Introduction to Containers and Container Networking

Frank Brockners
Questions?  
Use Cisco Spark to communicate with the speaker after the session

How
1. Find this session in the Cisco Live Mobile App
2. Click “Join the Discussion”
3. Install Spark or go directly to the space
4. Enter messages/questions in the space

cs.co/ciscollivebot#BRKSDN-2115
Abstract

Containers are becoming part of mainstream DevOps architectures and cloud deployments. Application owners and data center infrastructure teams are both aiming to shorten development life cycle and reduce operational cost and complexity by deploying containers. This session will provide an overview of container ecosystems and container tools including Docker. Join us and learn about the options to network containers with Docker and Kubernetes. Contiv, Cisco's open source container networking and policy project will be highlighted in this session. Finally, the audience will also learn the advantages that Cisco platforms provide in building a cloud platform for containers.
April, 26 1956 : SS IdealX leaves New York
Containers Spawned A New Eco-System

Today, 90+% of all cargo ships in a standard container

Faster delivery, reduced (dis-)charging times, increased security (less loss/damage of goods)

Average shipment cost reduced from >25% to <3%: Key enabler to globalization and just-in-time production
March 18, 1982: chroot() introduced to BSD
March 2000: FreeBSD Jails introduced
February 2004: Solaris Zones introduced
August 6, 2008: Initial release of LXC
March 13, 2013: Initial release of Docker
December 2014: Initial release of rkt

rkt
June, 2015: Open Container Initiative launched
June, 2015: Cloud Native Computing Foundation
Cloud Native Computing Foundation

“The Foundation’s mission is to create and drive the adoption of a new computing paradigm that is optimized for modern distributed systems environments capable of scaling to tens of thousands of self healing multi-tenant nodes.”
Linux Containers

• A Linux container lets you run a Linux system within another Linux system.

• A container is a group of processes on a Linux machine.

• Those processes form an isolated environment.

• Inside the container, it (almost) looks like a VM.

• Outside the container, it looks like normal processes running on the machine.

• It looks like a VM, but it is more efficient: Containers = Lightweight Virtualization
Containers and Virtual Machines

Containers are isolated but share OS and where appropriate bins/libraries

VM

Hypervisor (Type 2)

Guest OS

Host OS

Server

App A

Bins/Libs

Guest OS

Host OS

Server

App A’

Bins/Libs

Guest OS

Host OS

Server

App B

Bins/Libs

Guest OS

Host OS

Server

App A

App A’

App B

App B’

App C’

App C’
Containers are almost like Virtual Machines

- Containers have their own network interface (and IP address)
  - Can be bridged, routed... just like with Xen, KVM etc.

- Containers have their own filesystem
  - For example a Debian host can run Fedora container (and vice-versa)

- Security: Containers are isolated from each other
  - Two containers can't harm (or even see) each other

- Resource Control: Containers are isolated and can have dedicated resources
  - Soft & hard quotas for RAM, CPU, I/O...

Though...

- Apps in Containers share the kernel of the host OS (i.e. Linux guests only)
- Containers are light-weight, fast to start, allow for >10x density compared to VMs
Example - Density

• Simulating “The Internet”

• LXC container “the-internet” within which about 250 sub-containers are running, acting as BGP or OSPF routers.

• …on your Laptop!

https://github.com/nsec/the-internet
Example – Speed
Let’s start 10 simple containers…

```bash
#!/bin/bash
SECONDS=0
echo `date`
SCRIPT="while true; do echo Hello CiscoLive; sleep 1; done"
for i in `seq 1 10`
do
eval "docker run -d --name CiscoLive$i ubuntu /bin/sh -c "$SCRIPT""
done
echo `date`
echo "Elapsed time: $SECONDS seconds"
```

frank@vHennes:/tmp$
10 simple containers started.. in 4 seconds
Containers: Classes of Use-Cases

- Web-Scale Micro-Services
- Network Function Virtualization
- Extend Network Elements’ Functionality
Web-scale Mirco-Service Deployments Spurred Interest In Container Technology Again…

- Micro-Service design principles: 12factor.net

- Continuous Integration/Continuous Delivery:
  - Go from developer’s laptop, through automated test, to production, and through scaling without modification

- Alternative form of virtualization for multi-tenant services

- Scale-out:
  - Rapidly scale same application across hundreds or thousands of servers…and scale down as rapidly

- Cross Cloud Deployment
  - Move the same application across multiple clouds (public, private, or hybrid) without modification or noticeable delay
Containers for NFV

• Motivation: Containerize Virtual Network Functions
  • Light-weight approach to NFV
  • De-compose network functions into a set of containerized “component VNFs”:
    Compose use-case specific, optimized VNFs
  • Leverage alternate orchestration & lifecycle-management solutions

• Considerations
  • Most existing VNFs have specific kernel dependencies/requirements
  • Component-VNF concept still evolving – optimization vs. isolation vs.
    orchestration/management tradeoffs
  • Security/Isolation
  • Container & lifecycle management solutions mostly focused on micro-service
    deployments, less so on NFV/networking applications
  • NFV focused solution might also require a novel approach to current orchestration, incl. OSS/BSS
Motivation: Containers on Network Elements

- Network OS Independence
  - System Modularity
  - Limit kernel dependencies

- Enable multiple constituents to provide for device enhancements
  - Cisco Development, Cisco Services, Partners, Customers, ...

- Leverage existing tools/tool-chains and eco-systems
  - Integration of network device with server-centric tool chains

- Isolation / Security
  - Application-focused deployment, as opposed to base-OS focused deployment
    (“escape dependency hell”)
Qualities Of Containers

• Dedicated namespaces for containers
  • Hostname, Process, IPC, FileSystem, Network, User (subsets are possible)

• Resource Isolation between container and kernel (cgroups)

• Shared Kernel

• Option: Copy-on-write filesystem to allow for binary reuse between containers
Container Overview
Namespaces: Isolate System Resources

- Partition essential kernel structures to create virtual environments. E.g., you can have multiple processes with PID 42, in different environments; E.g., you can have multiple accounts called “frank” in different environments.

- Different kinds of namespaces
  - pid (processes)
  - mnt (mount points, filesystems)
  - net (network interfaces, routing...)
  - user (UIDs)
  - ipc (System V IPC)
  - uts (hostname)

- Namespace creation via the “clone()” system call with extra flags
  - new processes inherit the parent’s namespace
  - you can attach to an existing namespace (kernel >= 3.8)
PID Namespace

- Processes in a PID namespace don't see processes of the whole system
- Each pid namespace has a PID #1
- pid namespaces are actually nested
- A given process can have multiple PIDs
  - One in each namespace it belongs to
  - So you can easily access processes of children namespace
- Can't see/affect processes in parent/sibling namespace
PID Namespace
PID namespace isolates the Process ID, implemented as a hierarchy.

**PID namespace1 (Parent)**
(Level 0)

- pid:1
  - P1
  - ls /proc 1

**PID namespace2 (Child)**
(Level 1)

- pid:2
  - P2
  - pid:1
  - P1

**PID namespace3 (Child)**
(Level 1)

- pid:3
  - P3
  - pid:1
  - P1

- pid:4
  - P4
  - ls /proc 1 2 3 4

ls /proc
Mount Namespaces

Each mount namespace has its own file-system layout

- Mount namespaces are a “deluxe chroot()”
- A mount namespace can have its own rootfs
- Remount special filesystems, e.g. procfs (to see your processes)
Net Namespace
The Net namespace isolates the networking related resources

- Each net namespace has its own...
  - Network interfaces (and its own lo/127.0.0.1)
  - IP address(es)
  - routing table(s)
  - iptables rules
- Communication between containers:
  - UNIX domain sockets (=on the filesystem)
  - Pairs of veth interfaces

Net Namespace 1
- Net devices: eth0
- IP address: 1.1.1.1/24
- Route
- Firewall rule
- Sockets
- Proc
- sysfs
  ...

