Linux containers networking: performance and scalability of kernel modules

Joris Claassen*, Ralph Koning*, Paola Grosso*
*System and Network Engineering research group
University of Amsterdam
joris.claassen@gmail.com, r.koning@uva.nl, p.grosso@uva.nl

***Abstract*—Linux container virtualisation is gaining momentum as lightweight technology to support cloud and distributed computing. Applications relying on container architectures might at times rely on inter-container communication, and container networking solutions are emerging to address this need. Containers can be networked together as part of an overlay network, or with actual links from the container to the network via kernel modules. Most overlay solutions are not quite production ready yet; on the other hand kernel modules that can link a container to the network are much more mature. We benchmarked three kernel modules: *veth, macvlan* and *ipvlan*, to quantify their respective raw TCP and UDP performance and scalability. Our results show that the *macvlan* kernel module outperforms all other solutions in raw performance. All kernel modules seem to provide sufficient scalability to be deployed effectively in multi-containers environments.**

I. Introduction

Operating-system-level virtualization, or containers, are becoming a mainstream technology to support cloud and distributed computing applications. They are a lightweight alternative to full virtualization of virtual machines. Therefore, containers are being adopted in commercial applications and in high-performance scientific environments[1].

In contrast to virtual machines, where isolation is achieved at machine level and each virtual machine runs its own operating system, containers are isolated at kernel level using individual namespaces. Filesystem, process IDs (PID), inter-process communication (IPC) and network are some of the types of resources that can be isolated through namespaces[2].

Docker is the de-facto container choice, it used to rely on *lxc* [3] for containerisation, but now uses their own libcontainer [4] from version 1.0. Yet, containers remain interchangeable by using the open container standard, a standard developed by the Open Container Project *ocp*[5], based on AppC[6]. The open container standard allows vendors to compete using their own application container runtime and supporting services, e.g. clustering, scheduling and/or overlay networking. A system consists of one or more containers running on one or more nodes, which need to communicate amongst each other.

While container technologies are mature and growing in popularity, the networking solutions are still being developed. Two major classes of networking solutions for containers exist: overlay networking solutions and kernel module solutions. This work provides an overview and performance evaluation of the kernel module solutions. We were particularly interested in assessing the performance for TCP and UDP throughput while scaling up the amount of containers in single and multi node environments.

978-1-5090-0223-8/16/$31.00

II. Container technology

Containers use namespaces for isolation. The combination of several isolated namespaces provides a workspace for a container. Every container is limited to the resources that are assigned to it by its namespaces. The Namespaces that can be used by containers are: 1.PID - Used to isolation processes from each other; NET - Used to isolate network devices within a container; MNT - Used to isolate mount points; IPC - Used to isolate access to IPC resources; UTS - Used to isolate kernel and version identifiers.

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. *cgroups* limit the available hardware resources to each container. For example the amount of CPU cycles that a container can use.

Docker provides a service to easily deploy a container, using Namespaces and *cgroups*, 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 a specific container. These layers are implemented using storage backends, most of which are Copy-on-Write (COW), but there is also a non-COW fallback backend. Dockerfiles can be used to build images, which can be shared in a docker repository e.g. Docker Hub.

III. Container networking

An approach to provide network connectivity to containers is using Linux kernel modules, which can be used to create virtual network devices. These devices can then be attached to the correct network namespace. Four major modules are currently available and in the following sections we will describe them in detail.

1. veth.

The *veth* kernel module creates a pair of (virtual) networking devices that are connected to eachother. One of the ends can then be put in a different namespace, as visualised in the top part of Fig. 1. This technology was pioneered by Odin's Virtuozzo[7] and its open counterpart OpenVZ[8].

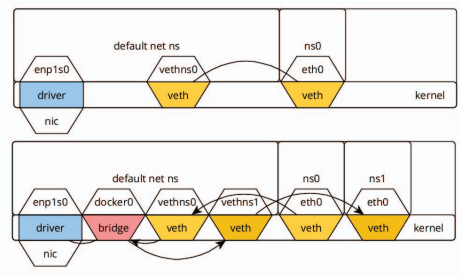

Fig. 1. Visualization of the veth network pipes

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 data stream from *ns0* to *ns1* using *veth* pipes (bridged) is visualised in the bottom part of Fig. 1.

2. Open vSwitch.

The *Open vSwitch* (OVS) kernel module comes as part of the mainline Linux kernel, but is operated by a separate piece of software. Open vSwitch provides a virtual switch, which also supports OpenFlow[9]. docker-ovs[10] uses *veth* pairs, to connect to the Open vSwitch in a similar way to Linux bridges using *veth* pairs. Open vSwitch in combination with *veth* pairs, shows a significant decrease in performance when running multipe threads. One solution is to use Open vSwitch patches [11].

