Networking in Containers and Container Clusters

Victor Marmol, Rohit Jnagal, and Tim Hockin

Google, Inc.
Mountain View, CA USA
{vmarmol,jnagal,thockin}@google.com

Abstract

Containers are becoming a popular way of deploying applications quickly, cheaply, and reliably. As with the early days of virtual machines, a variety of container networking configurations have been proposed to deal with the issues of discovery, flexibility and performance. Using Docker we present available networking configurations along with many of the popular networking setups, their uses, and their problems today. A second aspect we will explore are containers in clusters, as these systems rely even more heavily on the network. We use Kubernetes as an example of this type of system. We present the network setups of the core components of Kubernetes: pods and services.

Keywords

containers, Kubernetes, Docker, libcontainer, cgroups, namespaces, ipvlan, pods, microservices, container cluster, container manager.

Introduction

Containers, as a user-space abstraction, have been around for many years. At Google, our earliest uses of containers started around 2004 with basic chroot and ulimit setups and enhanced in 2006 with the introduction of cgroups [1]. Docker's [2] release in 2013 brought an ease of use to containers that have helped them to become ubiquitous. Developers targeting all kinds of applications, from large distributed systems to embedded networking components, are turning to containers (and Docker containers) to build, run, and deploy these applications. A common trend amongst all these varied uses is the heavy emphasis on network-based communications. More and more we see microservice architectures as the prefered way to use containers. This has reached a point where almost all container-based applications we see today rely on the network to perform their roles. The networking configurations used to achieve this communication are almost as varied as the uses themselves. This paper is a case study of networking in containers and container-based clusters today. Specifically we will explore the most popular networking setups for Docker containers. We will also explore the networking setups used in container-based clusters by taking a deeper look at the networking aspects of Kubernetes, Google's container cluster manager [3].

In this paper, when we refer to a container we are referring to a group of processes that are isolated via a set of Linux namespaces, cgroups, and capabilities.

History

The first container versions at Google were aimed at providing resource isolation while keeping high utilization. As such, the only networking goals for these containers were to provide a discoverable address and not lose performance. Port allocation, the scheduling and allotment of ports as a first-class resource, solved these problems nicely for some time. Since all internal Google applications were trusted, there was no need to isolate the applications' network views from each other, or protect the host from application containers. Outside Google there was a need for more isolated containers. LXC came up with multiple options to configure networking - starting with virtual Ethernet interfaces and over time covering VLAN, MACVLAN, and exposing dedicated physical devices [4]. These configurations provide a completely isolated networking view for apps running in containers. For clusters, the main networking goal is discovery and addressing for replicated containers as they move around through restarts, updates, and migrations.

Namespaces

Namespaces, as the name implies, restrict a particular kernel resource dimension to its own isolated view. The current set of supported namespaces are pid (process views), mnt (mounts), ipc (shared memory, message queues etc), UTS (host and domain names), user (uids and gids), and net (network view). Each net namespace gets its own isolated network interfaces, loopback device, routing table, iptable chains and rules. By assigning IPs to virtual interfaces within a container, multiple apps in different containers can, for example, all have a port 80 exposed.

Networking in Containers

The networking aspects of containers primarily center around the use of two of the Linux kernel's namespaces: network and UTS. These namespaces allow a container to present itself to the world and be routed to as if it were a host.

Docker is comprised of various subcomponents that manage container runtime, filesystem layers, and application images. The networking aspect is handled in container runtime setup. Docker's containers are, by default, created by libcontainer, a native implementation of Linux container runtime[5]. It is primarily written in Go,

with operations outside of the Go runtime written in C. Libcontainer provides a very low level API, exposing nearly all container-related kernel knobs. This makes libcontainer very powerful, but also difficult to configure at times. In this case study we will use Docker's use of libcontainer to showcase container network configurations.

Configurations

Libcontainer exposes its container configuration via a single JSON configuration file [6]. It allows the configuration of the container's hostname, routes, and networks. Hostname is directly used to set up UTS namespace. The routes configuration passes in any route table entries we want to install inside the namespace. Network configuration allows specifying how the container is exposed to the outside world. It allows the user to configure the MAC address, IP (both IPv4 and IPv6) address, and default gateway assigned to the container. It also allows the configuration of the MTU and queue length of the network device.