Net Namespace 2
- Net devices: eth1
- IP address: 2.2.2.2/24
- Route
- Firewall rule
- Sockets
- Proc
- sysfs
  ...

BRKSDN-2115 © 2018 Cisco and/or its affiliates. All rights reserved. Cisco Public
User Namespace

- Kernel-uid (kuid)/Kernel-Group-id (kgid): Original uid/gid, Global
- uid/gid: user id in user namespace, will be translated to kuid/kgid finally
- Only parent User namespace has rights to set map
IPC Namespace

IPC namespace isolates the inter-process communication resource (shared memory, semaphore, message queue)
UTS Namespace
Every UTS namespace has its own UTS related information

Unalterable
- ostype: Linux
- osrelease: 3.8.6
- version: ...

Alterable
- hostname: cisco1
- domainname: live1
- UTS namespace1

ostype: Linux
osrelease: 3.8.6
version: ...

hostname: cisco2
domainname: live2
UTS namespace2
Control Groups (cgroups)
Generic process grouping framework (kernel >= 2.6.24)

• Control the resources in a system with many tasks – Linux itself, VMs, containers…

• Subsystems
  • Group CPU Scheduler
  • CPU Accounting Controller
  • Cpuset
  • Memory
  • Block IO Controller
  • Device Whitelist Controller
  • Freezer
  • Namespace
Manage Containers

• Requirement: Userspace container management tool set
  • Manage containers
  • Create namespace
  • Create private filesystem layout for container
  • Resources controller by cgroup

• Solution options
  • liblxc
  • libvirt lxc
  • libcontainer/Docker
Container Management – Workflow
Example: Docker

Source Repository
- Dockerfile
  - Bin/Libs

App A
Build
Push
Base Libs Update
Build Host
- Docker Engine

Docker Image Registry
- App A
  - Bin/Libs
- App B
- App C
  - Bin/Libs
  - Docker Engine
Target Host
- Docker Engine
Containers as a Packaging Mechanism: Build once – Deploy Anywhere

• Containers combine content (the “RPM”) with context (the environment the RPM was built for)

• Containers resolve RPM dependency management shortcomings
  • How to resolve situations where different RPMs require different versions of a dependency package?
  • RPM built with a different toolchain does not guarantee ABI compatibility
Container World Taxonomy

• **Container Tools**
  • Docker, Rkt, repos registries
  • micro-OSs – CoreOS, RHEL Atomic, Ubuntu Snappy, ..

• **Cluster Control and Services**
  • Scheduler Job Monitor – Marathon, Aurora
  • Resource Managers – Mesos, Kubernetes, Docker Swarms
  • Distributed Key Value lock managers – zookeeper, etcd, consul

• **Service Orchestration Management**
  • Kubernetes, Mesosphere DCOS, Docker Swarm, HashiCorp Terraform, CoreOS Tectonic
Container Technology
A glimpse at Docker
Containers on Cisco devices
Container Orchestration and Networking
Cloud Native Container Networking
What is Docker?

- Docker is a container technology similar to Linux Containers (LXC) that…
  - Provides isolation for application processes from the host processes using Linux **namespaces**
  - Provides resource caps for the application using Linux **cgroups**
  - Provides industry preferred **packaging** model using docker images, docker index, and docker registry concepts
  - Provides the basis for application **lifecycle management** automation due to good integration with devops automation tools such as Puppet/Chef

- A rich repository of certified docker base images are easily available in public as well as private docker registries to cover a variety of application use cases
When Docker over LXC?

Docker provides the following advantages over LXC containers

- Docker packaging is becoming a **packaging model of choice** for application packaging and delivery in the industry.
- Standard packaging format provided by docker allows **easier integration with devops automation toolchains** such as puppet/chef/etc.
- Docker cli provides a **git like workflow** for developing containerized applications which is tool-friendly
- Docker provides **image management capabilities** using docker image, docker index and docker registry concepts
- Public and private docker registries provide a **repository of starter container images** for a variety of application use cases
- **Thin packaging/delivery** (Docker application download or upgrade is often a thin download due to sharing of common layers among multiple containers)
- **Lower footprint** for the containers. Docker layers, especially, the base layers can be shared among multiple containers
Installing Docker

- Docker can be easily installed on a wide variety of platforms (Ubuntu, Mac OS X, RHEL, CentOS and many more).

- Detailed instructions are here: https://docs.docker.com/engine/installation/
Let’s pull an image from Docker Hub…

Images consist of multiple layers…
Let’s pull an image from Docker Hub…

frank@vHennes:~$ sudo docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu

b3e1c725a85f: Pull complete
4daad8bdde31: Pull complete
63fe8c0068a8: Pull complete
4a70713c436f: Pull complete
bd842a2105a8: Pull complete
Digest: sha256:7a64bc9c8843b0a8c8b8a7e4715b7615e4e1b0d8ca3c7e7a76ec8250899c397a
Status: Downloaded newer image for ubuntu:latest
frank@vHennes:~$ sudo docker images

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>IMAGE ID</th>
<th>CREATED</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ubuntu</td>
<td>latest</td>
<td>104bec311bcd</td>
<td>4 weeks ago</td>
<td>129 MB</td>
</tr>
</tbody>
</table>

frank@vHennes:~$
A Docker image is made up of filesystems **layered** over each other.
The storage driver is responsible for presenting these layers as a single, unified file system.
When you start a container, Docker creates an empty, read-write layer on top of the stack – all changes are made in this layer.
Docker uses “**copy-on-write**” container layers.

If a file needs to be modified, it is copied into the read-write layer first.
These layers are read-only and never modified!
This means that multiple containers can share a single copy of the image.
Now that we have an image, what do we do with it?
Let’s use the image:
Start a container and run the bash shell.

/bin/bash
<table>
<thead>
<tr>
<th>Interactive</th>
<th>Terminal</th>
<th>Name of the container</th>
<th>Image</th>
<th>Command to run</th>
</tr>
</thead>
</table>

**Interactive Terminal**: Allows users to interact with the container's command line interface.

**Name of the container**: Specifies the unique identifier for the container.

**Image**: Indicates the base image from which the container is built.

**Command to run**: The specific command used to run the container.

```
frank@vHennes:$ sudo docker run -it --name CiscoLive ubuntu /bin/bash
```

```
frank@vHennes:$ sudo docker ps
```

```
CONTAINER ID   IMAGE     COMMAND                  CREATED             STATUS          PORTS                               NAMES
ca13282430da   ubuntu   "/bin/bash" 50 seconds ago Up 49 seconds
```

```
root@ca13282430da:/# ps -ef
```

```
UID  PID  PPID C STIME TTY      TIME CMD
root  1   0   0 14:24 ? 00:00:00 /bin/bash
root 11  1   0 14:24 ? 00:00:00 ps -ef
```
Docker keeps track of what you touch...

```
frank@vHennes:~$ sudo docker run -it --name CiscoLive ubuntu bash
root@d470be70c5d9:/# touch /tmp/CiscoLive
root@d470be70c5d9:/# rm /etc/legal
root@d470be70c5d9:/#
```

```
frank@vHennes:~$ sudo docker ps
CONTAINER ID   IMAGE              COMMAND                  CREATED               STATUS              PORTS                  NAMES
D470be70c5d9   ubuntu            /bin/sh                  43 seconds ago        Up 43 seconds        0.0.0.0:2222->22/tcp   CiscoLive
frank@vHennes:~$ sudo docker diff CiscoLive
C /etc
D /etc/legal
C /tmp
A /tmp/CiscoLive
frank@vHennes:~$
```
Let’s start 10 simple containers…

```
frank@vHennes:~/tmp$ cat 10containers.sh
#!/bin/bash
SECONDS=0
echo `date`
SCRIPT="while true; do echo Hello CiscoLive; sleep 1; done"
for i in `seq 1 10`
do
eval "docker run -d --name CiscoLive$i ubuntu /bin/sh -c "$SCRIPT"
" done
echo `date`
echo "Elapsed time: $SECONDS seconds"
frank@vHennes:~/tmp$
```
10 simple containers started.. in 4 seconds

