YARN-based Resource Management for Provisioning Scalable Container Cluster

Chia-Chuan Chuang*.

*National Center for High-performance Computing, No.22, Keyuan Rd., Taichung City, Taiwan (R.O.C.)

*Corresponding Author: jimmy_chuang@narlabs.org.tw

Abstract

Container technology is a modern technology for virtualization, and a container cluster is a virtual cluster that is composed of multiple containers launched from distributed servers. Launching containers from physical clusters in a resource-efficient manner is a major challenge, especially if the underlying hardware infrastructure where virtual clusters reside must be shared by multiple systems simultaneously. A global resource manager that can negotiate with various types of systems is required. YARN provides a unified resource management and scheduling mechanism; currently, various distributed computing technologies can be deployed on a single YARN cluster to share hardware infrastructure. In this paper, a provisioning platform for a YARN-based container cluster environment is proposed. With the resource management mechanism provided by YARN, existing computing technologies and container cluster provisioning can be combined on the same infrastructure.

Keywords: Resource Management, Container Cluster, YARN, Docker.

1. Introduction

Resolving software dependencies in a cluster has always been a difficult task for service providers, but this difficulty can be mitigated by building virtual clusters of virtual machines. The entire required software stack can be packaged into an image, and virtual machines can be launched on demand. Each virtual machine instance has its own hardware stack, including a virtualized BIOS, virtualized network adapters, and virtual CPU, as well as its own software stack from operating system to user applications. Container technology is another virtualization technology. It shares the host’s OS library and uses fewer resources than virtual machines. The difference between virtual machines and containers is shown in Fig. 1. A virtual machine is much more resource-intensive and takes minutes to boot up, but a container can provide services a few seconds after it is started.

To implement a lightweight, isolated microservice architecture [1], containers have been promoted. A single microservice application can be developed as a suite of small services, each of which runs in its own container. One example of a containerized microservice application is a container cluster; each container in the cluster separately packages specific software into a service required by the user. These containers are launched from one or more
physical hosts to form a distributed cluster of servers. If the underlying physical hosts are shared by multiple systems simultaneously, a global resource manager is required to schedule hardware resources. In this paper, a YARN-based container cluster provisioning platform is developed to enable container clusters to coexist with other distributed computing frameworks.

The remainder of this paper is as follows. Background information about container cluster management is described in Section 2; the proposed management platform is discussed in Section 3. System implementation is detailed in Section 4. In Section 5, conclusions are presented.

2. Background

2.1 Docker

Linux provides cgroup, namespace, and chroot to facilitate container virtualization. Because users find these mechanisms to be too complicated, these mechanisms are not widely used. Docker [2] is an implementation of container virtualization technology that utilizes Linux LXC and a multilayered file system. The Docker engine contains a set of command line tools for simplifying the cumbersome management of container technology. After the containers have been configured, required software dependencies are packaged into Docker images that can be distributed and shared. Service providers can use a single image to launch numerous containers that deliver identical functionality in seconds. This reduces the time cost to duplicated development efforts.

2.2 Cluster Managers

Microservice providers recommend that developers deliver applications by launching multiple components in containers. Usually, containers can be run on different physical hosts, and the Docker ecosystem must provide certain tools for container orchestration.

(a) Docker Swarm

Docker Swarm [3] is a master–slave cluster manager for Docker. A Swarm manager coordinates a set of physical hosts that run the Docker daemon and the daemon in each node communicates with the manager. The Swarm manager receives messages in the standard Docker API and forwards messages in that API to clients, according to a scheduling policy. Because Swarm uses the standard Docker API, any tool that functions with Docker can access Docker Swarm transparently to scale up to multiple hosts.

(b) Kubernetes

Kubernetes [4] is another system for managing scalable, containerized applications among multiple hosts. For administration purposes, Kubernetes defines pods (i.e., groups of containers that provide a service) as the smallest units that can be created, scheduled, and managed. Kubernetes is a master–slave architecture; the Kubernetes master deploys all containers of a single pod to run on the same Kubernetes slave node.

2.3 General Purpose Resource Manager

The Docker cluster managers introduced in Section 2.2 are specially designed for Docker containers, and have no inherent ability to cooperate with other systems. To coordinate resources among multiple systems, a global manager is required. Mesos [5] and YARN [6] both use a two-level scheduler model [7], and provide general resource management to support multiple systems. Container provisioning can be run on top of both frameworks.

(a) Marathon

Marathon [8] is a container orchestration system that deploys and manages containers on top of Mesos. Marathon makes a resource request from the Mesos master and then waits to accept an appropriate offer. Once an offer is accepted, the Mesos master sends the task’s specification to the slave, and the Mesos slave daemon can then launch a container through the Docker API.

(b) Docker on YARN

The big data engine Hadoop [9] uses the YARN architecture to manage resources; numerous distributing computing engines, such as Spark [10] and Storm [11], can be deployed on a single YARN cluster to share the hardware infrastructure simultaneously, which improves resource utilization. Thus, managing container provisioning on top of YARN is an effective solution.

