Software-Defined Networking tools for LXC (LinuX Containers)

38 commits 1 branch 0 releases 6 contributors

branch: master pipework /

Set BRTYPE to "linux" by default. ···

Jérôme Petazzoni jpetazzo authored 3 days ago latest commit a7956843c4

| | | |
|---|---|---|
| .gitignore | Ignore *~ files. | 5 months ago |
| LICENSE | Add Apache 2.0 license. (I hope it is OK with everyone!) | 4 months ago |
| README.md | Added optional parameter '-i <ifname>' to set container interface to … | 24 days ago |
| pipework | Set BRTYPE to "linux" by default. | 3 days ago |

README.md

# Pipework: Software-Defined Networking for Linux Containers

Pipework lets you connect together containers in arbitrarily complex scenarios.

**If you use VirtualBox**, you will have to update your VM network settings. Open the settings panel for the VM, go the the "Network" tab, pull down the "Advanced" settings. Here, the "Adapter Type" should be `pcnet` (the full name is something like "PCnet-FAST III"), instead of the default `e1000` (Intel PRO/1000). Also, "Promiscuous Mode" should be set to "Allow All". If you don't do that, bridged containers won't work, because the virtual NIC will filter out all packets with a different MAC address.

## Docker users: read this!

Pipework works with "plain" LXC containers (created with `lxc-start`), and therefore, it also works with the awesome Docker.

**Before using Pipework, please ask on the docker-user mailing list if there is a "native" way to achieve what you want to do *without* Pipework.**

In the long run, Docker will allow complex scenarios, and Pipework should become obsolete.

If there is really no other way to plumb your containers together with the current version of Docker, then okay, let's see how we can help you!

The following examples show what Pipework can do for you and your containers.

## LAMP stack with a private network between the MySQL and Apache containers

Let's create two containers, running the web tier and the database tier:

```
APACHE=$(docker run -d apache /usr/sbin/httpd -D FOREGROUND)
MYSQL=$(docker run -d mysql /usr/sbin/mysqld_safe)
```

Now, bring superpowers to the web tier:

```
pipework br1 $APACHE 192.168.1.1/24
```

This will:

- create a bridge named `br1` in the docker host;
- add an interface named `eth1` to the `$APACHE` container;
- assign IP address 192.168.1.1 to this interface,
- connect said interface to `br1`.

Now (drum roll), let's do this:

```
pipework br1 $MYSQL 192.168.1.2/24
```

This will:

- not create a bridge named `br1`, since it already exists;
- add an interface named `eth1` to the `$MYSQL` container;
- assign IP address 192.168.1.2 to this interface,
- connect said interface to `br1`.

Now, both containers can ping each other on the 192.168.1.0/24 subnet.

## Docker integration

Pipework can resolve Docker containers names. If the container ID that you gave to Pipework cannot be found, Pipework will try to resolve it with `docker inspect`. This makes it even simpler to use:

```
docker run -name web1 -d apache
pipework br1 web1 192.168.12.23/24
```

## Peeking inside the private network

Want to connect to those containers using their private addresses? Easy:

```
ip addr add 192.168.1.254/24 dev br1
```

Voilà!

## Setting container internal interface

By default pipework creates a new interface `eth1` inside the container. In case you want to change this interface name like `eth2`, e.g., to have more than one interface set by pipework, use:

`pipework br1 -i eth2 ...`

## Using a different netmask

The IP addresses given to `pipework` are directly passed to the `ip addr` tool; so you can append a subnet size using traditional CIDR notation.

I.e.:

```
pipework br1 $CONTAINERID 192.168.4.25/20
```

Don't forget that all containers should use the same subnet size; pipework is not clever enough to use your specified subnet size for the first container, and retain it to use it for the other containers.

## Setting a default gateway

If you want *outbound* traffic (i.e. when the containers connects to the outside world) to go through the interface managed by Pipework, you need to change the default route of the container.

This can be useful in some usecases, like traffic shaping, or if you want the container to use a specific outbound IP address.

This can be automated by Pipework, by adding the gateway address after the IP address and subnet mask:

```
pipework br1 $CONTAINERID 192.168.4.25/20@192.168.4.1
```

## Connect a container to a local physical interface

Let's pretend that you want to run two Hipache instances, listening on real interfaces eth2 and eth3, using specific (public) IP addresses. Easy!

```
pipework eth2 $(docker run -d hipache /usr/sbin/hipache) 50.19.169.157
pipework eth3 $(docker run -d hipache /usr/sbin/hipache) 107.22.140.5
```

Note that this will use `macvlan` subinterfaces, so you can actually put multiple containers on the same physical interface.

## Wait for the network to be ready

Sometimes, you want the extra network interface to be up and running *before* starting your service. A dirty (and unreliable) solution would be to add a `sleep` command before starting your service; but that could break in "interesting" ways if the server happens to be a bit slower at one point.

There is a better option: add the `pipework` script to your Docker image, and before starting the service, call `pipework --wait`. It will wait until the `eth1` interface is present and in `UP` operational state, then exit gracefully.

## Add the interface without an IP address

If for some reason you want to set the IP address from within the container, you can use `0/0` as the IP address. The interface will be created, connected to the network, and assigned to the container, but without configuring an IP address:

```
pipework br1 $CONTAINERID 0/0
```

## DHCP

You can use DHCP to obtain the IP address of the new interface. Just specify `dhcp` instead of an IP address; for instance:

```
pipework eth1 $CONTAINERID dhcp
```

You need three things for this to work correctly:

- obviously, a DHCP server (in the example above, a DHCP server should be listening on the network to which we are connected on `eth1`);
- the `udhcpc` DHCP client must be installed on your Docker *host* (you don't have to install it in your containers, but it must be present on the host);
- the underlying network must support bridged frames.

The last item might be particularly relevant if you are trying to bridge your containers with a WPA-protected WiFi network. I'm not 100% sure about this, but I think that the WiFi access point will drop frames originating from unknown MAC addresses; meaning that you have to go through extra hoops if you want it to work properly.

It works fine on plain old wired Ethernet, though.

## Specify a custom MAC address

If you need to specify the MAC address to be used (either by the `macvlan` subinterface, or the `veth` interface), no problem. Just add it as the command-line, as the last argument:

```
pipework eth0 $(docker run -d haproxy) 192.168.1.2 26:2e:71:98:60:8f
```

This can be useful if your network environment requires whitelisting your hardware addresses (some hosting providers do that), or if you want to obtain a specific address from your DHCP server. Also, some projects like Orchestrator rely on static MAC-IPv6 bindings for DHCPv6:

```
pipework br0 $(docker run -d zerorpcworker) dhcp fa:de:b0:99:52:1c
```

**Note:** if you generate your own MAC addresses, try remember those two simple rules:

- the lowest bit of the first byte should be `0`, otherwise, you are defining a multicast address;
- the second lowest bit of the first byte should be `1`, otherwise, you are using a globally unique (OUI enforced) address.

In other words, if your MAC address is `?X:??:??:??:??:??`, `X` should be `2`, `6`, `a`, or `e`. You can check Wikipedia if you want even more details.

## Support Open vSwitch

If you want to attach a container to the Open vSwitch bridge, no problem.

```
ovs-vsctl list-br
ovsbr0
pipework ovsbr0 $(docker run -d mysql /usr/sbin/mysqld_safe) 192.168.1.2/24
```

## Cleanup

When a container is terminated (the last process of the net namespace exits), the network interfaces are garbage collected. The interface in the container is automatically destroyed, and the interface in the docker host (part of the bridge) is then destroyed as well.