



JANUARY 2018

CONTAINERS AND CONTAINER NETWORKING

For Network Engineers

1	Introduction	2
1.1	The Digital Transformation Journey	2
1.2	What is Cloud Native?	4
1.3	Do Microservices=Containers and Containers=Microservices?	4
2	Container Basics	7
2.1	Container runtimes and formats	9
2.2	Docker Introduction	10
2.3	Container Operating Systems	12
3	Service Discovery	14
3.1	SkyDNS	15
3.2	Weave-DNS	15
3.3	CoreDNS	15
4	Consensus Management Tools	16
4.1	Consensus Protocols	16
4.1.1	Paxos	16
4.1.2	Raft	17
4.2	Distributed State Machines	17
4.2.1	Zookeeper	17
4.2.2	Etd	18
4.2.3	Consul	18
5	Container Clustering and Scheduling	19
5.1	Docker Swarm	20
5.2	Kubernetes	22
5.3	Mesosphere	24
6	Container Networking	26
6.1	Docker single host networking	26
6.2	Docker multi-host networking	30
6.3	Calico	30
6.4	Weave	31
6.5	Flannel	33
6.6	VMware NSX	34
6.6.1	NSX Native Container Networking	38
6.6.2	Microsegmentation of Microservices	42
6.6.3	Monitoring & Analytics for Microservices	47
7	Conclusion	56

Intended Audience

This document is targeted toward virtualization and network architects interested in deploying VMware NSX® network virtualization solutions in conjunction with containers and solutions that are part of the container ecosystem (ex. PaaS and CaaS platforms).

Revision History

Version	Updates	Comments
1.1	Fixed images/updates to consensus and schedulers	NSX-T 2.1

1 Introduction

This whitepaper provides guidance and information for network engineers wanting to learn more and potentially design environments that leverage container technology. We will also examine the benefits of the VMware NSX-T® platform as it relates to both containers and their associated technologies.

1.1 The Digital Transformation Journey

For many teams involved in IT, including many network engineers focused on the trends and innovations happening within the data center, the last couple of years has seen quite a lot of change. Many alternatives in architectures, protocols, connectivity, operations, virtualization, the emergence of software defined everything, API driven programmability and dev-ops, have drastically altered not only the landscape of companies and partnerships, but provided a platform so that said companies have completely changed the fundamental way they do business.

It's no secret in the modern business landscape that many of the most popular services and companies have completely grasped this shift or disruption in

technology and leveraged various aspects to completely revolutionize the industry that they are in. Let's consider some of the more commonly known examples:

- World's largest taxi company owns no taxis
- World's largest phone company own no telco infrastructure
- World's most popular media owner creates no original content
- World's largest movie warehouse owns no cinemas
- World's largest software vendors don't write many of the apps
- World's largest accommodation provider owns no real estate

It takes significant business acumen, forward thinking, leadership, market awareness, and innovation through new technology adoption to become the worlds largest or most popular anything. Luckily, a number of these companies are quite open and have documented or spoke publically about the changes they've gone through in their DC design and architecture. You can get examples of public and hybrid cloud app deployments, changes in app development process and time to market of products, leveraging of open source software and tools, workflow and lifecycle management, etc. The area we are going to discuss in this paper is around the applications specifically and efficiencies in application development.

Applications, historically speaking, have been created where process, logic, data, and UI design were interdependent. Instead, most of the companies listed above have started creating applications or modular services based on business process. These processes are typically sanctioned (approved by technical and management teams) and provide a tested technique for how a company wants to operate or conduct their business. Let's consider a basic example – an electronic commerce company might have services that would codify how a sales transaction should be processed:

1. Check inventory
2. Check credit
3. Remove from inventory

These three services are great candidates for a reusable container. This containerized service would then be available as an API to development teams that need to include that process in their application. So why is this so important? This containerization concept becomes very attractive, not only because it aligns directly to the business and business outcomes, but also eliminates the need to have isolated complex code that no one besides the author could understand/modify. It

also provides better application scale and agility by utilizing loosely coupled services that can scale up or down independently of other parts of the application. Leveraging containers and containerization in the data center is considered one of the hottest technology trends of 2018. In fact, according to IDC data, by 2020 fifty percent of the Global 2000 will see the majority of their business depend on their ability to create digitally-enhanced products, services, and experiences. Many of the previously mentioned companies are a bit ahead of the curve in this respect, and understand this trend toward digitally enhanced products and services; therefore, are already leveraging containers as a part of their larger application architecture methodology and building many new applications based on a microservices framework.

1.2 What is Cloud Native?

Two constants surrounding the entire conversation with containers and modernized application development are agility and speed. Many areas of IT, most specifically application developers and platform service teams, have begun talking about and looking deeper into the notion of “cloud native apps”. To really define what cloud native means, it’s important to define a couple of characteristics that are usually associated with the term. Most importantly, a cloud native app is an application designed to run optimally in a cloud environment. That means, we assume in a cloud environment everything (network, VMs/systems, storage, etc) can be programmatically provisioned, allowing for full automation using existing tools. The second part of the assumption is that failures are to be expected, which means our application needs to have resiliency built into the framework. Containers, which are designed to be lightweight and be run anywhere (and through the use of orchestrators can be highly available/resilient), align with the goals of cloud native applications and therefore we often discuss the two hand in hand.

1.3 Do Microservices=Containers and Containers=Microservices?

This is quite a common question – as well as misconception, so it’s appropriate to address this right of the bat while discussing microservices as an emerging application architecture. As engineers who have all been through the virtual machine trend, the well understood benefits of VMs as a prevailing infrastructure technology in the areas of efficiency, management, optimization and resource utilization etc. are rarely disputed. Containers themselves are just another type of emerging/modernized infrastructure technology (technically, the technology for

containers goes back a number of years – see the diagram below from the container Wikipedia page):

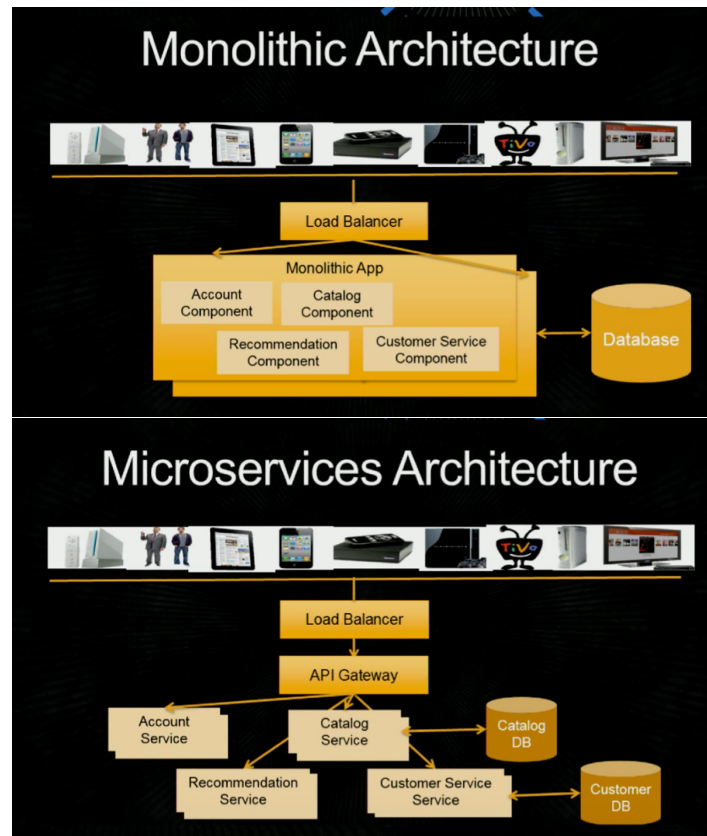
Implementations [\[edit \]](#)

Mechanism ↕	Operating system ↕	License ↕	Available since or between ↕
chroot	most UNIX-like operating systems	varies by operating system	1982
Docker	Linux , ^[7] FreeBSD , ^[8] Windows x64 (Pro, Enterprise and Education) ^[9] macOS ^[10]	Apache License 2.0	2013
Linux-VServer (security context)	Linux , Windows Server 2016	GNU GPLv2	2001
lxc	Linux	Apache License 2.0	2013
LXC	Linux	GNU GPLv2	2008
OpenVZ	Linux	GNU GPLv2	2005
Virtuozzo	Linux , Windows	Proprietary	2000 ^[16]
Solaris Containers (Zones)	illumos (OpenSolaris), Solaris	 CDDL , Proprietary	2004

While containers were developed many years ago, the emerging aspect of the technology is primarily made possible by maturing linux features combined with a broad and every growing ecosystem of companies focused on many aspects of the technology (which are all discussed later in this document).

As mentioned previously, one of the early companies publically discussing their use of containers publically was Netflix. A number of years ago they spoke at multiple conferences about how their application architecture was changing. Some business challenges arose that needed to be addressed - in order to scale systems and deliver content appropriately, as well as remaining competitive in the digital content market place, changes needed to be made. Netflix made a decision to move away from their traditional monolithic application architecture, that many of

enterprises' today still leverage, to more of a microservices based application architecture:

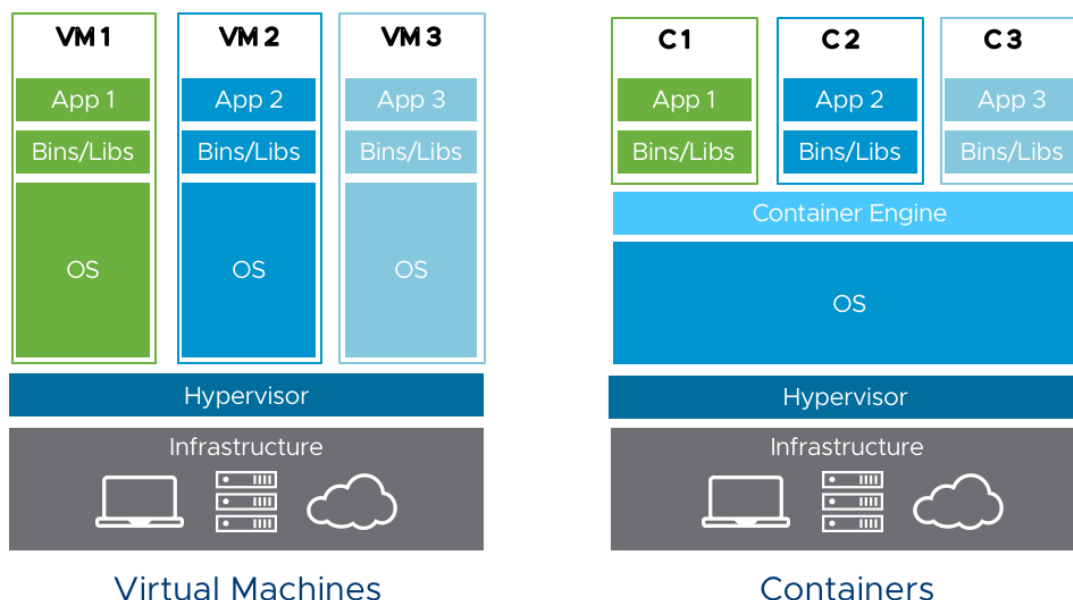


Leveraging containers, as an infrastructure technology, was a choice Netflix could or could not have decided to use to achieve their application architecture goals. The *microservices* piece of the example from Netflix is basically the change that each service (account, catalog, recommendation) became its own “application” with a single function, much like the payment system we talked about earlier, as opposed to having all those functions and their dependencies wrapped inside a large resource hungry single VM instance or bare-metal OS machine build that is used in the monolithic example. Containers are often deemed ideal technology for enabling microservices because the goals of containers (lightweight, easily packaged, can run anywhere) align with the microservices architecture goals and benefits, but they are not the same thing – a microservice may run in a container but it could also run as a fully provisioned VM.

This document is intended for a technical audience and primarily focuses on the technical aspects of containers, container orchestration, and container networking. Areas we will cover in this document include container basics and how they compare to virtual machines, we will also cover the various container formats and runtimes. We will look into operating systems that focus on containers and scheduling for containers as well as other frameworks. Additionally, we will take a look at some key technologies such as Service Discovery and Consensus Management tools that allow these distributed systems to function. Container networking will be the final focused topic, with a brief look across the solutions in the industry that exist for container networking. But more specifically what VMWare is doing with NSX to differentiate itself because it's important to remember, while NSX is viewed as industry leader in network virtualization, it's a lot more than "just networking".

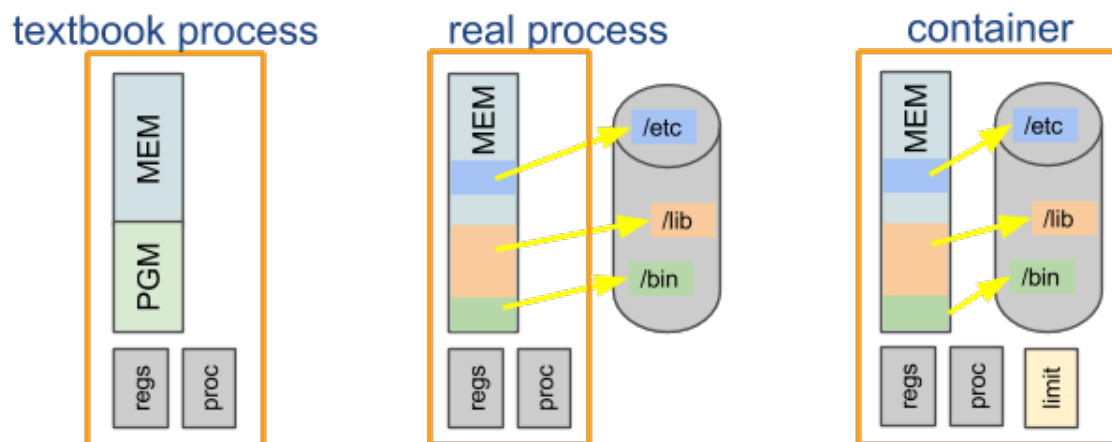
2 Container Basics

In their most basic form, containers are a method of operating system virtualization. Conversely, a virtual machine is a method or form of hardware virtualization. In the diagram below, the hypervisor is responsible for sharing system resources among the guest VMs. Each guest VM running on that hypervisor has its own operating system. Comparing this to the right side of the diagram, containers on a host all share a single operating system.



A fundamental difference between containers and VMs is that a container is a “running instance” of an image. Unlike a VM, you cannot copy it and run it on another host like you would an OVA or VM image file. Most hypervisors (ESX, Fusion, KVM) are considered to be a type of hardware virtualization – the hypervisor virtualizes the physical/compute layer so that multiple machines can share and reside on that hardware. Conversely, containers (LXC, docker) are a form of operating system virtualization (where there is only a single operating system on the physical/compute hardware). Containers are general referred to as ephemeral, which means to last for a very short time. Simply put: containers spin up, perform some task, and then typically are deleted and forgotten. Containers are essentially a process or a running instance of a program. Observing the diagram below, on the left there is a diagram showing a textbook process. In reality, processes need config files, libraries, etc. But a container “unshares” a process and its dependencies from the rest of the system providing isolation.

Containers vs. Processes



Containers accomplish this isolation by leveraging two Linux Kernel features: namespaces and cgroups. Namespaces are available for Interprocess calls (IPC), Networks, Mount points, Process IDs (PID), User and Unix Time Sharing (UTS – Host and Domain name). A namespace can be created for each of these resources and unshared from the rest of the system giving the container its own view of the

system. Cgroups or control groups allow the administrator to place limits on and account for system resources like CPU, memory, disk I/O, network, etc.

