An Investigation of the Linux Virtual Networking Latencies

Luca Abeni[1] and Harald Gustafsson[2] Fredrik Svensson[2] Tommaso Cucinotta[1]

[1] Scuola Superiore Sant'Anna, Pisa, Italy
{luca.abeni,tommaso.cucinotta}@santannapisa.it
[2] Ericsson {harald.gustafsson,fredrik.a.svensson}@ericsson.com

Abstract. Applications running in virtual environments often experience high communication latencies that might be problematic if such applications are characterized by strict temporal constraints. In this work, such high latencies are investigated with a focus on Kubernetes containers and on the software mechanisms used by many CNI plugins to interconnect them: network namespaces, virtual ethernet pairs, and software bridges. Our experiments show that virtual ethernet pairs and network namespaces introduce a negligible overhead, while most of the overhead is introduced by Netfilter rules and by traversing the network stack multiple times. Moreover, the worst-case latencies experienced by UDP packets exchanged between different containers are much larger than the average latencies because of the Linux "softirq" mechanism.

Keywords: Real-Time Computing · Virtual Networking · Operating Systems · Cloud Computing.

1 Introduction

In recent years, computing infrastructures have experienced an important paradigm shift, moving more and more applications to virtualized environments and clouds based on containers or VMs and virtualized networks. This tendency has important consequences for both compute and network virtualization, stressing some performance requirements in terms of compute and networking bandwidth, predictable execution and message passing, and low latency. As a result, cloud technologies are evolving to support these applications.

Focusing on applications characterized by strict temporal constraints, it could be interesting to take advantage of the cloud features to provide high reliability or fault tolerance. Guaranteeing that all applications' requirements, including temporal requirements, are respected under various conditions and failure scenarios. Unfortunately, current cloud infrastructures do not fully support this vision [10], and some work is needed from both the theoretical and practical points of view to implement the needed features. The theoretical aspects of this issue can be addressed by combining real-time theory with cloud computing in a novel way; for example, applications characterized by strict timing and reliability requirements can be modelled as Directed Acyclic Graphs (DAGs) composed

of *tasks* implemented by replicated microservices [1, 3]. This approach provides a powerful formal model to analyze the real-time and fault-tolerance properties of a virtualized system/application, and previous research works show that the analytical results obtained through this model match with simulative results.

However, when evaluating a practical implementation of this model on Kubernetes [12], it was noticed that a worst-case communication delay of about 300 μs between microservices needed to be accounted for to have a reasonable match between theoretical analysis and experimental results. This issue is due to that previous work focused on the computational aspects of real-time cloud systems without providing an appropriate model for the latencies introduced by inter-container (inter-service) communication. This paper starts investigating the worst-case communication delays between Kubernetes containers, to provide better support for a holistic model of a real-time fault-tolerant cloud that can be effectively implemented. After reviewing the related work (which mainly focuses on average performance) in Section 2, Section 3 recalls the basics of network virtualization in Kubernetes clusters, Section 4 presents the experimental setup used in this paper, and Section 5 presents an investigation of networking latencies based on it. Finally, Section 6 states our conclusions.

2 Related Work

Some previous research investigated the performance of some network virtualization mechanisms, such as the Kubernetes Container Network Interface (CNI) [17, 18]. However, most of these works focused on TCP throughput, UDP sustainable packet rate or average latencies, instead of worst-case latency. Some work has also been performed in connecting containers using eBPF to avoid the overhead introduced by the Linux networking stack [6]. That work's goal was again to improve the average throughput as measured by `iperf` or similar tools. Similarly, other works investigated the usage of eBPF through XDP [20].

The idea of bypassing the Linux networking stack has been considered in other contexts, too; some works compare the performance of different kernel bypassing solutions [4], while others [14, 13, 8] optimize the performance of user-space routers [5]. Some previous works identified the overhead in inter-container networking caused by traversing the network stack multiple times (as we also show in Section 5) and proposed some alternative switch architecture (based on intercepting the system calls used to send and receive packets) to improve the network throughput [21]. The same idea of intercepting system calls to avoid traversing the network stack multiple times has also been used in Socksdirect [16].

