Network Emulation using Network Namespaces

Antonio Carzaniga

17 March 2021

We want to experiment with networks, but without actually building them. One good way to do it is to *emulate* the network. And one way to do that is to use virtual machines. However, those are heavy on system resources. Instead, we will use a feature of the operating system called *network namespaces.* A network namespace is a very light-weight form of network virtualization.

In practice, we run everything on a single machine. This could be your laptop or even a virtual machine running within your laptop. In any case, everything we run—applications, configuration commands, scripts, etc.—runs as a regular process on the same machine. The trick is that a process running within a given namespace will see only the network interfaces, including virtual interfaces, forwarding tables, etc., that exist in that namespace. The operating system will then connect pairs of interfaces across namespaces, so that packets sent by one application in one namespace can be received by other applications in other namespaces. Applications can then also serve as switches or routers, by forwarding packets through these interfaces.

All of this effectively allows us to set up a network emulation environment. Network emulation means that links are virtual connections implemented by the operating system, but the code of the applications, as well as the code of the network stack, including for example the TCP stack, is exactly the same code that runs in a real environment.

The examples below are tested on a Linux system. Still, the concepts are quite general.

Basic Layer-2 Switch Setup Using Open vSwitch

As a first example, we want to create a network with two hosts, H_1 and H_2, connected by a *switch.* This means that H_1 and H_2 are connected by a level-2 network through a software switch (as opposed to a real switch in a physical box).

We start by creating the two network namespaces that represent the execution environments of the processes running on hosts H_1 and H_2, respectively. We will call these two namespaces "H1" and "H2". Notice that all the configuration commands in the following examples are executed with root privileges. So, for simplicity we start by running a root shell with something like `sudo bash`, then:

```
# ip netns add H1
# ip netns add H2
```

We then create the virtual network interfaces for the two hosts.

```
# ip link add H1-eth0 type veth peer name s-eth1
# ip link add H2-eth0 type veth peer name s-eth2
```

Each of these instructions creates a virtual link between two virtual Ethernet interfaces. Interface `H1-eth0` will be the main network interface for host H_1, and will be connected with interface `s-eth1`,

which will be one of the ports of the switch. Similarly, `H2-eth0` will be the main network interface for host H_2 and will be connected port `s-eth2` of the switch.

We can also list the links we just created with this command:

```
# ip link show
```

When we create these links, the interfaces are in the current network namespace, which we can check with `ifconfig -a`. So we have to move each interface into its intended namespaces.

```
# ip link set H1-eth0 netns H1
# ip link set H2-eth0 netns H2
```

Now `ifconfig -a` no longer shows interfaces `H1-eth0` and `H2-eth0`. Instead, we can find those in their respective namespace by running `ifconfig -a` within each namespaces. For example:

```
# ip netns exec H1 ifconfig -a
```

Notice however that `ifconfig -a` in the base namespace shows that the *peer* interfaces of `H1-eth0` and `H2-eth0`, namely `s-eth1` and `s-eth2`, respectively, are still in the base namespace. In fact, we want them to stay there, and we want to connect them to a switch, so that packets from one link can be correctly sent to the other, an vice-versa.

For that purpose, we use a software switch called Open vSwitch (`https://www.openvswitch.org/`). We use this switch as a level-2 *bridge*. In practice, the switch interconnects all its links in a single local-area network. This is effectively what happens when you connect devices to your wireless access point at home (although typically those access points also acts as a NAT routers).

So, this is how we create a virtual switch `s`:

```
# ovs-vsctl add-br s
```

And this is how we attach the two virtual interfaces `s-eth1` and `s-eth2` to switch `s`:

```
# ovs-vsctl add-port s s-eth1
# ovs-vsctl add-port s s-eth2
```

At this point, we have set up the layer-2 network (virtual) in terms of connectivity, but the network is not yet configured at the IP layer, and it is not running anyway. Notice in fact that if we run `ifconfig` within the `H1` and `H2` network namespaces, we don't see any active interface. Similarly, the two ports on the switch are also down.

```
# ip netns exec H1 ifconfig
# ip netns exec H2 ifconfig
# ifconfig
```

It is time to turn the network on the IP network for those two virtual hosts. We do that, within each host, by activating the loopback interface and by configuring and turning on the virtual Ethernet interface. In particular, we can assign addresses from the 10.0.0.0/8 private network address space.

```
# ip netns exec H1 ifconfig lo up
# ip netns exec H1 ifconfig H1-eth0 10.0.0.1
# ip netns exec H2 ifconfig lo up
# ip netns exec H2 ifconfig H2-eth0 10.0.0.2
```

Now, running `ifconfig` within the `H1` and `H2` namespaces shows that the network is configured and active.

The last step is to activate the two switch ports, `s-eth1` and `s-eth2`.

```
# ifconfig s-eth1 up
# ifconfig s-eth2 up
```

We are now ready to *use* the network. We first test connectivity using good-old *ping*. For example, we can run `ping` within `H1` to check connectivity with the address we assigned to `H2`:

```
# ip netns H1 ping 10.0.0.2
```

. . . magic!

We can then test the same connection between `H1` and `H2` using any application we like (netcat, web clients and servers, etc.). We can also capture and analyze traffic, using *wireshark*, *tshark*, or *tcpdump* on each network interface, just as we would on a real network interface.

Basic Setup Using a Host as a Layer-3 Switch/Router

In the previous example, we set up a layer-2 network to connect two hosts. We now see how to set up a layer-3 network (Internet) with proper IP forwarding. As a first example, we set up two hosts and a router to interconnect them. As before, the two "hosts" are nothing more than network name spaces. The router is also a simple host environment, in its own network namespace, in which we enable IP forwarding at the kernel level.