2.1 Container runtimes and formats

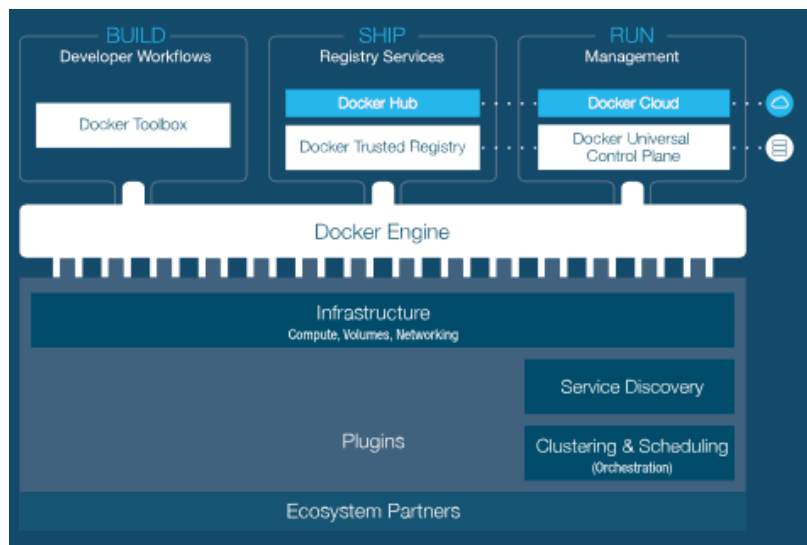
A runtime is an abstraction of the underlying operating systems features like namespaces, cgroups, Union File System, and capabilities that enable containers. The container format is the file system bundle along with associated metadata that the container needs to run. In June of 2015, a large number of companies got together to create the Open Container Initiative (OCI). Docker donated its runtime and container format, referred to as runC, to this effort. Around this same time, CoreOS and their rkt containers were building their own format and runtime called appc. CoreOS and Docker have since decided to work together and incorporate many of the features of appc into runC, so that the industry could adopt a common spec. The key difference between the two runtimes was that Docker starts all new containers under the docker daemon. So, Docker will effectively manage every container. Appc leveraged systemd to start and handle lifecycle of the containers. Today runC can be used with the docker-engine(daemon) or it can use systemd to manage the lifecycle of containers.

In addition to OCI there are still a few other runtimes/formats worth mentioning:

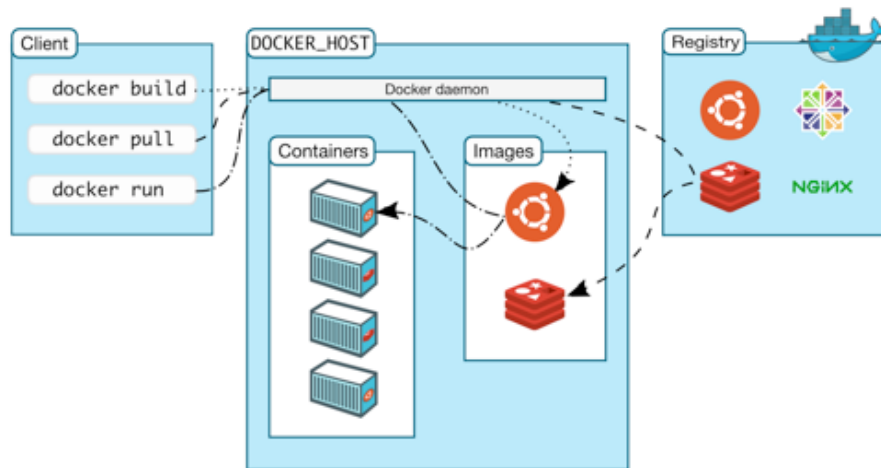
- LXC - this is the “classic” Linux container solution. LXC is an interface to the low level linux tools that enable containers. LXD is a project to improve upon the user experience that LXC offers. LXD provides an improved CLI and a REST API, an Openstack Nova plugin, as well as a Daemon which manages the containers.
- Clear Containers – a container effort by Intel. Clear containers take a completely different approach and leverage Intel VT technologies and KVM. Each Clear container is actually a full Linux operating system with all of the elements that talk to hardware removed. The premise is simple, why should the OS spend any time initializing “pretend” hardware? The result is that Clear containers can boot in under 150ms and have the benefits of VMs in terms of isolation.
- Garden – a long time effort in the marketplace, put together by Pivotal.

2.2 Docker Introduction

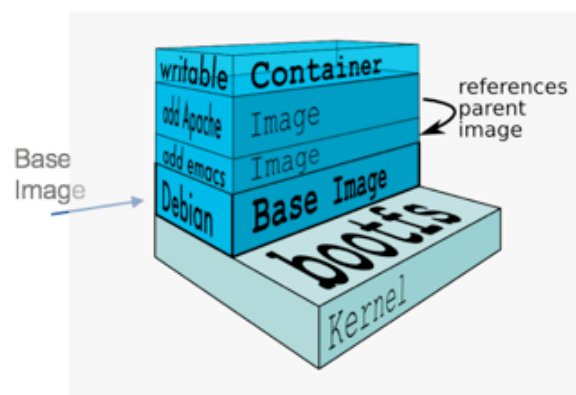
Docker was originally created by a Platform as a Service (PaaS) provider called DotCloud. Docker was designed as tool to allow developers to create an application along with its dependencies, package it, ship it, while allowing it to be able to be ran anywhere. While this flexibility can certainly be desired, in many small to large enterprises, this represents a separation of concerns. For example, the developer needs to ensure that the application works, while the IT/Infrastructure folks are only concerned with running the Docker-engine on a server:



Docker's architecture is simple and is outlined in the diagram below. The Docker client is a CLI interface that communicates with the Docker-Engine (formally called Docker-daemon). The Docker-engine is responsible for managing images and creating containers based on those images. Dockerhub, is an example of an image registry that Docker offers, holds official versions of popular software like Centos, Ubuntu, Nginx, etc. This provided registry gets used quite often due to the simplicity - any developer can create a Dockerhub account and post images of their software in the registry.



A key concept as depicted from the above diagram, in Docker, are the images. Containers are a running instance of an image/program. A Docker image leverages a Linux feature called Union File System. The Union File System allows “layered” images. Let’s consider an example: we have three applications - an Apache web server, a mail server, and a database and they all ran on Ubuntu. With this, we would be able to share the base Ubuntu image among all three applications. Not to mention, since Ubuntu is only ~187MB, this leads to savings in disk size, BW for downloads etc.



The layered file system illustrated in the diagram above shows the container as a running instance of the image and is depicted as a “writable Container”. Any changes the user makes to any files will be written on this layer and “overwrite” the layer below. The diagram example shows down one layer, that the user added “Apache”. One more layer down is “emacs” and finally at the bottom, there is a base layer of “Debian”.

2.3 Container Operating Systems

Containers can run on any Linux Distribution running a Kernel of 3.10 or later; however, many existing distributions (including RedHat and Canonical), including some new entrants, are promoting “slimmed down” versions of their distribution focused only on containers. The solutions do not have traditional package management system like Aptitude or YUM... rather, if you want to add a service or application, you simply run that service in a container. Since there is no package management system there is also no way to upgrade the OS. Most of these vendors offer simple ideas to addressing an upgrade solution – let’s say you have an active version and you went ahead and download an upgraded version... you simply reboot the machine into the upgraded version, and if need be you can perform a rollback to the previous version. In addition, the OS and application files are usually “read-only” to prevent tampering and mitigate security threats as much as possible.



RED HAT ATOMIC Red Hat Enterprise Linux Atomic – Atomic is based on RHEL 7 and has less than 200 installed packages vs. 6,500 for standard RHEL7. Docker and Kubernetes are installed by default. <http://www.projectatomic.io>



snappy Snappy Ubuntu Core – Ubuntu Core is a minimal OS that has been around for some time and runs on anything from a phone to Raspberry Pi. Snappy is a new packaging system that replaces aptitude. Snappy supports “Frameworks” and “apps”. In this system Docker is a Framework and containerized apps can be redistributed by Snappy. Snappy Ubuntu Core does not have docker loaded by default. <http://www.ubuntu.com/cloud/snappy>



RANCHER

RancherOS – is a minimal Linux distribution (less than 20MB) that runs everything in containers. In Rancher OS after the Kernel boots it immediately starts a Docker Daemon known as System Docker. It does not use init or systemd. System Docker then starts whatever services are needed. Rancher then starts “User

Docker” which is a Docker Daemon inside a container. All user containers run inside this container. Since all components of Rancher run in containers they are able to use docker packaging and distribution to deliver upgrades and rollbacks.

<http://rancher.com/rancher-os/>



PHOTON OS™

Photon OS – Photon is an open source project sponsored by VMware and designed for Containers including Docker, rkt, and Garden. Photon is a lightweight OS designed to integrate into vSphere.

<https://vmware.github.io/photon/>



CoreOS – is based on Chromium OS/Gentoo Linux. It does not have a package management system installed and so requires all services to be run as in containers. In addition, the only way to update the CoreOS system is to subscribe to automatic updates. There are three channels; alpha, beta, and stable. Updates are downloaded to the passive partition and when complete there are a couple of options on how quickly to reboot. CoreOS is designed to be clustered together. Several independent servers running CoreOS can operate as one by using CoreOS’ Fleet distributed init system and etcd consensus and discovery tool (these are all discussed later in this document). Fleet and etcd agents run on each node and update the other nodes with state of the server. Fleet is essentially a low-level scheduler of systemd. In early 2017, CoreOS announced its plan to depart from fleet by early 2018, and decided to embrace Kubernetes as the best tool for managing and automating container infrastructure at scale. CoreOS offers its own “enterprise ready kubernetes” solution called Tectonic.

<https://coreos.com/using-coreos/>

3 Service Discovery

The concept of service discovery is not unique to containers, in fact it's been gaining mindshare in mainstream system architecture principles, but given the dynamic nature of containers, it becomes a critical component of the solution. With the ability to scale containers pretty much on-demand, the more and more services running in an environment can expand exponentially; therefore, requiring some sort of mechanism to ensure there are no two services listening on the same port. While at the core, service discovery is as simple as knowing what process in a cluster is listening on a TCP or UDP port, a more modern summarization could be a facilitator of connections to ephemeral services. To that point, when running multiple machines, things get even more complicated across a distributed system (when systems fail, when we look to scale an application but try to use server resources most efficiently, etc.). In docker, there are many use-cases of when we would need to store and retrieve some data related to those services running in the cluster we are working with:

- Service registration refers to the process that stores the host and port a particular service is running on.
- Service discovery refers to the process that will allow others to discover information collected during the registration process.

The main goal of the following service discovery tools is to manage how processes and services in a cluster find and talk to one another. That means for each new instance of a service and/or an application, it would leverage service discovery to identify the current environment and then store that information. The storage itself comes in the form of a registry, typically leveraging a key/value format. The discovery service is typically used in a distributed system, so many of the inherent benefits of having multiple systems – scalability, fault tolerance/HA, distributed systems, etc. can be leveraged in a design.

Additionally, most discovery tools offer some sort of API that can be used by a new service to register itself, as well by others to find information about that service. The following list of tools is not exhaustive. In fact, some of these services can be combined with others in development (confd, registrar) to address additional areas of service discovery. A comprehensive service discovery solution should offer:

1. A consistent service directory/HA
2. A method to register services and monitor health
3. A method to lookup and connect to services

Generally, we can say there are multiple services running together that combine to make service discovery tools – some tools are DNS management based and others that fall under the Consensus management based category. DNS is used in some cases because it's widely known and understood, but in the world of containers, many developers think DNS is insufficient for a number of reasons:

1. DNS is not optimized for real-time changes
2. DNS has no native HA datastore
3. DNS doesn't automatically create/destroy entries
4. DNS designed for known ports (http=80 ssh=22) but not random service ports

SRV records were designed to address #4, but developers who use libraries or APIs don't/can't specify a SRV lookup

3.1 SkyDNS

SkyDNS is a distributed service for announcement and discovery of services, built on top of etcd. It utilizes DNS queries to discover available services. This is done by leveraging SRV records in DNS, giving configuration options around subdomains, priorities, and weights.

3.2 Weave-DNS

The Weave DNS server provides automatic discovery by answering name queries for services in a Weave network. This provides a simple way for containers to find each other via hostnames.

3.3 CoreDNS

CoreDNS is a CNCF incubating project that offers a number of benefits around simplicity, while known for being fast and efficient. It's DNS server can be used in a multitude of environments, but most recently we see customers running CoreDNS as the designated replacement for kube-dns in Kubernetes 1.9 for service discovery.

4 Consensus Management Tools

A number of the larger tech companies who were truly focused on understanding the requirements of building distributed systems decided to look at slightly different approaches. In 2006, Google released a research publication describing their Chubby lock service. Chubby is their distributed lock service, as well as reliable storage for a loosely-coupled distributed system. The project's design emphasis was on availability and reliability, as opposed to high performance. The link for the paper is available below, it describes the initial design and use, compares with actual use, and explains how the design was modified. Chubby implemented distributed consensus based on Paxos to provide a consistent key-value store. This could be used for resource locking, leader elections, reliable low-volume storage, etc.

<http://research.google.com/archive/chubby-osdi06.pdf>

Distributed consensus is considered to be the act of reaching agreement among a collection of nodes cooperating to solve a problem. Open source distributed computing and storage platforms have leveraged consensus algorithms as essential tools for replication and resiliency (removing single points of failure). Consensus protocols are the basis for the state machine replication approach to distributed computing. For example, Apache zookeeper utilizes consensus protocol to achieve fault tolerance by replicating the configuration repository across many servers.

4.1 Consensus Protocols

4.1.1 Paxos

Paxos is a family of protocols for solving consensus in a “network of unreliable processors”. The assumption is that the nodes or the communication medium between them will experience failure. Paxos is relatively older, first published in 1989 named after a fictional legislative consensus system on Paxos island in Greece, and has historically dominated both academic and commercial discourse relative to distributed consensus. But due to this perception that Paxos was too complicated, and it's apparent focus on the academics of how to reach consensus, a simpler alternative was published a few years ago called Raft.

4.1.2 Raft

Raft is a consensus algorithm, much like Paxos, but focused on being very easy to use. In fact, the Raft Consensus Algorithm claims equivalency to Paxos in fault tolerance and performance, but differs in the fact that it's very broken down into many independent sub-problems (like micro services architecture) which makes it very clean and easy to understand. The Raft github page provides lots of additional details on the project and some of its differences, but even very specific details like how Raft manages changes to cluster management have been enhanced and modernized. The protocol uses an approach where joint consensus is reached using two overlapping quorums, defined by the old and new cluster configuration, thus providing dynamic elasticity without disruption to operations. Raft has been positively embraced by the software development community and it's leveraged in many open source implementations written in a variety of languages.

4.2 Distributed State Machines

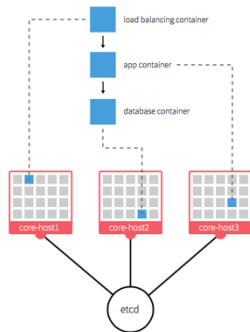
4.2.1 Zookeeper



Apache zookeeper was one of the earliest projects of this type. It originally started as a project to address and assist in the maintenance of the various components of the Hadoop Apache project (open source equivalent of Chubby). Learning from those principles, the project now boasts a centralized service for maintaining configuration and naming information. It also provides distributed synchronization and group services. In fact, consensus, group management, and presence protocols are all leveraged by Zookeeper so that services/applications do not need to implement them on their own.

