The platform we built at Iguazio is cloud native, using Docker-based microservices, etcd and home-grown cluster management. We are gradually migrating to the Kubernetes container orchestration engine, now that it has become more mature, leveraging its advanced functionality so that we can focus on delivering unique services.

Unlike most cloud-native apps, ours is real-time. We drive extreme performance by using low-level direct access to network, storage, CPU and memory resources. This challenge with containers and Kubernetes isn't trivial and has required some good, old fashioned hacking.

This post is the first in a series. I'll share how Kubernetes and the Container Networking Interface works with some hacking tricks to learn its internals and manipulate it. Future posts will cover high-performance storage and inter-process communications (IPC) tricks we use with containers.

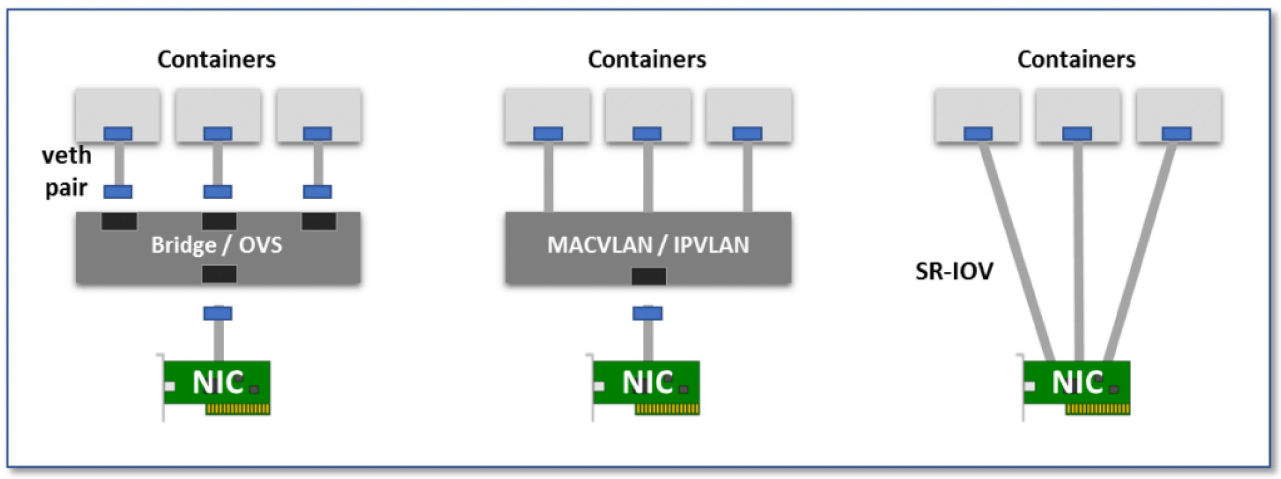
## Container Networking Basics

Containers use Linux partitioning capabilities called Cgroups and Namespaces. Container processes are mapped to network, storage and other namespaces. Each namespace "sees" only a subset of OS resources to guarantee isolation between containers.

On the network side, a namespace has its own network stack with interfaces, route tables, sockets and IPTABLE rules. An interface can only belong to one network namespace. Using multiple containers requires multiple interfaces. Another option is to generate pseudo-interfaces and soft wire them to a real interface (we can also map containers to the host network namespace, as used for daemons).

Here are few options for creating and wiring a pseudo-interface:

- **Virtual bridge**: Create virtual interface pairs (veth) with one side in the container and the other in the root namespace, and use Linux bridge or OpenvSwitch (OVS) for connectivity between containers and external (real) interfaces. Bridges may introduce some extra overhead when compared to a direct approach.
- **Multiplexing**: Multiplexing can consist of an intermediate network device that exposes multiple virtual interfaces, with packet forwarding rules to control which interface each packet goes to. MACVLAN assigns a MAC per virtual interface (outgoing packets are marked with that MAC, incoming packets are multiplexed based on the destination MAC). IPVLAN does the same, based on IP addresses and using a single MAC which makes it more VM-friendly.
- **Hardware switching**: most NICs today support Single Root I/O Virtualization (SR-IOV), which is a way to create multiple virtual devices. Each virtual device presents itself as a separate PCI device. It can have its own VLAN and hardware-enforced QoS association. SR-IOV provides bare-metal performance but is usually not available in the public cloud.




*Virtual networking modes: bridging, multiplexing and SR-IOV.*

In many cases, users want to create logical network subnets that span multiple L2/3 network segments. This requires overlay encapsulation protocols (the most common one is VXLAN, which encapsulate overlay traffic into UDP packets). VXLAN may introduce higher overhead and multiple VXLAN networks from different vendors are usually not interoperable due to lack of standardization in the control plane.

Kubernetes also makes extensive use of IPTABLES and NAT to intercept traffic going to a logical/virtual address and route it to the appropriate physical destination. Container networking solutions like Flannel, Calico and Weave use veth with a bridge/router and some overlay or routing/NAT manipulation.

See this great hands-on tutorial and usage guide for the various Linux networking options.

Beyond the expected packet manipulation overhead, virtual networking adds hidden costs and may negatively affect CPU and memory parallelism, for example:

- NICs direct traffic to cores based on headers. If the header is changed, traffic will go to the wrong cores and can reduce memory and CPU efficiency.
- NICs build packets and offload checksums to hardware to save a lot of CPU and memory cycles. If the overlay packets are built in software, or if we have overlay (container) on top of overlay (cloud/IaaS), the result will be a degradation of performance. New NICs can build VXLAN packets in hardware but the overlay solution must use it directly.