Only a few works considered the worst-case networking latency or high percentiles of such latency [11]. A notable exception to this attitude is KuberneTSN [9], which focuses on optimizing the worst-case network latency, but is based on TSN networks [7] and uses Polling Mode Drivers (PMD) that consume at least a dedicated CPU core to busy-wait for network packets. The usage of TSN technologies in virtualized environments has also been considered in other works [15].

In any case, most of the previous works are concerned with networking performance in the presence of packet flows at very high rates. In this paper, instead, the goal is to evaluate the networking latency experienced by packets sent at lower rates (for example, periodic flows of packets with period equal to $1ms$ or more). Moreover, the goal here is to avoid bypassing the networking stack.

3 Background

Motivated by the implementation of a real-time fault-tolerant cloud [1, 3, 12], this paper focuses on evaluating and analyzing the latencies introduced when exchanging packets between containerized applications. Containers can be implemented by using two mechanisms provided by the Linux kernel: *control groups* (controlling resource accounting and scheduling) and *namespaces* (controlling resource visibility). In particular, namespaces are associated with kernel subsystems and resources, and isolate a virtualized copy of a subsystem or resource for the processes executing inside a namespace; hence, such processes are given the illusion of having their own dedicated copy of the subsystem or resource. For example, processes executing in the network namespace see their own virtualized instance of the kernel networking stack, isolated from the network seen by processes running in different network namespaces. In other words, each network namespace has its own network interfaces, routing table, netfilter rules, etc...

As a result, thanks to the network namespace, each container cannot directly communicate with other containers and is isolated from them by not sharing any network resource between them. Kubernetes introduces the "Pod" abstraction and associates kernel namespaces to Pods (multiple containers can be in the same Pod). Also notice that the Linux kernel namespaces should not be confused with the Kubernetes namespaces, which are a mechanism to group Pods and other Kubernetes objects controlling their visibility.

A container can then communicate with the host system (or with other containers) through a *virtual ethernet pair* (veth), which is a pair of network devices (endpoints) connected point-to-point through a tunnel. If one of the two endpoints is inserted in a network namespace, then the processes running inside the namespace can send network packets that will arrive at the other endpoint. The second endpoint can be: 1) inserted in a different network namespace - creating a tunnel between the two namespaces, 2) connected to a software bridge - allowing to forward level-2 packets to other devices connected to the bridge, or 3) left in the host's default namespace so that packets arriving to it are forwarded at layer 3 using the host's routing tables and netfilter tables.

When a container is created, a container manager such as Kubernetes uses a low-level container runtime such as `crun`, `runc` or similar to setup the cgroups and namespaces for the containers (Kubernetes uses per-Pod network namespaces, so all the containers in a Pod share their network namespace). Then, the container can be given network connectivity by creating the appropriate virtual ethernet pairs, software bridges/switches, etc... In particular, Kubernetes uses a so-called "CNI plugin" (Container Network Interface plugin) for this. A CNI plu-

gin is an executable that can perform 5 different operations: `ADD`, `DEL`, `CHECK`, `GC`, and `VERSION`. The `ADD` operation is used to connect a container to the network by creating or configuring a virtual ethernet interface inside a network namespace. The CNI plugin is hence responsible for handling the virtual ethernet pair, attaching one of the two endpoints to a software bridge or switch, or defining IP routes for it and assigning an appropriate IP address to the endpoint that is inserted in the container's network namespace. IP address assignment can be done by an IPAM - IP Address Management - plugin.

