Container Network Solutions
Research Project 2

Joris Claassen

System and Network Engineering

University of Amsterdam

Supervisor:
Dr. Paola Grosso

July 2015

Abstract

Linux container virtualization is getting mainstream. This research focuses on the specific ways containers can be networked together; be it an overlay network, or the actual link from the container to the network. It gives an overview of these networking solutions. As most overlay solutions are not quite production ready yet, testing their performance is not really interesting. This research includes a literature study on the overlay solutions.

The kernel modules used to link the container to the network are much more mature, and they are benchmarked in this research. The results of the tests performed in this research show that while the veth kernel module is the most mature, it has been surpassed by the macvlan kernel module in raw performance. The new ipvlan kernel module also gets assessed in this research, but is still very immature and buggy.

Contents

Acronyms

ACR	App Container Runtime
cgroups	control groups
COW	copy-on-write
IPC	Inter-process communication
KVM	Kernel-based Virtual Machine
LXC	Linux Containers
MNT	Mount
NET	Networking
OCP	Open Container Project
PID	Process ID
SDN	Software Defined Networking
SDVN	Software Defined Virtual Networking
UTS	Unix Time-sharing System
VXLAN	Virtual Extensible LAN

1

Introduction

As operating-system-level virtualization is getting mainstream, so do the parts of computing that it relies on. Operating-system-level virtualization is all about containers in a Linux environment, zones in a Solaris environment, or jails in an BSD environment. Containers are used to isolate different namespaces within a Linux node, providing a system that is similar to virtual machines while all processes being run within containers on the same node interact with the same kernel.

At this moment Docker(3) is the de-facto container standard. Docker used to rely on Linux Containers (LXC)(10) for containerization, but has shifted to their own libcontainer(8) from version 1.0. On December 1, 2014, CoreOS Inc. announced they where going to compete with the Docker runtime by launching rkt(18); a more lightweight App Container Runtime (ACR) in conjunction with AppC(2); a lightweight container definition. As of June 22, 2015, both parties (and a lot of other tech giants) have come together, and announced a new container standard: the Open Container Project (OCP)(11), based on AppC. This allows for containers to be interchangeable while keeping the possibility for vendors to compete using their own ACR and supporting services (e.g. clustering, scheduling and/or overlay networking).

Containers can be used for a lot of different system implementations, of which almost all require interconnection; at least within the node itself. In addition to connecting containers within nodes, there are additional connectivity issues coming up when interconnecting multiple nodes.

The aim of this project is to get a comprehensive overview of most networking solutions available for containers, and what their relative performance trade-offs are.

There is a multitude of overlay networking solutions available, which will be compared based on features. The kernel modules that can be used to connect a container to the networking hardware will be compared based on features and evaluated for their respective performance. This poses the following question:

How do virtual ethernet bridges perform, compared to macvlan and ipvlan?

- In terms of TCP throughput?
- In terms of UDP throughput?
- In terms of scalability?

- In a single-node environment?
- In a switched multi-node environment?

2

Background

2.1 Container technology

2.1.1 Namespaces

A container is all about isolating namespaces. The combination of several isolated namespaces provides a workspace for a container. Every container can only access the namespaces it was assigned, and cannot access any namespace outside it. Namespaces that are isolated to create a fully isolated container are:

- Process ID (PID) - Used to isolation processes from each other
- Networking (NET) - Used to isolate network devices within a container
- Mount (MNT) - Used to isolate mount points
- Inter-process communication (IPC) - Used to isolate access to IPC resources
- Unix Time-sharing System (UTS) - Used to isolate kernel and version identifiers

2.1.2 cgroups

control groups (cgroups) are another key part of making containers a worthy competitor to VMs. After isolating the namespaces for a container, every namespace still has full access to all hardware. This is where cgroups come in. They limit the available hardware resources to each container. For example the amount of CPU cycles that a container can use could be limited.

2.2 Working with Docker

As stated in the introduction, Docker is *the* container standard at this moment. It has not gained that position by being the first, but by making container deployment accessible for the masses. Docker provides a service to easily deploy a container from a repository or by building a Dockerfile. A Docker container is built up out of several layers with an extra layer on top for the changes made for this specific container. These layers are implemented using storage backends, of which most are copy-on-write (COW), but there is also a non-COW fallback backend in case the module used is not supported by the used Linux kernel.

An example of the ease of use: to launch a fully functional Ubuntu container on a freshly installed system a user only has to run one command, as shown in code snippet 2.1.

Code snippet 2.1: Starting a Ubuntu container

```
core@node1 ~ $ docker run -t -i --name=ubuntu ubuntu
Unable to find image 'ubuntu:latest' locally
latest: Pulling from ubuntu
428b411c28f0: Pull complete
435050075b3f: Pull complete
9fd3c8c9af32: Pull complete
6d4946999d4f: Already exists
ubuntu:latest: The image you are pulling has been verified.
    ↪ Important: image verification is a tech preview feature
    ↪ and should not be relied on to provide security.
Digest: sha256:45e42b43f2ff4850dcf52960ee89c21cda79ec657302d36
faaaa07d880215dd9
Status: Downloaded newer image for ubuntu:latest
root@881b59f36969:/# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 14.04.2 LTS
Release:        14.04
Codename:       trusty
root@881b59f36969:/# exit
exit
```

Even when a container has exited the COW filesystems (including the top one) will remain on the system until the container gets removed. In this state the container is

not consuming any resources with exception of the disk apace of the top COW image, but it is still in standby to launch processes instantaneously. This can be seen in code snippet 2.2.

Code snippet 2.2: List of containers

```
core@node1 ~ $ docker ps -a
CONTAINER ID    IMAGE           COMMAND      CREATED
881b59f36969    ubuntu:latest   "/bin/bash" 6 minutes ago
STATUS                                  PORTS   NAMES
Exited (127) About a minute ago                 ubuntu
```

Docker can also list all images which are downloaded into the cache, as can be seen in code snippet 2.3. Images that are not available locally will be downloaded from a configured Docker repository (Docker Hub by default).

Code snippet 2.3: List of images

```
core@node1 ~ $ docker images
REPOSITORY  TAG     IMAGE ID        CREATED      VIRTUAL SIZE
ubuntu      latest  6d4946999d4f    3 weeks ago  188.3 MB
```

2.3 Related work

There is some related research done on containers and Software Defined Networking (SDN)s by Costache et al. (22). Research on Virtual Extensible LAN (VXLAN), which is another tunneling networking model that could apply to containers, is widely available. A more recent paper by Felter et al. (23) does report on performance differences on almost all aspects between Kernel-based Virtual Machine (KVM) (7) and Docker, but does not take into account different networking implementations. The impact of virtual ethernet bridges and Software Defined Virtual Networking (SDVN)s (in particular, Weave (20)) has already been researched on by Kratzke (24). Marmol et al. go a lot deeper into the theory of methods of container networking in *Networking in containers and container clusters*(25), but do not touch the performance aspect.

3

Container networking

Container networking could be about creating a consistent network environment for a group of containers. This could be achieved using an overlay network, of which multiple implementations exist.

There is also another part of container networking - the way a networking namespace connects to the physical network device. There are multiple Linux kernel modules that allow a networking namespace to communicate with the networking hardware.

3.1 Overlay networks

To interconnect multiple nodes that are running containers both consistent endpoints and a path between the nodes is a must. When the nodes are running in different private networks which are using private addressing, connecting internal containers can be troublesome. In most network environments there is not enough routable (IPv4) address space for all servers, although with IPv6 on the rise this is getting less and less of an issue.

3.1.1 Weave

Weave(20) is a custom SDVN solution for containers. The idea behind this is that Weave launches one router container on every node that has to be interconnected, after which the weave routers set up tunnels to each other. This enables total freedom to move containers between hosts without any reconfiguration. Weave also has some nice features that enable an administrator to visualize the overlay network. Weave also

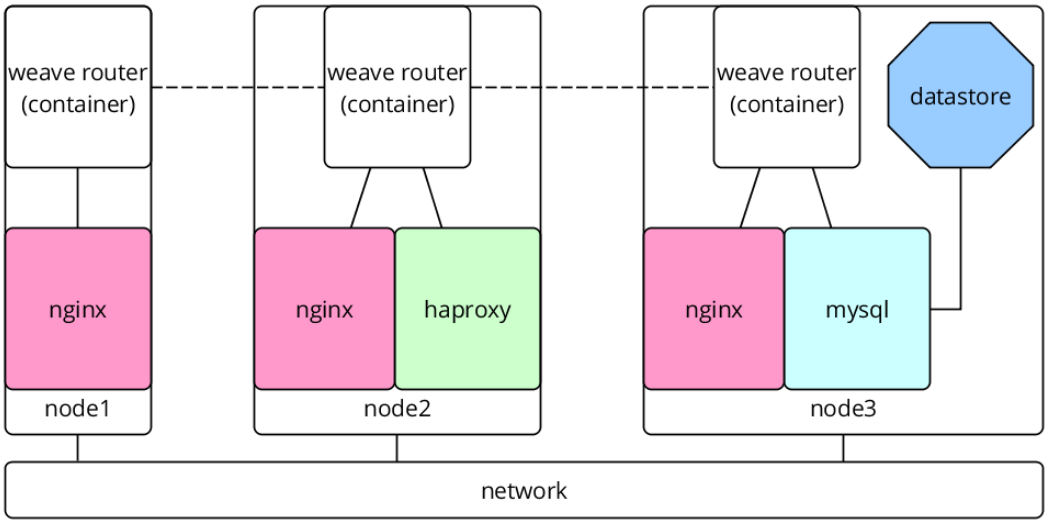

Figure 3.1: Example Weave overlay networking visualized

comes with a private DNS server; weaveDNS. It enables service discovery and allows to dynamically reallocate DNS names to (sets of) containers, which can be spread out over several nodes.