Frank@Ubuntu:~ /tmp$ sudo ./10containers.sh
Sun Jun 25 10:32:18 PDT 2017
64a0f95d19935f086eddf43cb2a74c0dff195fc0300b814c98193333e9cc3122
1d30e4a7f23813068721bff831468b7e15198230ded40b5b2fec8d33bc34a82f
068c034db5271249f14ce0427bbc3910b6a00ff6daf8fb4991012da1dee3f22c
1753e4ea73770933bacad423c4f856e69f66237f450c0d40c6f7611ff9e1e6010
400773e820c65452a8b3220a39d01b773353aaeac5d61969e58cb9baf565c939
a4ac99fbeb98c236d6d184760a3f165e5ab8d8b3e0f768a39ea2a6ddf26dcb34e
24dc000991f2e2b11083f85leeaf762466738a0512f3f62380278639d9eb2cc4c
178a60a78c9ea840b940d1b0376c3466a25aebf109721ad4601b6dea1d1a60e
6b97bd456a9a81094d126bdc86ad3351a5810696a0f0bc7f41b84ce70fe9b30711
b4abf7e1a1d81d52215c10764f8b710eb5b373652a92eb5a4d27ef9b412a3b46
Sun Jun 25 10:32:22 PDT 2017
Elapsed time: 4 seconds
Frank@Ubuntu:~ /tmp$
frankvHennes:~$ sudo docker ps
CONTAINER ID  IMAGE       COMMAND                  CREATED           STATUS          PORTS                  NAMES
49f23bc6d87b  ubuntu       "/bin/sh -c 'while tr"      2 minutes ago    Up 2 minutes    2/15                     CiscoLive10
2d5c9653559e  ubuntu       "/bin/sh -c 'while tr"      2 minutes ago    Up 2 minutes    2/15                     CiscoLive9
269351a903c2  ubuntu       "/bin/sh -c 'while tr"      2 minutes ago    Up 2 minutes    2/15                     CiscoLive8
015159b964fe  ubuntu       "/bin/sh -c 'while tr"      2 minutes ago    Up 2 minutes    2/15                     CiscoLive7
ab447c749141  ubuntu       "/bin/sh -c 'while tr"      2 minutes ago    Up 2 minutes    2/15                     CiscoLive6
d0872ef85852  ubuntu       "/bin/sh -c 'while tr"      2 minutes ago    Up 2 minutes    2/15                     CiscoLive5
aa0a2b1ce7b9  ubuntu       "/bin/sh -c 'while tr"      2 minutes ago    Up 2 minutes    2/15                     CiscoLive4
3986c3a153f4  ubuntu       "/bin/sh -c 'while tr"      2 minutes ago    Up 2 minutes    2/15                     CiscoLive3
4729ff08ce69  ubuntu       "/bin/sh -c 'while tr"      2 minutes ago    Up 2 minutes    2/15                     CiscoLive2
48142733e722  ubuntu       "/bin/sh -c 'while tr"      2 minutes ago    Up 2 minutes    2/15                     CiscoLive1

frankvHennes:~$ sudo docker attach CiscoLive1
Hello CiscoLive
Hello CiscoLive
Hello CiscoLive
Hello CiscoLive
^C
frankvHennes:~$ sudo docker stop CiscoLive1
CiscoLive1
frankvHennes:~$ sudo docker rm CiscoLive1
CiscoLive1
frankvHennes:~$ sudo docker ps
CONTAINER ID  IMAGE       COMMAND                  CREATED           STATUS          PORTS                  NAMES
49f23bc6d87b  ubuntu       "/bin/sh -c 'while tr"      3 minutes ago    Up 3 minutes    2/15                     CiscoLive10
2d5c9653559e  ubuntu       "/bin/sh -c 'while tr"      3 minutes ago    Up 3 minutes    2/15                     CiscoLive9
269351a903c2  ubuntu       "/bin/sh -c 'while tr"      3 minutes ago    Up 3 minutes    2/15                     CiscoLive8
015159b964fe  ubuntu       "/bin/sh -c 'while tr"      3 minutes ago    Up 3 minutes    2/15                     CiscoLive7
ab447c749141  ubuntu       "/bin/sh -c 'while tr"      3 minutes ago    Up 3 minutes    2/15                     CiscoLive6
d0872ef85852  ubuntu       "/bin/sh -c 'while tr"      3 minutes ago    Up 3 minutes    2/15                     CiscoLive5
aa0a2b1ce7b9  ubuntu       "/bin/sh -c 'while tr"      3 minutes ago    Up 3 minutes    2/15                     CiscoLive4
3986c3a153f4  ubuntu       "/bin/sh -c 'while tr"      3 minutes ago    Up 3 minutes    2/15                     CiscoLive3
4729ff08ce69  ubuntu       "/bin/sh -c 'while tr"      3 minutes ago    Up 3 minutes    2/15                     CiscoLive2

frankvHennes:~$
Creating your own images
Within my simple bash container, we try and run the "traceroute" command, but it isn’t installed:

```
root@d9b58462988a:/
frank@vHennes:$ sudo docker run -it --name CiscoLive ubuntu bash
root@46b58462988a:/# traceroute 8.8.8.8
bash: traceroute: command not found
root@46b58462988a:/#
```

We could install it inside my container, but as soon as we exit and start another container, traceroute is gone (as we have started a clean container from scratch).
What if I want to create a new image with the “traceroute” command already installed?

Two options:

1) Commit the container as a new image after making the changes.
2) Use a Dockerfile containing image creation instructions.
Option 1: Manually compose your container

```
root@46b58462988a:/ # apt-get update
Hit:1 http://archive.ubuntu.com/ubuntu xenial InRelease
Hit:2 http://archive.ubuntu.com/ubuntu xenial-updates InRelease
Hit:3 http://archive.ubuntu.com/ubuntu xenial-security InRelease
Reading package lists... Done
root@46b58462988a:/ # apt-get install inetutils-traceroute
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  ifupdown iproute2 isc-dhcp-client isc-dhcp-common libbatm1 libdns-export162 libisc-export160
  libmnl0 libxtables11 netbase
Suggested packages:
  ppp rdnsd iproute2-doc resolvconf avahi-autoipd isc-dhcp-client-ddns apparmor
The following NEW packages will be installed:
  ifupdown inetutils-traceroute iproute2 isc-dhcp-client isc-dhcp-common libbatm1 libdns-export162
  libisc-export160 libmnl0 libxtables11 netbase
0 upgraded, 11 newly installed, 0 to remove and 6 not upgraded.
Need to get 1839 kB of archives.
After this operation, 5536 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
```
Option 1: Commit the changes to your container

