

# Using network namespaces and a virtual switch to isolate servers

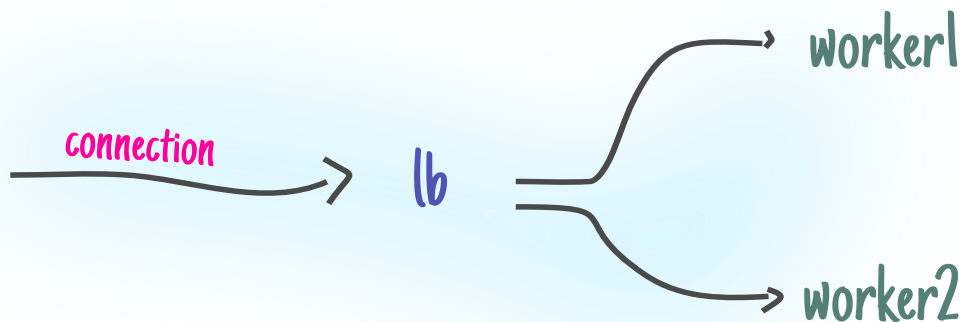
Yet another blog post on how to create a virtual network that connects network namespaces using a bridge, veth pairs and iptables.

by Ciro S. Costa - Jun 12, 2018

Hey,

I've been working on an `tc + ebpf`-based load-balancer that I'll soon talk about here in this blog, and one of the things I wanted to do was test such load-balancing feature.

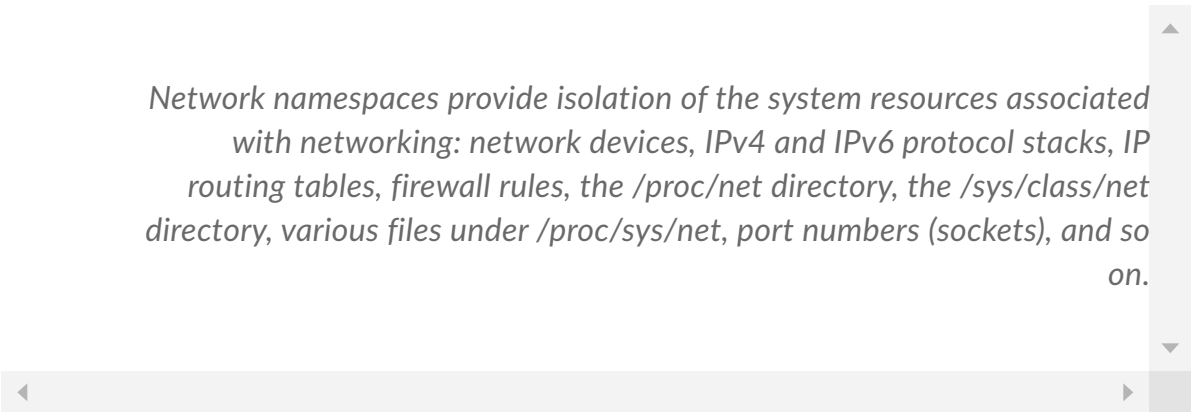
In my line of thought, I wondered: if I'm going to redirect the traffic by changing the destination address of the packets, that means that I'll need somehow to have a different internet set up in the machine where I could put processes listening on those addresses.



Having worked with Docker and implemented a tiny container runtime myself, it seemed clear that going with network namespaces, virtual interfaces and bridging would get the job done.

How these two relate to each other become clearer after looking at what jobs they do:

- network namespaces, according to `man 7 network_namespaces` :



*Network namespaces provide isolation of the system resources associated with networking: network devices, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, the `/proc/net` directory, the `/sys/class/net` directory, various files under `/proc/sys/net`, port numbers (sockets), and so on.*

- virtual interfaces provide us with virtualized representations of physical network interfaces; and
- the bridge gives us the virtual equivalent of a switch.

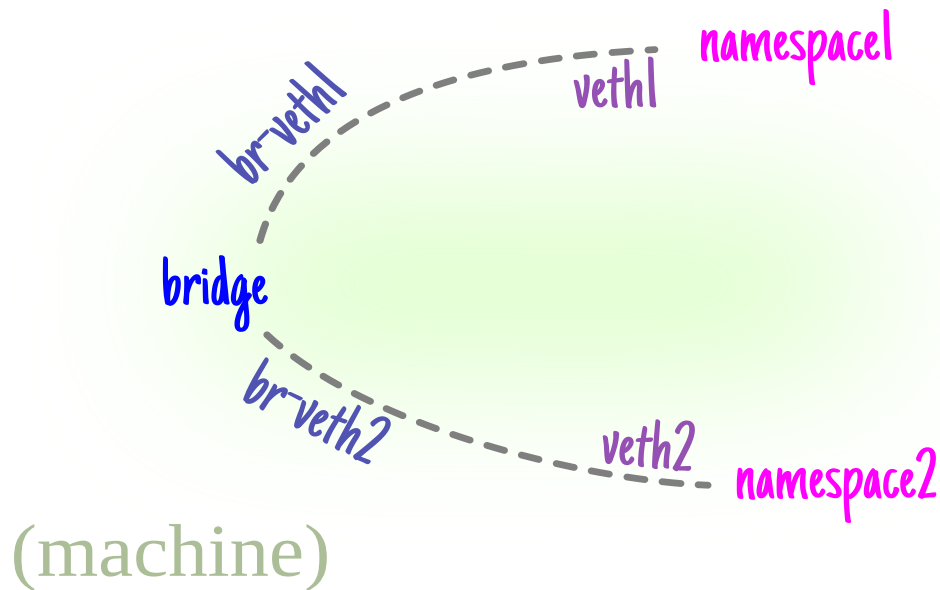
That said, we can combine these three virtual components and create a virtual network inside the host without the need of VMs. Very lightweight (and with a single dependency - `iproute2` ).