In figure 3.1 we see 3 nodes running 5 containers of which a common database container on one node is being used by multiple webservers. Weave makes all containers work as if they where on the same broadcast domain, allowing simple configuration.

Kratzke(24) has found Weave Net in its current form to decrease performance from 30 to 70 percent, but Weaveworks is working on a new implementation(21) based on Open vSwitch(13) and VXLAN that should dramatically increase performance. This new solution should work in conjunction with libnetwork.

3.1.2 Project Calico

While Project Calico(17) is technically not an overlay network but a "pure layer 3 approach to virtual networking" (as stated on their website), it is still worth mentioning because it does strive for the same goals as overlay networks. The idea of Calico is that data streams should not be encapsulated, but routed instead.

Calico uses a vRouter and BGP daemon on every node (in a privileged container), which instead of creating a virtual network modify the existing iptables rules of the

nodes they run on. As all nodes get their own private AS, the routes to containers running within the nodes are distributed to other nodes using a BGP daemon. This ensures that all nodes can find paths to the target container's destination node, while even making indirect routes discoverable for all nodes, using a longer AS path.

In addition to this routing software, Calico also makes use of one "orchestrator node". This node runs another container which includes the ACL management for the whole environment.

3.1.3 Socketplane and libnetwork

Socketplane is another overlay network solution specifically designed to work around Docker. The initial idea of Socketplane was to run one controller per node which then connected to other endpoints. Socketplane was able to ship a very promising technology preview before they got bought by Docker, Inc. This setup was a lot like the one of Weave, but it was based on Open vSwitch and VXLAN from the beginning.

Docker then put the Socketplane team to work on libnetwork(9). libnetwork includes an experimental overlay driver, which will use veth pairs, Linux bridges and VXLAN tunnels to enable an out of the box overlay network. Docker is hard at work to support pluggable overlay network solutions in addition to its current (limited) network support using libnetwork, which should ship with Docker 1.7.

Most companies working on overlay networks have announced support for libnetwork. These include, but are not limited to: Weave, Microsoft, VMware, Cisco, Nuage Networks, Midokura and Project Calico.

3.2 Kernel modules

Different Linux kernel modules can be used to create virtual network devices. These devices can then be attached to the correct network namespace.

3.2.1 veth

The veth kernel module is a pair of networking devices which are piped to each other. Every bit that enters the one end, comes out on the other. One of the ends can then be put in a different namespace, as visualized in the top part of figure 3.2. This technology is pioneered by Odin's Virtuozzo(19) and its open counterpart OpenVZ(15).

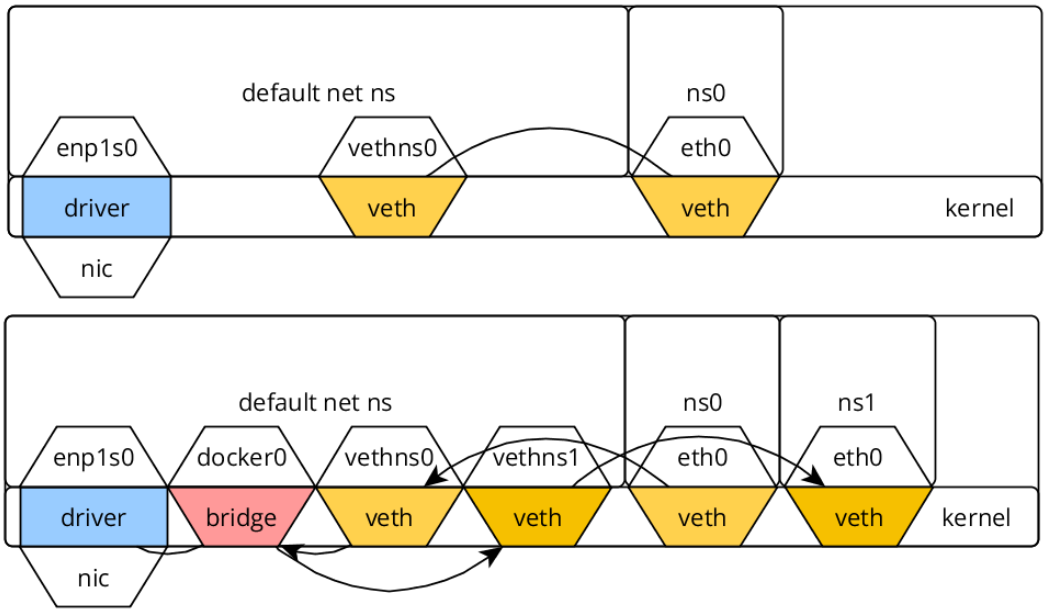

Figure 3.2: veth network pipes visualized

veth pipes are often used in combination with Linux bridges to provide an easy connection between a namespace and a bridge in the default networking namespace. An example of this is the docker0 bridge that is automatically created when Docker is started. Every Docker container gets a veth pair of which one side will be within the containers' network namespace, and the other connected to the bridge in default network namespace. A datastream from ns0 to ns1 using veth pipes (bridged) is visualized in the bottom part of figure 3.2.

3.2.2 openvswitch

The openvswitch kernel module comes as part of the mainline Linux kernel, but is operated by a seperate piece of software. Open vSwitch provides a solid virtual switch, which also features SDN through OpenFlow(12). docker-ovs(4) was created by and using Open vSwitch for the VXLAN tunnels between nodes, but was still using veth pairs oposed to internal bridge ports. Internal bridges have a higher throughput than veth pairs(14) when running multiple threads.

Putting these ports in a different namespace will confuse the Open vSwitch con-

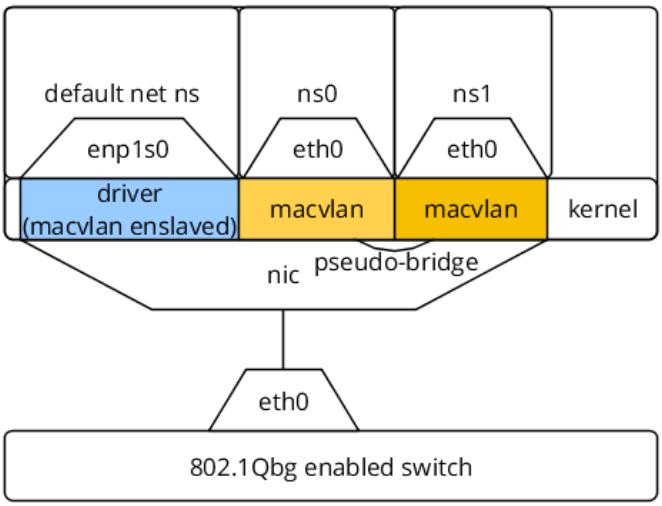

Figure 3.3: macvlan kernel module visualized

troller, effectively not making internal ports usable in container environments. Open vSwitch can still be a very good replacement for the default Linux bridges, but it will make use of veth pairs for the connection to other namespaces.

3.2.3 macvlan

macvlan is a kernel module which enslaves the driver of the NIC in kernel space. The module allows for new devices to be stacked on top of the default device, as visualized in figure 3.3. These new devices have their own MAC address and reside on the same broadcast domain as the default driver. macvlan has four different modes of operation:

- Private - no macvlan devices can communicate with each other; all traffic from a macvlan device which has one of the macvlan devices as destination MAC address get dropped.
- VEPA - devices can not communicate directly, but using a 802.1Qbg(1) (Edge Virtual Bridging) capable switch the traffic can be sent back to another macvlan device.
- Bridge - same as VEPA with the addition of a pseudo-bridge which forwards traffic using the RAM of the node as buffer.

- Passtru - passes the packet to the network; due to the standard behavior of a switch not to forward packets back to the port they came from it is effectively private mode.