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 shown in Fig. 2.

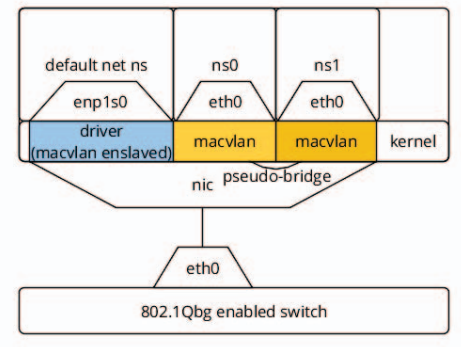

Fig. 2. Visualization of the macvlan kernel module

The new devices have their own MAC address and are placed within 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[12] (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 behaviour of a switch not to forward packets back to the port they came from it is effectively private mode.

4. ipvlan.

ipvlan is very similar to *macvlan*, it also enslaves the driver of the NIC in kernel space; however, it differs from *macvlan* because the packets sent all get the same MAC address on the wire. Forwarding to the correct virtual device is done based on the layer 3 address. *ipvlan* has two modes of operation:

- L2 mode - device behaves as a layer 2 device; all transmit 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 transmit 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.

IV. Tests

To quantify the current performance and scalability of container networking we compared three of the network techniques from section III: *veth* bridges, *macvlan* (in bridge mode), and *ipvlan* (in L3 mode).

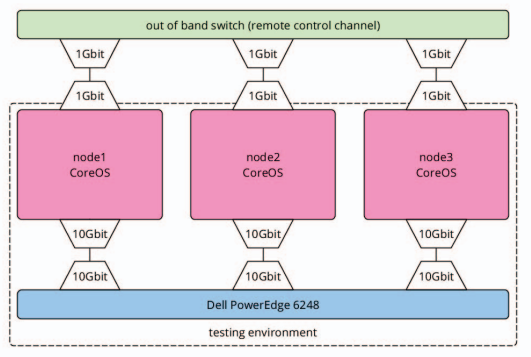

Fig. 3. Physical test setup: three physical nodes are interconnected via a 10gbps Ethernet switch

For our benchmark tests, we created the infrastructure shown in Fig. 3. For our experiments, we used three nodes running CoreOS, an OS with is focussed on running containers

and is actively mainained. The nodes were equipped with 3x Intel(R) Xeon(R) CPU E5620 @ 2.40GHz CPUs, had 24GB DDR3 1600Mhz RAM and 10Gb LR SM SFP+ networking interfaces. To connect the nodes we used a Dell PowerConnect 6248 equipped with three 10Gb LR SM SFP+ interfaces. Our testing environment was fully isolated so there where no external factors influencing the test results. Tests were done from within the containers using the *iperf3*[13] (3.0.11), a complete and more efficient rewrite of the original *iperf*. The tools we build for testing, rely on a modified Jérôme Petazzoni's excellent Pipework[14] tool to add support for ipvlan (in L3 mode).

We defined two types of tests scenarios: local and switched tests.

1) local tests: (Fig. 4) shows the local setup. A varying number of containers pairs (from 2 containers-1 pair up to 256 containers-128 pairs) where launched on one node, running 128 data streams at a time. The scrips where launched from the node itself to minimise any latencies.

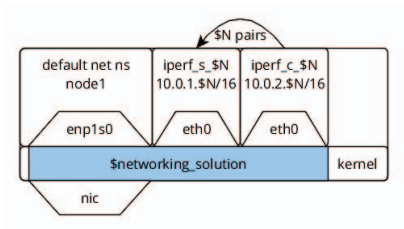

Fig. 4. Local setup, where all the N container pairs are running on the same physical node.

2) switched test: Fig.5 shows the switched setup, where each container of a pair runs on a different physical node.

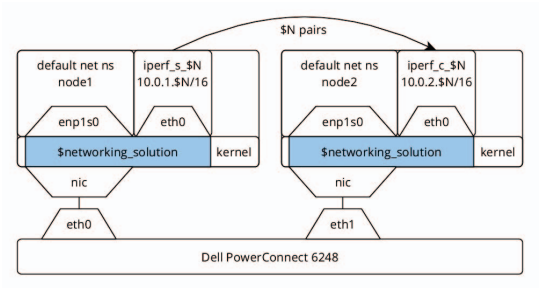

Fig. 5. Switched setup, where the N container pairs are running on two different physical nodes.