3. System Design

Although factors such as communication, container naming, and resource management may be simple for containerized applications deployed on a single host, when those containerized applications are scaled among multiple hosts, all of those factors become much more complicated. These issues are addressed in the following subsections.

3.1 Container Cluster Communication
Docker can provide native network isolation; in such isolated networks, containers on different hosts cannot communicate. A possible solution is connecting containers to a layer 2 network on which physical hosts reside. A customized bridge (i.e., br0) is deployed and the Docker engine is configured to use br0 instead of natively isolated networks, as shown in Fig. 2. Docker binds a physical Ethernet interface to br0, which enables containers to connect to the host network directly.

IP conflicts may occur if all containers are in the same layer 2 network. To avoid IP conflicts, each configuration is required to distinguish the feasible IP range of each Docker engine. Fig. 2 shows two subnets, 192.168.33.128/27 and 192.168.33.160/27, which are allowed on their respective hosts. In that scenario, if there is an excessive number of containers on this layer 2 network, a broadcast storm results, which degrades performance. Advanced tools, such as VLANs, MACVLANs, and GRE tunnels, can be adopted to overcome this issue.

3.2 Container Cluster Naming Resolution

When a container is launched, the Docker engine assigns its IP dynamically. However, the IP of each container must be maintained on all containers in advance when organizing a container cluster. If tens or hundreds of containers are initiated, extra effort is required by maintaining host lists on all containers. Consul [12] is adopted for efficient container naming, and is depicted in Fig. 3.

Consul is a service discovery mechanism; it provides a DNS server. A registrar on each host monitors container creation and destruction events in the Docker engine, and sends corresponding reports to Consul. If container creation events are detected, the registrar registers container information with Consul automatically and a DNS record is added into the Consul DNS server. Similarly, when a container is destroyed, that container’s information is deregistered and its DNS entry is removed. Hence, containers can use Consul DNS for naming resolution rather than maintaining a list of hosts.

3.3 Container Cluster Scaling

The benefit from the system designs described in Sections 3.1 and 3.2 is that containers in multiple hosts can engage in discovery and communicate with each other. The dynamic arrangement of containerized applications running on various hosts requires resource allocation. The YARN architecture, a general-purpose resource manager implementing a two-level scheduling model, is illustrated in Fig. 4. To enable a system to run on top of YARN, two specific modules, a YARN client and a YARN application master (AM), must be developed for that system. As shown in Fig. 4, Docker and Spark each have a custom client and AM, and can run on YARN simultaneously.

The Docker client for YARN first specifies the container number, application type, application arguments, and other information to submit a containerized application. The Docker AM is then initiated on a host, and is responsible for resource requests from that containerized application as long as the system runs. The YARN scheduler residing in the resource manager responds to an AM’s resource request by granting the amount of resources that satisfy the requirements of the AM’s request.
obtaining the resources, the Docker AM can ask the node manager to launch various containers. The YARN client initiates only the Docker AM. The YARN client cannot manage containers directly; it must delegate resource requests and container initiation requests to the AM. The YARN client can communicate with the Docker AM by using an RPC; the user can pass commands to the Docker AM by using the RPC through the YARN client.

4. Implementation

In this section, screenshots illustrate a successful implementation of container naming, communication, and scaling. Figs. 5 and 6 show resource management and service scaling using YARN. As shown in Fig. 5, two types of systems, one MapReduce application and two Docker on YARN (DORNE) applications, can run on top of a YARN cluster simultaneously. Fig. 6 illustrates that each DORNE application required three containers, and a total of six containers were distributed on two nodemanagers evenly. Note that only three containers were required, but four containers were running. The reason is that YARN initiates one additional container for the Docker AM. Fig. 7 shows DNS records generated during container creation. These records change frequently when containers are often created and destroyed. Fig. 8 shows that containers can ping each other using Consul DNS for container naming resolution. Host files are deprecated when running a multiple host containerized application; Consul delivers a scalable naming solution even in large container clusters.

5. Conclusions

In this paper, through the adoption of the YARN, a platform for provisioning scalable virtual clusters is proposed. It can be used to build a specific customized virtual cluster application. For example, various high performance computing science applications run in parallel. These applications can be built as containerized applications and run in containers [13]. The proposed YARN-powered system creates a specific type of virtual cluster on demand that can coexist with existing big data service platforms, thus reducing hardware construction costs and increasing infrastructure utilization.

References

(1) Sam Newman: “Building Microservices”, 2015
(2) Docker: https://www.docker.com/
(3) Docker Swarm: https://docs.docker.com/swarm/
(4) Kubernetes: http://kubernetes.io/


(8) Marathon: https://mesosphere.github.io/marathon/


(11) Apache Storm: http://storm.apache.org/

(12) Consul: https://www.consul.io/