3.2.4 ipvlan

ipvlan is very similar to macvlan, it does also enslave the driver of the NIC in kernel space. The main difference is that the packets that are being sent out all get the same MAC address on the wire. The forwarding to the correct virtual device is being done based on layer 3 address. It has two modes of operation:

- L2 mode - device behaves as a layer 2 device; all TX processing up to layer 2 happens in the namespace of the virtual driver, after which the packets are being sent to the default networking namespace for transmit. Broadcast and multicast are functional, but still buggy at the current implementation. This causes for ARP timeouts.
- L3 mode - device behaves as a layer 3 device; all TX processing up to layer 3 happens in the namespace of the virtual driver, after which the packets are being sent to the default network namespace for layer 2 processing and transmit (the routing table of the default networking namespace will be used). Does not support broad- and multicast.

4

Experimental setup

4.1 Equipment

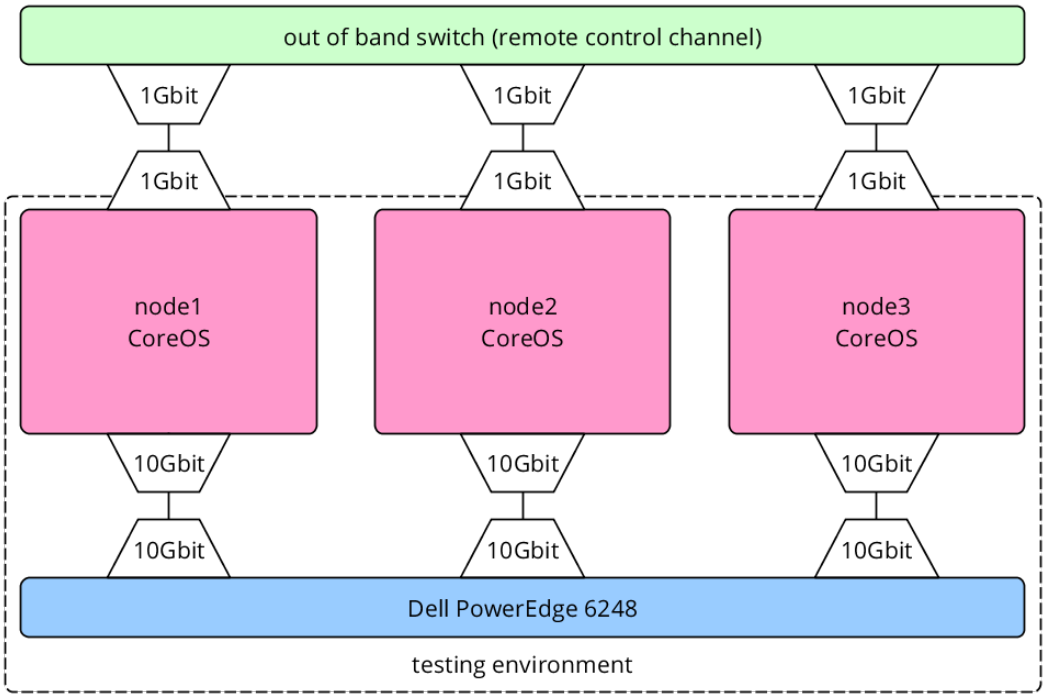

Figure 4.1: Physical test setup

Equipment used to perform the benchmark tests proposed in section 1 require a powerful hardware setup. The overview of the setup can be seen in figure 4.1. The hardware

Nodes	
CPU	3x Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
RAM	24GB DDR3 1600Mhz
NIC	10Gb LR SM SFP+

Switch	
Model	Dell PowerEdge 6248
NIC	3x 10Gb LR SM SFP+

Table 4.1: Hardware used for testing

used can be found in table 4.1. Nodes from the DAS4 cluster where reused to create this setup. The testing environment was fully isolated so there where no external factors influencing the test results.

4.2 Tests

The software used for the testing setup was CoreOS as this is an OS which is actively focused on running containers and has a recent Linux kernel (mainline Linux kernel updates are usually pushed within a month). The actual test setups where created using custom made scripts and can be found in appendix A and B.

To create the ipvlan interfaces required for the testing using these scripts, I modified Jérôme Petazzoni's excelent Pipework(16) tool to support ipvlan (in L3 mode). The code is attached in appendix C. All appendices can also be found on `https://github.com/jorisc90/rp2_scripts`.

Tests were ran using using the iperf3(5) measurement tool from within containers. iperf3 (3.0.11) is a recent tool that is a complete and more efficient rewrite of the orignal iperf. Scripts were created to launch these containers without too much delay.

All tests ran using exponentially growing numbers ($N in figure 4.2 and 4.3) of container pairs; ranging from 1 to 128 for TCP, and 1 to 16 for UDP. This is because UDP starts filling up the line in such a way that iperf3's control messages get blocked with higher number of container pairs, thus stopping measurements from being reliable. The tests where repeated 10 times to calculate the standard error values. The three compared network techniques are: veth bridges, macvlan (bridge mode) and ipvlan (L3 mode).

Due to the exponentially rising number of container pairs, a exponentially decreased throughput was expected both for TCP as with UDP. UDP performance was expected to be better than TCP because of the simpler protocol design.

4.2.1 Local testing

The local testing was performed on one node. This means a 2 up to 256 containers where being launched on one node for these tests, running 128 data streams at a time. The scrips where launched from the node itself to minimize any latencies.

An example of a test setup using 1 container and macvlan network devices (these commands are executed using a script):

Code snippet 4.1: Starting a macvlan server container

```
docker run -dit --net=none --name=iperf3_s_1 iperf/iperf
sudo ./pipework/pipework enp5s0 iperf3_s_1 10.0.1.1/16
docker exec iperf3_s_1 iperf3 -s -D
```

This spawns the container, after which it attaches a macvlan device to it. Then it launches the iperf3 server in daemon mode within the container.

To let the client testing commence, the following commands where used to start the test (also using a script):

Code snippet 4.2: Starting a macvlan client container

```
docker run -dit --net=none --name=iperf3_c_1 iperf/iperf
sudo ./pipework/pipework enp5s0 iperf3_c_1 10.0.2.1/16
docker exec iperf3_c_1 iperf3 -c 10.0.1.1 -p 5201 -f m -O 1 -M
    ↪ 1500 > iperf3_1.log &
```

This spawns the container, after which it attaches a macvlan device to it. Then it launches the iperf3 client which connects to the iperf3 server instance and saves the log files in the path where the script is executed.

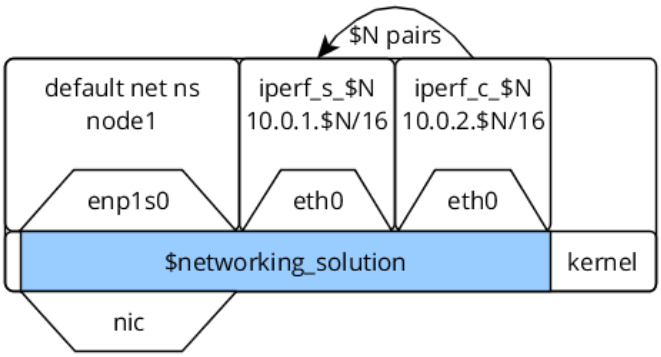

Figure 4.2: Local test setup

4.2.2 Switched testing

Switched testing was very similar to the local testing, but scripts had to be ran from an out-of-band control channel to ensure the timing was correct. This was achieved using node3 which ran commands using SSH over the second switch, while waiting for the command on node1 to finish before sending a new command to node2 and vice versa.

While the amount of data streams being ran are the same, the workload is split between two nodes; thus launching 1 up to 128 containers per node. This can be seen in figure 4.3.