This architecture allows Kubernetes to use different runtimes and mechanisms to implement containers and virtual networking. In particular, different CNI plugins can use various mechanisms to connect containers running on the same worker node or on different worker nodes:

- Containers running on the same worker node can be connected through a software bridge (or virtual switch) or by using the IP routing/forwarding mechanism provided by the host kernel
- Containers running on different worker nodes can be connected by encapsulating their traffic inside IP packets sent by the host (this allows containers having IP addresses in a network different from the host's network), or by directly routing the packets (then, the host needs to have routing tables for the containers' IP addresses)

4 Experimental Setup and Preliminary Experiments

The goal of this work is to analyze the delays, with a focus on worst-case latencies, introduced by the most important network virtualization mechanisms used by CNI plugins. The mechanisms used, virtual ethernet pair, network namespace, software bridge, etc are investigated how they impact the real-time performance of Kubernetes CNIs. To this end, some experiments have been performed using a "ping-pong" test application composed of 2 processes: a "ping" process that periodically sends UDP packets, and a "pong" process that receives such UDP packets and immediately sends back UDP replies to the "ping" process. Hence, the "ping" process can compute the Round-Trip Time (RTT) by measuring the time between sending a UDP packet and receiving back the reply. The process sends a large number of packets (100000 packets for the experiments reported on), computing the worst-case RTT, the average value, and a Cumulative Distribution Function (CDF) of the experienced RTTs.

The experiments reported in this paper were performed on a testbed composed of 3 servers based on an Intel(R) Xeon(R) E5-2650 CPU running at 2.60 GHz, configured for predictable performance [2]. The servers are based on Ubuntu 22.04.1 LTS with a 5.15 Linux kernel and run Kubernetes v1.24.3; one of them is used as a master node, while the other two are used as worker nodes. The average RTT between the two worker nodes (measured by using the "`ping`" Unix command) is 66 μs, and the worst-case is 531 μs. The "Weave Net" CNI plugin[3] is used to connect Kubernetes Pods to the network.

[3] https://github.com/weaveworks/weave

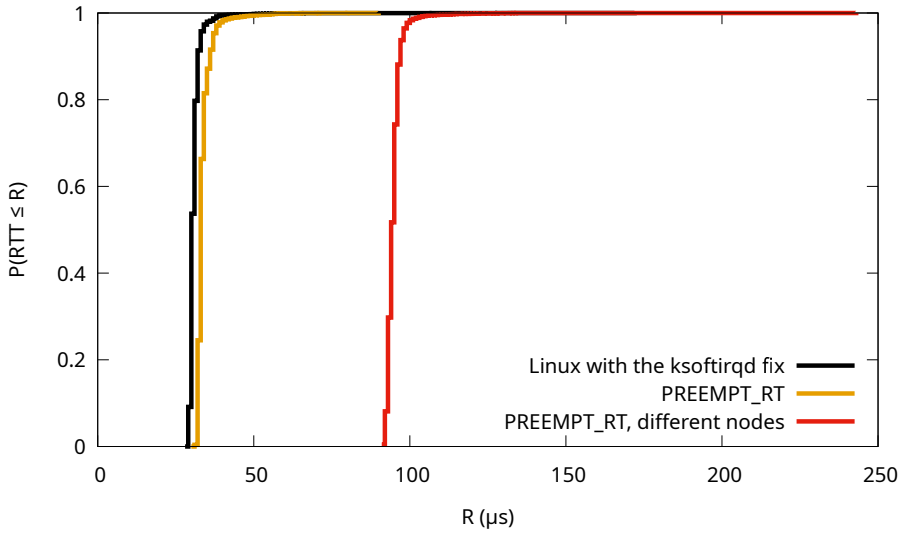

Fig. 1. Experimental CDF of the Round-Trip-Times measured with various kernels and configurations.

All the tests showed that the first "ping" and "pong" packets always experience a much larger RTT than all the other packets. Since this effect has been consistently observed on many different hardware platforms, kernel versions, and software configurations, we investigated it. Some experiments and measurements with `ftrace`[4] revealed that this initial delay is caused by the time needed to fill the IP routing caches in the Linux kernel. Hence, it has been decided to ignore the first RTT when computing the performance statistics.The first experiments performed starting the "ping" and "pong" processes in two different Kubernetes containers showed that the worst-case RTT obtained when the two containers executed on two different worker nodes was 919 μs. To investigate this large latency, the experiment was repeated, forcing the Kubernetes scheduler to place the two containers on the same worker node; in this case, the worst-case RTT was 517 μs, which again is larger than expected. Hence, the reason for this large value of the worst-case RTT needed to be investigated.

