Characterizing network performance of single-node large-scale container deployments

Conrado Boeira
Faculty of Computer Science
Dalhousie University
Conrado.Boeira@dal.ca

Miguel Neves
Faculty of Computer Science
Dalhousie University
mg478789@dal.ca

Tiago Ferreto
School of Technology
PUCRS
tiago.ferreto@pucrs.br

Israat Haque
Faculty of Computer Science
Dalhousie University
israat@dal.ca

***Abstract*—Cloud services have shifted from complex monolithic designs to hundreds of loosely coupled microservices over the last years. These microservices communicate via pre-defined APIs (e.g., RPC) and are usually implemented on top of containers. To make the microservices model profitable, cloud providers often co-locate them on a single (virtual) machine, thus achieving high server utilization. Despite being overlooked by previous work, the challenge of providing high-quality network connectivity to multiple containers running on the same host becomes crucial for the overall cloud service performance in this scenario. For that reason, this paper focuses on identifying the overheads and bottlenecks caused by the increasing number of concurrent containers running on a single node, particularly from a networking perspective. Through an extensive set of experiments, we show that the networking performance is mostly restricted by the CPU capacity (even for I/O intensive workloads), that containers can largely suffer from interference originated from packet processing, and that proper core scheduling policies can significantly improve connection throughput. Ultimately, our findings can help to pave the way towards more efficient large-scale microservice deployments.**

***Index Terms*—Cloud networking, Docker, microservices, performance evaluation.**

I. Introduction

Cloud services have progressively shifted from monolithic designs to hundreds of interconnected microservices over the last few years. Microservices are an appealing paradigm for several reasons, including modular application deployment, simplified debugging, and flexible coding as each microservice can be written in the programming language that best suits its needs. These benefits led several large cloud providers such as Amazon, Google, Twitter, Netflix and eBay to adopt this new application model.

Normally, each microservice runs on a separate container [1], [2]. In addition, to make the microservices model more profitable, cloud providers often co-locate them on a single (virtual) host, thus achieving high server utilization. One can easily expect a single machine to host dozens to hundreds of frequently launched and terminated containers as part of a microservice-based application [3], [4]. In this context, providing high-quality network connectivity to multiple containers running on the same host becomes crucial for the overall cloud service performance. Particularly, the high-density of containers on a single server requires that the network incurs minimal performance degradation to individual connections.

Previous research efforts have explored the performance of container network deployments while considering different networking drivers[1] and allocation scenarios (e.g., containers running on the same VM, same host, or different hosts). For example, Suo et al. [5] study the performance of various virtual network solutions to implement overlay networks (e.g., Docker overlay, Weave, Flannel, Calico) connecting containers spread among multiple VMs hosted on the same server. Later, Mentz et al. [6] extended their analysis to also encompass a real-world, data center oriented, traffic scenario. Unfortunately, both assume a small number of containers at each node (often only a single communicating pair) and overlook the challenges associated to single-node large-scale container deployments (i.e., when a large amount of containers is running on the same node).

Contribution. In this work, we conduct an extensive evaluation study of the network bottlenecks caused by the increasing number of concurrent containers running on a single virtualized host. Through a series of throughput and latency stress tests using traditional benchmarking tools (e.g., iperf, netcat, sockperf), we assess the performance of the virtualized network on a Docker environment while running three different drivers, namely bridge, macvlan and OVS. Our main findings show that:

- the network performance is mostly restricted by the CPU capacity under large numbers of containers (even for I/O intensive workloads). Moreover, OVS performed the best among the evaluated drivers, mainly due to its fast-path mechanism.
- containers can largely suffer from interference originated from packet processing overhead. More specifically, throughput among communicating containers can vary up to 3x and the flow completion time more than 2x as we increase the number of instances running on a VM.
- proper CPU core scheduling policies can significantly improve connection throughput. In particular, pinning communicating containers to different cores can hike throughput more than 20% for an OVS-based interface.