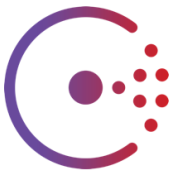
Since Zookeeper has been around for a long time, its biggest advantages come from maturity, robustness, and feature richness. Zookeeper is often criticized however for being overly complex compared to some of the other tools, and it also leverages Java – which has specific dependencies making it much more resource hungry.

4.2.2 Etcd



etcd is a key/value store written in Google Go Lang, developed by CoreOS, that provides simple access via HTTP. Etcd is preferred quite often because it's very easy to deploy, configure and use. Its architecture is fully distributed and provides a configurable system that can be used to build service discovery. Additional benefits include: secure (optional SSL cert auth), reliable data persistence (leverages raft), fast (1000s of writes per instance), very good documentation.

4.2.3 Consul



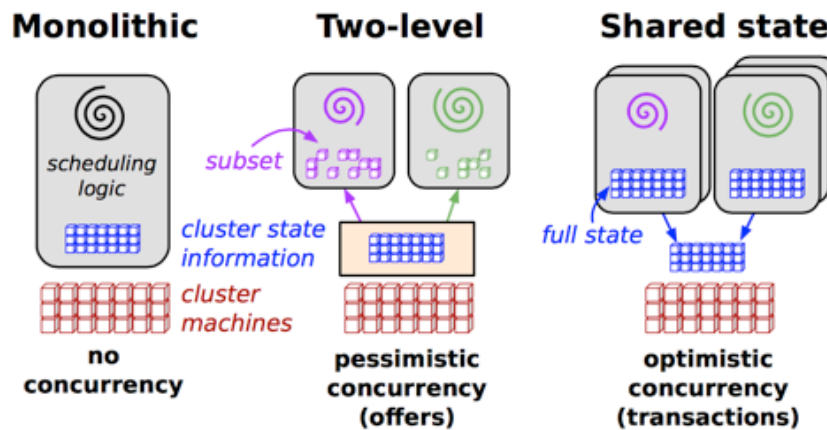
Consul is a consistent datastore that leverages a key/value store that can not only store data about the cluster, but also allows the use of “watches” to support numerous tasks associated around data changes, providing lots of out of the box customization and notifications built-in. Consul contains an imbedded or built-in service discovery framework that includes health checks of the nodes and services, and claims there is no need to run your own or use a third party one. Additionally, Consul's consensus mechanism is based on Raft and leverages a gossip protocol to propagate knowledge of the cluster out of the box, which works and can simplify setup for nodes in the same cluster, as well as across data centers. Main “core benefits” is a focal area for the project, some of these features really make Consul stand out: configurable health checks, ACL functions, HAproxy configuration, multiple DC support, and a useful Web UI.

5 Container Clustering and Scheduling

In the vast majority of environments, customers are going to want to run and manage containers on a group of machines in a similar way to how they manage VMs with vSphere and vCenter today. This is where container orchestration and the various projects focused on this area come into the mix. The Cloud Native Compute Foundation (CNCF) is a Linux Foundation Collaboration Project whose mission it is to standardize orchestration. (<https://cncf.io>) There's a ton of history here with the project, but let's just touch upon the highlights:

- Primary focus on “integrating the orchestration layer of the container ecosystem”
- Kubernetes is the seed technology
- Google contributed it's Kubernetes (k8s) project to CNCF
- For the purposes of this whitepaper, we will focus only on the three (the more widely deployed) projects in this space
 - I. Swarm
 - II. K8s (*in 2018, the leader and de-facto standard container orchestration*)
 - III. Meso Marathon
- Other projects in the ecosystem exist (ex. Rancher's Cattle, Nomad, Shipyard, Amazon's EC2 orchestration service, etc).

Container orchestration systems will generally include load balancing, service discovery, replication/high availability, affinity/anti-affinity but their primary job is scheduling. The orchestration system needs to understand what resources are available on each machine in the cluster. Schedulers are not unique to orchestration systems, applications like Hadoop or Data Center Operating Systems like we saw earlier rely on schedulers as well. There are generally three types of schedulers in use today as illustrated in the diagram from Google's whitepaper on their Omega scheduler below:



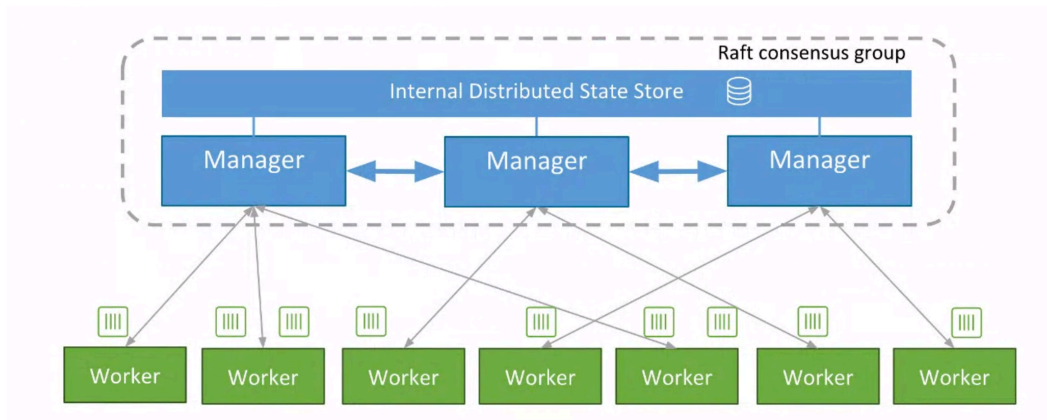
1. Monolithic – A single “all knowing” process that runs on one machine. This is the simplest scheduler and the one used by Docker Swarm, Fleet, and Hadoop.
2. Two-level – separates resource allocation with task placement. This is used by Mesos/Mesosphere. Two level schedulers first identify available resources and offer them to the framework specific scheduler. This allows multiple frameworks to share a given pool of resources and still be able to implement their own schedulers. In Mesos, Hadoop jobs can run side by side with Docker containers. The first schedule makes the offer to a given framework who either accepts or declines the offer. While the offer is being made the resources are locked and not useable by other frameworks. This is referred to as “pessimistic concurrency”.
3. Shared state – these schedulers are the most difficult to implement because every node needs to have the full state of every other node in the cluster. The schedulers have a common view of all available resources and can schedule jobs against them. This is referred to as “optimistic concurrency” and is more efficient than the two-level pessimistic concurrency found in the two-level scheduler. Shared state is used by Google’s Omega, Hashicorp Nomad and Microsoft Apollo.

5.1 Docker Swarm

Swarm is Docker’s own clustering solution. It aggregates a collection of individual hosts running Docker-engines into a “swarm”, presenting itself like a single Docker virtual host. Docker swarm relies on a consensus tool like etcd, that

was discussed previously, to discover the nodes as well as share state among all the nodes in the system. Swarm supports several scheduling strategies including bin pack (most loaded host first), spread (least loaded host), and random (which will not consider a nodes' CPU). In addition, Swarm has the notion of filters which allow containers to be placed based on Affinity/anti-affinity, dependencies, labels, node health, etc.

The high-level Swarm Architecture consist of the following:



Docker CLI – Swarm uses the Docker CLI (making it very easy for someone familiar with Docker to start using Swarm) and this interacts directly with the Swarm Managers.

Swarm manager – Controls the entire cluster, allocates resources. There is an HA feature available which supports creating multiple replicas. A Key/Value store is used to keep the replicas up to date and to elect a leader.

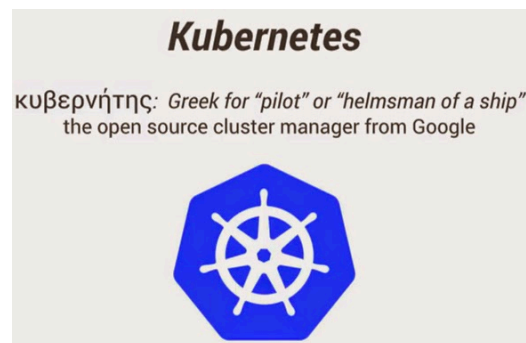
Docker-engine (formally daemon) - This is the same docker engine without swarm and runs on each worker node.

Discovery backend – Swarm supports multiple discovery backends to identify nodes in a cluster:

- I. Text file – list IP addresses of the cluster
- II. Dockerhub hosted token. You can request a token from Dockerhub and then let each host register with that token
- III. Key/Value stores like etcd, Consul, Zookeeper

5.2 Kubernetes

Kubernetes was developed and by Google and is the technology behind the Google Cloud Platform. Kubernetes is open source and is seed technology donated to the Cloud Native Compute Foundation. Kubernetes supports deployments on physical or virtual machines as well as public clouds. It has a substantial list of features including:

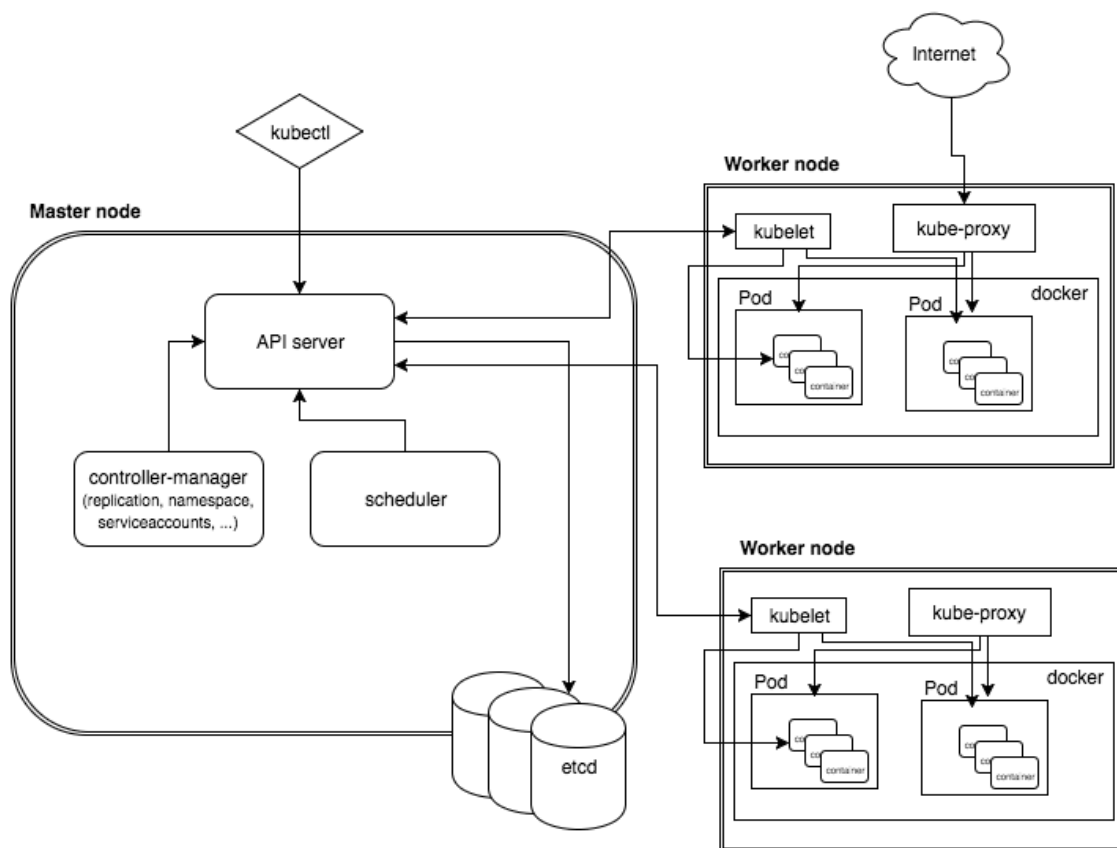


- Application configuration and secret management
- Manual and automatic scaling of applications
- Service discovery
- Load balancing
- Storage orchestration
- Automated rollouts/rollback
- Constraints and filters to control placement of containers
- Statistics gathering

Kubernetes differs from swarm and other container orchestration systems because it has the concept of a “pod”. A Kubernetes pod is a group of containers that form and application and share an IP stack, volume, and context (things like namespaces and cgroups). For example, a pod might consist of a web server container, an App container and a DB container. They would be deployed as a group, share IP and perhaps a volume. The Kubernetes scheduler could be given an instruction to “keep 10 pods running all on different workers” and the scheduler would keep track of the pods and ensure that there are always 10 copies running on unique hosts. If one host fails, Kubernetes will start a pod on another host.

Additionally, service discovery and load balancing are an integral part of Kubernetes. As pods are created they are assigned labels which identify what service they offer. The load balancers are updated with services and their corresponding IPs when pods are scheduled. On every node Google runs a statistics collector in a container called cAdvisor.

The Kubernetes architecture consists of the following components and is outlined in the diagram below:



Kubectl – this is basically the CLI method to interact directly with API server

API Server – this is the target for all operations to the data model

Controller manager – handles replication of pods, embeds core control loops (attempts to move from current state to desired state)

Scheduler – the K8s default scheduler is a monolithic scheduler. However, the K8s architecture falls into shared state scheduling (category 3 discussed above) because you can have multiple schedulers (even of the same type, e.g. kube-default) running, optimistically accessing the shared (etcd) state.

The master can run on any server and can be colocated with worker nodes depending upon the size of the cluster. K8s API servers uses etcd as a highly available durable key-value store. This is the main role of the etcd nodes in the k8s architecture. The worker nodes contain:

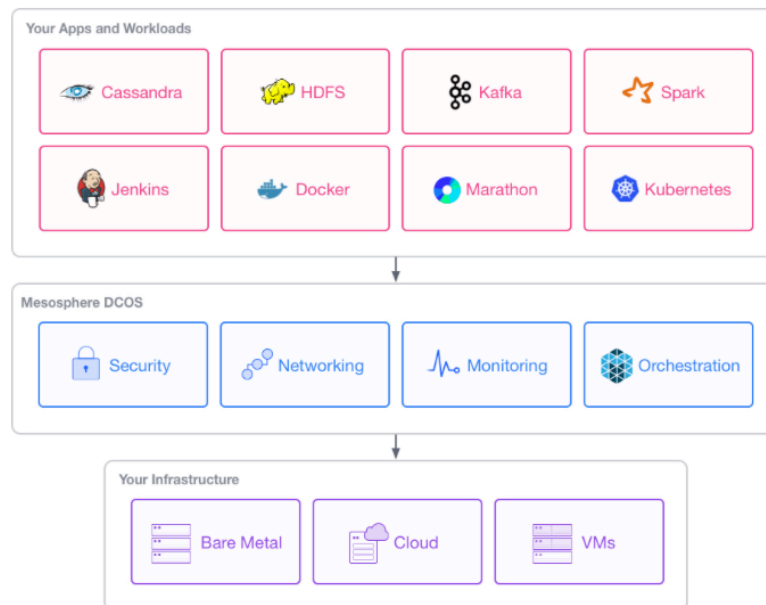
Kublet- this is the agent that communicates with the master Docker-engine

cAdvisor- this is a pod that collects statistics from the node and reports back to the master

Kube-proxy – east/west load balancing mechanism across nodes using NAT in iptables

5.3 Mesosphere

Mesosphere is the commercial product that implements Apache Mesos, more specifically Mesosphere Inc. is the company and DC/OS is their platform. This comes in two flavors – one being open source, “Enterprise” being the commercial platform of Mesosphere Inc. The solution allows multiple frameworks to use a pool of resources in a data center. Mesosphere supports 40 frameworks. Mesos’ actual scheduler, Marathon, deals with scheduling of containers specifically.