## Setting up

Assuming we want to simulate load-balancing across two different servers, that would mean that we'd set up the machine to have:

- two network namespace (in this case, we can think of each network namespace as a different computer);
- two veth pairs (we can think of each pair as two ethernet cards with a cable between them); and

- a bridge device that provides the routing to these two namespaces (we can think of this device as a switch).



First, let's start with the namespaces:

```
# Making use of the `ip` command from the `iproute2`
# package we're able to create the namespaces.
#
# By convention, network namespace handles created by
# iproute2 live under `/var/run/netns` (although they
# could live somewhere, like `docker` does with its
# namespaces - /var/run/docker/netns`).
ip netns add namespace1
ip netns add namespace2

# Check that iproute2 indeed creates the files
# under `/var/run/netns`.
tree /var/run/netns/
/var/run/netns/
├── namespace1
└── namespace2
```

Once the namespaces have been set up, we can confirm the isolation by taking advantage of `ip netns exec` and executing some commands there:

```
# List all the interfaces with their corresponding
# configurations.
#
# We can verify that inside the namespace, only the
# loopback interface has been set.
ip netns exec namespace1 \
    ip address show
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

At this point, we can start creating the veth pairs and associating one of their sides to their respective namespaces.

```
# Create the two pairs.
ip link add veth1 type veth peer name br-veth1
ip link add veth2 type veth peer name br-veth2

# Associate the non `br-` side
# with the corresponding namespace
ip link set veth1 netns namespace1
ip link set veth2 netns namespace2
```

These pairs act as tunnels between the namespaces (for instance, `namespace1` and the default namespace).

Now that the namespaces have an additional interface, check out that they're actually there:

```
# Differently from before, now we see the
# extra interface (veth1) that we just added.
ip netns exec namespace1 \
    ip address show
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
30: veth1@if29: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group d
    link/ether d2:92:25:3c:79:c5 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

As we can see, the `veth` interface has no IPV4 address associated with it.

We can do so by making use of `ip addr add` from within the corresponding network namespaces:

```
# Assign the address 192.168.1.11 with netmask 255.255.255.0
# (see the `/24` mask there) to `veth1`.
ip netns exec namespace1 \
    ip addr add 192.168.1.11/24 dev veth1

# Verify that the ip address has been set.
ip netns exec namespace1 \
    ip address show
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
30: veth1@if29: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group d
    link/ether d2:92:25:3c:79:c5 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.1.2/24 scope global veth1
        valid_lft forever preferred_lft forever

# Repeat the process, assigning the address 192.168.1.12 with
# netmask 255.255.255.0 to `veth2`.
ip netns exec namespace2 \
    ip addr add 192.168.1.12/24 dev veth2
```

Although we have both IPs and interfaces set, we can't establish communication with them.

That's because there's no interface in the default namespace that can send the traffic to those namespaces - we didn't either configure addresses to the other side of the veth pairs or configured a bridge device.

With the creation of the bridge device, we're then able to provide the necessary routing, properly forming the network:

```
# Create the bridge device naming it `br1`
# and set it up:
ip link add name br1 type bridge
ip link set br1 up

# Check that the device has been created.
ip link | grep br1
33: br1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKN
```

With the bridge created, now it's time to connect the bridge-side of the veth pair to the bridge device:

default

ces up too.

the bridge  
s their master.

master of the two  
that the two interfaces

```
ROADCAST,MULTICAST,UP> mtu 1500 master br1 state disabled priority 32 cost 2
ROADCAST,MULTICAST,UP> mtu 1500 master br1 state disabled priority 32 cost 2
```

Now, it's a matter of giving this bridge device an address so that we can target such IP in our machine's routing table making it a target for connections to those interfaces that we added to it:

```
# Set the address of the `br1` interface (bridge device)
# to 192.168.1.10/24 and also set the broadcast address
# to 192.168.1.255 (the `+` symbol sets the host bits to
# 255).
ip addr add 192.168.1.10/24 brd + dev br1
```