Some experiments with `ftrace` revealed that the worst-case RTT was caused by softirq handling in the Linux kernel. When a network packet is received, most of the processing is performed in the so-called "softirq context" (similar to the traditional BSD bottom-half). But, the Linux kernel used for the experiments can sometimes delay the softirq processing due to the usage of a "`ksoftirqd`" kernel thread. Further research revealed that this is a known issue that has been fixed in recent kernels[5]. Hence, we backported the fix to the kernel used for the

[4] https://www.kernel.org/doc/html/latest/trace/ftrace.html
[5] https://lore.kernel.org/lkml/87bkiu34fp.ffs@tglx/T/

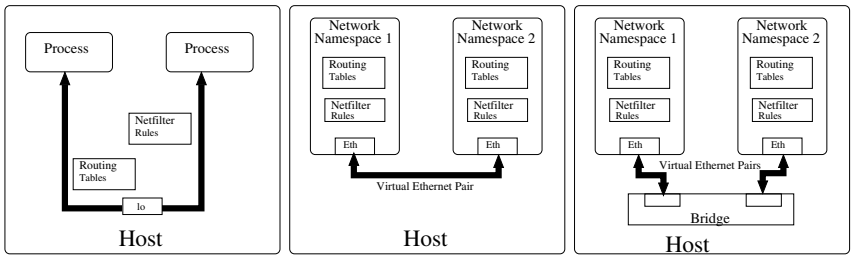

Fig. 2. Two "`ping`" and "`pong`" processes communicating through a network loopback device `lo` (left figure), executing in two different network namespaces and communicating through a virtual ethernet pair (center figure), and executing in two different network namespaces and communicating through two virtual ethernet pairs connected by a software bridge (right figure).

experiments. As a result, the worst-case RTT for containers executing on the same worker node went down to 140 μs. To further reduce the delays due to wakeup latencies, we switched to a real-time kernel using the PREEMPT_RT patchset (version 5.15.107-rt62). This allowed reducing the worst-case RTT to 89 μs. Figure 1 shows the CDFs of the RTTs measured in these experiments. From the figure, it is possible to see that most of the measured RTTs are near the average value, but the plots have some long tails, indicating some latencies much larger than the expected ones. One of the big differences between this investigation and previous works is that we focus on such long tails. For example, from the figure it is possible to see that using PREEMPT_RT allows reducing the length of the tail (the worst-case RTT) at the cost of increasing the average values (this is a well-known behaviour of real-time kernels). Hence, if the focus had been on reducing the average RTT, PREEMPT_RT would have been a bad choice.

To understand the sources of long tails shown in Figure 1 and the reason why the average RTT between two containers executing on different worker nodes is larger than the average RTT measured through the `ping` command, some more experiments have been performed, as described in the next section.

5 Analysis of the Round-Trip Times

To better understand the RTTs for two containers executing on the same worker node, we started by evaluating the performance of the various mechanisms used by Weave Net. Since this CNI plugin uses a Linux software bridge to connect containers placed on the same working node, the considered mechanisms are virtual ethernet pairs, the software bridge, and the network namespace.

As a baseline, we evaluated the RTTs when the two processes sending and receiving UDP packets execute on the same host (without containers) and use the loopback network interface `lo` for communicating, as shown in the left part

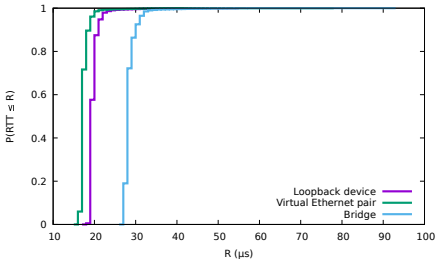

Fig. 3. Experimental CDF of the Round-Trip-Times for UDP packets exchanged locally with various network virtualization mechanisms.