frank@vHennes:$ sudo docker commit --help

Usage: docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

Create a new image from a container's changes

Options:
- -a, --author string  Author (e.g., "John Hannibal Smith <hannibal@a-team.com>")
- -c, --change value   Apply Dockerfile instruction to the created image (default [])
- -help                Print usage
- -m, --message string Commit message
- -p, --pause          Pause container during commit (default true)

frank@vHennes:$ sudo docker commit Ciscolive brockners/ciscolivedemo sha256:e183d36746f4a0bfe08ef0ea6353e30de0d272cd4522ead9bc413856c12d4dc7

frank@vHennes:$ sudo docker images

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>IMAGE ID</th>
<th>CREATED</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>brockners/ciscolivedemo</td>
<td>latest</td>
<td>e183d36746f4</td>
<td>About a minute ago</td>
<td>173.7 MB</td>
</tr>
<tr>
<td>ubuntu</td>
<td>latest</td>
<td>104bec311bcd</td>
<td>4 weeks ago</td>
<td>129 MB</td>
</tr>
</tbody>
</table>

frank@vHennes:$
**Option 1:**

Make your container available to others: Dockerhub

```
frank@vHennes:~$ sudo docker push brockners/ciscolivedemo
The push refers to a repository [docker.io/brockners/ciscolivedemo]
54d55c1f5f5a: Pushing 10.62 MB/44.75 MB
5972ebe5b524: Pushed
3d515508d4eb: Pushed
bb6cef52379: Pushed
87f743c24123: Pushed
32d75bc97c41: Pushing 9.695 MB/129 MB
```

```
frank@vHennes:~$ sudo docker push brockners/ciscolivedemo
The push refers to a repository [docker.io/brockners/ciscolivedemo]
54d55c1f5f5a: Pushed
5972ebe5b524: Pushed
3d515508d4eb: Pushed
bb6cef52379: Pushed
87f743c24123: Pushed
32d75bc97c41: Pushed
latest: Digest: sha256:e0a818f9b558a450612365de017d2653d3dc1f81cbe9867a5cb604164de7394d4 Size: 1569
frank@vHennes:~$ ```

Option 1:
Once pushed, the new image is available to others
The second (and recommended) method for building images is to use a Dockerfile.
Option 2: Let’s create a simple Dockerfile

- A Dockerfile is simply a text file containing instructions on how to build a Docker image – a “Makefile” for container construction.
- Each line in the Dockerfile creates a new image layer

```
frank@vHennes:~/tmp

frank@vHennes:~/tmp$ cat Dockerfile
# Dockerfile to install traceroute
FROM ubuntu
MAINTAINER Frank Brockners (fbrockne@cisco.com)
RUN apt-get update && apt-get install traceroute
frank@vHennes:~/tmp$
```
Option 2: Our –v2 Image is now available...

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>IMAGE ID</th>
<th>CREATED</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>brockners/ciscolivedemo-v2</td>
<td>latest</td>
<td>5ab4f2157463</td>
<td>42 seconds ago</td>
<td>169 MB</td>
</tr>
<tr>
<td>brockners/ciscolivedemo</td>
<td>latest</td>
<td>e183d36746f4</td>
<td>19 minutes ago</td>
<td>173.7 MB</td>
</tr>
<tr>
<td>ubuntu</td>
<td>latest</td>
<td>104bec311bcd</td>
<td>4 weeks ago</td>
<td>129 MB</td>
</tr>
</tbody>
</table>

```
frank@vHennes:~$ sudo docker images
REPOSITORY                TAG           IMAGE ID             CREATED               SIZE
brockners/ciscolivedemo-v2 latest  5ab4f2157463  42 seconds ago  169 MB
brockners/ciscolivedemo     latest  e183d36746f4  19 minutes ago  173.7 MB
ubuntu                     latest  104bec311bcd  4 weeks ago     129 MB
frank@vHennes:~$  
```
Option 2: Our –v2 Image is now available...

... and we can run it

```
frank@vHennes:~$ sudo docker images
REPOSITORY                  TAG        IMAGE ID             CREATED              SIZE
brockners/ciscolivedemo-v2  latest     5ab4f2157463         42 seconds ago      169 MB
brockners/ciscolivedemo     latest     e183d36746f4         19 minutes ago       173.7 MB
ubuntu                      latest     104bec311bdc          4 weeks ago          129 MB

frank@vHennes:~$ sudo docker run -it brockners/ciscolivedemo-v2 /bin/bash
root@15db82b18d6a:/# traceroute 8.8.8.8
traceroute to 8.8.8.8 (8.8.8.8), 30 hops max, 60 byte packets
  1  172.17.0.1 (172.17.0.1)  0.000 ms  0.011 ms  0.007 ms
  2  192.168.132.2 (192.168.132.2)  0.149 ms  0.071 ms  0.121 ms^C
root@15db82b18d6a:/#
```
Container Technology

Docker

Containers on Cisco devices

Container Orchestration and Networking

Cloud Native Container Networking
Containers on Cisco Devices: Containers in XR

- 64-bit Open Embedded Linux support
  - Container support
  - Standard Linux Toolchains
  - Third-Party Applications Support
  - NCS 5500, NCS 5000, NCS1002, XRv 9000, ASR 9000
    - LXC and Docker XR 6.1.2

- IOS-XR offers application hosting
  - natively on the Host OS (WRL 7)
  - within a Container (managed through libvirt)

- See also IOS-XR Application Hosting Configuration Guide
Cisco developed packages for core network functions (BGP, MPLS, etc.)

Yocto packages for standard Linux tools and libraries (bash, python, tcpdump, etc.).

Runs processes responsible to perform system diags, monitor env. variables, and manage HW components.

First container to be booted by the host, responsible for the start and maintenance of the Control Plane container.

Runs any 64-bit Linux distribution. Launched from the XR container using virsh and libvirtd.

Access XR interfaces through shared network namespaces.

Cisco Public
Global-VRF Network Name Space

- Provide visibility of fabric attached interfaces outside of XR CLI.
- Available to processes in the XR containers or Third Party containers.
- Requires that the interface is Up
- Routing handled by XR.
Third Party Network Name Space

RP/0/RP0/CPU0:ios#run
Wed Oct 28 18:45:56.168 IST

[XR-vm_node0_RP0_CPU0:~]$ ip netns exec tpnns bash
[XR-vm_node0_RP0_CPU0:~]$ ifconfig
Gi0_0_0_0 Link encap:Ethernet  HWaddr 52:46:04:87:19:3c
    inet addr:192.164.168.10  Mask:255.255.255.0
    inet6 addr: fe80::5046:4ff:fe87:193c/64 Scope:Link
    UP RUNNING NOARP MULTICAST  MTU:1514  Metric:1
    RX packets:0  errors:0  dropped:0  overruns:0  frame:0
    TX packets:3  errors:0  dropped:0  overruns:0  carrier:0
    collisions:0  txqueuelen:1000
    RX bytes:0 (0.0 B)  TX bytes:210 (210.0 B)
Details:
NCS-5500/5000 Architecture
(Container Based)
XR Control Plane LXC

Named network namespace (vrf)

XR control plane

East-west interface

• XR processes (Routing protocols, etc.)
  • XR CLI

XR FIB

Third party Container

Named network namespace (vrf)

Update Source IP

Mgmt subnet

64-bit Linux Host OS

Mgmt0/RP0/CPU0/0

---

Update Source IP

Mgmt subnet

BRKSDN-2115   © 2018 Cisco and/or its affiliates. All rights reserved. Cisco Public 80
64-bit Linux Host OS

XR Control Plane LXC

- Named network namespace (vrf)
  - fwd_ew
  - fwdintf
- XR control plane
  - East-west interface
  - XR processes (Routing protocols, etc.)
  - XR CLI
- XR FIB
  - fwdintf

3rd party Docker container

- Named network namespace (vrf)
  - fwd_ew
  - fwdintf
- Update Source IP
- Mgmt subnet

XR Linux

- Docker Client
- Update Source IP
- Mgmt subnet

- Docker Daemon
- Mgmt0/RP0/CPU0/0
- Gig0/0/0/2
- Gig0/0/0/2
- Gig0/0/0/2

© 2018 Cisco and/or its affiliates. All rights reserved. Cisco Public
Details:
ASR 9k Architecture (VM Based)
XR Control Plane VM (ASR9k)

- XR processes (Routing protocols, etc.)
- XR CLI

Mgmt subnet

Libvirtd Daemon

3rd party LXC

Named network namespace (vrf)

East-west interface

fwd_ew
fwdintf

Update Source IP

Mgmt subnet

virsh Client

XR linux

Update Source IP

XR processes (Routing protocols, etc.)

XR FIB

Update Source IP

Mgmt subnet

64-bit Linux Host OS

Gig0/0/2
Gig0/0/2
Gig0/0/2

Gig0/0/2
Let’s leverage our learnings on Docker in IOS-XR
User orchestrates docker containers in XR bash shell

- XR LXC Bash
  - Docker Client Binary
  - Loopback1
- Docker Container 1
  - App 1 Binaries
- Docker Container n
  - App 2 Binaries

- Shared Network Namespace (global-vrf)
- XR LXC only Storage Volume
- Docker Storage Pool (upto 1.5 to 2G)
  - libdevmapper/libcontainer
- Remote REST API <localhost:2376>
- Docker Daemon 1.10

- XR Router
Docker Application Workflow

• Create a docker image
• Pull down using docker client in XR control plane
• Spin up the docker container on host OS by executing “docker run” in XR linux shell.