To ensure the timing of the bandwidth tests was correct we used node 3 to coordinate, by waiting for the command on node 1 to finish before sending a command to node 2, and run commands via SSH over an out-of-band switch. While the amount of data streams are the same as in the local test, the workload is split between two nodes; thus launching 1 up to 128 containers per node.

All tests ran using exponentially growing numbers ($N in Fig. 4 and Fig.5) of container pairs; ranging from 1 to 128 for TCP, and 1 to 16 for UDP. The lower number of pairs possible in the UDP tests was due the protocol filling up the line in such a way that *iperf3*'s control messages were blocked; this was particularly evident when running with higher number of container pairs, making measurements unreliable. The tests where repeated 10 times to calculate the standard error values. Due to the exponentially rising number of container pairs, an exponentially decreasing throughput was expected both for TCP and UDP. UDP performance was expected to be better than TCP because of the simpler protocol design. Furthermore we used two different MTU sizes for the Ethernet frames: the default 1500 bytes frame and the 9000 bytes Jumbo frames.

V. Results

In the following sections we present the results for the local and switched scenarios respectively.

A. Local testing results

1) TCP: As can be seen in Fig. 6, the *ipvlan* (L3 mode) kernel module achieves the best performance in our tests. Given 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 significant difference in performance behaviour between MTU 1500 and MTU 9000.

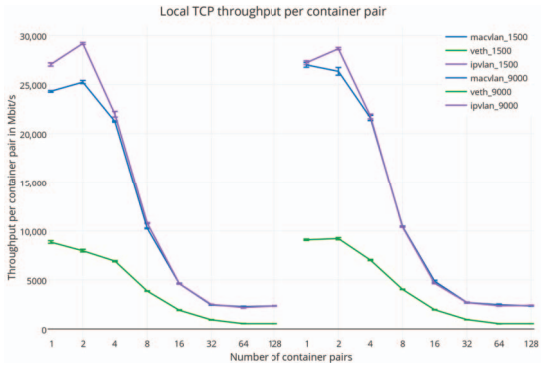

Fig. 6. Local TCP test results for MTU 1500 (left) and MTU 9000 (right)

2) UDP: The performance shown in Fig. 7 is different from the one of Fig. 6. The performance difference could be caused because iperf3 is consuming more CPU cycles in trying to measuring the jitter of the UDP packets 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 is required to explain 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 behaviour of exponential decreasing performance after two container pairs. *macvlan* (bridge mode) does perform reasonably well, probably thanks to the network pseudo-bridge running in the RAM. The total throughput is still 3.5 times as low as the one we observed with the *macvlan* (bridge mode) TCP traffic streams.

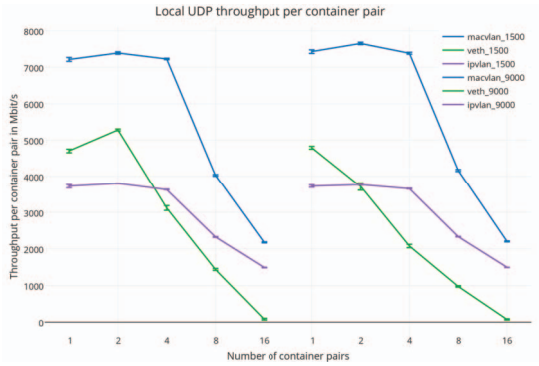

Fig. 7. Local UDP test results for MTU 1500 (left) and MTU 9000 (right)

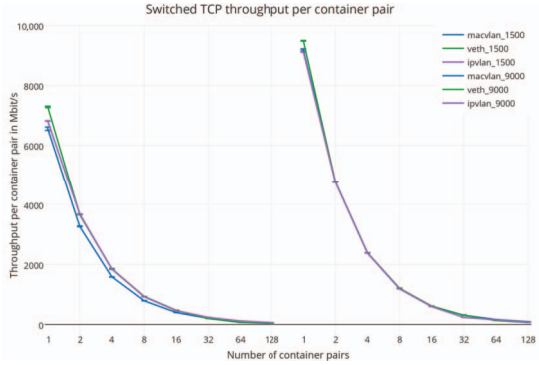

Fig. 8. Switched TCP test results for both MTU 1500 (left) and MTU 9000 (right)

B. Switched testing

1) TCP: Fig. 8shows that the throughput of all three kernel modules are similar and decreases exponentially, as we expected. Interesting to note that there is a 2Gbit/sec extra performance when using an MTU of 9000 instead of and MTU 1500.

