Network namespaces

Linux namespaces are a relatively new kernel feature which is essential for implementation of containers. A namespace wraps a global system resource into an abstraction which will be bound only to processes within the namespace, providing resource isolation. In this article I discuss network namespace and show a practical example.

Posted by Diego Pino García on April 10, 2016

Namespaces and **cgroups** are two of the main kernel technologies most of the new trend on software containerization (think Docker) rides on. To put it simple, cgroups (https://en.wikipedia.org/wiki/Cgroups) are a metering and limiting mechanism, they control how much of a system resource (CPU, memory) you can use. On the other hand, namespaces (https://en.wikipedia.org/wiki/Linux_namespaces) limit what you can see. Thanks to namespaces processes have their own view of the system's resources.

The Linux kernel provides 6 types of namespaces: **pid**, **net**, **mnt**, **uts**, **ipc** and **user**. For instance, a process inside a *pid namespace* only sees processes in the same namespace. Thanks to the *mnt namespace*, it's possible to attach a process to its own filesystem (like *chroot*). In this article I focus only in network namespaces.

If you have grasped the concept of namespaces you may have at this point an intuitive idea of what a network namespace might offer. Network namespaces provide a brand-new network stack for all the processes within the namespace. That includes **network interfaces**, **routing tables** and **iptables rules**.

Network namespaces

From the system's point of view, when creating a new process via `clone()` syscall, passing the flag CLONE_NEWNET will create a brand-new network namespace into the new process. From the user perspective, we simply use the tool `ip` (package is *iproute2*) to create a new persistent network namespace:

```
$ ip netns add ns1
```

This command will create a new network namespace called *ns1*. When the namespace is created, the ip command adds a *bind* mount point for it under */var/run/netns*. This allows the namespace to persist even if there's no process attached to it. To list the namespaces available in the system:

```
$ ls /var/run/netns
ns1
```

Or via `ip`:

```
$ ip netns
ns1
```

As previously said, a network namespace contains its own network resources: interfaces, routing tables, etc. Let's add a loopback interface to *ns1*:

```
1 $ ip netns exec ns1 ip link set dev lo up
2 $ ip netns exec ns1 ping 127.0.0.1
3 PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
4 64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.115 ms
```

- Line 1 brings up the loopback interface inside the network namespace ns1.

- Line 2 executes the command `ping 127.0.0.1` inside the network namespace.

An alternative syntax to bring up the loopback interface could be:

```
$ ip netns exec ns1 ifconfig lo up
```

However, I tend to use the command ip (http://baturin.org/docs/iproute2/) as it has become the preferred networking tool in Linux, obsoleting the old but more familiar commands `ifconfig`, `route`, etc. Notice that `ip` requires root privileges, so run it as *root* or prepend *sudo*.

A network namespace has its own routing table too:

```
$ ip netns exec ns1 ip route show
```

Which at this point returns nothing as we haven't add any routing table rule yet. Generally speaking, any command run within a network namespace is prepend by the prologue:

```
$ ip netns exec <network-namespace>
```

A practical example

One of the consequences of network namespaces is that only one interface could be assigned to a namespace at a time. If the root namespace owns *eth0*, which provides access to the external world, only programs within the root namespace could reach the Internet. The solution is to communicate a namespace with the root namespace via a **veth pair**. A *veth pair* works like a patch cable, connecting two sides. It consists of two virtual interfaces, one of them is assigned to the root network namespace, while the other lives within a network namespace. Setting up their IP addresses and routing rules accordingly, plus enabling NAT in the host side, will be enough to provide Internet access to the network namespace.

Additionally, I feel like I need to make a clarification at this point. I've read in several articles about network namespaces that physical device interfaces can only live in the root namespace. At least that's not the case with my current kernel (Linux 3.13). I can assign *eth0* to a namespace other than the root and when setting it up properly have Internet access from the namespace. However, the limitation of one interface living only in one single namespace at a time still applies, and that's a reason powerful enough to need connecting network namespace via a veth pair.