Starting from the previous configuration, we must first clean things up. We could do things in reverse, turning off interfaces, etc., but we'll be a bit more drastic. We just delete the software bridge:

```
# ovs-vsctl del-br s
```

and then delete the `H1` and `H2` network namespaces.

```
# ip netns del H1
# ip netns del H2
```

This will automatically delete all the virtual links that have interfaces in those namespaces.

So, now we are ready to start with the layer-3 setup. We first create the three namespaces for the hosts and the router, respectively:

```
# ip netns add H1
# ip netns add H2
# ip netns add R
```

Then we create the links and move their endpoints in their intended namespaces. For clarity, we indicate the intended namespace in the name of the endpoints.

```
# ip link add H1-eth0 type veth peer name R-eth1
# ip link add H2-eth0 type veth peer name R-eth2
# ip link set H1-eth0 netns H1
# ip link set H2-eth0 netns H2
# ip link set R-eth1 netns R
# ip link set R-eth2 netns R
```

We now need to configure the IP addresses and activate all the network interfaces:

```
# ip netns exec H1 ifconfig lo up
# ip netns exec H1 ifconfig H1-eth0 10.0.1.2 netmask 255.255.255.0
# ip netns exec H1 ifconfig H1-eth0 up
# ip netns exec H2 ifconfig lo up
# ip netns exec H2 ifconfig H2-eth0 10.0.2.2 netmask 255.255.255.0
# ip netns exec H2 ifconfig H2-eth0 up
# ip netns exec R ifconfig lo up
# ip netns exec R ifconfig R-eth1 10.0.1.1 netmask 255.255.255.0
# ip netns exec R ifconfig R-eth1 up
# ip netns exec R ifconfig R-eth2 10.0.2.1 netmask 255.255.255.0
# ip netns exec R ifconfig R-eth2 up
```

Now, notice that we have connectivity within each single link. So, for example, we can ping the router R from host H_1:

```
# ip netns exec H1 ping -c 3 10.0.1.1
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1: icmp_seq=1 ttl=64 time=0.058 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.123 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=0.169 ms

--- 10.0.1.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2036ms
rtt min/avg/max/mdev = 0.058/0.116/0.169/0.045 ms
```

And vice-versa, we can ping H_1 from R

```
# ip netns exec R ping -c 3 10.0.1.2
PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data.
64 bytes from 10.0.1.2: icmp_seq=1 ttl=64 time=0.076 ms
64 bytes from 10.0.1.2: icmp_seq=2 ttl=64 time=0.114 ms
64 bytes from 10.0.1.2: icmp_seq=3 ttl=64 time=0.194 ms

--- 10.0.1.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2026ms
rtt min/avg/max/mdev = 0.076/0.128/0.194/0.049 ms
```

However, we can not reach H_2 from H_1 (and vice-versa)

```
# ip netns exec H1 ping -c 3 10.0.2.2
ping: connect: Network is unreachable
# ip netns exec H2 ping -c 3 10.0.1.2
ping: connect: Network is unreachable
```

In fact, we can not even reach the second interface of the router from the first host (or the first interface of the router from the second host):

```
# ip netns exec H1 ping -c 3 10.0.2.1
ping: connect: Network is unreachable
# ip netns exec H2 ping -c 3 10.0.1.1
ping: connect: Network is unreachable
```

This is because we don't even have basic routing information on the hosts. The configuration of the IP addresses for the host interfaces implicitly sets up the most basic routing entries for the local network. For example, on H_1, the routing tables are as follows:

```
# ip netns exec H1 ip route
10.0.1.0/24 dev H1-eth0 proto kernel scope link src 10.0.1.2
```

However, with that, host H_1 does not know how to send to any address outside its local network 10.0.1.0/24. We must therefore configure H_1 to use router R, and more specifically its interface 10.0.1.1, as its default gateway router.

```
# ip netns exec H1 ip route add default via 10.0.1.1 dev H1-eth0
```

With that, the routing tables on H_1 are complete:

```
# ip netns exec H1 ip route show
default via 10.0.1.1 dev H1-eth0
10.0.1.0/24 dev H1-eth0 proto kernel scope link src 10.0.1.2
```

We do the same for H_2

```
# ip netns exec H2 ip route add default via 10.0.2.1 dev H2-eth0
```

However, we still cannot reach H_2 from H_1 (and vice-versa):

```
# ip netns exec H1 ping -c 3 10.0.2.2
PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.

--- 10.0.2.2 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2026ms
```

Why? Notice that the message is different right now. Earlier, the system immediately fails saying that the network is unreachable. Now, H_1 does have the routing information to send those packets, which it does, but then the packets fail to get to their destination and/or back to H_1.

You might think that that is because the router does not have the proper routing information. However, R does in fact have that information, again just be virtue of the fact that we correctly configured the address and netmask of its two interfaces:

```
# ip netns exec R ip route
10.0.1.0/24 dev R-eth1 proto kernel scope link src 10.0.1.1
10.0.2.0/24 dev R-eth2 proto kernel scope link src 10.0.2.1
```

So, why is R not doing its job as a router? The answer is a somewhat minor technical detail of the Linux system we use in these examples. The router is in fact a regular host that needs to perform IP forwarding, but that is disabled by default. So, all we need to do is to enable IP forwarding:

```
# ip netns exec R sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 0
# ip netns exec R sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```

So, now we have connectivity from H_1 to H_2 and back, through router R.