```
docker pull <>
docker run <>
docker exec <>
```
Linux Build Machine: Compose your App

- Create Dockerfile on a Linux Build Host describing your App

```bash
frank@vHennes:~/tmp
frank@vHennes:~/tmp$ cat XR-Dockerfile
# Template Dockerfile
FROM <customer-repo>/ubuntu:<label>
MAINTAINER <name>

RUN apt-get -y update && apt-get -y install \ 
    build-essential \ 
    ...

RUN git clone <app-git-repo>
WORKDIR <app-build-dir-name>
RUN { \ 
    make <app-target>; \ 
    ... \ 
}

ENV <app-variable-name=value>
CMD ["<param1>" "<param2>" ...]
ENTRYPOINT ["<app-cmd-name>
frank@vHennes:~/tmp$
```
Linux Build Machine: Compose your App

• Build and image

• Upload to your repository of choice (custom or Dockerhub)

Find existing docker images (initially this will show existing images)

```
docker images
```

Using the Dockerfile created earlier build the docker image in the directory containing created dockerfile

```
docker build -t myimage .
docker images  # should now show myimage
```

Push docker image to a customer repository

```
docker push <customer-registry-name:port>/myimage>
```

Alternatively, one can just export the docker image as a tarball. This requires creating a docker container.

```
docker run --net=host --name=mycont --it myimage /bin/bash
docker export --output "myimage" <docker-container-name>
```
XR Router: Install your App using the tarball

- Download image tarball
- User downloads docker image in bash shell created using bash command
- If so required – clean up 😊

Find existing docker images (initially this will show no images)

```bash
docker images
```

Create a docker image using a tarfile
(do "docker export --output "myimage" <docker-container-name>" on another machine and scp the file "myimage" to XR bash shell)

```bash
docker import myimage
docker images # should now show myimage
```

Remove a docker image
(must stop and then remove all running containers using that image)

```bash
docker rmi myimage
docker images " should show myimage is removed"
```
XR Router: Install your App using the Docker Registry

- User downloads docker image from registry in bash shell created using `bash` command
- If so required – clean up 😊

Find existing docker images (initially this will show no images)
```
docker images
```

To pull a docker image from a customer's private registry, download and install ca.crt for the customer private registry
```
mkdir -p /etc/docker/certs.d/<customer-registry-name:port>
scp <source-path>/ca.crt /etc/docker/certs.d/<customer-registry-name:port>/ca.crt
```

Pull docker images from registry
```
docker pull <customer-registry-name:port>/myimage
docker images # should now show myimage
```

Remove a docker image (must stop and then remove all running containers using that image)
```
docker rmi myimage
docker images “ should show myimage is gone”
```
XR Router: Start your Containers

- User orchestrates/manages docker containers in bash shell created using `bash` command

```
Find existing docker containers (with their container-id/container-name)

   docker ps -a

Create and start a new docker container (creates, starts and attaches to the container bash)

   docker run --net=host --name=mycont --restart "always" --it myimage
   /bin/bash

Alternatively, create but do not start a new docker container

   docker create --net=host --name=mycont --restart "always" --itd myimage
   /bin/bash

Start existing stopped docker container

   docker start -i <container-id or container-name>

Run a new process (e.g., bash shell) in an existing running docker container

   docker exec -it|-d <container-id or container-name> /bin/bash >
```
XR Router: Stop and Remove your Containers

- User orchestrates/manages docker containers in bash shell created using bash command

  Find existing docker containers (with their container-id/container-name)
  
  ```
  docker ps -a
  ```

  Stop existing running docker container
  
  ```
  docker stop <container-id or container-name>
  ```

  Remove stopped docker container
  
  ```
  docker rm <container-id or container-name>
  ```

  Check that the docker container is completely removed
  
  ```
  docker ps -a
  ```
User debugs docker containers in bash shell created using `bash` command

- Check the state of existing docker containers
  ```
docker ps -a
  ```

- Capture docker logs to capture activity that has already occurred
  ```
docker logs --details <container-id or container-name>
  ```

- Capture docker logs to capture future docker activity
  ```
docker logs --follow --details <container-id or container-name>
  ```

- To debug what is happening in a running container, run another bash shell in the existing docker container
  ```
docker exec --it <container-id or container-name> /bin/bash > ps --ef
  ```
  # run this to check processes running inside docker container
Examples
Example 1: Router “State” enforcement

The Traditional Approach :
Enforce state through periodic “Commit Replace”
Example 1: Router “State” enforcement

The DevOps approach:

- Config Management tool to enforce intended state
- Stream real-time Telemetry data to detect “state” discrepancy
- Remediate “diff” through a chef/puppet/ansible run and/or raise alerts!
Example 2: Testing and Deployment

The Traditional Approach:

Weeks of Testing on Physical hardware: Expensive, non-representative

Multiple Step “Manual” Deployment Process:
- Load Images, config manually
- Verify connectivity
- Remediate errors manually
Example 2: Testing and Deployment

The DevOps Approach:

Automated Testing of PI features on Virtual Hardware:
- Representative Topologies
- Cheaper

Automated Deployment process:
- Auto-provisioning fetches images/config automatically
- Configuration Management tools enforce intended state
Try out IOS-XR App-hosting on your laptop!

To get your own copy, head over to https://xrdocs.github.io/application-hosting/tutorials/iosxr-vagrant-quickstart
Container Technology

Docker

Containers on Cisco devices

Container Orchestration and Networking

Cloud Native Container Networking
The default networking in Docker is fairly basic...
Docker Containers are connected using a bridge

Host

docker0 Bridge
172.17.0.1

vethaaa2e22@if10  veth9a598b1@if12

Eth0  
172.17.0.2  CiscoLive

Eth0  
172.17.0.3  CiscoLive2
Container: CiscoLive