Docker containers can by default reach the outside world, but the outside world cannot talk to the container. To expose container endpoints, Docker requires explicit specification of what ports to expose. A container can either expose all ports, or specific ones. Container ports can be bound to specific host ports or docker can pick dynamic ports from a predefined range. All port mappings are managed through **`iptables`** rules. Docker sets up a MASQUERADE rule for traffic from docker containers, and rules to drop packets between containers unless specifically whitelisted (Figure 1).

```
$ sudo iptables -t nat -L -n
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16     0.0.0.0/0

Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0          0.0.0.0/0
tcp dpt:8080 to:172.17.2.13:8080
```

Figure 1. MASQUERADE rules made by Docker on a host.

Strategies

Libcontainer supports the following network configurations.

loopback Creating a new network namespace always creates a loopback device. This strategy is primarily used to enable the interface and perform any specific configuration or for containers that don't require external networking.

veth veth (virtual ethernet) creates a pair of network interfaces that act like a pipe. The usual setup involves creating a veth pair with one of the peer interfaces kept in the host network namespace and the other added to the container namespace. The interface on the host network namespace is added to a network bridge. In the Docker world, most of the veth setup is done by the Docker daemon. Libcontainer just expects the network pair to be already set up and it coordinates adding one of the interfaces to a newly-created container namespace (Figure 2).

On host

```
$ ip address list
...
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460
qdisc noqueue state UP group default
  link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    valid_lft forever preferred_lft forever
1037: veth694884b: <BROADCAST,UP,LOWER_UP> mtu 1460
qdisc pfifo_fast master docker0 state UP group default
qlen 1000
  link/ether 12:0f:8b:dc:27:ac brd ff:ff:ff:ff:ff:ff
```

Inside container

```
$ ip address list eth0
1036: eth0: <BROADCAST,UP,LOWER_UP> mtu 1460 qdisc
pfifo_fast state UP group default qlen 1000
  link/ether 02:42:ac:11:02:08 brd ff:ff:ff:ff:ff:ff
    valid_lft forever preferred_lft forever
```

Figure 2. Example of a veth container setup in Docker inside and outside the container. Container's eth0 and host's veth694884b interfaces form a veth pair linked to the docker bridge docker0.

In practice, use of separate bridges can allow creating virtual networks between containers.

In addition to setting up veth and bridge interfaces, containers are linked together and with the wider world through iptable rules. This strategy with a network bridge is found to be significantly less performant than using the host network interface directly, since it requires forwarding/routing which is not penalty free.

netns The netns strategy allows multiple containers to share the same network namespace and have access to the same network devices. A new container can be configured by passing in the name of the container that has loopback and/or veth configured. This option can also be used to directly expose physical host networking to a container by sharing the host network namespace.

MACVLAN/VLAN Libcontainer supports creation of VLAN and MACVLAN interfaces, but these are not used directly by Docker today. Instead tools like pipework [7] can be used to set up complex networking configurations. This strategy has the best performance of the options mentioned here for environments where VLANs can be used. Libcontainer does not yet support the recently released ipVLAN which works similarly to MACVLAN, but switching using L3 [8].

Networking in Container Clusters

Container clusters are compute clusters where the primary unit of deployment is a container. In a container cluster, all jobs are run in containers and that becomes the primary abstraction exposed to jobs (as opposed to a machine or a virtual machine). These types of clusters have risen in

popularity as containers provide a lightweight and flexible way to isolate and deploy applications. They're also thought to be more efficient than the alternatives and thus provide higher overall machine utilization.

Kubernetes is an open-source cluster container manager released by Google and based on over 10 years of running container clusters internally. It schedules, deploys, and manages groups of containers to clusters of machines. Systems like Kubernetes are designed to provide all the primitives necessary for microservice architectures, and for these the network is vital. Applications perform nearly all their functions through the network. Discovery, communication, and synchronization are all done through the network. This makes the network setup of any Kubernetes cluster one of the most important aspects.