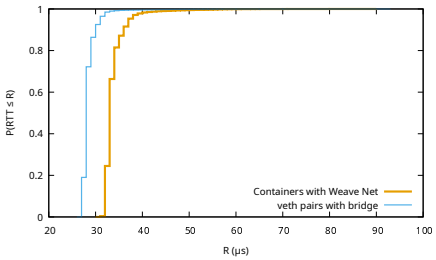

Fig. 4. Experimental CDF of containers connected through Weave Net versus software bridge with virtual ethernet pairs.

of Figure 2. In this case, the IP routing and forwarding mechanism of the Linux kernel is used to move packets from one program to the other and vice-versa.

Then, the two processes have been connected through a virtual ethernet pair, inserting the two endpoints in two different network namespaces (one per process) as shown in the center part of Figure 2. In this case, the Linux IP routing and forwarding is not stressed because the two endpoints of the virtual ethernet pair are directly connected. The overhead introduced by the kernel to cross the namespaces is evaluated instead.

Finally, the two processes have been executed in dedicated network namespaces again, using two different virtual ethernet pairs, with one endpoint connected to the software bridge, and the other one inside the dedicated network namespace, and used by the test process as shown in the right part of Figure 2. In this case, we measured the overhead of the virtual ethernet pair, network namespace, and software bridge. The results of this experiment are reported in Figure 3, which immediately shows some interesting things. First of all, the RTTs experienced when using a virtual ethernet pair with 2 network namespaces are generally smaller than the RTTs experienced when using the `lo` device without namespaces (hence, the network namespaces do not seem to introduce any overhead and seem to improve the network performance instead). While this result first seems to be counterintuitive, some additional investigation showed that it can be perfectly justified: when connecting the two processes through the `lo` device, the host kernel's routing tables are used to forward the packets while using the virtual ethernet pair this overhead is not needed (packets sent from an endpoint are automatically tunneled to the other endpoint). Moreover, the two network namespaces have no netfilter rules to be applied. Hence, using the `lo` device causes much more time to be spent in the `fib_table_lookup()` and `nf_*` kernel functions. The second thing to be noticed is that the configuration using 2 virtual ethernet pairs and a bridge results in RTTs considerably larger than the other two configurations. This can be explained by looking at the path traversed by the packets in the various setups. In the first two configurations, a packet sent from one process to the other traverses the UDP/IP stack two

times - one for sending a packet and one for receiving it. On the other hand, in the bridge-based configuration the packet traverses the UDP/IP stack twice for going from the process to the bridge and twice for going from the bridge to the second process.

Finally, all the RTT CDFs for all configurations are characterized by long tails with similar sizes. We further investigated this by using `ftrace` to understand the source of those large worst-case RTTs. It turned out that the long delays were caused by interference from other, unrelated, softirqs. PREEMPT_RT kernels reduce the interference on real-time tasks caused by softirq processing by allowing softirqs to be preempted by user processes. However, some of the code in the Linux kernel networking stack uses `spin_lock_bh()` to protect critical sections, and this function blocks the invoking task if a softirq is currently running, waiting until the softirq is terminated. This causes a so-called *priority inversion* [19]: if a high-priority task needs to process a network packet while a low-priority softirq is executing, the high-priority task blocks until the low-priority softirq is finished. Since softirqs are per-CPU-core, if the sender process and the receiver process run on two different cores a single packet can suffer 2 interferences from softirqs; hence, an RTT can be increased by 4 interferences from softirqs (2 times for the "ping" packet and 2 times for the "pong" packet).

Since the inspection of `ftrace` data revealed that a softirq might require up to 20 μs to be processed, this is about 80 μs per RTT independent of the network virtualization mechanism. To avoid this issue, it would be necessary to remove the calls to `spin_lock_bh()` from the networking stack or to bypass the networking stack by using eBPF, XDP, or DPDK-like techniques.