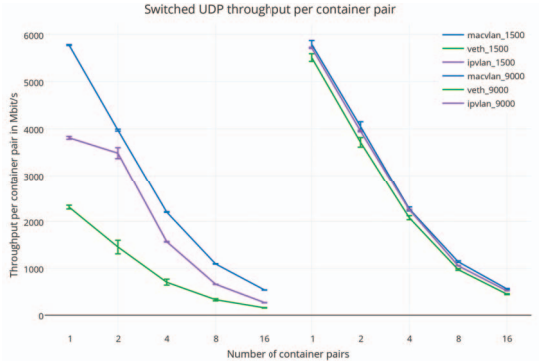

Fig. 9. Switched UDP test results for MTU 1500 (left) and MTU 9000 (right)

2) UDP: Fig. 9 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 utilised using our benchmarks. Using mtu 9000 results are close and decrease exponentially when less frames are sent.

VI. Related work

Containers are a fairly new technology; limited work has been carried out in the scientific community to assess their performance and consequently little literature is available to compare with. A recent paper by Felter et al. [15] does report on performance differences aspects between KVM [16] and Docker, but does not take into account different networking implementations. Given that containerised applications will rely heavily on inter-container communication we believe that understanding, evaluating and benchmarking the networking capabilities of containers is an area that requires focus.

Overlay and tunnelling technologies can also be used to interconnect containers. The impact of virtual ethernet bridges and SDVNs, in particular Weave, has already been researched on by Kratzke[17]. There is some related research done on containers and SDNs by Felter et al.[18]. Research on VXLAN, which is another tunnelling networking model that could apply to containers, is widely available. We consider these technologies still not mature enough and effectively less appealing for use in production environments so we decided to concentrate on the more mature kernel modules.

In *Networking in containers and container clusters*, Marmol et al.[19] provide a theoretical overview of the methods used in container networking; however, they do not provide a side-by-side comparison of kernel modules for container networking as in this paper.

VII. Conclusion

Given the emergence of containers in distributed cloud applications we were interested in assessing the maturity of networking solutions supporting container communication. We evaluated three kernel module solutions available for container networking *veth*, *macvlan*, and *ipvlan* in both single node and multi node environments. We conclude that for the best 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 imposes.

In switched environments we did not observe any significant performance difference, but the features of *macvlan* could be attractive for a lot of applications. In particular a separate MAC address on the wire allows for better separation on the network level.

VIII. Future work

The strangely low UDP performance, which has also been reported on in OpenStack environments requires further investigation. Also, *ipvlan* (L2 mode), needs to be evaluated after it has been correctly patched for reliable ARP support. Yet, the most interesting work is in investigating the functionality and performance of different overlay networks, as they will likely see the largest adoption among distributed containers applications.

Acknowledgment

Dr. Grosso would like to thank the Dutch national program COMMIT.

REFERENCES

[1] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, and C. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, Feb 2013, pp. 233–240.

[2] E. W. Biederman, "Multiple instances of the global linux namespaces," *Proc. of the Linux Symposium*, 2006.

[3] Linux containers. [Online]. Available: https://linuxcontainers.org/

[4] docker/libcontainer • github. [Online]. Available: https://github.com/docker/libcontainer

[5] Open container project. [Online]. Available: https://www.opencontainers.org/

[6] App container • github. [Online]. Available: https://github.com/appc

[7] Virtuozzo - odin. [Online]. Available: http://www.odin.com/products/virtuozzo/

[8] Virtual ethernet device - openvz virtuozzo containers wiki. [Online]. Available: https://openvz.org/VirtualEthernetdevice

[9] Openflow - open networking foundation. [Online]. Available: https://www.opennetworking.org/sdn-resources/openflow

[10] socketplane/docker-ovs • github. [Online]. Available: https://github.com/socketplane/docker-ovs

[11] Switching performance chaining ovs bridges — open cloud blog. [Online]. Available: http://www.opencloudblog.com/?p=386

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

[13] iperf3 - iperf3 3.0.11 documentation. [Online]. Available: http://software.es.net/iperf/

[14] jpetazzo/pipework • github. [Online]. Available: https://github.com/jpetazzo/pipework

[15] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," vol. IBM Research Report RC25482 (AUS1407-001).

[16] Kvm. [Online]. Available: http://www.linux-kvm.org/

[17] N. 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)*, 2015, pp. 165–169.

[18] 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.

[19] V. Marmol, R. Jnagal, and T. Hockin, "Networking in containers and container clusters," *Proceedings of netdev 0.1*, Feb. 2015. [Online]. Available: http://people.netfilter.org/pablo/netdev0.1/papers/Networking-in-Containers-and-Container-Clusters.pdf