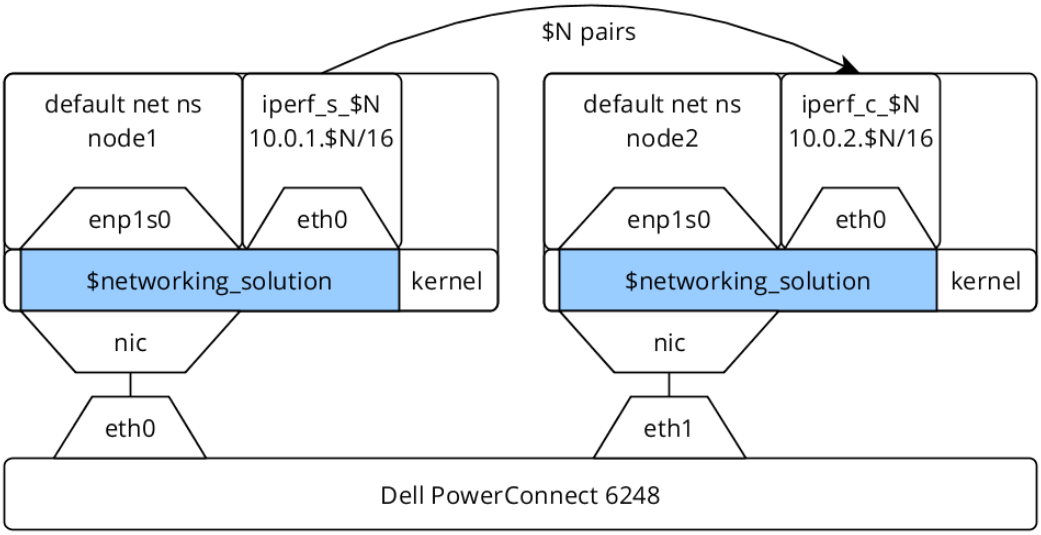

Figure 4.3: Switched test setup

4.3 Issues

While building a test setup like this there are always some issues that are being ran into. A quick list of the issues that cost me the most time to solve:

- Figuring out that one of the 4 10Gbit NIC **slots** on the switch was broken, instead of the SFP+ module or the fiber
- Getting CoreOS to install and login without a IPv4 address (coreos-install has to be modified)
- Figuring out that the problem of iperf3 control messages working but actual data transfer being zero was due to jumbo frames feature not being enabled on the switch; and not due to a setting on the CoreOS nodes
- Debugging ipvlan L2 connectivity and finding(6) that broadcast/multicast frames still get pushed into the wrong work-queue - instead opted for L3 mode for the tests

5

Performance evaluation

This chapter describes the results of the tests defined in chapter 4.

5.1 Local testing

5.1.1 TCP

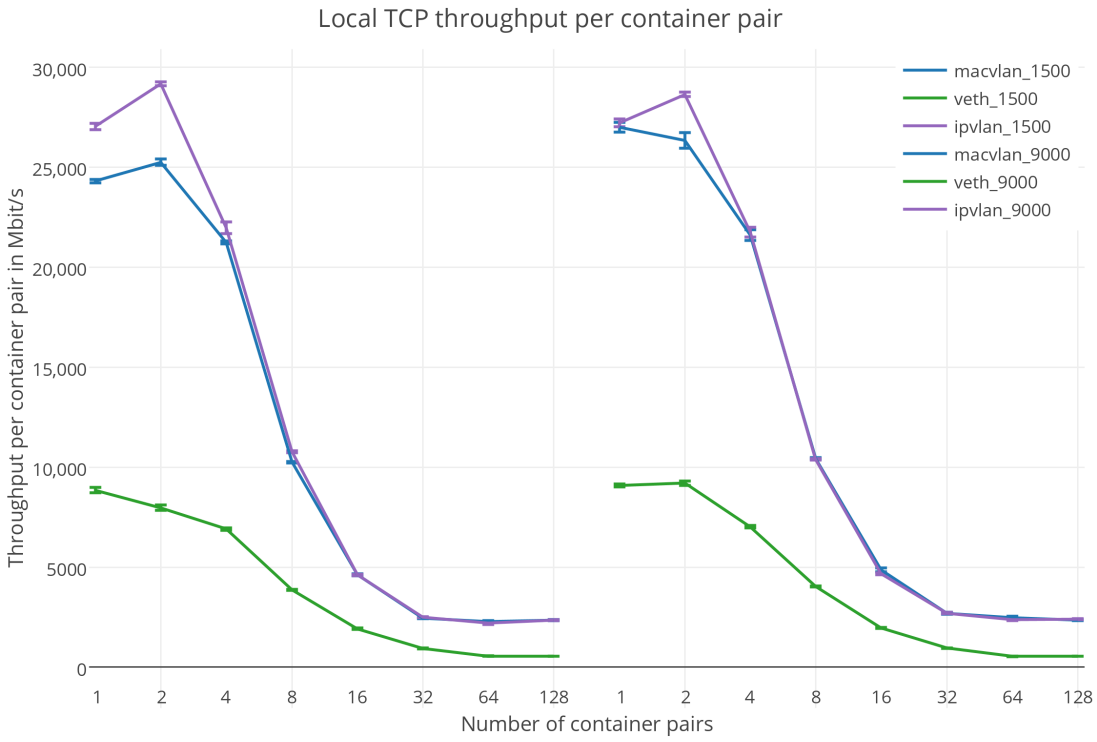

Figure 5.1: Local TCP test results for MTU 1500 (left) and MTU 9000 (right)

As can be seen in figure 5.1, the ipvlan (L3 mode) kernel module has the best performance in the tests. Note that because the device is in L3 mode, the routing table of the default namespace is used, and no broad- and/or multicast traffic is forwarded to these interfaces. veth bridges perform 2.5 up to 3 times as low as macvlan (bridge mode) and ipvlan (L3 mode). There is no big difference in performance behavior between MTU 1500 and MTU 9000.

5.1.2 UDP

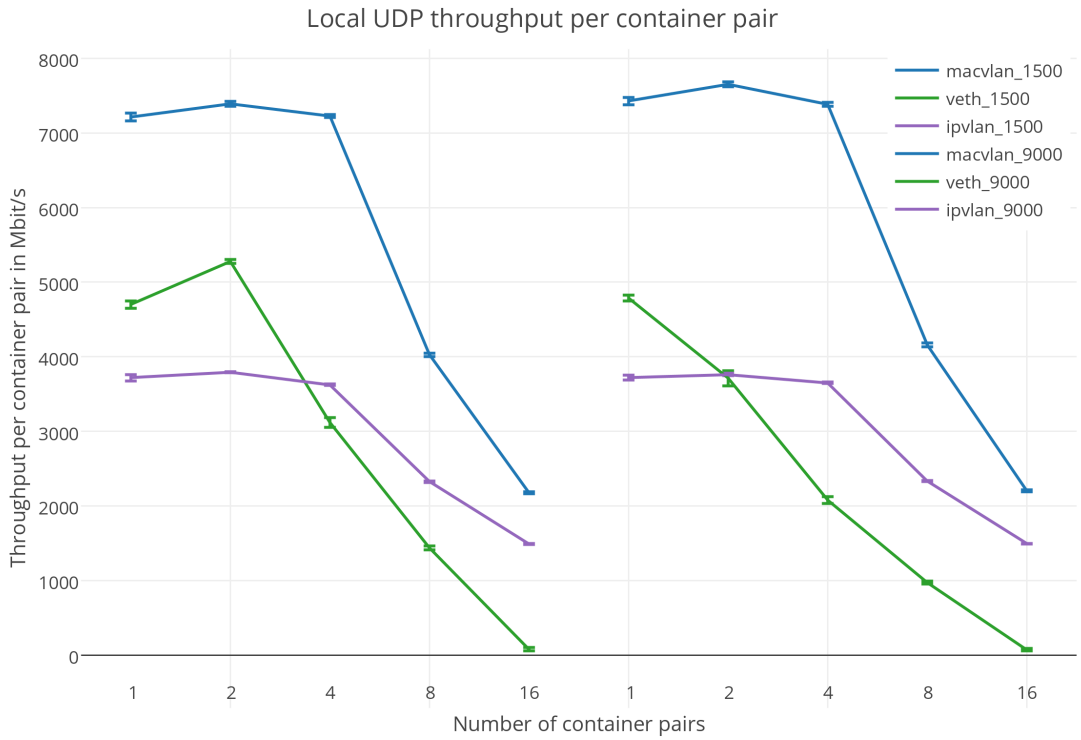

Figure 5.2: Local UDP test results for MTU 1500 (left) and MTU 9000 (right)

The performance displayed in figure 5.2 is very different from the one of figure 5.1. The hypothesis is that this is because either iperf3 is consuming more CPU cycles by measuring the jitter of the UDP packets and/or the UDP offloading in the kernel module (of either the node's 10Gbit NIC or one of the virtual devices) is not optimal. More research should be put looking into this anomaly.

The measured data shows that ipvlan (L3 mode) and veth bridges do not perform well in UDP testing. veth bridges do show the expected behavior of exponential de-

creasing performance after two container pairs. macvlan (bridge mode) does perform reasonably well, probably accounting to the network pseudo-bridge in RAM. The total throughput is still 3.5 times as low as with macvlan (bridge mode) TCP traffic streams.

