limitations and rules are used to define the resource behavior of programs within their cgroups so that each has their “fair share”.

Much like with CPU scheduling, cgroups are often used to evenly distribute resources amongst its processes, but the difference is that it does not need to be equal. For example, when running unfamiliar applications, one thought might be to run all user-level programs in their own cgroup so as to limit their usage of the system resources to a certain extent such that they won’t be able to monopolize and steal away these resources from the system. Whether the exhaustion of a resource is intentional or accidental, cgroups provide a safety barrier (similar to that of saving 5% of the hard disk for root) that can be used for the safety of the host as a whole.

In terms of how Docker uses cgroups is that Docker will place all the containers, unless told otherwise, into their own cgroup that are each set to equally share the system resources. However, by passing in certain arguments, one can raise the priority of certain containers, lower the I/O permitted by others, and all around customize to the best efforts of the system. Additionally, since cgroups occur at the system level, none of the applications will ever need to be aware that they are being throttled or administrated over in any way, again providing the illusion of each container belonging on its own host.

Similar to the namespaces, cgroups manifest themselves in the virtual filesystem. Depending on your installation, they are often found at **/sys/fs/cgroup/** or **/cgroup/**. From this root directory, you can see each subsystem has its own directory within which you can create cgroups within. Aside from that, you are also capable of mounting a cgroup and assigning the subsystems desired to be nested within using the **mount** command. Cgroups subscribe to a hierarchical structure in order to create nested groups of related tasks such that related

processes can be stored together. This can be extended to running all tasks related to the webserver within one cgroup, internally, creating different cgroups for the database, backend, etc.