To start with, let's create a new network namespace called *ns1*.

```
# Remove namespace if it exists.
ip netns del ns1 &>/dev/null

# Create namespace
ip netns add ns1
```

Next, create a *veth pair*. Interface *v-eth1* will remain inside the root network namespace, while its peer, *v-peer1*, will be moved to the *ns1* namespace.

```
# Create veth link.
ip link add v-eth1 type veth peer name v-peer1

# Add peer-1 to NS.
ip link set v-peer1 netns ns1
```

Next, setup IPv4 addresses for both interfaces and bring them up.

```
# Setup IP address of v-eth1.
ip addr add 10.200.1.1/24 dev v-eth1
ip link set v-eth1 up

# Setup IP address of v-peer1.
ip netns exec ns1 ip addr add 10.200.1.2/24 dev v-peer1
ip netns exec ns1 ip link set v-peer1 up
ip netns exec ns1 ip link set lo up
```

Additionally I brought up the loopback interface inside *ns1*.

Now it's necessary we make all external traffic leaving *ns1* to go through *v-eth1*.

```
ip netns exec ns1 ip route add default via 10.200.1.1
```

However this won't be enough. As with any host sharing Its internet connection, it's necessary to enable IPv4 forwarding in the host and enable masquerading.

```
# Share internet access between host and NS.

# Enable IP-forwarding.
echo 1 > /proc/sys/net/ipv4/ip_forward

# Flush forward rules, policy DROP by default.
iptables -P FORWARD DROP
iptables -F FORWARD

# Flush nat rules.
iptables -t nat -F

# Enable masquerading of 10.200.1.0.
iptables -t nat -A POSTROUTING -s 10.200.1.0/255.255.255.0 -o eth0 -j MASQUERADE

# Allow forwarding between eth0 and v-eth1.
iptables -A FORWARD -i eth0 -o v-eth1 -j ACCEPT
iptables -A FORWARD -o eth0 -i v-eth1 -j ACCEPT
```

If everything went fine, it would be possible to ping an external host from *ns1*.

```
$ ip netns exec ns1 ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=50 time=48.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=50 time=50.8 ms
```

This is how the routing table inside *ns1* would look like after the setup:

```
$ ip netns exec ns1 ip route sh
default via 10.200.1.1 dev v-peer1
10.200.1.0/24 dev v-peer1  proto kernel  scope link  src 10.200.1.2
```

Prepending the `ip netns exec` prologue for every command to run from the namespace might be a bit tedious. Once the most basic features inside the namespace are setup, a more interesting possibility is to run a bash shell and attach it to the network namespace:

```
$ ip netns exec ns1 /bin/bash --rcfile <(echo "PS1=\"namespace ns1> \"")
namespace ns1> ping www.google.com
PING www.google.com (178.60.128.38) 56(84) bytes of data.
64 bytes from cache.google.com (178.60.128.38): icmp_seq=1 ttl=58 time=17.6 ms
```

Type *exit* to leave end the bash process and leave the network namespace.

Conclusion

Network namespaces, as well as other containerization technologies provided by the Linux kernel, are a lightweight mechanism for resource isolation. Processes attached to a network namespace see their own network stack, while not interfering with the rest of the system's network stack.

Network namespaces are easy to use too. A similar network-level isolation could have been set up using a VM. However, that seems a much more expensive solution in terms of system resources and time investment to build up such environment. If you only need process isolation at the networking level, network namespaces are definitively something to consider.

The full script is available as a GitHub gist at: ns-inet.sh (https://gist.github.com/dpino/6c0dca1742093346461e11aa8f608a99).

More readings

- Introducing Linux Network Namespaces (http://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces/) by Scott Lowe.
- Namespaces in operation, part 7: Network namespaces (https://lwn.net/Articles/580893/) by Jake Edge.
- Linux's namespaces man page (http://man7.org/linux/man-pages/man7/namespaces.7.html)
- Linux's ip-netns man page (http://man7.org/linux/man-pages/man7/namespaces.7.html)

networking (/dpino/tag/networking) igalia (/dpino/tag/igalia)

← **PREVIOUS POST (/DPINO/2016/04/02/IPV6-TUNNEL/)**

NEXT POST → (/DPINO/2016/05/02/NETWORK-NAMESPACES-IPV6/)

● (/dpino/feed.xml) ● (https://twitter.com/diepg)

● (https://github.com/dpino)