5.2 Switched testing

5.2.1 TCP

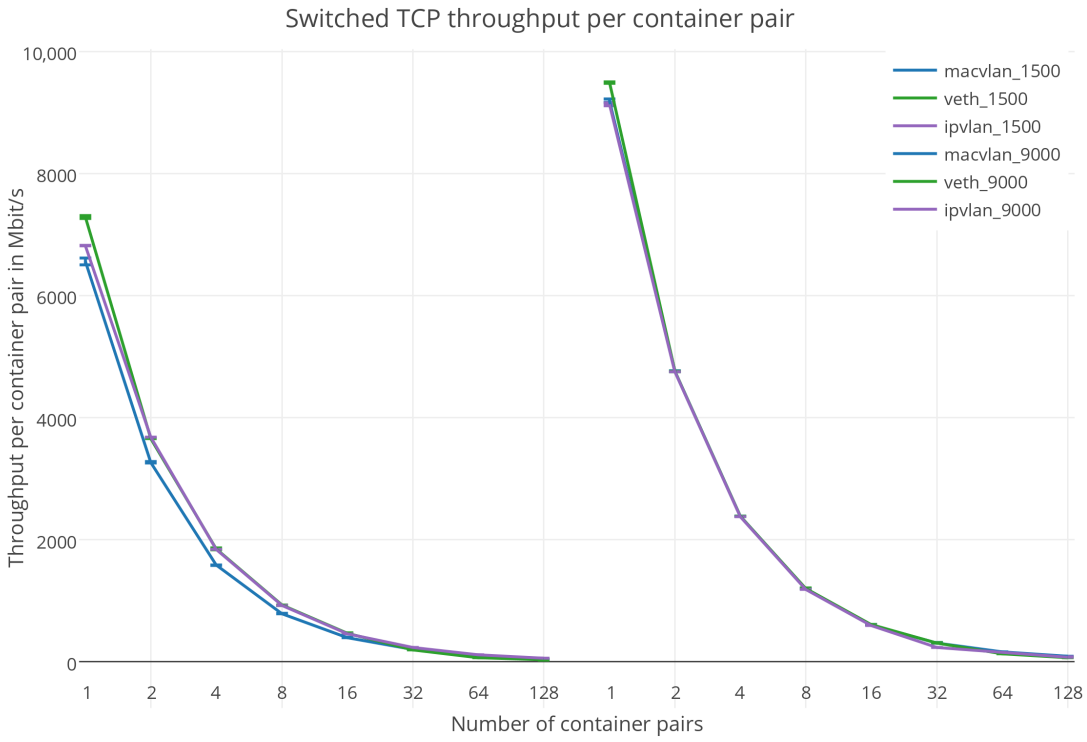

Figure 5.3: Switched TCP test results for both MTU 1500 (left) and MTU 9000 (right)

The switched TCP performance is a very close call between all networking solutions, as can be seen in figure 5.3. The throughput of all three kernel modules is exponentially decreasing as was expected. Displayed is that the MTU 9000 performance gives 2Gbit/sec extra performance over MTU 1500.

5.2.2 UDP

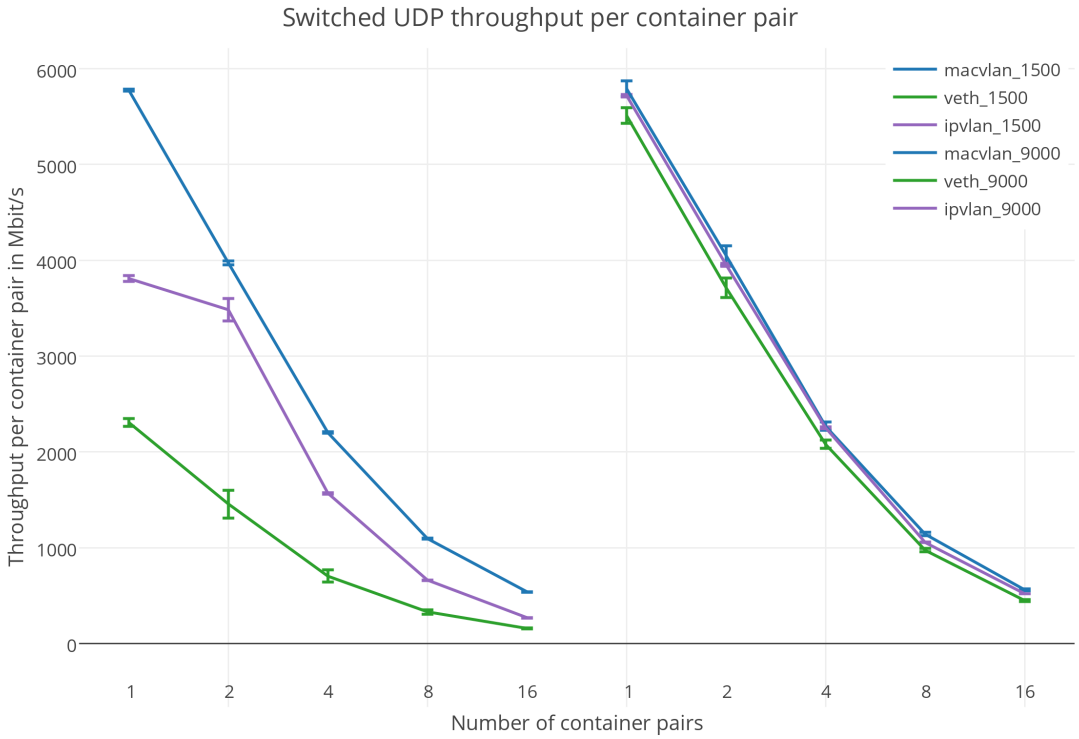

Figure 5.4: Switched UDP test results for MTU 1500 (left) and MTU 9000 (right)

Again, the UDP performance graphs are not so straight forward. Figure 5.4 shows that on MTU 1500 macvlan (brdige mode) outperforms both ipvlan (L3 mode) and veth bridges. As with local UDP testing the CPU gets fully utilized using these benchmarks. The MTU 9000 results show that performance is really close and exponentially decreasing when there are less frames being sent.

6

Conclusion

When this project started, the aim was to get a comprehensive overview of most networking solutions available for containers, and what their relative performance tradeoffs were.

There are a lot of different options to network containers. Whereas most overlay solutions are not quite production ready just yet, they are in very active development and should be ready for testing soon. Though overlay networks are nice for easy configuration, they per definition impose overhead. Therefore, where possible, it would be preferable to route the traffic over the public network without any tunnels (the traffic should of course still be encrypted). Project Calico seems like a nice step in-between, but it still imposes overhead through running extra services in containers.

There was specific interest in the performance of kernel modules used to connect containers to network devices. This resulted in a performance evaluation of these modules.

Only veth and macvlan are production ready for container deployments yet. ipvlan in L3 mode is getting there, but L2 mode is still buggy and getting patched. For the best local performance within a node, macvlan (bridge mode) should be considered. In most cases, the performance upsides outweigh the limits that a separate MAC address for each container impose. There is a downside for very massive container deployments on one NIC: the device could go into promiscuous mode if there are too many MAC addresses associated to it. This did not occur in my tests, but they where limited to 256 containers per node. If the container deployment is in a shortage of IP(v6) resources

(and there should really be no reason for this), veth bridges can be used in conjunction with NAT as a stopgap.

In switched environments there is really not so much of a performance difference, but the features of macvlan could be attractive for a lot of applications. A separate MAC address on the wire allows for better separation on the network level.

6.1 Future work

There are some results in that are not completely explainable:

- The strangely low UDP performance which has also been reported on in OpenStack environments could be looked into and hopefully further explained.

There is also some work that could not be done yet:

- The performance of ipvlan (L2 mode) should be reevaluated after it has been correctly patched to at least allow for reliable ARP support.
- The functionality and performance of different overlay networks could be (re)evaluated. Weave is working on their fast datapath version, and the Socketplane team is busy implementing new functions in libnetwork. There are numerous third parties working on Docker networking plugins that can go live after the new plugin system launch.

References

[1] Ieee 802.1: 802.1qbg - edge virtual bridging. URL http://www.ieee802.org/1/pages/802.1bg.html. 10

[2] App container • github. URL https://github.com/appc. 1

[3] Docker - build, ship and run any app, anywhere, . URL https://www.docker.com/. 1

[4] socketplane/docker-ovs • github, . URL https://github.com/socketplane/docker-ovs. 9

[5] iperf3 - iperf3 3.0.11 documentation. URL http://software.es.net/iperf/. 13

[6] [patch next 1/3] ipvlan: Defer multicast / broadcast processing to a work-queue. URL https://www.mail-archive.com/netdev%40vger.kernel.org/msg63498.html. 16

[7] Kvm. URL http://www.linux-kvm.org/. 5

[8] docker/libcontainer • github, . URL https://github.com/docker/libcontainer. 1

[9] docker/libnetwork, . URL https://github.com/docker/libnetwork. 8

[10] Linux containers. URL https://linuxcontainers.org/. 1

[11] Open container project. URL https://www.opencontainers.org/. 1

[12] Openflow - open networking foundation, . URL https://www.opennetworking.org/sdn-resources/openflow. 9

[13] Open vswitch, . URL http://openvswitch.org/. 7

[14] Switching performance chaining ovs bridges — open cloud blog, . URL http://www.opencloudblog.com/?p=386. 9

[15] Virtual ethernet device - openvz virtuozzo containers wiki, . URL https://openvz.org/Virtual_Ethernet_device. 8