Organization. Section II provides necessary background and the motivation for this work. Section III-A describes the evaluation setup. Section III-B presents the evaluation

[1]We use the terms *driver* and *interface* interchangeably throughout the paper.

results and key takeaways, followed by a literature review in Section IV. Section V discusses potential extensions to our work. Finally, we conclude the paper in Section VI.

II. Background

This section presents the necessary background to understand the proposed work. Usually, we need an abstraction of the physical resources (e.g., NIC) to deploy VMs or containers. There are different network drivers, which we briefly introduce below.

Bridge. Linux Bridges [7] can function like a switch connecting VMs or containers and allowing communication between the virtual and physical (host) networks. A Bridge consists of four components: a set of physical or virtual network interfaces; a control plane to prevent loops and crashes in the network; a forwarding plane for MAC-based packet forwarding; and a MAC learning database [8].

Macvlan. Macvlan [9] allows a NIC (the parent interface) to have multiple MAC and IP addresses. Thus, a VM or container can have a Macvlan interface to directly interact with the host network using its MAC and IP addresses. In case of multiple containers, Macvlan operates in the bridge mode, i.e., containers communicate over that bridge. Unlike Linux Bridges, Macvlan does not need the MAC learning database or loop prevention algorithm as the MAC addresses are known, which makes Macvlan a lightweight driver. Nonetheless, a Macvlan bridge depends on the parent interface and can suffer from the single point of failure.

OVS. Open vSwitch or OVS [10] is another network driver and consists of a multilayer software that acts as a switch to connect virtualized hosts. OVS is commonly deployed in an emulator like Mininet [11] to connect hosts and mimic a Software-Defined Networking (SDN) platform. The SDN controller communicates to OVS using the OpenFlow protocol. OVS also supports other commonly used technologies like NetFlow, sFlow, and VLAN.

III. Evaluation

This section presents the evaluation setup and discussion on results following key takeaway lessons[2].

A. Setup

Testbed. We conducted our experiments on a Dell PowerEdge R720 server equipped with a 16-core Intel Xeon E5-2650 2.0GHz processor, 64GB DDR3 RAM, and a 300GB 15K RPM SAS hard disk. The server hosts a virtual machine running Ubuntu Server 18.04.3 LTS with 8 CPU cores and 8GB of RAM. VMware ESXi 7.0 is used as hypervisor and Docker 20.10.6 is used for container deployment inside the virtual machine. Fig. 1 summarizes our testbed. We are mostly interested in investigating the performance of the virtual network under high container loads (i.e., large numbers of containers). For that, we deploy containers in pairs, with client and server sides communicating through the virtual network layer (bridge, macvlan or OVS).

[2]Source code is available at *https://github.com/conradoboeira/Containers-Network-Contention*.

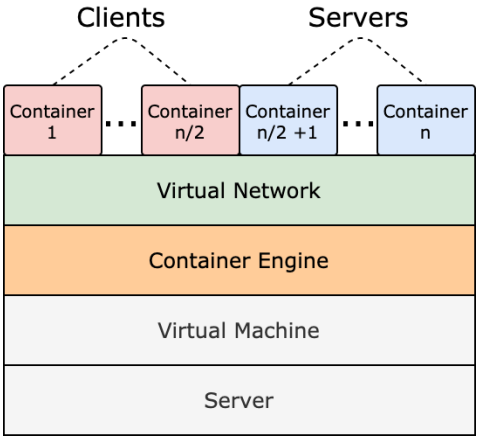

Fig. 1: Testbed used in the experiments.

Workload. We used three well-known benchmarking tools in our experiments. For throughput stress tests, we adopted *iperf* 2.0.10 [12] under TCP mode. In this case, all client containers concurrently establish a connection to their matching servers and send packets at the maximum possible rate for at least two minutes. *sockperf* (version 3.7) [13] is used for the same test in scenarios with varying packet sizes as iperf does not provide that flexibility. To analyze the effect of large numbers of communicating containers on the network latency, we used Netcat (version 1.187) [14] to send a fixed amount of data (5 MB) from client to server containers. We also considered an imbalanced traffic scenario (*Elephant/Mice*) for this test, common on current data centers [15], [16], where 20% of the clients send a much larger flow (50 MB) while the remaining ones are kept short-lived. Both latency-oriented workloads use TCP flows.