```shell
root@73878bbdb884:/# ifconfig eth0
eth0  Link encap:Ethernet HWaddr 02:42:ac:11:00:02
      inet addr:172.17.0.2 Bcast:0.0.0.0 Mask:255.255.0.0
      inet6 addr: fe80::42:acff:fe11:2/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
      RX packets:12523 errors:0 dropped:0 overruns:0 frame:0
      TX packets:8406 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0
      RX bytes:2783964 (27.0 MB) TX bytes:460945 (460.9 KB)
```

Container: CiscoLive2

```shell
root@90f72d9afca8:/# ifconfig eth0
eth0  Link encap:Ethernet HWaddr 02:42:ac:11:00:02
      inet addr:172.17.0.2 Bcast:0.0.0.0 Mask:255.255.0.0
      inet6 addr: fe80::42:acff:fe11:3/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
      RX packets:11955 errors:0 dropped:0 overruns:0 frame:0
      TX packets:8349 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0
      RX bytes:26812833 (26.8 MB) TX bytes:458995 (458.9 KB)
```

Docker Networking

- Bridge network driver (--driver=bridge)
- None network driver (--driver=none)
- Host network driver (--driver=host)
- Overlay network driver (--driver=overlay) – Multi-Host using VXLAN
- MACVLAN network driver (--driver=macvlan)
- Remote drivers – compatible with CNM
  - Contiv, Weave, Calico, Kuryr
Networking typically means multiple hosts..

Host

docker0 Bridge

vethaaa2e22@if10

Eth0

CiscoLive

Host

docker0 Bridge

veth9a598b1@if12

Eth0

CiscoLive2
Docker version 1.9 introduced some key enhancements to networking:

- Multi-host networking
- Multiple interfaces on containers
- Multiple named networks
Docker also now uses a plugin model for networking, with available plugins including:

- Calico
- Weave
- Contiv
- Midokura
- VMware
- Nuage Networks
- …
What is “Project Contiv”? 
“An open source project defining infrastructure operational policies for container-based application deployment.”
Contiv Overview

• Policy-based Networking for Microservice deployments
  • Simplify deployment of micro-services
  • Implement policy-driven networking abstraction layer for microservices

• Comprehensive feature set
  • Common network abstraction layer for all types of workloads: Baremetal, VMs, containers
  • Flexible groupings: Tenant level isolation, isolate control and data traffic
  • Support for major container schedulers: Kubernetes, OpenShift
  • Common networking concept for different transports: L2, L3 (BGP), Overlay (VXLAN) and ACI modes

• Two major components make up Contiv
  • Netmaster
  • Netplugin (Contiv Host Agent)
Contiv Network Components

Contiv CLI/UI

- Tools to manipulate Contiv objects
- Implements CRUD using REST I/F
- Expected to be used by Infra/Ops Teams

Contiv Master

- Distributed Cluster wide Function
- Stateless: useful in node failure/restart, upgrade
- Implements cluster wide network and policy
- Manage Global Resources: IPAM, VLAN/VXLAN Pools

Contiv Agent

- Container Networking for:
  - Kubernetes, Mesos, Nomad, and Swam
- Route Distribution using BGP or Json RPC
- Custom openflow pipeline for host networking
  - Allows implementing various features (details later)
- Exports Data about: App Connectivity, Stats, Peer

Node 1

Node 2

Node-n

BRKSDN-2115 © 2018 Cisco and/or its affiliates. All rights reserved. Cisco Public 113
Contiv network plugin: Multi-tenancy

Container 1
Container 2
Container 3
Tenant A

Container 1
Container 2
Container 3
Tenant B

Container 1
Container 2
Container 3
Tenant C

Overlapping IP subnets between tenants.
Contiv network plugin: Security policies

- Group “Web”
  - Container 1
  - Container 2
  - Container 3
  - Policy “Web”

- Group “App”
  - Container 1
  - Container 2
  - Container 3
  - Policy “DB”

- Group “DB”
  - Container 1
  - Container 2
  - Container 3
Contiv Summary

- LDAP+ RBAC
- All New User Experience and Workflow
- Kubernetes 1.8+ Support
- OpenShift 3.6+ Integration
- Simple 1-click Install

Commercially Supported Contiv

- Cisco Advance Services
- Cisco Technical Services Solution Support for Contiv

Open Source at http://contiv.github.io/
Cisco ACI + Contiv (ACI mode)

Solution Highlights

- Docker containers/OpenShift pods data plane connectivity is provided by ACI fabric
- ACI policies can be extended across physical, virtual machines, and Docker containers/OpenShift Pods

Unified Policy Automation and Enforcement Across Physical, Virtual, and Containers
Kubernetes
What is Kubernetes (K8s)?

- Container orchestrator
- Runs and manages containers
- Supports multiple cloud and bare-metal environments
- Inspired and informed by Google's experiences and internal systems
- 100% Open source, written in Go
- Manage applications, not machines
- Rich ecosystem of plug-ins for scheduling, storage, networking
Kubernetes Deployment

- Master(s)
  - Scheduler
  - Controller Manager
  - etcd
  - API

- Node(s)
  - Container
  - Pod
  - Kubelet
  - Kube-Proxy

- API
- CLI
- UI

FOR YOUR REFERENCE
K8s Resource Types

Kubernetes has five main resource types that can be created and configured using a YAML or a JSON format file.

- **Pods:** Represent a collection of containers (usually running an application or a set of applications tightly coupled) and other resources with a shared context that are grouped together by Kubernetes. It is the basic unit of work Kubernetes manages. Since each container from a pod may require customization using environment variables, they can be passed as parameter during pod customization.

- **Services:** Defines a single IP/Port combination that provides access to a pool of pods and by default connects the client to a pod in a round-robin fashion. This solves the dilemma of having to keep up with every transient IP address assigned by Docker.

- **Replication Controllers:** A framework for defining pods that are meant to be horizontally scaled. A replication controller includes a template, which is a pod definition that is to be replicated.

- **Persistent Volumes (PV):** Provisions persistent networked storage to pods that can be mounted inside a container to store data. Normally provided by the Kubernetes administrator. It is already mounted and accessible for immediate consumption. The storage will not be destroyed when a container is destroyed.

- **Persistent Volume Claims (PVC):** Represents a request for storage by a pod to Kubernetes to be bounded for an available PV. This approach allows multiple containers to use the same PV and share its contents.
Kubernetes Deployment

Master nodes:
- Controls and manages the cluster
- Kubectl (command line)
- REST API (communication with workers)
- Scheduling and replication logic

Worker nodes:
- Hosts the K8s services
- Kubelet (K8s agent that accepts commands from the master)
- Kubeproxy (network proxy service responsible for routing activities for inbound or ingress traffic)
- Docker host

Nodes: Hosts that run Kubernetes applications
Kubernetes Deployment

Pods:
- Smallest deployment unit in K8s
- Collection of containers that run on a worker node
- Each has its own IP
- Pod shares a PID namespace, network, and hostname

Replication controller:
- Ensures availability and scalability
- Maintains the number of pods as requested by user
- Uses a template that describes specifically what each pod should contain

Labels:
- Metadata assigned to K8s resources
- Key-value pairs for identification
- Critical to K8s as it relies on querying the cluster for resources that have certain labels

Service:
- Collections of pods exposed as an endpoint
- Information stored in the K8s cluster state and networking info propagated to all worker nodes
**Nodes, Pods, Containers**

- **Node:**
  - A server

- **Cluster:**
  - Collection of nodes

- **Pod:**
  - Collection of containers; Nodes can run multiple Pods
Nodes, Pods, Containers, Labels

• Labels:
  • key/value pairs applied to container

• Pod addressing/reachability
  • Each pod gets exactly 1 interface with exactly 1 IP
  • All containers in Pod share a localhost
  • All Pods (subject to network policy) can talk to each other via L3
  • Containers within a Pod share a common networking namespace

• Nodes
  • are assigned unique IP subnets
Example Setup with the Kubernetes default bridge cbr0

"Fabric": logical "router" (provided by CNI)
Example Setup with the Kubernetes: Pod Addressing

“Fabric”: logical “router” (provided by CNI)

Node 1

External: 10.20.30.1
Internal: 10.100.10.1 / 24

eth0

cbr0

veth

host/root ns

eth0

10.100.10.10

10.100.10.20

POD 1

POD 2

Node 2

External: 10.20.30.2
Internal: 10.100.20.1 / 24

eth0

cbr0

veth

host/root ns

eth0

10.100.20.10

10.100.20.20

POD 1

POD 2

Node 3

External: 10.20.30.3
Internal: 10.100.30.1 / 24

eth0

cbr0

veth

host/root ns

eth0

10.100.30.10

10.100.30.20

POD 1

POD 2

Black address managed by operator
Blue address managed by K8s
Namespaces

- K8s namespaces represent Linux ‘namespaces’
- K8s namespaces replace the need for having a ‘tenant’ concept
Selectors

- Selectors are
  - simple
  - powerful
  - can do everything you’re used to do with EPGs
Services Overview

• “Pods can come and go, services stay”

• Define a single IP/Port combination that provides access to a pool of pods

• By default a service connects the client to a Pod in a round-robin fashion

• This solves the dilemma of having to keep up with every transient IP address assigned by Docker
Services Overview: Example Setup:
Service Addressing:

```
Service Address: 10.11.12.13:80
```

"Fabric": logical "router" (provided by CNI)

Node 1
- eth0
- cbr0
- veth
- host/root ns
- 10.100.10.10
- 10.100.20.10
- POD 1
- POD 2

Node 2
- eth0
- veth
- host/root ns
- cbr0
- veth
- 10.100.20.10
- 10.100.30.10
- POD 1
- POD 2

Node 3
- eth0
- veth
- host/root ns
- cbr0
- veth
- 10.100.30.10
- 10.100.20.10
- POD 1
- POD 2

External: 10.20.30.1
Internal: 10.100.10.1 / 24

Service Address:
- Src: 10.100.20.10
- Dst: 10.100.20.10
- Src: 10.100.30.10
- Dst: 10.100.30.10
- Src: 10.100.20.10
- Dst: 10.100.20.10
- Src: 10.100.30.10
- Dst: 10.100.30.10
- Src: 10.100.30.10
- Dst: 10.100.20.10
# Kubernetes: Services – Deployments, EndPoints

## Service Definition

```yaml
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376
```

*Service Definition (not connected to any running app yet)*

## Deployment

```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

*Example: Create a ‘Deployment’ standing up 3 ‘replicas’ (Pods) to provide that service*

## Endpoints

```yaml
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
spec:
  subsets:
    - addresses:
        - ip: 1.2.3.4
          - ip: 4.3.2.1
          - ip: 1.1.1.1
        ports:
          - port: 9376
```

*‘Endpoints’ connect ‘Service’ to ‘location’ via IP/port; K8s creates an endpoint for each Pod*
Kubernetes: Services – Summary

A Service may LB to multiple Pods

- Service1
  - Endpoint1
    - Pod1
      - Selector1: ...
  - Endpoint2
    - Pod2
      - Selector1: ...
  - Endpoint3
    - Pod 3
      - Selector3: ...

A Pod may provide multiple Services

- Service2
  - Endpoint4
    - Pod3
      - Selector3: ...
      - Selector4: ...
  - Endpoint5
  - Endpoint6

Logical
Actual
Kubernetes Service, Pods, Replication Controllers, Labels

Front End Service
app=webapp, role=frontend, version=v1

Front End v1 Pod
app=webapp, role=frontend, version=v1

Front End v1 Pod
app=webapp, role=frontend, version=v1

Front End v2 Pod
app=webapp, role=frontend, version=v2

Front End v1 Controller
Desired Count = 2
app=webapp, role=frontend, version=v1

Front End v2 Controller
Desired Count = 1
app=webapp, role=frontend, version=v2
Kubernetes Networking – Key Communication patterns

4 distinct communication patterns:

<table>
<thead>
<tr>
<th>Communication Pattern</th>
<th>Network Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highly-coupled container-to-container communications</td>
<td>Pod concept and localhost communications</td>
</tr>
<tr>
<td>Pod-to-Pod communications</td>
<td>Network Plugins</td>
</tr>
<tr>
<td>Pod-to-Service communications</td>
<td>Kubernetes services concept / ClusterIP</td>
</tr>
<tr>
<td>External-to-Service communications</td>
<td>Kubernetes services concept / NodePort, ExternalIP</td>
</tr>
</tbody>
</table>
Kubernetes Network Plugins

• Kubernetes fundamental requirements for any networking implementation
  • all containers can communicate with all other containers without NAT
  • all nodes can communicate with all containers (and vice-versa) without NAT
  • the IP that a container sees itself as is the same IP that others see it as

• Network Plugins provide various levels of sophistication – from simple to highly feature rich and performant

Examples
• Cilium
• Contiv
• Contrail
• Flannel
• Google Compute Engine (GCE)
• Kube-router
• L2 networks and linux bridging
• Multus
• NSX-T
• Nuage Networks VCS
• OpenVSwitch
• OVN (Open Virtual Networking)
• Project Calico
• Romana
• Weave Net from Weaveworks
• CNI-Genie from Huawei
Container Technology

Docker

Containers on Cisco devices

Container Orchestration and Networking

Cloud Native Container Networking
What Container Network Stacks Provide Today:

- Lifecycle management for application Containers
- Overlay connectivity for application Containers:
  - NAT communication with external world
  - Policy controlled overlay, may extend policy control to DC fabric in some cases
  - Network policy addresses security/connectivity
  - Designed for Data Center applications / use cases
- Good start, but not sufficient for NFV use cases (Firewalls, Customs VNFs)
What Container Networks Stacks Lack for NFV Use Cases:

- NVF-specific policy APIs (e.g. QoS, placement considering network resources, etc.)

- Networking:
  - NAT is a bottleneck, not suitable for NFV use cases
  - VNFs often require more than 1 interface / IP address
  - No support for high-speed wiring of NFVs:
    - To the outside world; To application containers; Between NFV containers; Creation of Service Function Chains (mixed physical & virtual)

- Management/Control:
  - Containerized NVFs not really in the data plane (except for the vswitch)
  - No support for cloud-native, high-performance NVFs

- Forwarding:
  - Kernel or OVS used for forwarding
Cloud-Native Network Function Summary

- Container-based
- Container stack lifecycle (same as application containers)
- 12-factor app design for management/control
- High-performance forwarding
- High-performance networking
- Seamless virtual/physical world interworking
- Common policy control with application containers (as much as possible)
- Must solve 3 key problems:
  - Lifecycle management
  - High-Performance Networking (wiring/config, operations)
  - Easy installation, operation
Contiv Networking Control: Contiv-VPP
https://github.com/contiv/vpp

- Contiv is a networking plugin for Kubernetes that:
  - Allocates IP addresses to Pods (IPAM)
  - Programs the underlying infrastructure it uses (Linux TCP/IP stack, OVS, VPP, …) to connect the Pod’s to other Pods in the cluster and/or to the external world.
  - Implements K8s network policies that define which pods can talk to each other.
  - Implements K8s services; a service exposes one or more (physical) service instances implemented as K8s pods to the other pods in the cluster and/or to external clients as a virtual instance (e.g. as a virtual “service” IP address).