[16] jpetazzo/pipework • github. URL https://github.com/jpetazzo/pipework. 13

[17] Project calico — a pure layer 3 approach to virtual networking. URL http://www.projectcalico.org/. 7

[18] Coreos is building a container runtime, rkt. URL https://coreos.com/blog/rocket/. 1

[19] Virtuozzo - odin. URL http://www.odin.com/products/virtuozzo/. 8

[20] Weaveworks • weave - all you need to connect, observe and control your containers, . URL http://weave.works/. 5, 6

[21] Weave fast datapath — all about weave, . URL http://blog.weave.works/2015/06/12/weave-fast-datapath/. 7

[22] C. Costache, O. Machidon, A. Mladin, F. Sandu, and R. Bocu. Software-defined networking of linux containers. *RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference, 2014*, 9 2014. doi: http://dx.doi.org/10.1109/RoEduNet-RENAM.2014.6955310. 5

[23] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. IBM Research Report RC25482 (AUS1407-001). 5

[24] Nane Kratzke. About microservices, containers and their underestimated impact on network performance. In *Proceedings of CLOUD COMPUTING 2015 (6th. International Conference on Cloud Computing, GRIDS and Virtualization)*, pages 165–169, 2015. 5, 7

[25] Victor Marmol, Rohit Jnagal, and Tim Hockin. Networking in containers and container clusters. *Proceedings of netdev 0.1*, February 2015. URL http://people.netfilter.org/pablo/netdev0.1/papers/Networking-in-Containers-and-Container-Clusters.pdf. 5

Appendix A

server_ipvlan

```
#!/bin/bash
if [ -z "$1" ]; then
  echo "Enter number of containers to be created:"
  exit 1
fi
for ((i=1; i<=$1; i++)); do
  echo "Creating server iperf3_s_$i listening on port 5201..."
  docker run -dit --net=none --name=iperf3_s_$i iperf/iperf
  echo "Attaching pipework ipvlan interface 10.0.1.$i/16"
  sudo ./pipework/pipework enp5s0 --ipvlan iperf3_s_$i 10.0.1.
     ↪ $i/16
done

wait

for (( i=1; i <= $1; i++ ))
do
  echo "Running iperf3 on server iperf3_s_$i"
  docker exec iperf3_s_$i iperf3 -s -D
done
```

server_macvlan

```
#!/bin/bash
if [ -z "$1" ]; then
  echo "Enter number of containers to be created:"
  exit 1
fi
for ((i=1; i<=$1; i++)); do
```

```
  echo "Creating server iperf3_s_$i listening on port 5201..."
  docker run -dit --net=none --name=iperf3_s_$i iperf/iperf
  echo "Attaching pipework macvlan interface 10.0.1.$i/16"
  sudo ./pipework/pipework enp5s0 iperf3_s_$i 10.0.1.$i/16
done

wait

for (( i=1; i <= $1; i++ ))
do
  echo "Running iperf3 on server iperf3_s_$i"
  docker exec iperf3_s_$i iperf3 -s -D
done
```

server_veth

```
#!/bin/bash
if [ -z "$1" ]; then
  echo "Enter number of containers to be created:"
  exit 1
fi
for ((i=1; i<=$1; i++)); do
  echo "Creating server iperf3_s_$i listening on port $((5200+
    ↪ $i))..."
  docker run -dit --name=iperf3_s_$i -p $((5200+$i)):$((5200+
    ↪ $i)) -p $((5200+$i)):$((5200+$i))/udp iperf/test
  docker exec iperf3_s_$i iperf3 -s -p $((5200+$i)) -D
done
```

Appendix B

client_ipvlan_tcp

```
#!/bin/bash
if [ -z "$1" ] || [ -z "$2" ] || [ -z "$3" ]; then
  echo "run ./script <number of containers> <mtu> <number of
     ↪ times to run>"
  exit 1
fi

for (( i=1; i <= $1; i++ ))
do
  echo "Creating client iperf3_c_$i ..."
  docker run -dit --net=none --name=iperf3_c_$i iperf/iperf
  wait
  echo "Attaching pipework ipvlan interface 10.0.2.$i/16"
  sudo ./pipework/pipework enp5s0 --ipvlan iperf3_c_$i 10.0.2.
     ↪ $i/16
done
wait
sleep 5

for (( j=1; j <= $3; j++ )); do
  for (( i=1; i <= $1; i++ )); do
    echo "Running iperf3 on client iperf3_c_$i for time $j..."
    docker exec iperf3_c_$i iperf3 -c 10.0.1.$i -p 5201 -f m -
       ↪ O 1 -M $2 > iperf3_$i.log &
  done
  wait
  for (( i=1; i <= $1; i++ )); do
    cat iperf3_$i.log | awk 'FNR == 17 {print}' | awk -F" " '{
       ↪ print $5","",$6","$7","",$8}' >> ~/
       ↪ output_$1_$2_$3_$i.csv
```

```
    cat iperf3_$i.log | awk 'FNR == 17 {print}' | awk -F" " '{
      ↪ print $5,",",$6,"$7,",",$8}' >> ~/
      ↪ output_ipvlan_total_$1_$2_$3.csv
  done
done
```

client_ipvlan_udp

```
#!/bin/bash
if [ -z "$1" ] || [ -z "$2" ] || [ -z "$3" ]; then
  echo "run ./script <number of containers> <mtu> <number of
    ↪ times to run>"
  exit 1
fi

for (( i=1; i <= $1; i++ ))
do
  echo "Creating client iperf3_c_$i ..."
  docker run -dit --net=none --name=iperf3_c_$i iperf/iperf
  wait
  echo "Attaching pipework ipvlan interface 10.0.2.$i/16"
  sudo ./pipework/pipework enp5s0 --ipvlan iperf3_c_$i 10.0.2.
    ↪ $i/16
done
wait
sleep 5

for (( j=1; j <= $3; j++ )); do
  for (( i=1; i <= $1; i++ )); do
    echo "Running iperf3 on client iperf3_c_$i for time $j..."
    docker exec iperf3_c_$i iperf3 -u -c 10.0.1.$i -p 5201 -f
      ↪ m -O 1 -M $2 -b 10000M > iperf3_$i.log &
  done
  wait
  for (( i=1; i <= $1; i++ )); do
    cat iperf3_$i.log | awk 'FNR == 17 {print}' | awk -F" " '{
      ↪ print $5,",",$6,"$7,",",$8,",",$9,",",$10,",",$11
      ↪ ,",",$12}' >> ~/output_$1_$2_$3_$i.csv
    cat iperf3_$i.log | awk 'FNR == 17 {print}' | awk -F" " '{
      ↪ print $5,",",$6,"$7,",",$8,",",$9,",",$10,",",$11
      ↪ ,",",$12}' >> ~/output_ipvlan_total_$1_$2_$3.csv
  done
```

```
done
```

client_macvlan_tcp

```
#!/bin/bash
if [ -z "$1" ] || [ -z "$2" ] || [ -z "$3" ]; then
  echo "run ./script <number of containers> <mtu> <number of
     ↪ times to run>"
  exit 1
fi

for (( i=1; i <= $1; i++ ))
do
  echo "Creating client iperf3_c_$i ..."
  docker run -dit --net=none --name=iperf3_c_$i iperf/iperf
  wait
  echo "Attaching pipework macvlan interface 10.0.2.$i/16"
  sudo ./pipework/pipework enp5s0 iperf3_c_$i 10.0.2.$i/16
done
wait
sleep 5

for (( j=1; j <= $3; j++ )); do
  for (( i=1; i <= $1; i++ )); do
    echo "Running iperf3 on client iperf3_c_$i for time $j..."
    docker exec iperf3_c_$i iperf3 -c 10.0.1.$i -p 5201 -f m -
       ↪ O 1 -M $2 > iperf3_$i.log &
  done
  wait
  for (( i=1; i <= $1; i++ )); do
    cat iperf3_$i.log | awk 'FNR == 17 {print}' | awk -F" " '{
       ↪ print $5","",$6","$7","",$8}' >> ~/
       ↪ output_$1_$2_$3_$i.csv
    cat iperf3_$i.log | awk 'FNR == 17 {print}' | awk -F" " '{
       ↪ print $5","",$6","$7","",$8}' >> ~/
       ↪ output_macvlan_total_$1_$2_$3.csv
  done
done
```