Measurement collection. We report our results in terms of throughput and flow completion time (FCT). Unless stated otherwise, the results are an average of 10 repetitions. All measurements are collected using basic statistics provided by our benchmarking tools.

B. Results

Overall performance. We first compare the overall performance of different container networking drivers in our single-VM multi-container setup. Fig. 2 shows the total throughput (i.e., the sum of the throughput from all individual flows) as we vary the number of containers depicted in pairs. We can observe that the throughput increases up to 5 communicating pairs when it faces a slight drawback and then stabilizes at an upper bound, which varies depending on the networking driver, e.g., around 56 Gbps and 70 Gbps for bridge and OVS, respectively. The main reason is that the performance is restricted by the CPU capacity (despite the I/O intensive workload), as all containers run on the same VM, and therefore none of their flows reach the hardware NIC.

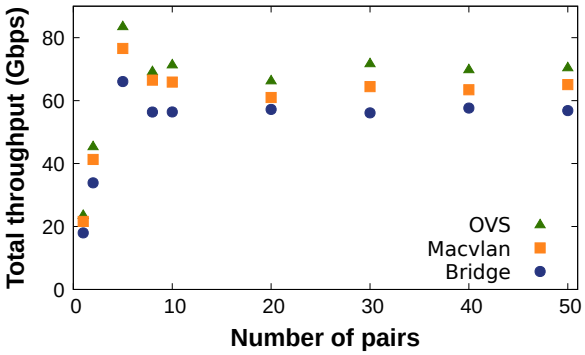

Fig. 2: Total Throughput for OVS, Macvlan and Bridge from 1 to 50 pairs.

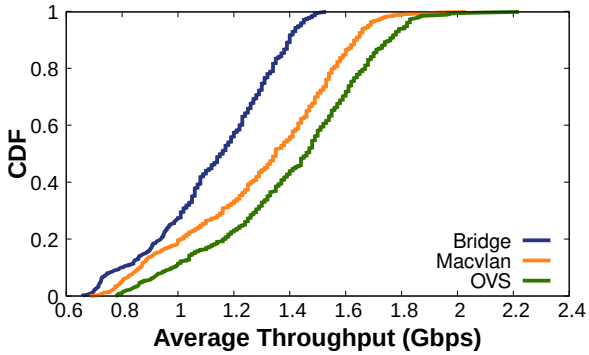

Fig. 3: Cumulative distribution function for the individual throughput of containers in a scenario with 50 pairs.

The difference in performance among the networking drivers is in line with the literature (e.g., [6], [17]) and stems from the fact that simpler drivers tend to have better performance. For example, macvlan does not have the overhead for topology discovery and frame copying between paired devices when compared to a bridge [17]. Interestingly, OVS performed the best among the three evaluated drivers. That is mainly due to its fast-path, which can cache forwarding rules and process packets completely inside kernel space and thus takes advantage of the performance assurances from the latter [18].

Network interference. Next, we look at how containers interfere with each other at the network level as we increase the number of containers in a node. Ultimately, this interference can cause unpredictable slowdowns to applications and result in missed deadlines or SLA violations. Fig. 3 presents a CDF of the average throughput (in Gbps) for each client container. We considered 100 containers (50 clients and 50 servers) in this experiment. As it can be seen, the throughput varies up to 3x among clients for all drivers, which unveils a significantly unfair resource sharing (particularly the CPU) among containers. For instance, the lowest and highest observed throughput for a bridge were approximately 0.6 and 1.6 Gbps, respectively.

To better understand the source of this variability, we analyze the number of re-transmitted packets for each client container in a bridge-based scenario. Fig. 4 shows the results, which we plot as a function of the client throughput. Interestingly, clients presenting larger numbers of re-transmissions are also able to achieve higher throughput. This is because while all clients are facing consistent packet loss (expected as part of the TCP congestion control algorithm [19]), not all of them are having an equal opportunity to re-transmit, in this case, due to the scarcity of CPU resources.