Figure 4 compares the performance of the bridged virtual ethernet pairs with the performance of containers connected through Weave Net. The difference between these RTTs has been investigated, and turned out to be due to the Netfilter rules installed by Weave Net.

Finally, the differences between the RTTs for containers running on the same worker node and for containers running on different worker nodes (see the difference between the light blue line and the green line in Figure 1) have been investigated. It turned out that such latencies are due to the transmission of the packets on the network and to VXLAN encapsulation.

6 Conclusions

This paper presented an analysis of the latencies introduced by the most important mechanisms used by Kubernetes CNI plugins such as Weave Net to interconnect Kubernetes containers. The results of this investigation show that while the average latencies are caused by the packet routing and filtering mechanisms used to implement virtual networking, the worst-case latency is caused by some interference by the softirq mechanism on the kernel's networking stack.

In particular, the biggest contributions to the average latencies come from the `fib_table_lookup()` function used to access IP forwarding tables and from applying Netfilter rules. Moreover, the software bridges (or virtual ethernet

switches) used to forward packets between containers force such packets to traverse the network stack multiple times, introducing more overhead. On the other hand, the large worst case for the latency experienced during inter-container communications (i.e., the long tail in the CDFs of such latency) is caused by the usage of `spin_lock_bh()`, which ends up waiting for the termination of currently executing softirqs in the Linux networking stack.

As a future work we plan to investigate the removal of softirq synchronization from the network stack, to reduce the worst-case communication latencies experienced by containerized applications.

References

1. Abeni, L., Andreoli, R., Gustafsson, H., Mini, R., Cucinotta, T.: Fault tolerance in real-time cloud computing. In: Proceedings of the 26th IEEE International Symposium on Real-Time Distributed Computing (ISORC). pp. 170–175 (2023). https://doi.org/10.1109/ISORC58943.2023.00031
2. Abeni, L., Cucinotta, T., Pinczel, B., Mátray, P., Srinivasan, M.K., Lindquist, T.: On the use of linux real-time features for ran packet processing in cloud environments. In: Anzt, H., Bienz, A., Luszczek, P., Baboulin, M. (eds.) High Performance Computing. ISC High Performance 2022 International Workshops. pp. 371–382. Springer International Publishing, Cham (2022)
3. Andreoli, R., Gustafsson, H., Abeni, L., Mini, R., Cucinotta, T.: Design-time analysis of time-critical and fault-tolerance constraints in cloud services. In: Proceedings of the 16th IEEE International Conference on Cloud Computing (CLOUD). pp. 415–417 (2023). https://doi.org/10.1109/CLOUD60044.2023.00056
4. Ara, G., Lai, L., Cucinotta, T., Abeni, L., Vitucci, C.: A framework for comparative evaluation of high-performance virtualized networking mechanisms. In: Ferguson, D., Pahl, C., Helfert, M. (eds.) Cloud Computing and Services Science. pp. 59–83. Springer International Publishing, Cham (2021)
5. Barbette, T., Soldani, C., Mathy, L.: Fast userspace packet processing. In: 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS). pp. 5–16 (2015). https://doi.org/10.1109/ANCS.2015.7110116
6. Behl, D., Huang, H., Kodeswaran, P., Sen, S.: On ebpf extensions to kubernetes cni datapath. In: 2023 15th International Conference on COMmunication Systems and NETworkS (COMSNETS). pp. 207–209 (2023). https://doi.org/10.1109/COMSNETS56262.2023.10041357
7. Farkas, J., Bello, L.L., Gunther, C.: Time-sensitive networking standards. IEEE Communications Standards Magazine **2**(2), 20–21 (2018). https://doi.org/10.1109/MCOMSTD.2018.8412457
8. Farshin, A., Roozbeh, A., Jr., G.Q.M., Kostić, D.: Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20). pp. 673–689. USENIX Association (Jul 2020), https://www.usenix.org/conference/atc20/presentation/farshin
9. Garbugli, A., Rosa, L., Bujari, A., Foschini, L.: Kubernetsn: a deterministic overlay network for time-sensitive containerized environments. In: ICC 2023 - IEEE International Conference on Communications. pp. 1494–1499 (2023). https://doi.org/10.1109/ICC45041.2023.10279214