client_macvlan_udp

```
#!/bin/bash
if [ -z "$1" ] || [ -z "$2" ] || [ -z "$3" ]; then
  echo "run ./script <number of containers> <mtu> <number of
     ↪ times to run>"
  exit 1
fi

for (( i=1; i <= $1; i++ ))
do
  echo "Creating client iperf3_c_$i ..."
  docker run -dit --net=none --name=iperf3_c_$i iperf/iperf
  wait
  echo "Attaching pipework macvlan interface 10.0.2.$i/16"
  sudo ./pipework/pipework enp5s0 iperf3_c_$i 10.0.2.$i/16
done
wait
sleep 5

for (( j=1; j <= $3; j++ )); do
  for (( i=1; i <= $1; i++ )); do
    echo "Running iperf3 on client iperf3_c_$i for time $j..."
    docker exec iperf3_c_$i iperf3 -u -c 10.0.1.$i -p 5201 -f
       ↪ m -O 1 -M $2 -b 10000M > iperf3_$i.log &
  done
  wait
  for (( i=1; i <= $1; i++ )); do
    cat iperf3_$i.log | awk 'FNR == 17 {print}' | awk -F" " '{
       ↪ print $5","$6","$7","$8","$9","$10","$11
       ↪ ","$12}' >> ~/output_$1_$2_$3_$i.csv
    cat iperf3_$i.log | awk 'FNR == 17 {print}' | awk -F" " '{
       ↪ print $5","$6","$7","$8","$9","$10","$11
       ↪ ","$12}' >> ~/output_macvlan_total_$1_$2_$3.csv
  done
done
```

client_veth_tcp

```
#!/bin/bash
if [ -z "$1" ] || [ -z "$2" ] || [ -z "$3" ]; then
  echo "run ./script <number of containers> <mtu> <number of
     ↪ times to run>"
```

```
  exit 1
fi

for (( i=1; i <= $1; i++ ))
do
  echo "Creating client iperf3_c_$i ..."
  docker run -dit --name=iperf3_c_$i iperf/iperf
done
wait
sleep 5

for (( j=1; j <= $3; j++ )); do
  for (( i=1; i <= $1; i++ )); do
    echo "Running iperf3 on client iperf3_c_$i for time $j..."
    docker exec iperf3_c_$i iperf3 -c 10.0.0.1 -p $((5200+$i))
      ↪ -f m -O 1 -M $2 > iperf3_$i.log &
  done
  wait
  for (( i=1; i <= $1; i++ )); do
    cat iperf3_$i.log | awk 'FNR == 17 {print}' | awk -F" " '{
      ↪ print $5","",$6","$7","",$8}' >> ~/
      ↪ output_$1_$2_$3_$i.csv
    cat iperf3_$i.log | awk 'FNR == 17 {print}' | awk -F" " '{
      ↪ print $5","",$6","$7","",$8}' >> ~/
      ↪ output_veth_total_$1_$2_$3.csv
  done
done
```

client_veth_udp

```
#!/bin/bash
if [ -z "$1" ] || [ -z "$2" ] || [ -z "$3" ]; then
  echo "run ./script <number of containers> <mtu> <number of
    ↪ times to run>"
  exit 1
fi

for (( i=1; i <= $1; i++ ))
do
  echo "Creating client iperf3_c_$i ..."
  docker run -dit --name=iperf3_c_$i iperf/iperf
done
```

```
wait
sleep 5

for (( j=1; j <= $3; j++ )); do
  for (( i=1; i <= $1; i++ )); do
    echo "Running iperf3 on client iperf3_c_$i for time $j..."
    docker exec iperf3_c_$i iperf3 -u -c 10.0.0.1 -p $((5200+
      ↪ $i)) -f m -O 1 -M $2 -b 10000M > iperf3_$i.log &
  done
  wait
  for (( i=1; i <= $1; i++ )); do
    cat iperf3_$i.log | awk 'FNR == 17 {print}' | awk -F" " '{
      ↪ print $5","$6","$7","$8","$9","$10","$11
      ↪ ","$12}' >> ~/output_$1_$2_$3_$i.csv
    cat iperf3_$i.log | awk 'FNR == 17 {print}' | awk -F" " '{
      ↪ print $5","$6","$7","$8","$9","$10","$11
      ↪ ","$12}' >> ~/output_veth_total_$1_$2_$3.csv
  done
done
```