In addition to the throughput, we also compare the flow completion time (FCT) for each communicating pair of containers. Fig. 5 shows the CDF of the FCT when 20 and 100 containers (10 and 50 pairs, respectively) are running on the VM. As expected, OVS (which achieves the highest throughput in our tests) presents the shortest FCT in general. Interestingly, the FCT grows up significantly as we increase the number of containers. For example, the 99th-percentile raises about 31.3%

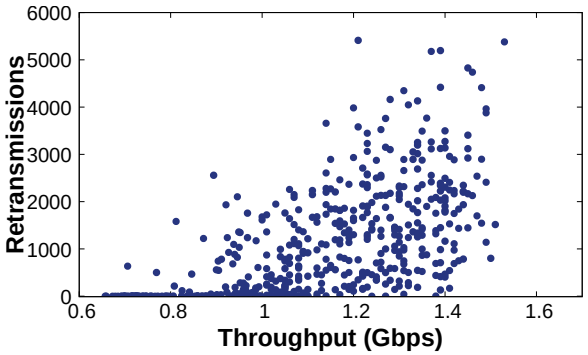

Fig. 4: Number of retransmissions for the throughput achieved in a test with 50 pairs using Bridge as the network driver and with test time duration of 2 minute.

(from 154.3 ms to 202.6 ms), 39.8% (from 115.9 ms to 162.0 ms), and 32.0% (from 110.7 ms to 146.1 ms) for a bridge, Macvlan and OVS-based interface, respectively. Moreover, we observe large tail values in the graph as some containers take a longer time to access resources and use more time to send the same amount of data compared to others.

Furthermore, the *FCT Scenario* shows that the time containers can take to send a specific amount of data can significantly vary even between instances running concurrently on the same host. Specific services can take up to 3 seconds to complete a flow, while others can finish within a second. Although the available bandwidth is supposed to be fairly shared among all instances, the contention for the network and subsequently the CPU can lead to significant divergence in completion time, which can be the source of problems when trying to guarantee a specific level of performance for services running on top of containers.

The following scenario was the *Elephant/Mice Scenario*. We considered the three network drivers, and the results for 50 and 100 pairs can be seen in Fig. 6. The difference between the network virtualization solutions is narrower. When looking at the first 80% of the flows, which consists of the mice flows, the FCT achieved with all drivers are similar. Specifically,

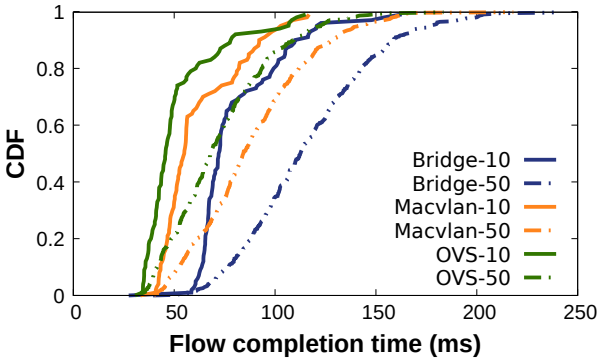

Fig. 5: CDF of the flow completion time for different network drivers and numbers of pairs of containers.

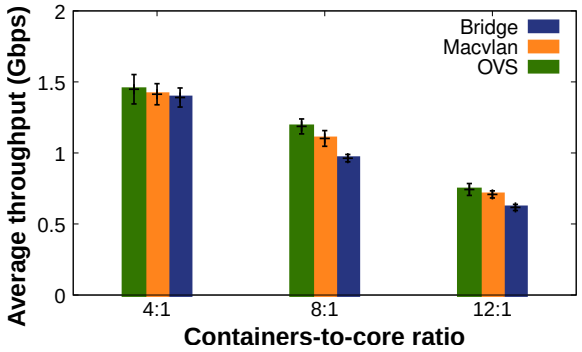

Fig. 7: Throughput comparison between different container-to-core ratios.