Kubernetes provides a set of key abstractions users can use to architect their applications. We will examine two of those abstractions, pods and services, and how their networks are configured.

Pod

A pod, like a pod of whales, is a group of tightly coupled containers [9]. These containers are always deployed side by side on the same machine. In a pod, all containers share resources and share fate. All containers in a pod are placed inside the same network namespace such that they can talk to each other via localhost and share the port space. The pod is also assigned its own routable IP address. In this way pods are exposed outside of the Kubernetes node, are uniquely addressable, and can bind to any network port. To the external world, a pod looks like a virtual machine.

Kubernetes pods are comprised of groups of Docker containers. We create a "POD" infrastructure container with a small no-op binary. The other containers in the pod are then created, joining the existing network namespace of the POD container. Having an infrastructure pod helps in persisting network setup as individual containers in the pod are added, removed, or restarted. Today, we use the veth bridge networking strategy provided by Docker and assign it the pod's IP address. While the performance is tolerable for now, we would like to replace veth with an ipVLAN setup instead. Kubernetes is unable to use MACVLAN since most software defined networks (including Google Compute Engine's) don't support this option.

An alternate to the IP per pod approach is dynamic port allocation and mapping. We've found that systems choosing the latter approach are significantly more complex, permeate ports throughout all of their configurations, are plagued by port conflicts, and can suffer from port exhaustion. Ports become a schedulable resource which brings a tremendous amount of complexity. Applications must be able to change what ports they bind to at runtime and communicate their port(s) to their users and the service-discovery infrastructure. No longer can applications communicate with an IP and a pre-defined port, they now must know the IP and the associated port. We find that the complexity of assigning a routable IP to each pod to be preferable over making ports a resource throughout Kubernetes.

Services

In Kubernetes pods are considered ephemeral: they can come and go. Machine maintenance can, for example, cause them to be replaced with different instances that serve the same purpose. This makes it inconvenient and incorrect to address a pod by its IP address. This is where services come in. A service is an abstraction that allows the stable addressing of a group of pods (sometimes called a microservice) [10]. It functions very similarly to a group of pods placed in front of a load balancer. A service has an IP that is guaranteed to be stable. Requests to that IP are then load balanced to active pods that are behind the service. As pods come and go, the service updates the routes it can provide to incoming requests.

Services are implemented using a combination of **`iptables`** routes and a user space service proxy running on all Kubernetes nodes. When a pod on a node makes a request to a service through the latter's IP address the **`iptables`** rules re-route the request to the service proxy on the node. The service proxy keeps an updated list of all the pods that can answer requests for this particular service. The proxy watches the shared cluster state for changes in service membership and can enact the change quickly. Given the list of member pods, the service proxy does simple client-side round robin load balancing across the member pods. This service proxy allows applications to function unmodified in a Kubernetes cluster. If the overhead of the service proxy is a concern, applications can perform the membership queries and routing themselves, just as the proxy does. We have not noticed a significant performance impact from the use of the service proxy. We have work underway to replace the service proxy completely with `iptables` routes in order to remove the need for the proxy.

Services in Kubernetes tend to not be exposed to the outside world since most microservices simply talk to other microservices in the cluster. Public-facing services are extremely important as well since some microservice must eventually provide a service to the outside world. Unfortunately, public-facing services are not completely handled today in Kubernetes since there is no external proxy to act in a similar manner to the on-node service proxy. The current implementation of public services have a load balancer targeting any node in a Kubernetes cluster. Requests that arrive on this node are then rerouted by the service proxy to the correct pod answering requests for the service. This is an active area of work where a more complete solution is still being designed.

An alternative to the Kubernetes service abstraction is to use DNS to redirect to pods. The reason this approach was not chosen was due to the inconsistent handling of DNS lookups in DNS client libraries. Many clients do not respect TTLs, do not round robin members, or do not re-do lookups when membership changes, and we felt that DNS clients would not update membership quickly enough.