Appendix C

pipework

```
#!/bin/sh
# This code should (try to) follow Google's Shell Style Guide
# (https://google-styleguide.googlecode.com/svn/trunk/shell.
    ↪ xml)
set -e

case "$1" in
  --wait)
    WAIT=1
    ;;
esac

IFNAME=$1

# default value set further down if not set here
IPVLAN=
if [ "$2" = "--ipvlan" ]; then
  IPVLAN=1
  shift 1
fi

CONTAINER_IFNAME=
if [ "$2" = "-i" ]; then
  CONTAINER_IFNAME=$3
  shift 2
fi

HOST_NAME_ARG=""
if [ "$2" == "-H" ]; then
  HOST_NAME_ARG="-H $3"
```

```
  shift 2
fi

GUESTNAME=$2
IPADDR=$3
MACADDR=$4

case "$MACADDR" in
  *@*)
    VLAN="${MACADDR#*@}"
    VLAN="${VLAN%%@*}"
    MACADDR="${MACADDR%%@*}"
    ;;
  *)
    VLAN=
    ;;
esac

[ "$IPADDR" ] || [ "$WAIT" ] || {
  echo "Syntax:"
  echo "pipework <hostinterface> [--ipvlan] [-i
    ↪ containerinterface] <guest> <ipaddr>/<subnet>[
    ↪ @default_gateway] [macaddr][@vlan]"
  echo "pipework <hostinterface> [--ipvlan] [-i
    ↪ containerinterface] <guest> dhcp [macaddr][@vlan]"
  echo "pipework --wait [-i containerinterface]"
  exit 1
}

# Succeed if the given utility is installed. Fail otherwise.
# For explanations about `which` vs `type` vs `command`, see:
# http://stackoverflow.com/questions/592620/check-if-a-program
  ↪ -exists-from-a-bash-script/677212#677212
# (Thanks to @chenhanxiao for pointing this out!)
installed () {
  command -v "$1" >/dev/null 2>&1
}

# Google Styleguide says error messages should go to standard
  ↪ error.
warn () {
  echo "$@" >&2
}
```

```
die () {
  status="$1"
  shift
  warn "$@"
  exit "$status"
}

# First step: determine type of first argument (bridge,
    ↪ physical interface...),
# Unless "--wait" is set (then skip the whole section)
if [ -z "$WAIT" ]; then
  if [ -d "/sys/class/net/$IFNAME" ]
  then
    if [ -d "/sys/class/net/$IFNAME/bridge" ]; then
      IFTYPE=bridge
      BRTYPE=linux
    elif installed ovs-vsctl && ovs-vsctl list-br|grep -q "^${
        ↪ IFNAME}$"; then
      IFTYPE=bridge
      BRTYPE=openvswitch
    elif [ "$(cat "/sys/class/net/$IFNAME/type")" -eq 32 ];
        ↪ then # Infiniband IPoIB interface type 32
      IFTYPE=ipoib
      # The IPoIB kernel module is fussy, set device name to
          ↪ ib0 if not overridden
      CONTAINER_IFNAME=${CONTAINER_IFNAME:-ib0}
    else IFTYPE=phys
    fi
  else
    case "$IFNAME" in
      br*)
        IFTYPE=bridge
        BRTYPE=linux
        ;;
      ovs*)
        if ! installed ovs-vsctl; then
          die 1 "Need OVS installed on the system to create an
              ↪  ovs bridge"
        fi
        IFTYPE=bridge
        BRTYPE=openvswitch
        ;;
```

```
      *) die 1 "I do not know how to setup interface $IFNAME."
         ↪ ;;
    esac
  fi
fi

# Set the default container interface name to eth1 if not
   ↪ already set
CONTAINER_IFNAME=${CONTAINER_IFNAME:-eth1}

[ "$WAIT" ] && {
  while true; do
    # This first method works even without `ip` or `ifconfig`
       ↪ installed,
    # but doesn't work on older kernels (e.g. CentOS 6.X). See
       ↪ #128.
    grep -q '^1$' "/sys/class/net/$CONTAINER_IFNAME/carrier"
       ↪ && break
    # This method hopefully works on those older kernels.
    ip link ls dev "$CONTAINER_IFNAME" && break
    sleep 1
  done > /dev/null 2>&1
  exit 0
}

[ "$IFTYPE" = bridge ] && [ "$BRTYPE" = linux ] && [ "$VLAN" ]
   ↪ && {
  die 1 "VLAN configuration currently unsupported for Linux
     ↪ bridge."
}

[ "$IFTYPE" = ipoib ] && [ "$MACADDR" ] && {
  die 1 "MACADDR configuration unsupported for IPoIB
     ↪ interfaces."
}

# Second step: find the guest (for now, we only support LXC
   ↪ containers)
while read _ mnt fstype options _; do
  [ "$fstype" != "cgroup" ] && continue
  echo "$options" | grep -qw devices || continue
  CGROUPMNT=$mnt
done < /proc/mounts
```

```
[ "$CGROUPMNT" ] || {
    die 1 "Could not locate cgroup mount point."
}

# Try to find a cgroup matching exactly the provided name.
N=$(find "$CGROUPMNT" -name "$GUESTNAME" | wc -l)
case "$N" in
  0)
    # If we didn't find anything, try to lookup the container
    ↪ with Docker.
    if installed docker; then
      RETRIES=3
      while [ "$RETRIES" -gt 0 ]; do
        DOCKERPID=$(docker inspect --format='{{ .State.Pid }}'
        ↪ "$GUESTNAME")
        [ "$DOCKERPID" != 0 ] && break
        sleep 1
        RETRIES=$((RETRIES - 1))
      done

      [ "$DOCKERPID" = 0 ] && {
        die 1 "Docker inspect returned invalid PID 0"
      }

      [ "$DOCKERPID" = "<no value>" ] && {
        die 1 "Container $GUESTNAME not found, and unknown to
        ↪ Docker."
      }
    else
      die 1 "Container $GUESTNAME not found, and Docker not
      ↪ installed."
    fi
    ;;
  1) true ;;
  *) die 1 "Found more than one container matching $GUESTNAME
  ↪ ." ;;
esac

if [ "$IPADDR" = "dhcp" ]; then
  # Check for first available dhcp client
  DHCP_CLIENT_LIST="udhcpc dhcpcd dhclient"
  for CLIENT in $DHCP_CLIENT_LIST; do
```

```
    installed "$CLIENT" && {
      DHCP_CLIENT=$CLIENT
      break
    }
  done
  [ -z "$DHCP_CLIENT" ] && {
    die 1 "You asked for DHCP; but no DHCP client could be
      ↪ found."
  }
else
  # Check if a subnet mask was provided.
  case "$IPADDR" in
    */*) : ;;
    *)
      warn "The IP address should include a netmask."
      die 1 "Maybe you meant $IPADDR/24 ?"
      ;;
  esac
  # Check if a gateway address was provided.
  case "$IPADDR" in
    *@*)
      GATEWAY="${IPADDR#*@}" GATEWAY="${GATEWAY%%@*}"
      IPADDR="${IPADDR%%@*}"
      ;;
    *)
      GATEWAY=
      ;;
  esac
fi

if [ "$DOCKERPID" ]; then
  NSPID=$DOCKERPID
else
  NSPID=$(head -n 1 "$(find "$CGROUPMNT" -name "$GUESTNAME" |
    ↪ head -n 1)/tasks")
  [ "$NSPID" ] || {
    die 1 "Could not find a process inside container
      ↪ $GUESTNAME."
  }
fi

# Check if an incompatible VLAN device already exists
```

```
[ "$IFTYPE" = phys ] && [ "$VLAN" ] && [ -d "/sys/class/net/
    ↪ $IFNAME.VLAN" ] && {
  ip -d link show "$IFNAME.$VLAN" | grep -q "vlan.*id $VLAN"
      ↪ || {
    die 1 "$IFNAME.VLAN already exists but is not a VLAN
        ↪ device for tag $VLAN"
  }
}

[ ! -d /var/run/netns ] && mkdir -p /var/run/netns
rm -f "/var/run/netns/$NSPID"
ln -s "/proc/$NSPID/ns/net" "/var/run/netns/$NSPID"

# Check if we need to create a bridge.
[ "$IFTYPE" = bridge ] && [ ! -d "/sys/class/net/$IFNAME" ] &&
    ↪  {
  [ "$BRTYPE" = linux ] && {
    (ip link add dev "$IFNAME" type bridge > /dev/null 2>&1)
        ↪ || (brctl addbr "$IFNAME")
    ip link set "$IFNAME" up
  }
  [ "$BRTYPE" = openvswitch ] && {
    ovs-vsctl add-br "$IFNAME"
  }
}

MTU=$(ip link show "$IFNAME" | awk '{print $5}')
# If it's a bridge, we need to create a veth pair
[ "$IFTYPE" = bridge ] && {
  LOCAL_IFNAME="v${CONTAINER_IFNAME}pl${NSPID}"
  GUEST_IFNAME="v${CONTAINER_IFNAME}pg${NSPID}"
  ip link add name "$LOCAL_IFNAME" mtu "$MTU" type veth peer
      ↪ name "$GUEST_IFNAME" mtu "$MTU"
  case "$BRTYPE" in
    linux)
      (ip link set "$LOCAL_IFNAME" master "$IFNAME" > /dev/
          ↪ null 2>&1) || (brctl addif "$IFNAME" "
          ↪ $LOCAL_IFNAME")
      ;;
    openvswitch)
      ovs-vsctl add-port "$IFNAME" "$LOCAL_IFNAME" ${VLAN:+tag
          ↪ ="$VLAN"}
      ;;
```

```
  esac
  ip link set "$LOCAL_IFNAME" up
}

# Note: if no container interface name was specified, pipework
    ↪ will default to ib0
# Note: no macvlan subinterface or ethernet bridge can be
    ↪ created against an
# ipoib interface. Infiniband is not ethernet. ipoib is an IP
    ↪ layer for it.
# To provide additional ipoib interfaces to containers use SR-
    ↪ IOV and pipework
# to assign them.
[ "$IFTYPE" = ipoib ] && {
  GUEST_IFNAME=$CONTAINER_IFNAME
}

# If it's a physical interface, create a macvlan subinterface
[ "$IFTYPE" = phys ] && {
  [ "$VLAN" ] && {
    [ ! -d "/sys/class/net/${IFNAME}.${VLAN}" ] && {
      ip link add link "$IFNAME" name "$IFNAME.$VLAN" mtu "
          ↪ $MTU" type vlan id "$VLAN"
    }
    ip link set "$IFNAME" up
    IFNAME=$IFNAME.$VLAN
  }
  GUEST_IFNAME=ph$NSPID$CONTAINER_IFNAME
  [ "$IPVLAN" ] && {
    ip link add link "$IFNAME" "$GUEST_IFNAME" mtu "$MTU" type
        ↪ ipvlan mode l3
  }
  [ ! "$IPVLAN" ] && {
    ip link add link "$IFNAME" dev "$GUEST_IFNAME" mtu "$MTU"
        ↪ type macvlan mode bridge
    ip link set "$IFNAME" up
  }
}

ip link set "$GUEST_IFNAME" netns "$NSPID"
ip netns exec "$NSPID" ip link set "$GUEST_IFNAME" name "
    ↪ $CONTAINER_IFNAME"
```

```
[ "$MACADDR" ] && ip netns exec "$NSPID" ip link set dev "
    ↪ $CONTAINER_IFNAME" address "$MACADDR"
if [ "$IPADDR" = "dhcp" ]
then
  [ "$DHCP_CLIENT" = "udhcpc" ] && ip netns exec "$NSPID" "
     ↪ $DHCP_CLIENT" -qi "$CONTAINER_IFNAME" -x "hostname:
     ↪ $GUESTNAME"
  if [ "$DHCP_CLIENT" = "dhclient" ]; then
    # kill dhclient after get ip address to prevent device be
       ↪ used after container close
    ip netns exec "$NSPID" "$DHCP_CLIENT" $HOST_NAME_ARG -pf
       ↪ "/var/run/dhclient.$NSPID.pid" "$CONTAINER_IFNAME"
    kill "$(cat "/var/run/dhclient.$NSPID.pid")"
    rm "/var/run/dhclient.$NSPID.pid"
  fi
  [ "$DHCP_CLIENT" = "dhcpcd" ] && ip netns exec "$NSPID" "
     ↪ $DHCP_CLIENT" -q "$CONTAINER_IFNAME" -h "$GUESTNAME"
else
  ip netns exec "$NSPID" ip addr add "$IPADDR" dev "
     ↪ $CONTAINER_IFNAME"
  [ "$GATEWAY" ] && {
    ip netns exec "$NSPID" ip route delete default >/dev/null
       ↪ 2>&1 && true
  }
  ip netns exec "$NSPID" ip link set "$CONTAINER_IFNAME" up
  [ "$GATEWAY" ] && {
    ip netns exec "$NSPID" ip route get "$GATEWAY" >/dev/null
       ↪ 2>&1 || \
    ip netns exec "$NSPID" ip route add "$GATEWAY/32" dev "
       ↪ $CONTAINER_IFNAME"
    ip netns exec "$NSPID" ip route replace default via "
       ↪ $GATEWAY"
  }
fi

# Give our ARP neighbors a nudge about the new interface
if installed arping; then
  IPADDR=$(echo "$IPADDR" | cut -d/ -f1)
  ip netns exec "$NSPID" arping -c 1 -A -I "$CONTAINER_IFNAME"
     ↪  "$IPADDR" > /dev/null 2>&1 || true
else
  echo "Warning: arping not found; interface may not be
     ↪ immediately reachable"
```

```
fi

# Remove NSPID to avoid `ip netns` catch it.
rm -f "/var/run/netns/$NSPID"

# vim: set tabstop=2 shiftwidth=2 softtabstop=2 expandtab :
```