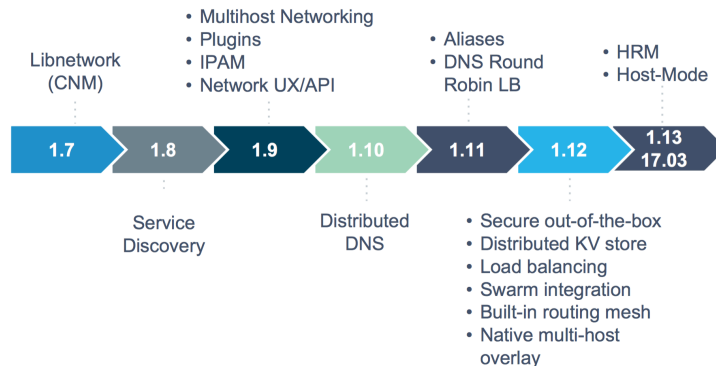


Marathon is an open source container orchestration platform for Mesos as well as Mesosphere DCOS. Marathon was essentially designed to be the scheduler for Mesos environments. In addition to containers, it can also be used to schedule any long running service - including other schedulers like Chronos (a framework to replace cron) Typically in a Mesos environment, Marathon is the first framework started (this essentially replaces init for the Mesos cluster). All long running apps, including containers are started with marathon. Marathon supports the following features:

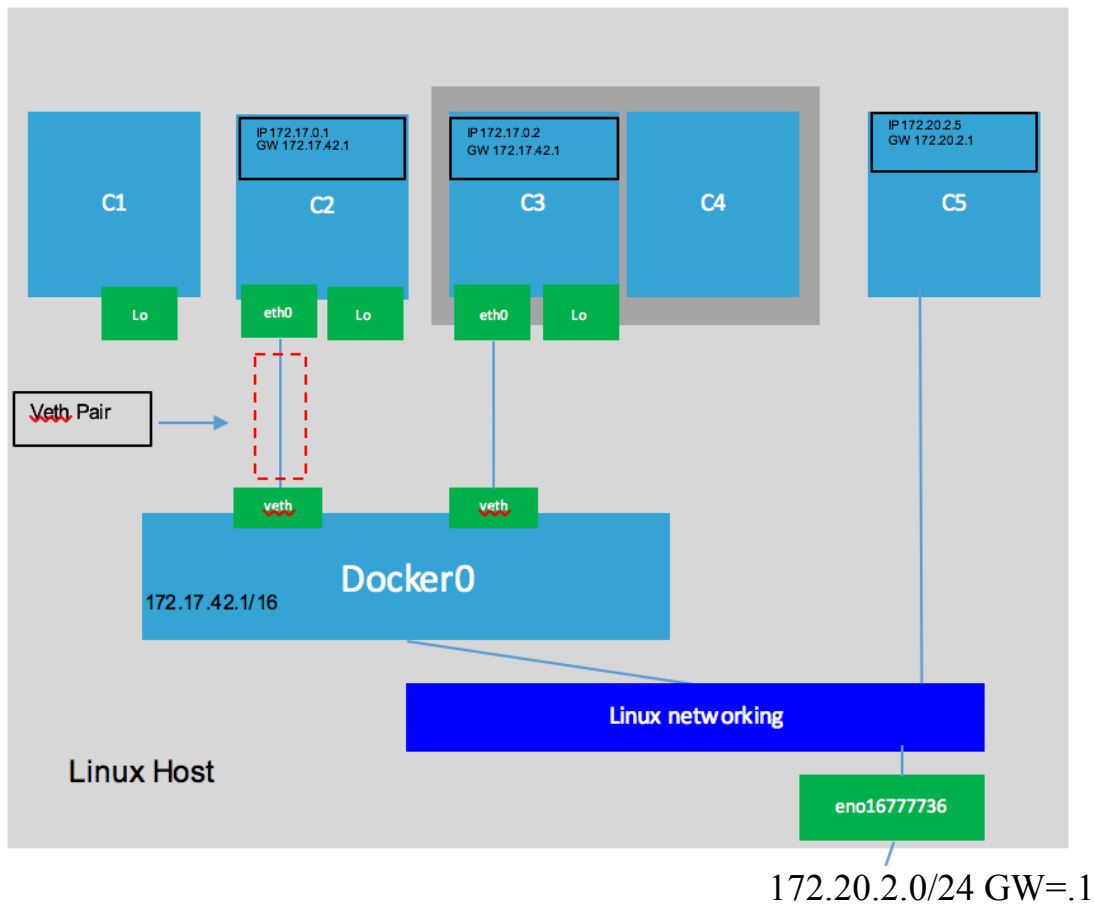
- High availability – active/passive cluster
- Constraints – to influence container placement
- Service discovery
- Load balancing
- Health checks
- Event subscription
- Metrics
- Built-in Web Interface, handy if you prefer a GUI for managing clusters

6 Container Networking

6.1 Docker single host networking

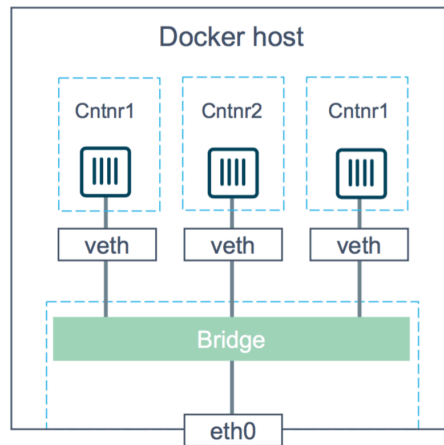


Prior to Docker 1.9 there was very limited support for networking. A container had three options 1) no network 2) Docker0 bridge 3) Linked to another container. In 1.9 and later, Docker has deprecated the option to directly link two containers and replaced it with the ability for containers to share networking. The docker CLI uses the “`—network=`” option to allow the administrator to specify networking. The section below describes options in v1.9 and later.



--net=none – (container C1 in above diagram) the container is provided with a loopback address only.

--net=Bridge – (container C2 and C3 in above diagram) the bridge created by the bridge driver for the bridge network is called docker0. Each container is connected to a bridge network via the Veth pair.



In 1.9 and later you can create a bridge via the docker network create command and specify a variety of options (things like IP address, external connectivity etc). C2 and C3 are basically placed in their own network namespace, are allowed to communicate because they are on the same bridge, but that bridge is a private network restricted to that single host. Simply put, containers on different bridge networks cannot communicate (even within the same host). External access requires port mapping; therefore, Docker will automatically create an IP tables outbound source SNAT rule allowing the container outbound access.

We can see the outbound rule by entering:

```
iptables -t nat -L
```

```
Chain POSTROUTING (policy ACCEPT)
target     prot opt source                destination
MASQUERADE all  --  172.17.0.0/16          anywhere
```

`--net=Container:NAME_or_ID (C4)` This option allows two containers to share networking. This could be useful in cases where apps are bundled or it can be used to conserve IP addresses.

`--net=host(C5)` this option allows a container to share all of the networking with the host.

In order for inbound traffic to reach the container, Docker will create a DNAT entry in iptables. If the container is started with a `-P` flag, the mapping will be automatic in the sense that Docker will either get the ports that should be exposed

from the metadata of the image or from the composition file. In this instance, Docker will also ensure that there is no one else using the port on the host. If the container is started with `-p` the user needs to specify the ports.

`docker run -d -P nginx`

A `docker ps` will show the mappings:

```
[root@Centos7 ~]# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
bf1703556c11	nginx	"nginx -g 'daemon off'"	34 hours ago	Up 34 hours	0.0.0.0:32773->80/tcp, 0.0.0.0:32772->443/tcp
e3ce3991f5c0	nginx	"nginx -g 'daemon off'"	34 hours ago	Up 34 hours	443/tcp, 0.0.0.0:4800->80/tcp

In this case the docker engine has selected unused ports on the host 32773 -> 80 and 32772 -> 443 and mapped them to the nginx container.

We can see the iptables rule by entering the following command:

`iptables -t nat -L -n`

```
Chain DOCKER (2 references)
target     prot opt source                destination
RETURN     all  --  0.0.0.0/0              0.0.0.0/0
DNAT       tcp  --  0.0.0.0/0              0.0.0.0/0          tcp dpt:4800 to:172.17.0.3:80
DNAT       tcp  --  0.0.0.0/0              0.0.0.0/0          tcp dpt:32772 to:172.17.0.4:443
DNAT       tcp  --  0.0.0.0/0              0.0.0.0/0          tcp dpt:32773 to:172.17.0.4:80
```

If the administrator specifies the `-p` option they must also include the port information:

Host port Container port

 \$ `docker run -p 8080:80 ...`

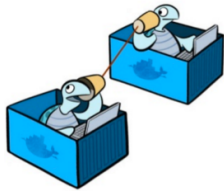
`Docker run -d -p 4800:80 nginx`