Some applications (like Iguazio's) use advanced NIC capabilities such as RDMA, the DPDK fast network processing library or encryption to offload messaging, have tighter control over CPU parallelism, reduce interrupts or eliminate memory copies. This is only possible when using a direct network interface or an SR-IOV virtual interface.

## Playing with POD Networking

There is no easy way to see network namespaces, as Kubernetes and Docker don't register them ("**ip netns**" won't work with Kubernetes and Docker). But we can use a few tricks to see, debug, manage and configure POD networking from the host.

Network namespaces are listed in **/proc/<PID>/ns/net** so we need to find the process ID (PID) for our POD. First, let's find a container ID with the command below, and only take the first 12 digits:

```
kubectl get po <POD-NAME> -o jsonpath='{.status.containerStatuses[0].containerID}' | cut -c 10-21
```

Next, let's use Docker commands to find the PID:

```
PID=$(docker inspect --format '{{ .State.Pid }}' <ContainerID>)
```

Once we have the PID, we can monitor and configure networking for that POD. Use the **nsenter** utility to run any command with the POD namespace, for example:

```
nsenter -t ${PID} -n ip addr
```

...will show us all the POD interfaces along with their IPs. We can then use other commands, like **ping** or **curl**, to check connectivity, use privileged operations or ones that are not installed in the POD container to monitor, debug or configure that POD (e.g. ip route, nslookup, etc.)

```
$ nsenter -t 13537 -n ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if45: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 0a:58:0a:f4:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.244.0.2/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::30d0:a8ff:fe43:fe80/64 scope link tentative dadfailed
       valid_lft forever preferred_lft forever
```

If we're not happy with a single interface per POD, we can just take or create interfaces from the host namespace and assign them to the POD:

```
ip link set netns ${PID} <IFNAME>
```

And if we want them back, we just return them to the host:

```
nsenter -t ${PID} -n ip link set <IFNAME> netns 1
```

## Automating Kubernetes Networking with CNI

Kubernetes uses CNI plug-ins to orchestrate networking. Every time a POD is initialized or removed, the default CNI plug-in is called with the default configuration. This CNI plug-in creates a pseudo interface, attaches it to the relevant underlay network, sets the IP and routes and maps it to the POD namespace.

Kubernetes unfortunately still supports only one CNI interface per POD with one cluster-wide configuration. This is very limiting since we may want to configure multiple network interfaces per POD, potentially using different overlay solutions with different policies (subnet, security, QoS).

Let's see how we can bypass those limitations.

When the **Kubelet** Kubernetes local agent configures POD networking, it looks for a CNI json configuration file in **/etc/cni/net.d/** and a relevant plug-in binary (based on the type attribute) located in **/opt/cni/bin/**. A CNI plug-in can invoke a secondary IP Address Management (IPAM) plug-in to set the IP address per interface (e.g. host-local or **dhcp**). It is also possible to use other paths through **Kubelet** command options:

```
$ cat >/etc/cni/net.d/10-mynet.conf
{
    "cniVersion": "0.2.0",
    "name": "mynet",
    "type": "bridge",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.22.0.0/16",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
    }
}
```

**Kubelet** invokes the CNI plug-in with environment variables containing command parameters (CNI_ARGS, CNI_COMMAND, CNI_IFNAME, CNI_NETNS, CNI_CONTAINERID, CNI_PATH) and streams the json.conf file through **stdin**. The plug-in responds with json output text, describing the results and status. See more detailed explanation and examples here. It's relatively simple to develop your own CNI plug-in if you know the Go programming language, as the framework does much of the magic and you can use or extend one of the existing plug-ins here.

**Kubelet** will pass the POD name and namespace as part of the CNI_ARGS variable (for example "K8S_POD_NAMESPACE=default;K8S_POD_NAME=mytests-1227152546-vq7kw;" ). We can use this to customize the network configuration per POD or POD namespace (e.g. put every namespace in a different subnet). Future Kubernetes versions will treat networks as equal citizens and include network configuration as part of the POD or namespace spec just like memory, CPUs and volumes. For the time being, we can use annotations to store configuration or record POD networking data/state.

The CNI plug-in does its own logic once invoked. It can also attach multiple network interfaces to the same POD and bypass the current limitations, with one caveat: Kubernetes will only be aware of the one we report in the results (which is used for service discovery and routing). I recently stumbled upon a great Intel open source CNI plug-in called Multus which does exactly that. Linux programmer Doug Smith wrote a detailed Multus walk-through after a recent Slack chat.

Multus accepts a hierarchical conf file with an array of CNI definitions. It will configure each POD with multiple interfaces, one per definition, and we can specify which interface is the "masterplugin" to be recognized by Kubernetes. We use this as a baseline coupled with the POD name and POD annotations to create flexible and unique network configurations per POD.

The same Intel git also includes another interesting CNI driver for SR-IOV and DPDK support.

## Summary

I love cloud-native and microservices, the impact they are having on agile and continuous application delivery is immense. However, it seems like container networking projects are still nascent and just taking their first steps, which is essentially no more than solving the challenge of cross-segment/cloud connectivity. These projects still require customization to make them fit the broader application base and higher-performance. Hopefully, it will quickly improve and get to the functionality level of other more mature software-defined networking solutions like OpenStack Neutron or VMware NSX.

How do we make all these different overlay and underlay control planes interoperable, allowing two different vendor/cloud solutions in both ends of a logical network? This is the $64,000 question and I welcome your feedback. It's clearly a key area that requires more attention and standardization.

*For more on Kubernetes networking and related topics, come to CloudNativeCon +KubeCon Europe 2017 in Berlin, Germany March 29-30.*

*Feature image via Pixabay.*