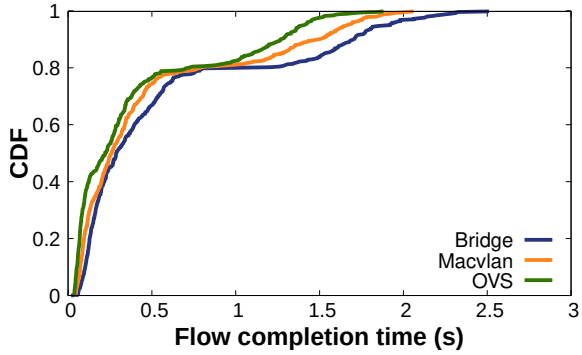

Fig. 6: CDF results for the flow completion time in the Elephant/Mice scenario for 50 pairs.

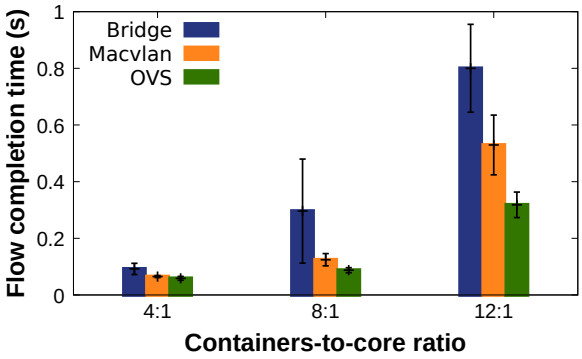

Fig. 8: Flow completion time comparison between different container-to-core ratios.

there is still a slightly better performance achieved by OVS, followed by Macvlan and Bridge. However, the difference is more significant for the elephant flows. The highest FCT values of Bridge are consistently higher than OVS and Macvlan.

Further observation from the graphs is how the elephant flows affect the completion time of the mice flows (see Fig. 5 and Fig. 6). The mice flows can take more than 0.5 seconds to complete, twice longer compared to *FCT Scenario*. Once again, the results confirm that Open vSwitch incurs less CPU usage than the other two, which allows it to perform better in these contention scenarios. CPU contention is even more of an issue as Elephant flows can use much of the processor capacity and impact other flows.

Another takeaway from this test is how containers that do not need to send larger amounts of data can be affected by other replicas. This can become an issue when we consider that such containers can have a critical function inside a microservices architecture. For example, containers sending small amounts of control data can be drowned out by replicas from a different service running in the same host.

CPU contention. To assess the impact of scarce CPU resources on single-node multi-container networks, we analyze their performance with varying container-to-core ratios. Fig. 7 shows the average throughput for three ratios. The throughput degrades with the higher ratios, which tend to become the norm as the number of containers in an application increases [3], [4]. This performance trend is due to two reasons: first, many containers share a core with a high ratio, which naturally causes them to get a smaller slice of the CPU. Second, the number of interrupts and context switches on the CPU is high under the high ratio that incurs a high management overhead [2], [20].

Fig. 8 shows the average flow completion time as we vary the container-to-core ratio, where the FCT increases with the increasing ratios. In particular, it increased more than twice every time we step up the ratio for a bridge interface. Interestingly, the performance gap among interfaces also increases with the container-to-core ratio. The main reason is that the CPU takes longer to handle each interrupt for processing a packet (or group of packets) when the interface is more complex, as in the case of a bridge [21].

To better understand the cost of interrupts and context switching in a single-VM multi-container network, we study how the network performance varies for different CPU pinning policies. This test considers 96 containers (48 pairs) running on an 8-core VM. We test two policies, which uniformly pin each container to a core while ensuring the communicating containers are pinned to i) the same core or ii) different cores.

Fig. 9 shows the total throughput for the evaluated policies

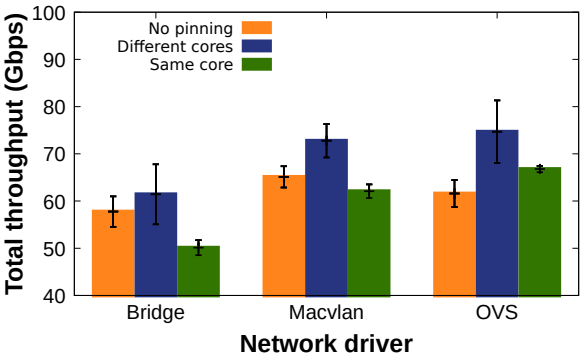

Fig. 9: Throughput comparison for 48 containers with and without CPU pinning.