10. García-Valls, M., Cucinotta, T., Lu, C.: Challenges in real-time virtualization and predictable cloud computing. Journal of Systems Architecture **60**(9), 726–740 (2014). https://doi.org/https://doi.org/10.1016/j.sysarc.2014.07.004, https://www.sciencedirect.com/science/article/pii/S1383762114001015
11. Ghasemirahni, H., Barbette, T., Katsikas, G.P., Farshin, A., Roozbeh, A., Girondi, M., Chiesa, M., Jr., G.Q.M., Kostić, D.: Packet order matters! improving application performance by deliberately delaying packets. In: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22). pp. 807–827. USENIX Association, Renton, WA (Apr 2022), https://www.usenix.org/conference/nsdi22/presentation/ghasemirahni
12. Gustafsson, H., Svensson, F., Mini, R., Abeni, L., Andreoli, R., Cucinotta, T.: Rtilience: Fault-tolerant time-critical kubernetes. In submission (2024)
13. Katsikas, G.P., Barbette, T., Kostić, D., Maguire, J.G.Q., Steinert, R.: Metron: High-performance nfv service chaining even in the presence of blackboxes. ACM Trans. Comput. Syst. **38**(1–2) (jul 2021). https://doi.org/10.1145/3465628, https://doi.org/10.1145/3465628
14. Katsikas, G.P., Barbette, T., Kostić, D., Steinert, R., Jr., G.Q.M.: Metron: NFV service chains at the true speed of the underlying hardware. In: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18). pp. 171–186. USENIX Association, Renton, WA (Apr 2018), https://www.usenix.org/conference/nsdi18/presentation/katsikas
15. Leonardi, L., Bello, L.L., Patti, G.: Towards time-sensitive networking in heterogeneous platforms with virtualization. In: 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). vol. 1, pp. 1155–1158 (2020). https://doi.org/10.1109/ETFA46521.2020.9212116
16. Li, B., Cui, T., Wang, Z., Bai, W., Zhang, L.: Socksdirect: datacenter sockets can be fast and compatible. In: Proceedings of the ACM Special Interest Group on Data Communication. p. 90–103. SIGCOMM '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3341302.3342071, https://doi.org/10.1145/3341302.3342071
17. Qi, S., Kulkarni, S.G., Ramakrishnan, K.K.: Understanding container network interface plugins: Design considerations and performance. In: 2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN). pp. 1–6 (2020). https://doi.org/10.1109/LANMAN49260.2020.9153266
18. Qi, S., Kulkarni, S.G., Ramakrishnan, K.K.: Assessing container network interface plugins: Functionality, performance, and scalability. IEEE Transactions on Network and Service Management **18**(1), 656–671 (2021). https://doi.org/10.1109/TNSM.2020.3047545
19. Sha, L., Rajkumar, R., Lehoczky, J.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Transactions on Computers **39**(9), 1175–1185 (1990). https://doi.org/10.1109/12.57058
20. Vieira, M.A.M., Castanho, M.S., Pacífico, R.D.G., Santos, E.R.S., Júnior, E.P.M.C., Vieira, L.F.M.: Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. ACM Comput. Surv. **53**(1) (feb 2020). https://doi.org/10.1145/3371038, https://doi.org/10.1145/3371038
21. Zhuo, D., Zhang, K., Zhu, Y., Liu, H.H., Rockett, M., Krishnamurthy, A., Anderson, T.: Slim: OS kernel support for a Low-Overhead container overlay network. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). pp. 331–344. USENIX Association, Boston, MA (Feb 2019), https://www.usenix.org/conference/nsdi19/presentation/zhuo