Checking our routing table (from the default namespace), we can see that any requests with a destination to our namespaces ( 192.168.1.0/24 ) go through our bridge device:

```
default via 10.0.2.2 dev enp0s3 proto dhcp src 10.0.2.15 metric 100
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.15
10.0.2.2 dev enp0s3 proto dhcp scope link src 10.0.2.15 metric 100
192.168.1.0/24 dev br1 proto kernel scope link src 192.168.1.10 # <<<<<<<<<
```

We can verify that we indeed have connectivity:

```
# Check the connectivity from the default namespace (host)
ping 192.168.1.12
PING 192.168.1.12 (192.168.1.12) 56(84) bytes of data.
64 bytes from 192.168.1.12: icmp_seq=1 ttl=64 time=0.050 ms
64 bytes from 192.168.1.12: icmp_seq=2 ttl=64 time=0.061 ms

# We can also reach the interface of the other namespace
# given that we have a route to it.
ip netns exec namespace1 \
    ip route
192.168.1.0/24 dev veth1 proto kernel scope link src 192.168.1.11

# Let's reach the other then iface then.
ip netns exec namespace1 \
    ping 192.168.1.12
```

Given that the routing table from namespace1 doesn't have a default gateway, it can't reach any other machine from outside the 192.168.1.0/24 range.

```
# Try to reach Google's DNS servers (8.8.8.8).
#
# Given that there's no route for something that doesn't
# match the `192.168.1.0/24` range, 8.8.8.8 should be unreachable.
ip netns exec namespace1 \
    ping 8.8.8.8
connect: Network is unreachable
```

To fix that, the first step is giving the namespaces a default gateway route:

```
# Execute the command to add the default gateway in all
# the network namespaces under `/var/run/netns`.
#
# The command is going to add a default gateway which should
# be used for all connections that doesn't match the other
# specific routes.
#
# 192.168.1.10 corresponds to the address assigned to the
# bridge device - reachable from both namespaces, as well as
# the host machine.
ip -all netns exec \
    ip route add default via 192.168.1.10

# Check how the routing table looks inside the namespace
ip netns exec namespace1 \
    ip route
default via 192.168.1.10 dev veth1
192.168.1.0/24 dev veth1 proto kernel scope link src 192.168.1.11
```

Show we be able to reach the internet now? Not yet.

```
# Try to reach Google's DNS servers (8.8.8.8).
ip netns exec namespace1 \
    ping 8.8.8.8
    PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
^C
--- 8.8.8.8 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2043ms
```

Although the network is now reachable, there's no way that we can have responses back - packets from external networks can't be sent directly to our 192.168.1.0/24 network.

To get around that, we can make use of NAT (network address translation) by placing an `iptables` rule in the `POSTROUTING` chain of the `nat` table:

```
# -t specifies the table to which the commands
# should be directed to. By default it's `filter`.
#
# -A specifies that we're appending a rule to the
# chain the we tell the name after it;
#
# -s specifies a source address (with a mask in
# this case).
#
# -j specifies the target to jump to (what action to
# take).
iptables \
    -t nat \
    -A POSTROUTING \
    -s 192.168.1.0/24 \
    -j MASQUERADE
```

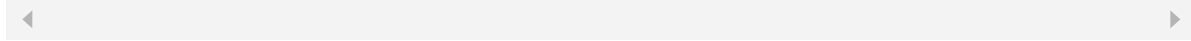
Although `NAT` is set to work for the packets originating from 192.168.1.0/24 , there's still (yet another) one more configuration to do: enable packet forwarding (maybe this is already active in your case):

```
# Enable ipv4 ip forwarding
sysctl -w net.ipv4.ip_forward=1

# Send some packets to 8.8.8.8
```



```
ip netns exec namespace1 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=61 time=43 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=61 time=35 ms
```



Now (finally), we're good! We have connectivity all the way:

- the host can direct traffic to an application inside a namespace;
- an application inside a namespace can direct traffic to an application in the host;
- an application inside a namespace can direct traffic to another application in another namespace; and
- an application inside a namespace can access the internet.

## Closing thoughts

Did we need bridge at all? That depends.

If the intention was to have the communication going through the host to a namespace (or vice-versa) directly, just setting the pairs would be enough.

Just in case you spot any mistake, please let me know! I'm cirowrc on Twitter.

Have a good one!

*finis*

### Stay in touch!

From time to time I'll deliver some content to you.

The emails are not automatic - it's all about things I thought were worth sharing that I'd personally like to receive.

[JOIN THE GROUP](#)

If you're into Twitter, reach me at [@cirowrc](#).

[About](#)

[Tags](#)

[Advertise](#)

[Twitter](#)

[GitHub](#)

[LinkedIn](#)

© [Ciro da Silva da Costa](#), 2018.