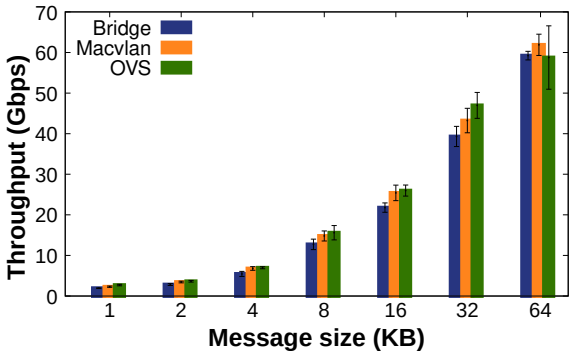

Fig. 10: Throughput results for different message sizes with 50 container pairs.

as we vary the networking driver. We can observe that pinning communicating containers can substantially increase the total throughput compared to a baseline scenario with no pinning depending on the network driver. For example, pinning communicating containers to different cores has increased OVS throughput around 21.2% compared to the scenario without pinning. We suspect that such improvement is due to the reduced overhead originating from cache misses that follows the occurrence of context switches when a container resumes on a different core [20]. Pinning communicating containers to the same core, on the other hand, can drop the total throughput depending on the driver (e.g., up to 13.2% for a bridge interface). The primary reason is that both containers are depending on the same processing unit and thus cannot run in parallel, i.e., every interrupt from one container is directly affecting the other.

Packet size. Fig. 10 shows the total throughput under different packet sizes for 100 communicating containers (50 pairs). Note that the maximum transmission unit (MTU) is given by the VM loopback interface and thus containers can send packets of up to 65KB.

We analyzed the influence of different packet sizes on the system throughput. In Fig. 10 we present the results for 7 different message sizes with the three studied network drivers. One can see the progressive increase in throughput as the packets become larger.

This behavior is expected; as the packet size increases, we need to generate a smaller number of them to achieve a specific throughput. The results show that this behavior in networks can also be seen in intra-host communications.

C. Key takeaways

In the following, we summarize the key takeaways from our investigation based on the above results.

Drivers' Performance. We compared the performance of three network drivers, Bridge, Macvlan, and Open vSwitch, in various scenarios. The results confirm that the OVS ensures the best performance compared to the other two. Specifically, it offers higher throughput and lower flow completion time in all scenarios, mainly due to its kernel-based enhancements. The worst performing driver in all tests was the Bridge, which is commonly used in container solutions.

The Importance of CPU Resources. CPU resources have a significant impact on the network performance between containers on the same VM. In particular, in the case of different CPU to container ratios, having more cores per replica can greatly benefit the performance of the network. Moreover, the possibility of binding container to specific cores can also benefit a system that depends on communication between containers. This technique allows a container to better utilize the faster memories closer to the core and access its context faster when returning from handling an interruption.

Resource Sharing Fairness. We also observed a lack of fairness in resource sharing for containers running on top of a single Virtual Machine. Our results show that some containers might achieve higher throughput (almost twice) compared to others. Containers that are able to use the available resources first can achieve a better network performance than others, leading to unbalanced resource utilization.

IV. Related Work

With the increasing popularity of containers, many researches have started investigating the performance of container networking solutions. Qi et al. [22] analyze the overheads and bottlenecks of different Container Network Interface (CNI) plugins in Kubernetes. Unlike their work, we focus on evaluating performance at a container (rather than pod) granularity. Anderson et al. [23], on their turn, focus on evaluating the network performance of service function chains (SFCs) deployed over containers. Similar to our work, the authors measure the cost of transfers between containers on the same host, though we assume a nested virtualization scenario (i.e., all containers are running inside a virtual machine rather than directly on the host). Deploying containers on top of VMs has become a standard practice specially in public clouds [24].