```
[root@Centos7 ~]# docker run -d -p 4800:80 nginx
e3ce3991f5c081743ceede6eaa8ebf4a755759b1e941b4675955181d34d3357a
[root@Centos7 ~]# docker ps
```

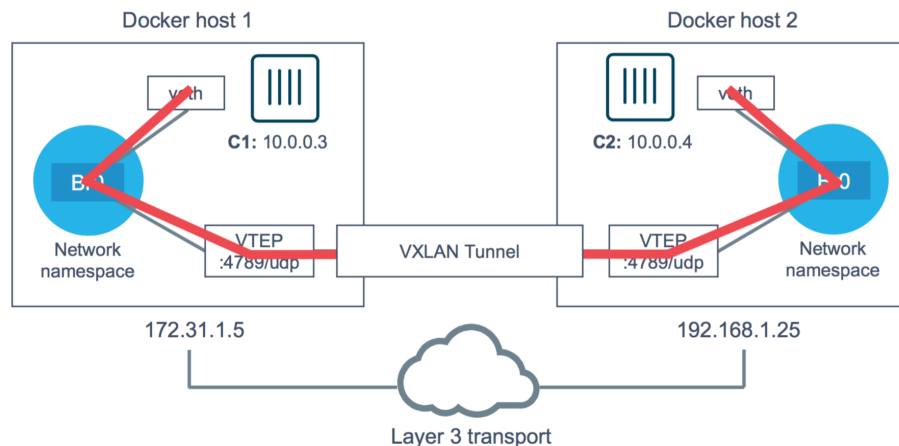
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e3ce3991f5c0	nginx	"nginx -g 'daemon off'"	3 seconds ago	Up 2 seconds	443/tcp, 0.0.0.0:4800->80/tcp	condescending_hopper
065ae8e27b64	ubuntu	"/bin/bash"	About an hour ago	Up About an hour		tender_raman
be165756df6e	_ ubuntu	"/bin/bash"	About an hour ago	Up About an hour		determined_leakey

In the example above we specified a port mapping of 4800:80. Docker will create the iptables rules and docker-proxy just as with the `-P` option but it will not guarantee 4800 is not already in use.

6.2 Docker multi-host networking



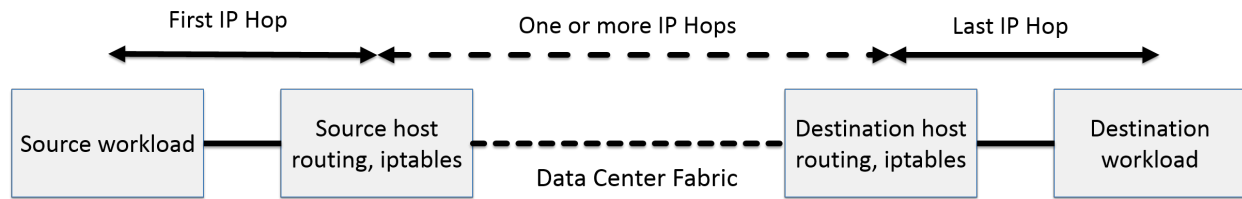
As of Docker 1.9 standard install, Docker Engine supports multi-host networking through the overlay network driver. The support comes from Docker's leverage of libnetwork, a VXLAN based overlay network driver. Docker's built-in overlay network solution is a simple VXLAN implementation - at a basic level, a Layer 2 overlay scheme is built over a Layer 3 network:



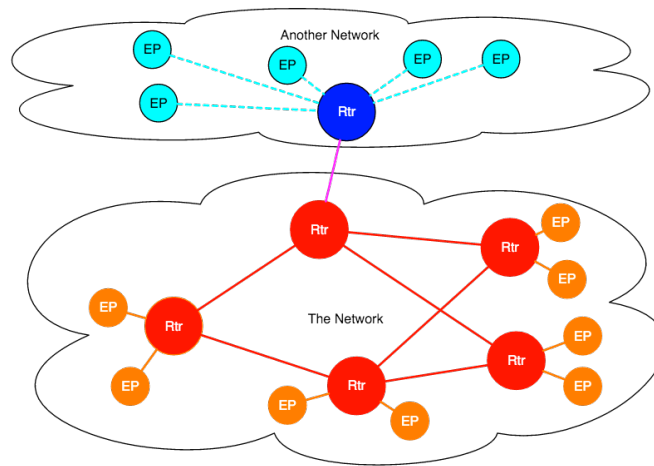
6.3 Calico



Project Calico is an approach to both Openstack and Docker networking. It uses a purely L3 based approach, as opposed to the VXLAN overlay based models of many other container based multi-host networking options. This pure IP based network, combined with BGP for route redistribution, is the key to understanding how Calico scales and builds the virtual network from the host (leveraging native linux kernel for L3 forwarding and ACL enforcement).



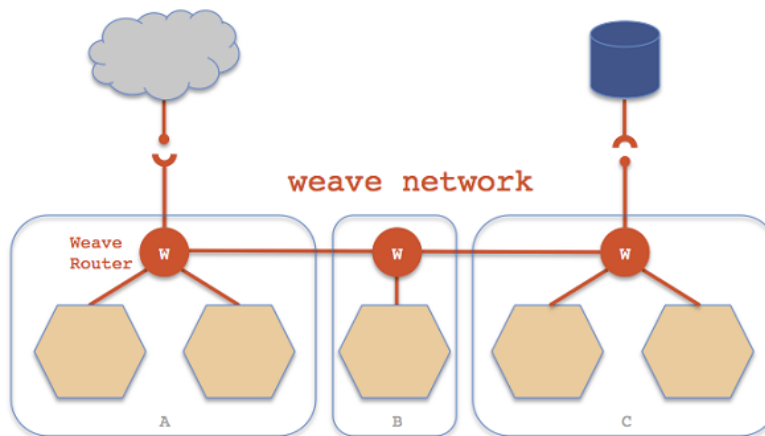
Architecturally, calico is quite simple. Most network engineers are familiar with the “vswitch” concept from the hypervisor part of the compute stack. In calico, instead of relying on vswitches, the project treats each physical server/host as its own VRouter, and then runs BGP (via BIRD, an open source BGP stack) between each one of the routers:



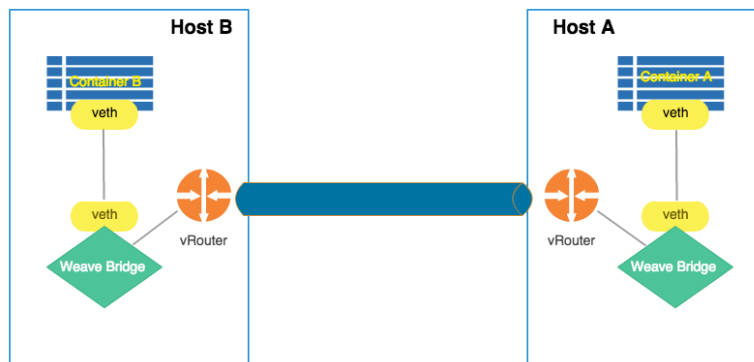
6.4 Weave



Weave, developed by Weaveworks, is another alternative approach to multi-host Docker networking. A weave network consists of a number of peers (aka weave routers) residing on different hosts. Weave routers establish both TCP and UDP sessions between one another. The TCP connections are primarily used for protocol handshaking and topology update exchange. The UDP connections are created and used to carry encapsulated network packets. This particular point is one of the main things Weave touts about their networking architecture. Both TCP and UDP connections can be encrypted:



Let's look a bit deeper into how Weave works. Weave starts by creating a weave network bridge on each host. Each container is directly connected to this bridge. Also connected to this bridge is a weave router container running on each host. The weave router sniffs packets in promiscuous mode (via pcap) from its weave bridge connected interface. When locally switching traffic between containers, and between host and local containers, weave router is not used and the linux kernel routes the packets via weave bridge. The captured packets by weave router are forwarded over the UDP channel to other weave router peers on other hosts.



Additionally, Weave offers an integrated IP address allocator and Weave DNS (which was mentioned earlier) services so that containers can discover each other by hostname (ap1 or ap1.weave.local).

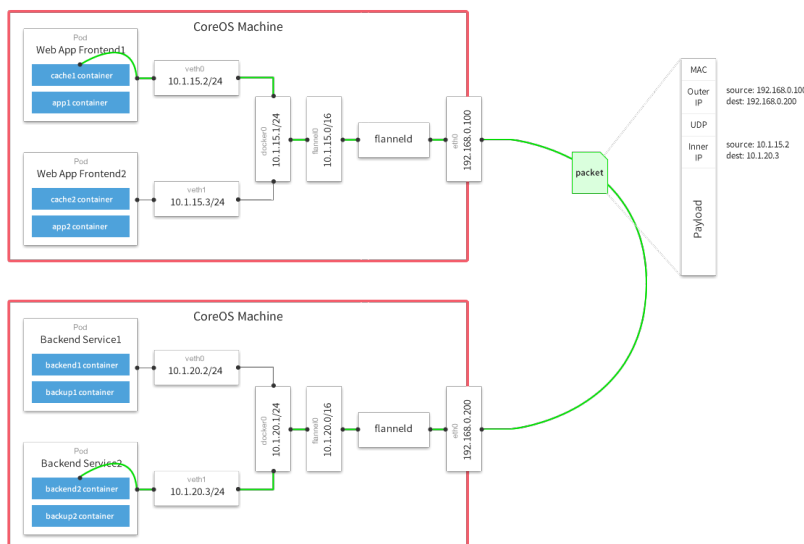
6.5 Flannel



Flannel is another UDP or VXLAN based approach to multi-host Docker networking. Flannel looks to differentiate itself by providing a simple solution so that each container gets a unique IP that can be used for container to container communication. Flannel uses packet encapsulation to create a virtual overlay network that spans every host in the cluster. A flannel agent, `flanneld`, gives each host an IP subnet (/24 by default) from a preconfigured address space, which Docker is then able to allocate IPs to individual containers. Flannel leverages `etcd` to store all network configuration, allocated subnets, and auxiliary data (ex. Host IPs). A `flanneld` daemon runs on each host while is primarily used for routing of packets and watching information in `etcd`. Packet forwarding is achieved by a number of different methods. These strategies are known as backends:

- `udp`: use UDP to encapsulate the packets.
- `vxlan`: use in-kernel VXLAN to encapsulate the packets.
- `host-gw`: create IP routes to subnets via remote machine IPs.
- `aws-vpc`: create IP routes in an Amazon VPC route table.

The easiest and most simplistic backend is UDP. UDP backend encapsulates every IP fragment in a UDP packet, building a simple overlay network between hosts. The diagram below shows this overlay between two hosts:



UDP is the default and most simplistic backend, but some of the tradeoffs might be a worth consideration. Since UDP uses the linux userspace to encapsulate every packet, there might be considerable performance difference, compared to the vxlan backend. The vxlan mode not only offers some hardware features to enhance performance, it leverages the native linux kernel VxLAN support to provide a much faster network.

6.6 VMware NSX

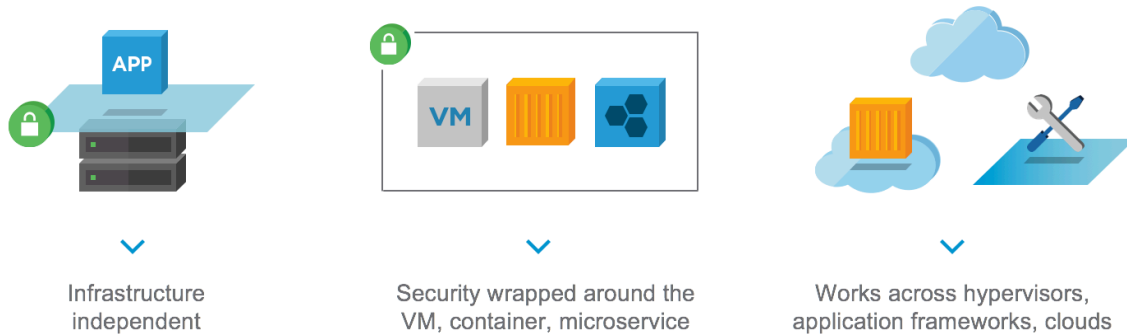


The software-defined data center (SDDC) stack has been embraced for compute and storage functionalities, and for network virtualization, we've certainly come a long way as well. NSX has played a key role in redefining and modernizing networking in the data center. NSX has subsequently emerged as the industry's leader in software defined networking (SDN) by providing a network virtualization and security platform for the enterprise, that has enabled our customers to make this transition to the digital era.

Earlier we covered the concept of digital transformation, and this is one of the key priorities for any CIO in the enterprise. The challenge is when IT receives infrastructure sounding goals and objectives like security of applications and data, speed of delivery, and application high availability, the lightbulb goes off – a traditional enterprise network strategy may not be sufficient any longer a to meet the demands of new approaches in application development and new application architectures (containers, microservices, PaaS).

VMware NSX-T is designed to address these emerging application frameworks and architectures that have heterogeneous endpoints and technology stacks. NSX allows IT and development teams to choose the technologies best suited for their particular applications. NSX is also designed for management, operations and consumption by development organizations – in addition for IT. As developers embrace these newer technologies like containers, and the percentage of workloads running in public clouds increases, network virtualization must expand to offer a full range of networking and security services, native, in these environments. And that's exactly where we are with NSX – a network virtualization solution for a multi-cloud and multi-hypervisor environments. By providing seamless network virtualization for workloads running on either VMs or containers, the NSX

technology you are already familiar with is now supporting cloud and container environments:



The NSX-T architecture is designed around four primary fundamental attributes. These attributes enable greater decoupling, not just at the infrastructure layer itself (hardware & hypervisor), but also at the public cloud (AWS and Azure) and container level (Kubernetes and Pivotal).

Policy and Consistency: this attribute focuses on intended policy definition and realizable end state via RESTful API. NSX-T maintains unique & multiple inventories and controls to enumerate desired outcomes across diverse domains

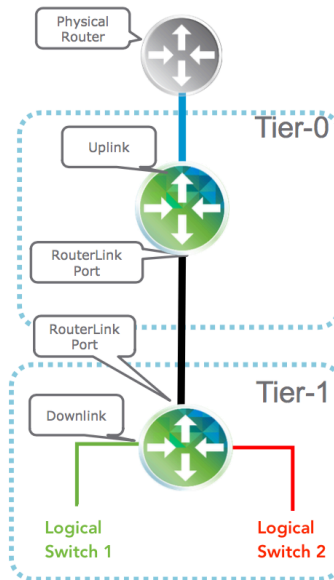
Networking & Connectivity: this attribute is focused on the consistent logical switching and distributed routing supporting multiple vSphere and KVM nodes, without being tied to a compute manager/domain. This connectivity further morphed with containers and cloud via domain specific implementation (k8s or PCF) while still providing connectivity across heterogeneous end points

Security & Services: this attribute allows for a unified security policy model, just like networking connectivity. The services such as load balancer, NAT, Edge FW and DFW across multiple compute domains without lock-in to specific compute. Allowing for consistent security between VMs and container workloads is essential to assuring the integrity of the overall security framework design set forth by security operations

Visibility: this attribute becomes critical for customers attempting to operationalize mixed workloads, like VM and containers. It enables for consistent monitoring, port counters and stats, SPAN, traceflow, etc. via a common toolset, across the domains, that network operations and administrators are familiar with

Before we take a deeper look into how these goals and visions for NSX are working for containers, let's take a quick look at how NSX networking works. NSX leverages an overlay networking model, similar to ones we've discussed prior, but let's focus on what are the differences and solution differentiators. The NSX overlay topology control is enhanced by introduction of GENEVE-based encapsulation. GENEVE is a tunneling mechanism for tunnel end points (TEPs) to carry layer 2 information over Layer 3, while still leveraging the traditional offload capabilities offered by NICs for performance. In the NSX logical network, each of the compute nodes is considered a transport node and will have a TEP. TEPs are the overlay tunnel endpoints that are used to encapsulate and de-encapsulate packets between hosts. One of the primary changes between VXLAN and GENEVE is in the length and types of fields in each packet header. The ability to insert additional context into the overlay header by using Metadata as TLV (Type, Length, Value) unlocks doors for future innovations and new features in context awareness, end-to-end telemetry, security, and encryption. More information on GENEVE can be found on the IETF website (<https://datatracker.ietf.org/doc/draft-ietf-nvo3-geneve/>).

NSX provides next-generation optimized networking (routing and switching), which allows for minimal configuration while supporting both distributed and localized forwarding. As an example, NSX distributed routing allows for routing between subnets on a ESXi or KVM (multi-hypervisor) to be handled in kernel, meaning the traffic never has to leave the hypervisor. Another important out of the box concept - as you can see by the diagram below, the idea of tenancy is built right into the routing model:



Let's define a few important terms to help understand the above diagram:

- **Logical Switch** is a broadcast domain which can span across multiple compute hypervisors. VMs in the same subnet would connect to the same logical switch.
- **Logical Router** provides North-South, East-West routing between different subnets & has two components: Distributed component that runs as a kernel module in hypervisor and Centralized component to take care of centralized functions like NAT, DHCP, LB and provide connectivity to physical infrastructure. The Tier 0 router is owned/configured by the infrastructure teams, and provides a gateway service between logical and physical network. Tier 0 supports dynamic routing and ECMP to the physical and is usually tied to an edge node. Tier 1 logical routers are owned/configured by the tenant, and is always connected to a Tier 0 logical router.

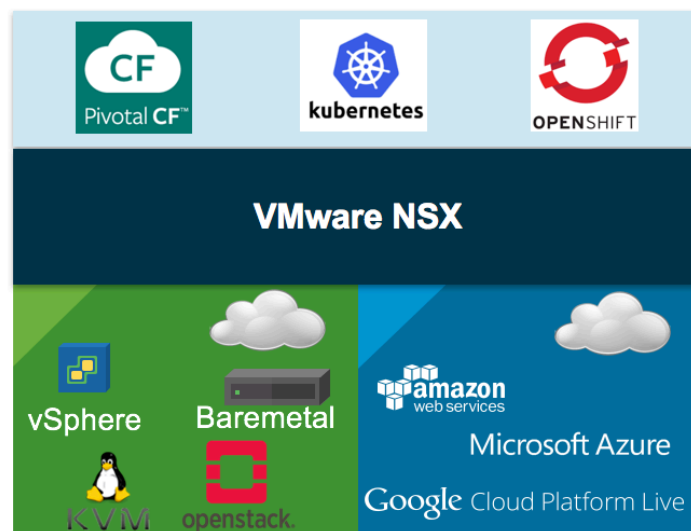
Types of interfaces on a Logical Router

- **Downlink**- Interface connecting to a Logical switch.
- **Uplink**- Interface connecting to the physical infrastructure/physical router.
- **RouterLink**- Interface connecting two Logical routers.

- **Edge nodes** are appliances with a pool of capacity to run the centralized services and would be an on/off ramp to the physical infrastructure. Edge nodes contain or host one or multiple Logical routers to provide centralized services and connectivity to physical routers. Edge node will be a transport node just like compute node and will also have a TEP IP to terminate overlay tunnels. They are available in two form factors: Bare Metal or VM (both leveraging Linux Foundation's Data Plane Development Kit (DPDK) Technology).

6.6.1 NSX Native Container Networking

Now let's take a deeper look into how NSX integrates and supports containers. NSX integration with Platform as a Service (PaaS) and Containers as a Service (CaaS) platforms provides networking, security, monitoring, and multi-tenancy for cloud native and containerized applications. The next few sections will cover an in-depth deeper dive on three use-cases customers are looking to solve surrounding containers and how NSX works with these container platforms – native container networking, micro-segmentation of microservices, and monitoring & analytics for microservices.

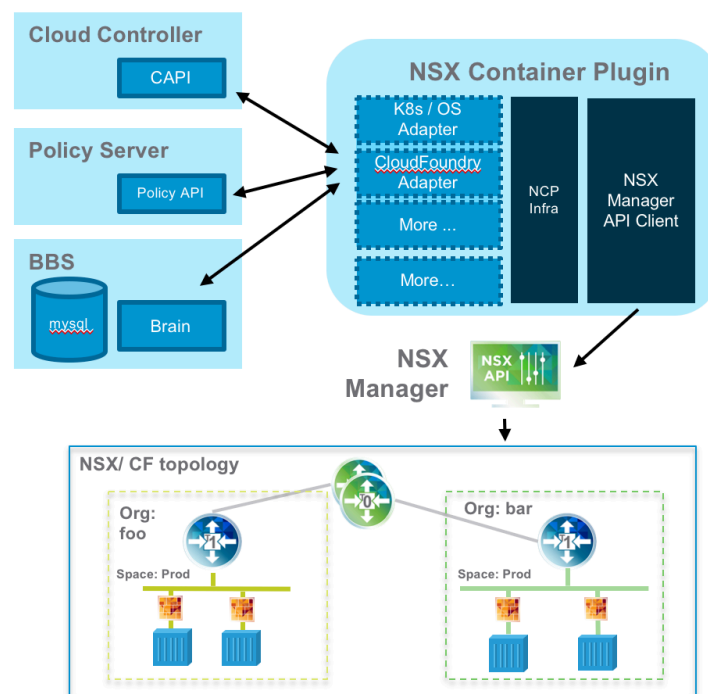


The NSX native container networking integrates with something called the container network interface (CNI). This is a network driver used by Kubernetes, OpenShift, Cloud Foundry and Mesos. The CNI spec is an alternate/opposition

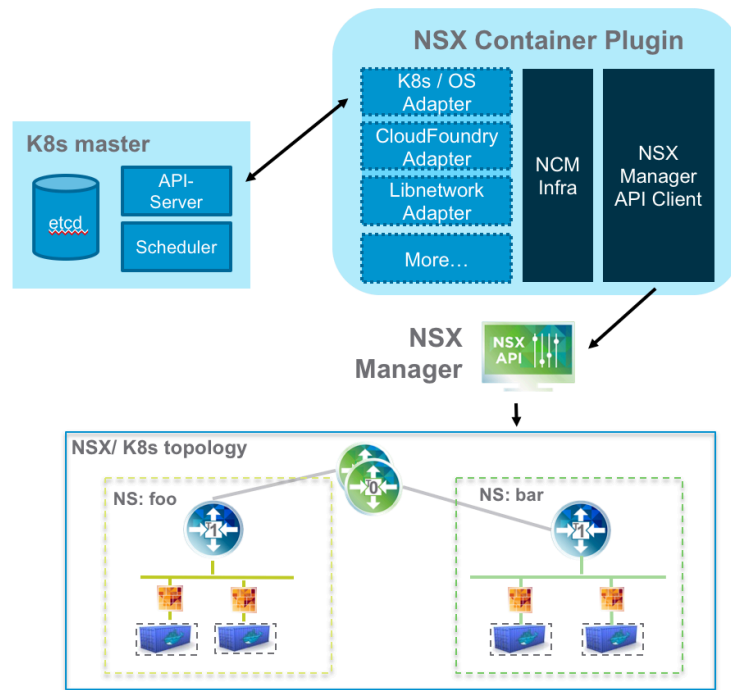
proposal to Docker's libnetwork container network model (CNM) that we discussed previously. NSX supports direct integration with Kubernetes and Openshift, as well as integration with Pivotal Cloud Foundry (PCF). Additionally, NSX is leveraged as the networking option in the Pivotal Container Service (PKS).

The NSX-T Container Plug-in (NCP) was built to provide direct integration with a number of environments where container-based applications could reside. The primary component of NCP runs in a container and communicates with both NSX Manager and the Kubernetes API server, in the case of K8S/OpenShift. NCP monitors changes to containers and other resources, but it also manages networking resources such as logical ports, switches, routers, and security groups for the containers through the NSX API. Below are examples of the interaction between k8s/OS and PCF with NCP:

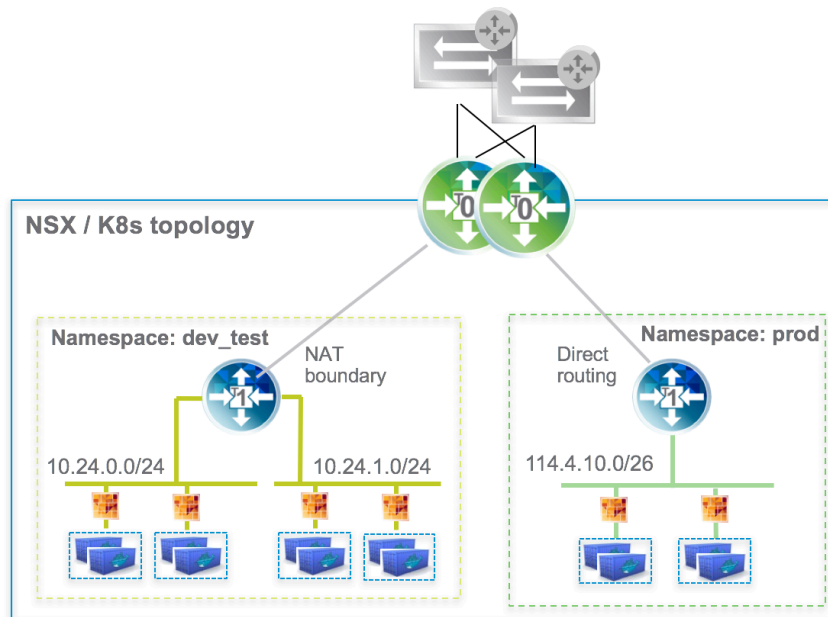
Integration PCF + NSX example



Integration Kubernetes + NSX example

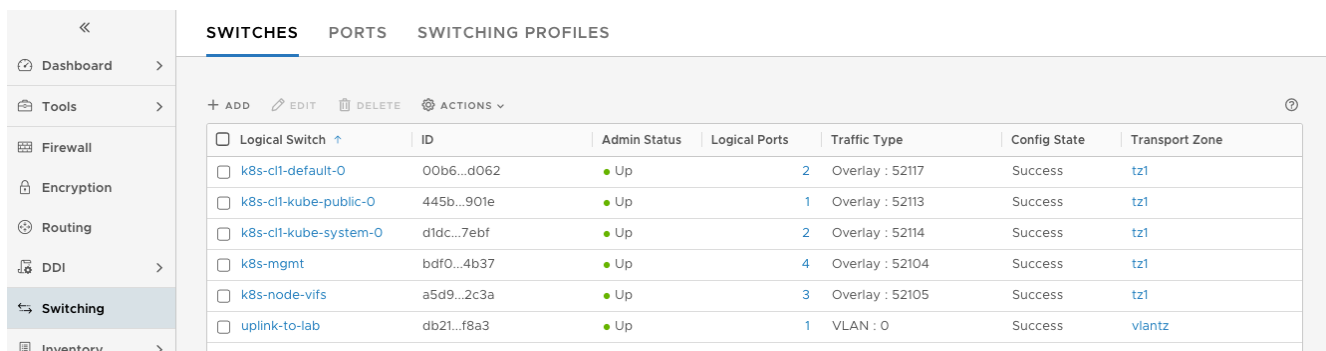


Example Kubernetes NSX topology



In this example topology, the NSX and k8s integration dynamically builds separate network topologies per k8s namespace. This means every k8s namespace gets one or more logical switches (each k8s pod has its own logical port on the NSX logical switch) and one Tier 1 router. The “blue containers” near the bottom represent k8s pods, which are simply a group of one or more containers. Additionally, every node can have pods from different Namespaces and different IP subnets/topologies. The Kubernetes NSX integration takes advantage of the optimized high-performing east/west and north/south routing topology (including dynamic routing to the physical network) that we discussed earlier. From the security standpoint, every pod has distributed firewall (DFW) rules applied on its interface. Further solution highlights include the ability to leverage NSX IP address management (IPAM), which supplies subnets from IP Blocks to k8s Namespaces, and individual IP/MACs to pods. In the example above, we’ve leveraged two separate IP Blocks to provide routed addresses to one Namespace, and non-routable internal NAT’d IPs to the other Namespace. Lastly, NSX brings its network toolbox to the table within this integration, in order to address visibility and troubleshooting in a container networking environment. Most network administrations and operators rely on certain tools and features to do their job effectively, and NSX 2.0 currently supports port counters, SPAN/mirroring, IPFIX, the traceflow/port connection tool, and spoofguard.

Looking deeper into a NSX and Kubernetes setup, we can view the default settings in from a Kubernetes deployment. The standard out of the box k8s includes a default, kube-system, and kube-public namespace. Quickly from the NSX GUI or via API call, we are able to locate the logical switches and Tier 1 routers associated with this basic k8s configuration, and verify our NAT and no-NAT IPAM configuration:



Logical Switch	ID	Admin Status	Logical Ports	Traffic Type	Config State	Transport Zone
<input type="checkbox"/> k8s-ctl-default-O	00b6...d062	Up	2	Overlay : 52117	Success	tz1
<input type="checkbox"/> k8s-ctl-kube-public-O	445b...901e	Up	1	Overlay : 52113	Success	tz1
<input type="checkbox"/> k8s-ctl-kube-system-O	d1dc...7ebf	Up	2	Overlay : 52114	Success	tz1
<input type="checkbox"/> k8s-mgmt	bdf0...4b37	Up	4	Overlay : 52104	Success	tz1
<input type="checkbox"/> k8s-node-vifs	a5d9...2c3a	Up	3	Overlay : 52105	Success	tz1
<input type="checkbox"/> uplink-to-lab	db21...f8a3	Up	1	VLAN : 0	Success	viantz

ROUTERS NAT

Logical Router	ID	Type	Connected Tier-0 Route	High Availability Mode	Transport Zone	Edge Cluster
Central-T0-Router	0029...3a71	Tier-0		Active-Standby	tz1	EdgeCluster1
k8s-clt-default	08af...b685	Tier-1	Central-T0-Router		tz1	
k8s-clt-kube-public	73a5...a766	Tier-1	Central-T0-Router		tz1	
k8s-clt-kube-system	2cdf...df0b	Tier-1	Central-T0-Router		tz1	

IPAM

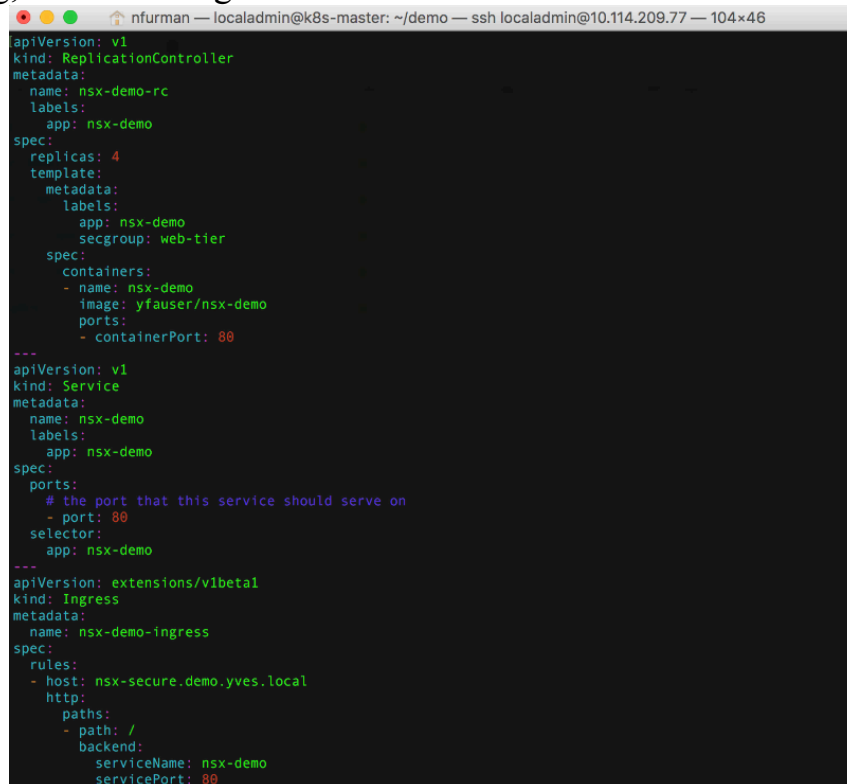
IP Blocks	ID	CIDR
k8s-ip-block1	fa5d...9219	10.4.0.0/14
k8s-no-nat	1c65...cdeb	172.20.0.0/17

6.6.2 Microsegmentation of Microservices

Security is an immensely important conversation today to anyone working in information technology because applications run the business and we've seen very recently with some organizations, a breach can have severe repercussions to the business if things are comprised. Therefore, more and more questions about the safety around these applications and their data get asked daily. Some customers are looking at additional perimeter security measures in the data center, while others have looked at implementing a zero-trust security model within the data center network. Network virtualization helps reduce the risk and supports that higher-level security strategy. That being said, microsegmentation provided through network virtualization paves the way and is one of the key building blocks to a successful zero-trust model. Micro-segmentation is used to decrease the level of risk and increase the security posture of the modern data center. This is achieved through discovery and modeling of the communication patterns between all the machines in an environment, then recommending security policies and distributed firewall rule recommendations. Micro-segmentation also helps with ongoing security operations, audit and compliance, providing real-time visibility into

security group memberships and firewall rules. NSX offers support for the microsegmentation of containers leveraging our NSX and k8s integration.

Let's consider a simple scenario. There's a new Namespace created in k8s called "nsx-secure" and we're going to deploy some container based applications there. The image below shows a simple YAML application spec that calls for four pods to be running, all containing the "nsx-demo" container:



```

nrfurman — localadmin@k8s-master: ~/demo — ssh localadmin@10.114.209.77 — 104x46
apiVersion: v1
kind: ReplicationController
metadata:
  name: nsx-demo-rc
  labels:
    app: nsx-demo
spec:
  replicas: 4
  template:
    metadata:
      labels:
        app: nsx-demo
        secgroup: web-tier
    spec:
      containers:
      - name: nsx-demo
        image: yfauser/nsx-demo
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nsx-demo
  labels:
    app: nsx-demo
spec:
  ports:
    # the port that this service should serve on
    - port: 80
  selector:
    app: nsx-demo
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: nsx-demo-ingress
spec:
  rules:
  - host: nsx-secure.demo.yves.local
    http:
      paths:
      - path: /
        backend:
          serviceName: nsx-demo
          servicePort: 80

```

What's also important to note from this file is that in the metadata, a security group of "web-tier" has been configured. Essentially, this configuration will add a tag to all four of the containers that get spun up and automatically add them into the NSX security group. Let's again verify the configuration. First, we'll verify that the pods have been built:


```

nrfurman — localadmin@k8s-master: ~/demo — ssh localadmin@10.114.2...
Every 2.0s: kubectl -n nsx-secure get all      Sat May 13 17:12:34 2017

NAME                                READY    STATUS              RESTARTS   AGE
po/nsx-demo-rc-09vqf               1/1     Running             0           8s
po/nsx-demo-rc-jsys1               1/1     Running             0           8s
po/nsx-demo-rc-ljff75              0/1     ContainerCreating   0           8s
po/nsx-demo-rc-zq986               1/1     Running             0           8s

NAME                                DESIRED   CURRENT   READY    AGE
rc/nsx-demo-rc                      4         4         3        8s