- Contiv is a user-space based, high-performance, high-density networking plugin for Kubernetes. Contiv-VPP leverages FD.io/VPP as the industry’s highest performance data plane.
The Universal Dataplane - FD.io Vector Packet Processor (VPP)

- VPP is a rapid packet processing development platform for highly performing network applications
  - 18+ MPPS, single core
  - Multimillion entry FIBs
  - 480Gbps bi-dir on 24 cores
- Runs on commodity CPUs and leverages DPDK
- Creates a vector of packet indices and processes them using a directed graph of nodes – resulting in a highly performant solution.
- Runs as a Linux user-space application
- Ships as part of both embedded & server products, in volume; Active development since 2002
- FD.IO (The Fast Data Project)
Ligato – Accelerate Cloud-Native Development
Platform and code samples for development of cloud native VNFs

Components:

- Cloud-Native Infrastructure – a Golang platform for building cloud-native microservices
- VPP-agent - Golang implementation of a control/management plane for FD.io/VPP based cloud-native Virtual Network Functions (VNFs)
- SFC-Controller - Service Function Chain (SFC) Controller for stitching virtual and physical networks
- BGP-Agent - BGP Agent is a BGP information provider

https://ligato.github.io/
Putting it all together
Enabling Production-Grade Native Cloud Network Services at Scale

Production-Grade Container Orchestration
- kubernetes

Performance-Centric Container Networking
- Contiv

Cloud-native NF Orchestration
- LIGATO

Containerized Fast Data Input/Output
- RIO

Network Function and Network Topology Orchestration
- Container Networking
- Kubernetes
- API Proxies

Containerized Network Data Plane
- Networking Plugin
- Contiv-VPP Netplugin

Service Policy
Service Topology
Lifecycle
Ligato Agent: a CNF Development Platform
www.github.com/ligato/cn-infra
Ligato VPP Agent: a CNF Development Platform
www.github.com/ligato/vpp-agent
Cloud-native Network Micro-Services Putting It All Together Now – The System Design

Functional Layered Diagram

System Components

Putting It All Together Now – The System Design

- Service Policy
- Service Topology
- Lifecycle

- Production-Grade Container Orchestration
  - Kubernetes
  - API Proxies

- Network Function and Network Topology Orchestration
  - Container Networking
    - Contiv-VPP Netmaster

- Containerized Network Data Plane
  - Networking Plugin
    - Contiv-VPP Netplugin
  - Kubelet

- Control and Management Plane
  - Applications
    - clientv1
  - Contiv-VPP Netmaster
  - SFC Controller
    - clientv1
  - Tools (e.g. agentctl)
    - clientv1

- Inter-Process Communication
  - Data Store
    - etcd
    - Configuration
    - Operational State
  - Message Bus
    - k8s
    - Configuration Notifications

- Containers Lifecycle Orchestration
  - Containerized Switch
  - Containerized Network Functions
  - Containerized Network Functions
  - Containerized Switch

- Container Networking
  - Networking Plugin
    - Contiv-VPP Netplugin
  - Kubelet

- Containerized Network Data Plane

- Cloud-native Network Micro-Services
Network Micro-Service Use Case: Service Function Chaining with Cloud-Native NFs
A glimpse at performance: Example topology

Topology: Containerized Switch with one or more Containerized Network Functions
Example Benchmark: memif 10x faster than veth
Cloud-native Networking

**Prepare System Environment**
1. Install Container orchestration infrastructure: K8s, CRI Shim
2. Define Contiv-VPP setup: deployment .yaml file, VPP startup.conf
3. Define App Pods in .yaml files

**Deploy App Pods**
4. Deploy defined Contiv-VPP PODs with K8s
5. Deploy defined App PODs with K8s
6. Agents configure their local VPPs containerized in K8s PODs

**Verify Service is Up**
7. Verify Containerized network topology is configured correctly

**Benchmark for Service Acceptance**
8. Benchmark Containerized network topology
What else is missing for microservices?

- Visibility
- Resiliency & Efficiency
- Traffic Control
- Security
- Policy Enforcement

Enter: Service Mesh
Vanilla Kubernetes

[Diagram showing a Kubernetes Pod with components: App Container, TLS, Retries, Metrics]
Introducing a sidecar – Istio/Envoy
Proxy (sidecar) details

- Deployed as a sidecar per Pod
- Proxy acts as a client and server

**Service A Cluster**
- Service A
- Envoy

**Service B Cluster**
- Service B
- Envoy

**External Network**
- HTTP/1.1, HTTP/2, TLS

**Edge Proxy**
- HTTP/2, gRPC, REST
Service Mesh Overview

- **Outbound features**
  - Service authentication
  - Load balancing
  - Retry and circuit breaker
  - Fine-grained routing
  - Telemetry
  - Request Tracing
  - Fault Injection

- **Inbound features:**
  - Service authentication
  - Authorization
  - Rate limits
  - Load shedding
  - Telemetry
  - Request Tracing
  - Fault Injection
Using Istio/Envoy
Example: Identity based request routing
Example: Identity based request routing (2/3)
Example: Identity based request routing (3/3)

The Comedy of Errors

Summary: Wikipedia Summary: The Comedy of Errors is one of William Shakespeare's early plays. It is his shortest and one of his most farcical comedies, with a major part of the humour coming from slapstick and mistaken identity, in addition to puns and word play.

<table>
<thead>
<tr>
<th>Book Details</th>
<th>Book Reviews</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: paperback</td>
<td></td>
</tr>
<tr>
<td>Pages: 200</td>
<td></td>
</tr>
<tr>
<td>Publisher: PublishersA</td>
<td></td>
</tr>
<tr>
<td>Language: English</td>
<td></td>
</tr>
<tr>
<td>ISBN-10: 1234567890</td>
<td></td>
</tr>
</tbody>
</table>

An extremely entertaining play by Shakespeare. The slapstick humour is refreshing!
— Reviewer1

★★★★★

Absolutely fun and entertaining. The play lacks thematic depth when compared to other plays by Shakespeare.
— Reviewer2

★★★★★☆
Summary
Summary

- Containers – a lightweight tool for virtualization, rapid development and deployment
  - Web-Scale MicroServices / NFV / Custom Apps on Network-Device
  - Efficient tool chains for containers drove acceptance

- Containers on routers and switches
  - Fast/Custom feature delivery: 3rd-party integration tools (Puppet, Chef, …), Guest-Shell, Your stuff…

- Cloud Native Container Networking
  - Kubernetes – Container orchestration and scheduling
  - Ligato – Cloud-native container control and wiring
  - Contiv-VPP – Container networking and policy fabric with scale and performance
Questions?

Use Cisco Spark to communicate with the speaker after the session

How

1. Find this session in the Cisco Live Mobile App
2. Click “Join the Discussion”
3. Install Spark or go directly to the space
4. Enter messages/questions in the space

cs.co/ciscolivebot#BRKSDN-2115
• Please complete your Online Session Evaluations after each session

• Complete 4 Session Evaluations & the Overall Conference Evaluation (available from Thursday) to receive your Cisco Live T-shirt

• All surveys can be completed via the Cisco Live Mobile App or the Communication Stations

Don’t forget: Cisco Live sessions will be available for viewing on-demand after the event at www.ciscolive.com/global/on-demand-library/.
Continue Your Education

• Demos in the Cisco campus
• Walk-in Self-Paced Labs
• Tech Circle
• Meet the Engineer 1:1 meetings
• Related sessions
Thank you
You're