Zhao et al. [17] study the performance of multiple networking drivers (Bridge, Macvlan, OVS) in distinct container allocation scenarios (e.g., on the same VM, same host or different hosts). They use a single pair of containers in their experiments and vary the number of flows. Interestingly, the authors found

that containers placed on the same VM can achieve faster communication compared to those running directly on the same host when source and destination of intra-VM flows are bind to the same NUMA node. Suo et al. [5] extend this analysis to more networking drivers, including overlay-based ones (e.g., Docker overlay [25], Weave [26], Flannel [27], Calico [28]) which encapsulate packets when sending them to a different host. Mentz et al. [6] evaluate the performance of container networking solutions under a real-world traffic scenario. The authors mimic a data center traffic load comprised of many low volume flows and occasional transfers of large volume ones.

Taking into consideration the papers previously mentioned, it is possible to see that our work is the first to explore large deployments of containers. Works such as [6], [17] use only 2 pairs of containers. Even in [5], where they analyzed the contention for network resources, this number of pairs rose only to 8. The paper that has the largest deployment scenario is [22], as they experiment with 50 containers in a single host. However, this work focus on scenarios where an orchestrator (Kubernetes) is used, meaning containers are usually organized in pods. This differs their work from ours as they do not experiment with different network drivers at the container level.

V. Discussion

In this section, we discuss possible ways to extend the current evaluation. Although we cover an extensive set of experiments (e.g., varying number of containers, different flow sizes, multiple network interfaces), there are still interesting scenarios to explore. For example, we did not consider a scenario where a single container can send multiple flows. As each of these flows can run over a single CPU thread, one would expect the network behaviour to be similar to the behaviour we observed for large numbers of containers, i.e., when the number of containers is much greater than the number of CPU cores. Nonetheless, the manner in which the different flows can be balanced and scheduled among CPU cores can lead to different performance trends. We plan to explore that in the future.

Also, we have focused on exploring single-VM scenarios in this work as they allow us to understand better the contention of the network drivers and CPU resources. However, some applications may require deploying containers over different machines in cloud environments, e.g., due to scarce resources or fault tolerance. When splitting an application among VMs (or hosts), the competition for CPU resources, which we found in this work to be a critical aspect of network performance, can be diminished and better handled. Nonetheless, this separation forces the operator to use network solutions such as Docker Overlay [25], where extra overhead is created due to the need for encapsulating packets leaving from a container in one machine to another. Thus, we are interested in exploring other container allocation scenarios like those using different VMs and hosts.

Talking about possibilities to resolve the contention for CPU resources and the unfairness issues we observed in our experiments, one alternative would be relying on CPU resource reservation. Docker has a native option to set the amount of CPU cores reserved and the CPU shares, which allows a system's scheduler to prioritize some containers over others [29]. During our tests, we found out that the option to limit the percentage of CPU cores used by a container was ineffective. We leave a detailed analysis as future work.

In our current evaluation, client-server communication occurs over TCP connections. Even though we expect there will be no difference in the observed trends if we perform the same set of experiments over UDP (we validated this hypothesis empirically for some scenarios), the latter may lead to lower throughput due to the lack of kernel optimizations, such as GRO [30], that allow TCP to achieve better performance [17]. We plan to analyze the performance of different protocols, e.g., QUIC for container communication in the future.

Finally, this work raised a question over the idea of fair resource sharing in container environments. As shown in our experiments, a virtual machine cannot fairly share its resources among multiple containers, and some of them achieve much higher performance than others with the same configuration. This can become a major issue when we have critical containers that need to offer strict performance guarantees. One alternative is applying flow prioritization mechanisms (e.g., priority scheduling over CPU cores) to alleviate the problem, but the overhead these mechanisms incur may be unacceptably large. Priority-based deployments have not been extensively studied and can be an interesting avenue to explore.