Discovery

Services are the key to all communication within a Kubernetes cluster. Discovering the IP address of those services is done in two ways. The first approach is through environment variables exposed in the pods. Each pod in the cluster has a set of environment variables exposed with the IP addresses of each service. Of course, this does not scale well with larger clusters. More recently service discovery was delegated to an internal cluster DNS service. Each service gets a DNS address in the cluster that resolves to the service IP. This was found not to have the DNS problems mentioned before since the service IP address is unique and stable.

Configurations

We'll briefly describe how the Kubernetes network is configured in some of the more popular network strategies.

Andromeda Andromeda is the software defined network underlying Google Compute Engine [11]. Andromeda allows us quite a bit of flexibility in programing the underlying network fabric. We allocate a subnet of 256 internal 10-dot IPv4 IPs on each node to be used for pods and ask the fabric to route this /24 range to the specific node. Service IP addresses are allocated from a different part of the 10-dot pool. Pod to pod communication uses the internal 10-dot IPs, but communication with the internet gets NAT'ed through the node as those have internet routable IPs.

Flannel Flannel is an overlay network developed by CoreOS [12]. Flannel uses universal TUN/TAP and UDP to encapsulate IP packets. It creates a virtual network very similar to the one described above for Andromeda by allocating a subnet to each host for use by its pods.

Others Using a virtual bridge on a node and connecting them across is being used to deploy more complex topologies. OVS on Kubernetes is enabled with such a setup [13]. Weave uses a similar technique to build an overlay network [14].

Resource Management

At the cluster level, networking resources can be managed by detecting network capacity at each node and allocating it to running containers like any other resource. However, a practical resource model for inter-cluster networking requires detailed topology information to be really effective. In most setups, network bandwidth management responsibility is shared between the fabric provider and the cluster manager. Given a topology model, containers can be scheduled to (re-)balance the networking load. In addition, when hotspots are identified, a node-local action can be initiated to limit bandwidth-hogging containers.

Future Work

Kubernetes would like to move to virtual migratable IPs so that container migration becomes possible within the cluster. There is also work underway to introduce "real" load balancing in the service proxy. This will allow the load balancer to balance on things like the utilization and health of pods behind the service.

Acknowledgements

The authors would like to acknowledge and thank the entire Kubernetes, Docker, and libcontainer teams, contributors, and communities. Their work is being highlighted and described here.

We would also like to thank John Wilkes and Mahesh Bandewar for their review and guidance on this paper.

References

1. Victor Marmol, Rohit Jnagal, "Containers @ Google", Sunnyvale, CA, accessed February 6, 2014, http://www.slideshare.net/vmarmol/containers-google
2. Docker, "Docker", accessed February 6, 2014, https://github.com/docker/docker
3. Kubernetes, "Kubernetes", accessed February 6, 2014, https://github.com/googlecloudplatform/kubernetes
4. Daniel Lezcano, "lxc.container.conf", accessed February 6, 2014, http://man7.org/linux/man-pages/man5/lxc.container.conf.5.html
5. Libcontainer, "Libcontainer", accessed February 6, 2014, https://github.com/docker/libcontainer
6. Libcontainer, "Libcontainer Config", accessed February 6, 2014, https://github.com/docker/libcontainer/blob/master/config.go
7. Jérôme Petazzoni, "Pipework", accessed February 6, 2014, https://github.com/jpetazzo/pipework
8. Mahesh Bandewar, "ipvlan: Initial check-in of the IPVLAN driver.", accessed February 6, 2014, http://lwn.net/Articles/620087/
9. Kubernetes, "Pods", accessed February 6, 2014, https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/pods.md
10. Kubernetes, "Services", accessed February 6, 2014, https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/services.md
11. Amin Vahdat, "Enter the Andromeda zone - Google Cloud Platform's latest networking stack", accessed February 6, 2014, http://googlecloudplatform.blogspot.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html
12. CoreOS, "Flannel", accessed February 6, 2014, https://github.com/coreos/flannel
13. Kubernetes, "OVS Networking", accessed February 6, 2014, https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/ovs-networking.md
14. Zettio, "Weave", accessed February 6, 2014, ttps://github.com/zettio/weave