NAME                                CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
svc/nsx-demo                        10.108.231.21 <none>        80/TCP      8s

```

Then let's take a look at and verify the NSX configuration (the logical switch, the logical ports on the LS, and the security tags):

+

✎

🗑

⚙

▼

☐ Logical Switch
 ☐ k8s-node-ls
 ☐ k8scl-one-default
 ☐ k8scl-one-kube-system
 ☐ k8scl-one-nsx-open
 ☒ **k8scl-one-nsx-secure**
☐ vlan41-uplink

↶

k8scl-one-nsx-secure

Summary

Monitor

Manage ▼

Related ▼

Summary

Namek8scl-one-nsx-secure

ID782ecb03-2c6e-46f7-853f-3e560ef07566

Description

Admin Status● Up

Replication ModeHierarchical Two-Tier replication

VNI50058

Logical Ports5

Traffic TypeOverlay

Transport ZoneGTZ

Created5/13/2017, 8:18:01 PM by admin

Last Updated5/13/2017, 8:18:01 PM by admin

↶

k8scl-one-nsx-secure

Summary

Monitor

Manage ▼

Related ▼

All Ports on Switch

+

ADD

✎

EDIT

🗑

DELETE

⚙

ACTIONS ▼

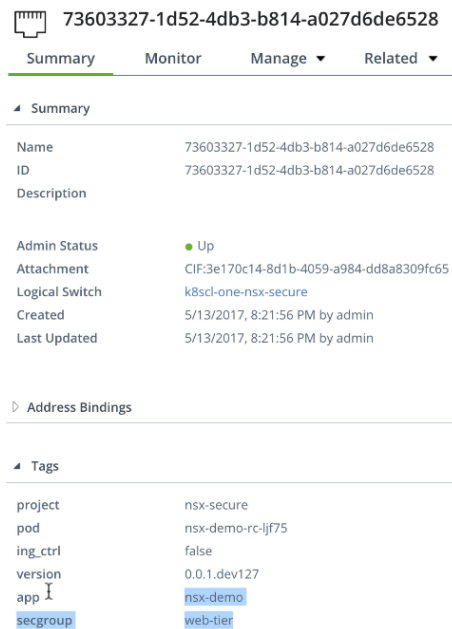
📄

COLUMNS ▼

<input type="checkbox"/> Logical Port	ID	Admin Status	Operational Status	Switching Profiles	Attachment
<input type="checkbox"/> 1721062a-4004-...	1721...6e22	● Up	● Up	nsx-default-swit...	LR:d5ae...b70b
<input type="checkbox"/> 3e83167e-bea4-...	3e83...e8ac	● Up	● Up	nsx-default-swit...	CIF:255e...08fd
<input type="checkbox"/> 73603327-1d52-...	7360...6528	● Up	● Up	nsx-default-swit...	CIF:3e17...fc65
<input type="checkbox"/> b7666078-ff05-4...	b766...9235	● Up	● Up	nsx-default-swit...	CIF:5da5...b32a
<input type="checkbox"/> d9a2dc8b-5f34-...	d9a2...4bbd	● Up	● Up	nsx-default-swit...	CIF:6843...07bd

44

vmware®



73603327-1d52-4db3-b814-a027d6de6528

Summary Monitor Manage Related

Summary

Name	73603327-1d52-4db3-b814-a027d6de6528
ID	73603327-1d52-4db3-b814-a027d6de6528
Description	
Admin Status	● Up
Attachment	CIF:3e170c14-8d1b-4059-a984-dd8a8309fc65
Logical Switch	k8scl-one-nsx-secure
Created	5/13/2017, 8:21:56 PM by admin
Last Updated	5/13/2017, 8:21:56 PM by admin

Address Bindings

Tags

project	nsx-secure
pod	nsx-demo-rc-lj7f5
ing_ctrl	false
version	0.0.1.dev127
app	nsx-demo
secgroup	web-tier

In our simple application example, when defining our zero-trust policy, the app owner knows that none of these web containers should be communicating with one another. Meaning the intent is clear - block all of these containers from using any services (for example http or https) between one another, even though they exist across multiple k8s worker nodes or possibly in the same node. By leveraging the distributed firewall (DFW) feature in NSX, we can construct a simple rule to stop any service from communicating between containers in the source “web-tier” group to a destination of “web-tier” group. Based on the dynamic group membership of our containers in the “web-tier” security group we verified in the previous step, the pods are automatically made members of this group. Since every k8s pod has DFW rules applied directly to it, we can verify quickly that our containers are secure - in the environment, each container has a port scan app embedded to scan and check if port 80 is open for all IP addresses in the same IP subnet, and with this basic DFW configuration in place, our port scan shows that port 80 is indeed blocked for the other container IPs in the subnet:

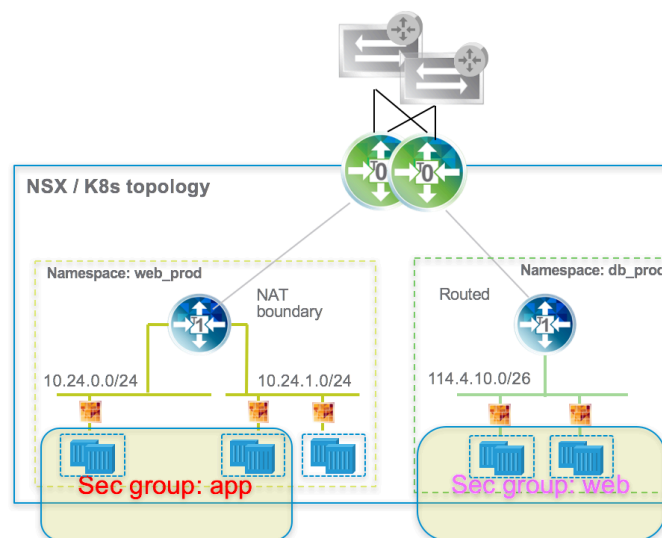
UP DOWN COLUMNS FILTER OBJECTS

	Name	ID	Sources	Destinations	Services	Action	Applied To	Log
⊞	k8s-section (1) Applied To: 0	3f4ca693-daa...						
1	block-web-to-web	1030	web-tier	web-tier	Any	Drop	All	No
s	Default Layer3 Section (1)	820004a4-ce...						

```

Scan of Host: 172.24.47.1, open ports: []
Scan of Host: 172.24.47.2, open ports: []
Scan of Host: 172.24.47.3, open ports: []
Scan of Host: 172.24.47.4, open ports: []
Scan of Host: 172.24.47.5, open ports: []
    