VI. Conclusion

In this work, we presented an exploration of network contention considering several containers communication among themselves in a single host. We detailed the differences between three available network drivers: Open vSwitch, Macvlan, and Linux Bridge. Our experiments have shown that OVS presents better throughput and flow completion time than Macvlan and Bridge; the latter exhibits a considerably lower performance than the other two. We also have another novel observation under various CPU to container ratios. Specifically, binding container to specific cores ensures a better performance, especially under high ratios. Finally, containers running on a single VM can suffer from resource utilization fairness. Overall, our in-depth evaluations at scale can help service providers determine how to efficiently utilize their resources while deploying microservices.

Acknowledgements. We thank the anonymous reviewers for their feedback. This work is partially supported by the Natural Sciences and Engineering Research Council (NSERC) Discovery Grant.

References

[1] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 152–166.

[2] J. Lei, M. Munikar, K. Suo, H. Lu, and J. Rao, "Parallelizing packet processing in container overlay networks." in *EuroSys*, 2021, pp. 261–276.

[3] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, 2019, pp. 19–33.

[4] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: practical and scalable ml-driven performance debugging in microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 135–151.

[5] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of container networks," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 189–197.

[6] L. L. Mentz, W. J. Loch, and G. P. Koslovski, "Comparative experimental analysis of docker container networking drivers," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, 2020, pp. 1–7.

[7] "Linux bridges," 2021. [Online]. Available: https://wiki.linuxfoundation.org/networking/bridge

[8] N. Varis, "Anatomy of a linux bridge," in *Proceedings of Seminar on Network Protocols in Operating Systems*, vol. 58, 2012.

[9] "Macvlan," 2021. [Online]. Available: https://docs.docker.com/network/macvlan/

[10] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 117–130.

[11] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010, pp. 1–6.

[12] "iperf," 2021. [Online]. Available: https://iperf.fr/

[13] "Mellanox/sockperf: Network benchmarking utility," 2021. [Online]. Available: https://github.com/Mellanox/sockperf

[14] "Netcat," 2021. [Online]. Available: https://www.commandlinux.com/man-page/man1/nc.1.html

[15] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 202208. [Online]. Available: https://doi.org/10.1145/1644893.1644918

[16] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010, pp. 267–280.

[17] Y. Zhao, N. Xia, C. Tian, B. Li, Y. Tang, Y. Wang, G. Zhang, R. Li, and A. X. Liu, "Performance of container networking technologies," in *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, 2017, pp. 1–6.

[18] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 117–130.

[19] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," Internet Requests for Comments, RFC Editor, RFC 5681, September 2009. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5681.txt

[20] D. Ghatrehsamani, C. Denninnart, J. Bacik, and M. Amini Salehi, "The art of cpu-pinning: Evaluating and improving the performance of virtualization and containerization platforms," in *49th International Conference on Parallel Processing-ICPP*, 2020, pp. 1–11.

[21] K. Suo, Y. Shi, A. Lee, and S. Baidya, "Characterizing networking performance and interrupt overhead of container overlay networks," in *Proceedings of the 2021 ACM Southeast Conference*, 2021, pp. 93–99.

[22] S. Qi, S. G. Kulkarni, and K. Ramakrishnan, "Assessing container network interface plugins: Functionality, performance, and scalability," *IEEE Transactions on Network and Service Management*, 2020.

[24] M. Bacou, G. Todeschi, D. Hagimont, and A. Tchana, "Nested virtualization without the nest," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[25] "Docker overlay network - use overlay networks," 2021. [Online]. Available: https://docs.docker.com/network/macvlan/

[26] "Weave," 2021. [Online]. Available: https://github.com/weaveworks/weave

[27] "Flannel," 2021. [Online]. Available: https://github.com/coreos/flannel/

[28] "Calico," 2021. [Online]. Available: https://github.com/projectcalico/calico-containers

[29] "Docker runtime options with memory, cpus, and gpus," 2021. [Online]. Available: https://docs.docker.com/config/containers/resource_constraints/#configure-the-realtime-scheduler

[30] H. Xu, "Generic receive offload," in *Japan Linux Symposium*, 2009.