```

The traditional web/app/db 3-tier model that often gets talked about when speaking about application architecture makes sense to most folks. If pods get deployed into multiple security groups based on similar labels or nomenclature, this microsegmentation between k8s pods is a very common use case our customers are looking to automate and solve. Leveraging NSX and the DFW, in our setup, we have provided security policy between our containers inside of the “web” group and between the “app” group:



L3 FIREWALL L2 FIREWALL EXCLUSION LIST SETTINGS									
+ ADD RULE ▾ ADD SECTION ▾ DELETE RULE ↑ MOVE UP ↓ MOVE DOWN ACTIONS ▾ FILTER OBJECTS ⓘ									
#	Name	ID	Sources	Destinations	Services	Action	Applied To	Log	Stats
	Admin-Section	512c8f65-1c10-4526							0 Rules
	hc-lp-k8s-clt-5b87f3d1-5176-4584	2d278ddf-7cc6-48b							1 Rule
	k8s-clt-nsx-demo-policy	7ad40b30-e58e-4c							1 Rule
2	r-k8s-clt-nsx-demo...	2058	si-k8s-clt-nsx-... sg-k8s-clt-nsx-d...	dg-k8s-clt-nsx...	TCP	Allow	dg-k8s-clt-ns...	No	packets: 0 bytes: 0 sessions: 0
	is-k8s-clt-nsx-ujo	93a52be0-3d31-47e3-9ac2-52c3...		Stateful			Applied To: 1		2 Rules

6.6.3 Monitoring & Analytics for Microservices

Software defined networking and new application architectures (like microservices) impose new requirements upon enterprise networking teams and present new ways of developing and managing network architectures. Goals like delivering highly availability service level agreements (SLAs) for mission-critical applications has demanded an increase in visibility and tooling. Consistent uptime, availability, and responsiveness to issues on the network can only be achieved with tools and visibility provided by network virtualization. Here we're going to examine some of the tools and visuals available from NSX:

- Logical Switch/Logical Router detail

The screenshot shows the VMware NSX Manager interface. On the left is a navigation pane with categories like Dashboard, Tools, Firewall, Encryption, Routing, DDI, Switching (selected), Inventory, Fabric, and System. The main area is titled 'SWITCHES' and contains a list of logical switches. The switch 'k8s-cl1-nsx-demo-0' is selected and highlighted. To the right, the 'Overview' tab is active, displaying details for 'k8s-cl1-nsx-demo-0'.

k8s-cl1-nsx-demo-0	
Overview Monitor Manage Related	
Summary EDIT	
Name	k8s-cl1-nsx-demo-0
ID	58dd9f88-4e3b-463a-90d9-acf72a3b1cca
Description	
Admin Status	● Up
Replication Mode	Hierarchical Two-Tier replication
VNI	54165
Logical Ports	10
Traffic Type	Overlay
Transport Zone	tz1
Created	8/10/2017, 12:55:33 PM by admin
Last Updated	8/10/2017, 12:55:33 PM by admin

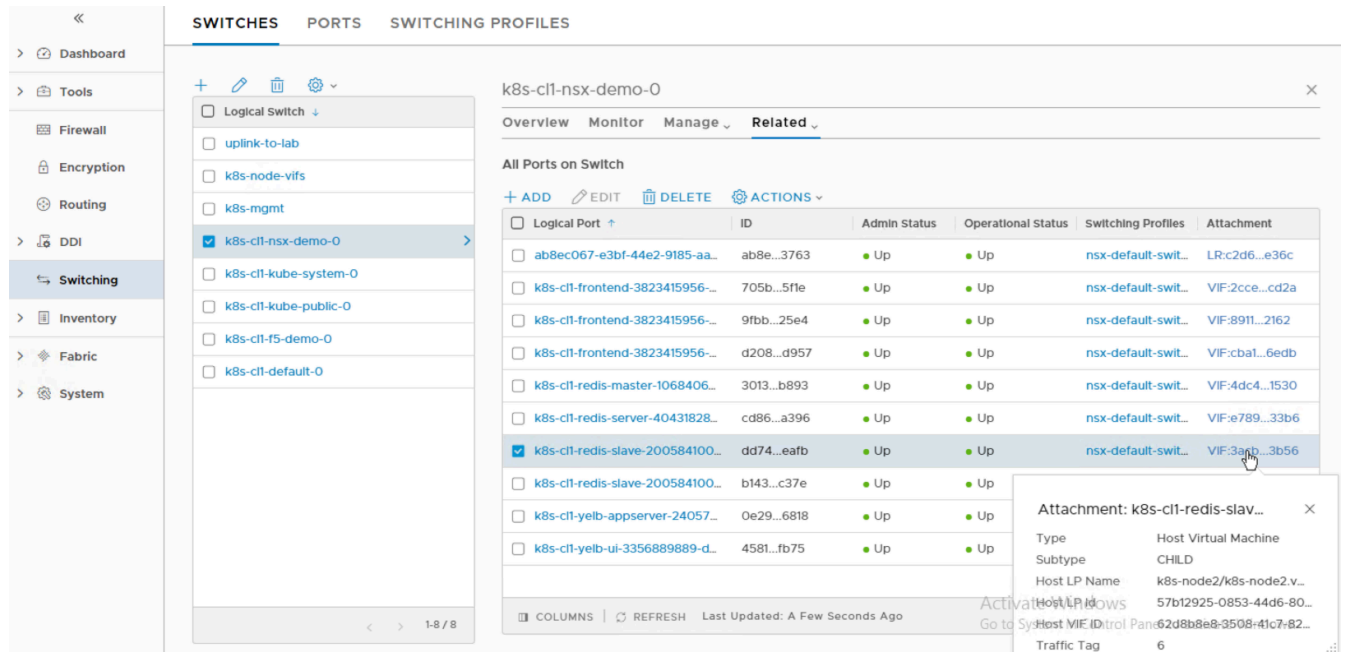
- Logical Switch/Logical Router interface counter (BUM, drops, etc) detail

The screenshot shows the VMware NSX Manager interface with the 'Monitor' tab selected for the logical switch 'k8s-cl1-nsx-demo-0'. It displays various statistics including operational status, VIF traffic statistics, and blocked packets statistics.

Switch Status & Statistics				
Operational Status: Success				
Failure Message: None				
Last Updated at: 8/10/2017, 2:14:06 PM				
VIF Traffic Statistics				
Traffic (Cumulative)	Transmitted Bytes	Transmitted Packets	Received Bytes	Received Packets
Unicast	10.58 MB	41,311	4.28 MB	42,393
Multicast & Broadcast	53.61 KB	768		
Dropped		0		280
TOTAL	10.63 MB	42,079	4.28 MB	42,673

Blocked Packets Statistics		
Traffic / Feature	Blocked By	Packets
IPv6	Spoof Guard	274

- Logical Interface (connected to POD/container) detail

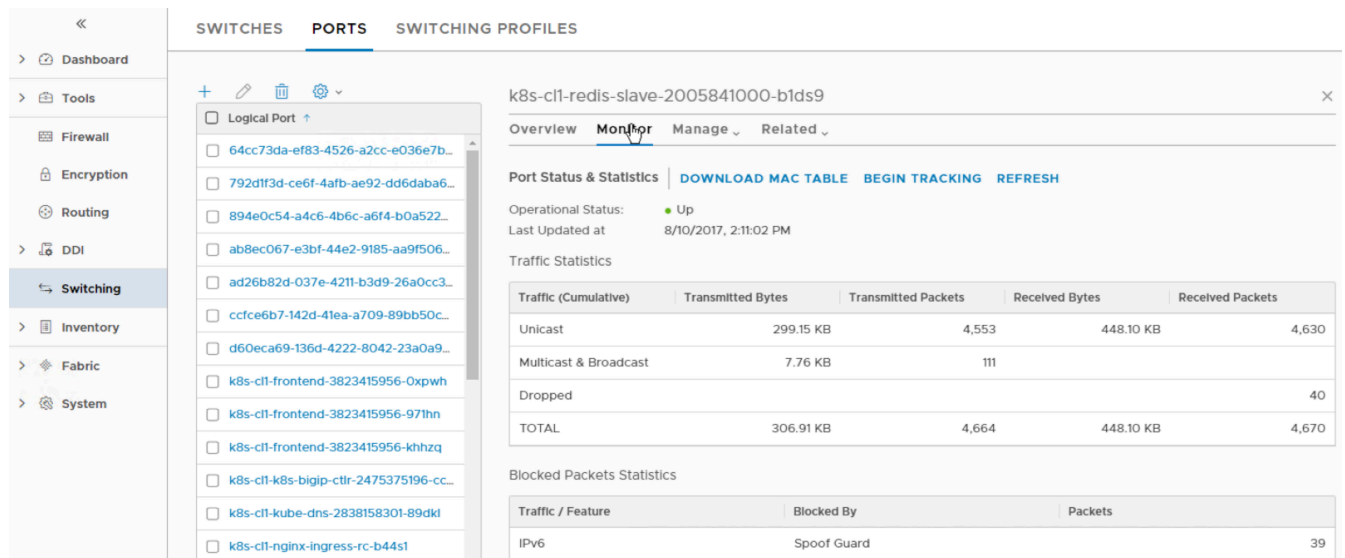


The screenshot displays the VMware NSX Manager interface. On the left, a navigation pane shows the 'Switching' section selected. The main panel shows the 'Logical Switch' configuration for 'k8s-cl1-nsx-demo-0'. The 'Related' tab is active, showing a table of ports on the switch. The port 'k8s-cl1-redis-slave-2005841000-b1ds9' is highlighted, and its details are shown in a pop-up window.

Logical Port	ID	Admin Status	Operational Status	Switching Profiles	Attachment
ab8ec067-e3bf-44e2-9185-aa...	ab8e...3763	Up	Up	nsx-default-swit...	LR:c2d6...e36c
k8s-cl1-frontend-3823415956...	705b...5f1e	Up	Up	nsx-default-swit...	VIF:2cce...cd2a
k8s-cl1-frontend-3823415956...	9fbb...25e4	Up	Up	nsx-default-swit...	VIF:8911...2162
k8s-cl1-frontend-3823415956...	d208...d957	Up	Up	nsx-default-swit...	VIF:cbal...6edb
k8s-cl1-redis-master-1068406...	3013...b893	Up	Up	nsx-default-swit...	VIF:4dc4...1530
k8s-cl1-redis-server-40431828...	cd86...a396	Up	Up	nsx-default-swit...	VIF:e789...33b6
k8s-cl1-redis-slave-2005841000...	dd74...eafb	Up	Up	nsx-default-swit...	VIF:3anp...3b56
k8s-cl1-redis-slave-2005841000...	b143...c37e	Up	Up		
k8s-cl1-yelb-appserver-24057...	Oe29...6818	Up	Up		
k8s-cl1-yelb-ui-3356889889-d...	4581...fb75	Up	Up		

Attachment: k8s-cl1-redis-slav...
 Type: Host Virtual Machine
 Subtype: CHILD
 Host LP Name: k8s-node2/k8s-node2.v...
 Host VIF ID: 57b12925-0853-44d6-80...
 Traffic Tag: 6

- Logical interface (connected to POD/container) interface counter detail



The screenshot displays the VMware NSX Manager interface. On the left, a navigation pane shows the 'Switching' section selected. The main panel shows the 'Logical Interface' configuration for 'k8s-cl1-redis-slave-2005841000-b1ds9'. The 'Monitor' tab is active, showing traffic statistics and blocked packets statistics.

Traffic (Cumulative)	Transmitted Bytes	Transmitted Packets	Received Bytes	Received Packets
Unicast	299.15 KB	4,553	448.10 KB	4,630
Multicast & Broadcast	7.76 KB	111		
Dropped				40
TOTAL	306.91 KB	4,664	448.10 KB	4,670

Traffic / Feature	Blocked By	Packets
IPv6	Spoof Guard	39

Port connection tool – this tool is a visualization of connectivity between two container logical ports. As the topology is built, realized state data like machine information, logical port status, and tunnel health status, gets represented as hop by hop connectivity between various points in the path.

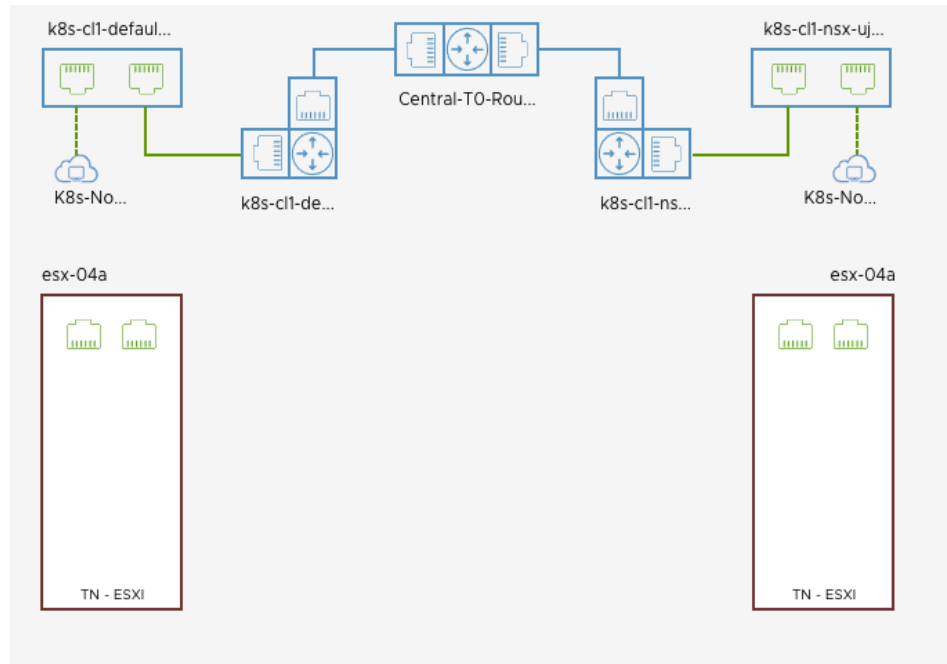
- Define logical source port and destination port

The screenshot shows the 'PORT CONNECTION' tool interface. On the left is a sidebar with navigation links: Dashboard, Tools (expanded), Port Connection (selected), Traceflow, Port Mirroring Session, IPFIX, and Firewall. The main panel is titled 'PORT CONNECTION' and contains two columns: 'Source' and 'Destination'. Each column has a 'Type' dropdown set to 'Logical Port' and a 'Port' dropdown. The Source Port is 'a0bca7ef-64d4-4060-8cd7-918c15aa4dac' and the Destination Port is 'eaf3f607-5ede-4f8e-9f82-9c78febf42f'. A blue 'GO' button is located below the Source Port dropdown.

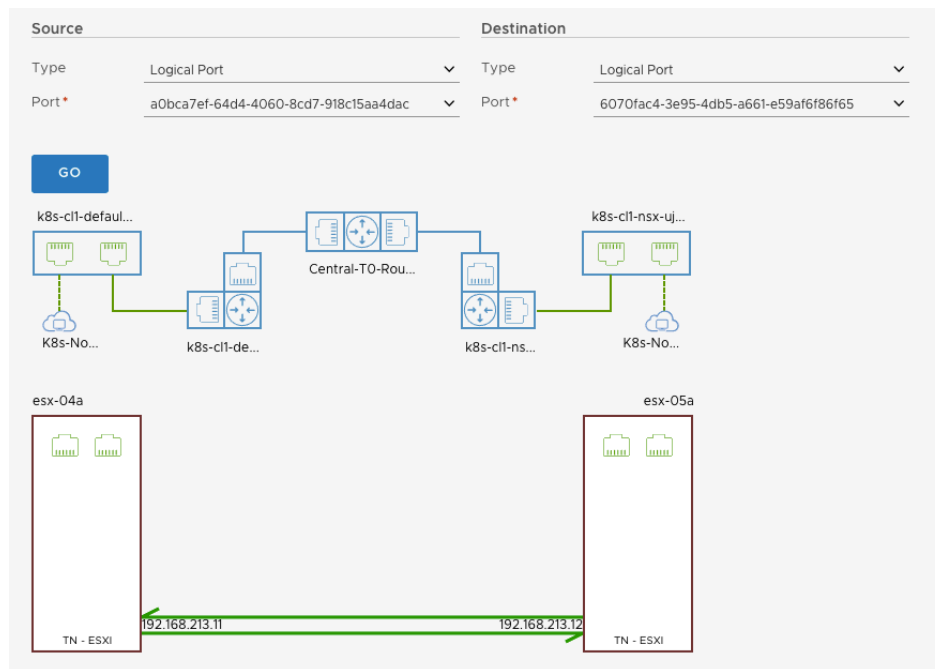
Source		Destination	
Type	Logical Port	Type	Logical Port
Port *	a0bca7ef-64d4-4060-8cd7-918c15aa4dac	Port *	eaf3f607-5ede-4f8e-9f82-9c78febf42f

GO

- Path output between two containers on same host machine



- Path output between two containers on different host machines (green is overlay)



Traceflow – this tool expands upon the visualization of connectivity between two logical ports by allowing an administrator to inject user-defined packet types into the network topology

- Define traceflow between ingress pod interface and web pod interface

TRACEFLOW

Traffic Type: Unicast

Source		Destination	
Type	Logical Port	Type	Logical Port
Port	a0bca7ef-64d4-4060-8cd7-918c15aa4dac	Port	eaf3f607-5ede-4f8e-9f82-9c78febfb42f
IP Address	10.4.0.67	IP Address	10.4.0.101
MAC Address	02:50:56:00:50:02	MAC Address	02:50:56:00:50:05

> Advanced

TRACE

- Packets between ingress pod and web pod are dropped by DFW

TRACEFLOW

Source: Port a0bca7ef-64d4-4060-8cd7-918c15aa4dac, IP/MAC 10.4.0.67/02:50:56:00:50:02

Destination: Port eaf3f607-5ede-4f8e-9f82-9c78febfb42f, IP/MAC 10.4.0.101/02:50:56:00:50:05

RE-TRACE **EDIT** **NEW TRACE**

Trace Results

Show: **ALL** 0 DELIVERED 1 DROPPED

Physical Hc	Observation Type	Transport Node	Component
0	Dropped by ...	esx-04a	eaf3f607-5ede-4f8e-9f82-9c78feb...

- Define traceflow between container interface and L3 IP outside NSX network

TRACEFLOW

Traffic Type: Unicast ▼

Source

Type Logical Port ▼

Port eaf3f607-5ede-4f8e-9f82-9c78febfb42f ▼

IP Address 10.4.0.101

MAC Address 02:50:56:00:50:05

Destination

Type IP - MAC ▼

IP Address 192.168.110.10

Trace Type ☐ Layer 2 ☒ Layer 3

> Advanced

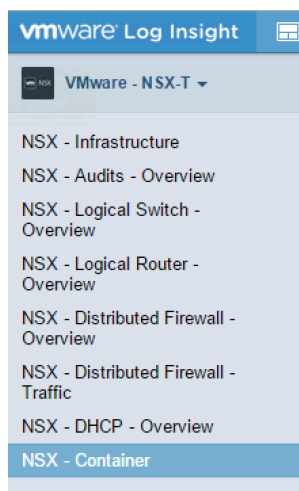
TRACE

- Packets between web pod and external to NSX IP address are successful

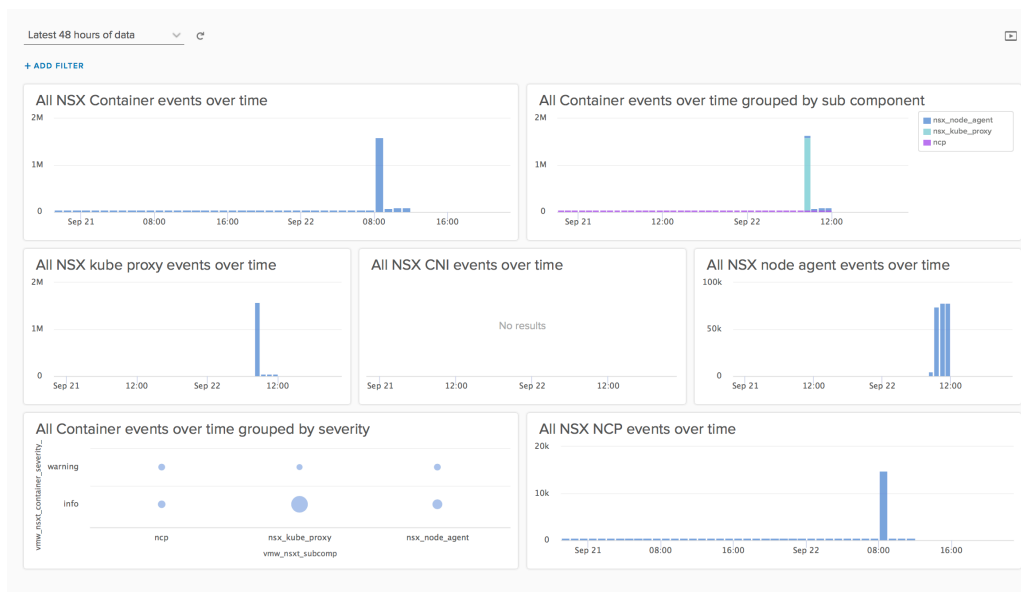
TRACEFLOW

Source	Destination		
Port	eaf3f607-5ede-4f8e-9f82-9c78febfb42f	IP/MAC	192.168.110.10/FF:FF:FF:FF:FF:FF
IP/MAC	10.4.0.101/02:50:56:00:50:05		
Trace Results			
Show: ALL 1 DELIVERED 0 DROPPED			
Physical Hop Count	Observation Type	Transport Node	Component
0	Injected	esx-04a	
0	Received	esx-04a	Distributed Firewall
0	Forwarded	esx-04a	Distributed Firewall (Rule ID: 2053)
0	Forwarded	esx-04a	k8s-clt-nsx-uj-o
0	Received	esx-04a	k8s-clt-nsx-uj-o
0	Forwarded	esx-04a	k8s-clt-nsx-uj-o
0	Forwarded	esx-04a	transit-rl-b9ab137f-3418-4caa-a49a-7b6537e6a57a
0	Received	esx-04a	Central-TO-Router
0	Forwarded	esx-04a	Central-TO-Router
0	Received	esx-04a	transit-bp-0029b8d1-4f28-48e4-9e2b-28185e4b3a71
0	Forwarded	esx-04a	Remote IP : 192.168.213.10
1	Received	edge.corp.local	Remote IP : 192.168.213.11
1	Received	edge.corp.local	Edge Tunnel
1	Forwarded	edge.corp.local	transit-bp-0029b8d1-4f28-48e4-9e2b-28185e4b3a71
1	Received	edge.corp.local	Central-TO-Router
1	Delivered	edge.corp.local	Uplink-to-Lab

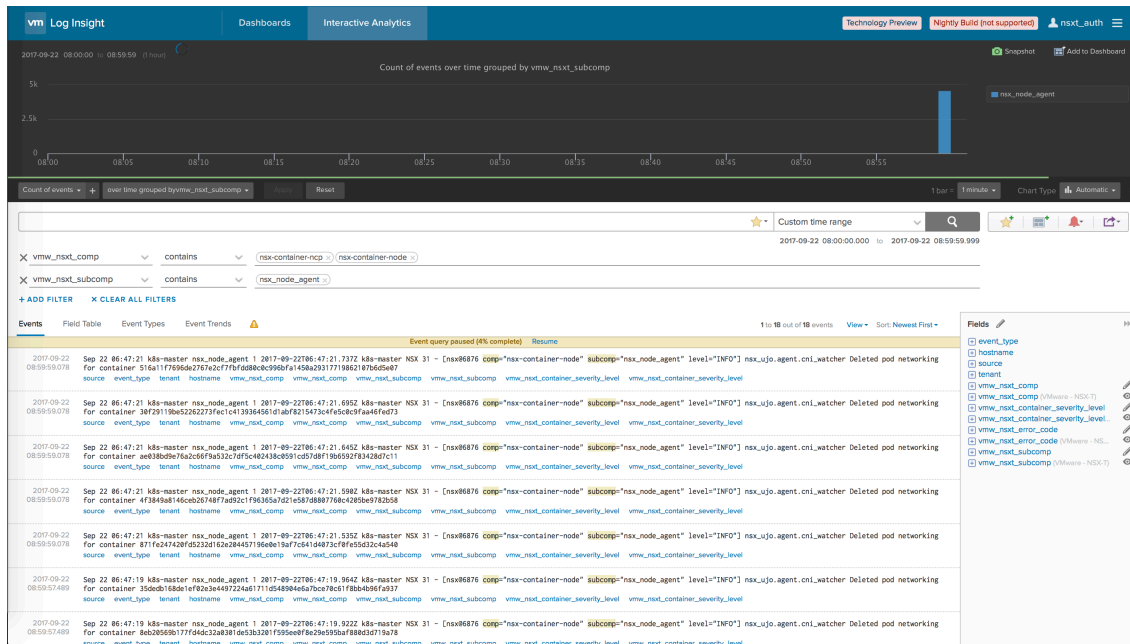
vRealize Log Insight – this tool from VMware is focused on intelligent log management and analytics. Log Insight delivers heterogeneous and highly scalable log management with intuitive, actionable dashboards, sophisticated analytics and broad third-party extensibility. It provides deep operational visibility and faster troubleshooting across physical, virtual and cloud environments. LogInsight has long supported NSX, and a LogInsight content pack for NSX and containers is supported:



- Dashboard showing all NSX Container events



- Searchable raw log data from all NSX Container related events



Container Networking Solution Comparison

Feature/criteria	Calico	Weave	Flannel	NSX
Built-in visibility and operational tools				•
Option/feature to encrypt data path		•		•3
Great for BGP enthusiasts	•			
Enterprise support		•1	•2	•
Native microseg/firewall support	•	•		•
Support for NAT and No NAT topologies				•
Relatable Source IP				•

1 only when using their Weave cloud offering
2 only supported with CoreOS Tectonic distro
3 DNE (distributed network encryption) supported with NSX-T

7 Conclusion

The VMware NSX® network virtualization and security platform can help organizations achieve the full potential of cloud-native apps and bring a number of benefits to the table. NSX enables advanced networking and security across any application framework, helps speed the delivery of applications by removing bottlenecks in developer and IT team workflows, enables micro-segmentation down to the microservice level, enhances monitoring and analytics for microservices, and has reference designs to help organizations get started. It enables a single network overlay and micro-segmentation for both VMs and containers as well as common monitoring and troubleshooting for traditional and cloud-native apps. Additionally, NSX integrates with existing tools in the data center and public cloud for IT teams, while easily plugging into the container ecosystem to empower developers without slowing them down.

NSX empowers both developers and IT teams to work together to the benefit of both, as well as the businesses they support, by enabling common networking, security, workflows, and management across any device, any app, any framework, and any infrastructure. Increased speed and agility for developers coupled with increased connectivity, security, visibility, and control for IT teams mean that the entire organization can operate in tandem to drive the digital transformation of their businesses forward.