Survey on Performance Models of Container Networks

David de Andrés Hernández
Technical University of Munich
Email: deandres.hernandez@tum.de

***Abstract*—When *cloudifying* any system, the choice of an appropriate container network solution is critical. The wrong container network choice can turn a working system unviable and eclipse the main benefits of cloud architectures: reliability, scalability, and flexibility. Yet the number of solutions targeting the cloud environment is vast and the angles approaching the challenges are varied. Therefore, understanding and evaluating the benefits and limitations of the available solutions and technologies is a tedious but crucial process. To support this process, this paper surveys nine published benchmarks and three performance models searching for the major affecting factors and estimators. The findings are the following: the CPU prevails as bottleneck for performance in most scenarios; although many performance benchmarks exist, not many target scenarios with large-scale container pairs; performance models for containers with focus on the CPU resources provide the most precise estimators.**

I. Introduction

Cloud architectures are motivated by both economies of scale and resource optimization and are enabled by virtualization technologies. But virtualization often incurs in performance degradations and a reduction in the system's portability. To mitigate this effect, containers, a form of lightweight virtualization, emerged. Thanks to their alternative means of partitioning resources, the virtualization overhead is drastically reduced, and the deployment process is expedited. These benefits favoured the standardization of containers and that in turn made portability an additional strength of this technology.

It's popularity rising, a new architecture exploiting container's strengths emerged: microservices. This architecture borrows the encapsulation principle of software development and breaks applications into stand-alone containers. In this approach, each container is only responsible for a single function and by doing so, operators can update or replace components without impact to the remaining services. Moreover, applications can be granularly scaled by load-balancing a service into several instances of the same container. If in addition instances are stateless, containers can be re-spawned in-service and without downtime. Nontheless, the price to pay for microservices is the need of automation and orchestration software to overcome the explosion in the number of components. Overall, microservices and cloud principles are aligned.

In the light of the above, network connectivity among containers is paramount for the correct functioning of microservice-based applications. Yet, there are two important challenges. First, a microservice-populated cloud changes constantly within seconds and containers can be considered ubiquitous. Second, all possible traffic patterns take place: between containers in the same VM, between containers in different VMs, between containers in different hosts, between containers in different data centres, and many more. In this scenario, the host's OS becomes an important component of the networking infrastructure of the cloud. With one remark, it was not conceived for such scale. For this reason, the networking performance of containers must be carefully considered as it can turn a working system unviable when migrated to cloud architectures. The main cause of this degradation is due to overhead processing of packets through the OS's networking stack. Another source of degradation is the processing of headers belonging to overlay networks. Overlay networks, although complex, allow the communication of containers in constant move. But again, the host's OS where not conceived to process headers efficiently at this scale. Willing to overcome the performance challenge, operators have created sophisticated frameworks for high-performance packet IO.

Due to the immense number of choices for container networking, an evaluation of the solutions is of vital importance. To be thorough, this evaluation requires of several steps. The first step is to analyse the solutions to confirm whether the technological requirements of the system to be *cloudified* can be satisfied, or not. Further, consulting benchmarks can give an indication of the bottlenecks and viability. In the last stages, a high-fidelity model can estimate the results that configuration changes can have in the system's behaviour.

II. Background

A. Networking stacks

Most container environments make use of the kernel's network stack because it is feature rich, reliable and easy to use. However in high-performance scenarios, custom-made stacks running in user space can replace the kernel's implementation to provide additional performance at the cost of additional complexity.

1) Kernel's stack: Current OS's include a rich network stack providing a socket-based user-level interface for transmitting and receiving packets; handling a wide variety of protocols; as well as managing the underlying hardware. Figure 1 presents on the left the different layers which packets traverse before being handed over to a user-space application. At the bottom we find the device driver, which is the layer responsible for interacting directly with the HW. This includes: claiming control of a device; requesting memory ranges and IO ports; setting the DMA mask; and registering functions to send, receive and manipulate packets. Next in the stack, we find the Network Device Driver Interface (NDDI), which

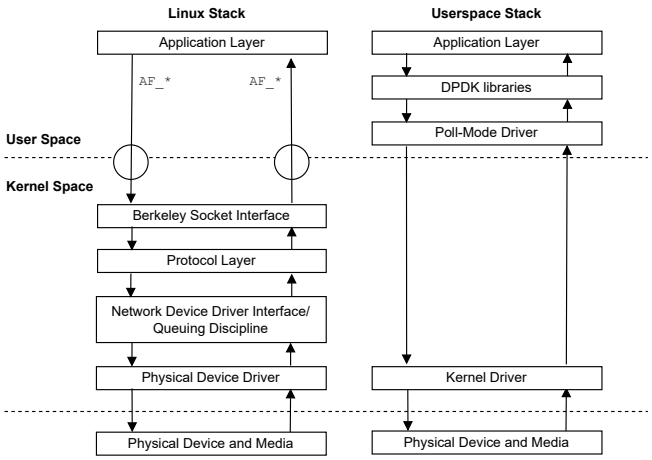

Fig. 1. Kernel's network stack [28] and DPDK stack [self-made].

enables, multiple and perhaps different, network devices to be used simultaneously. Furthermore, the NDDI includes a packet scheduler implementing queuing disciplines. Moving upwards, the protocol layer is where the different protocols are implemented. Each protocol must interact to the north with the socket interface and to the south with the NDDI. To do this, each protocol is associated with a protocol family (PF_*) northbound, and with a protocol type southbound. At the top of the kernel stack we find the Berkeley Socket Interface, which allows user space programs to communicate with the remote devices. Is the last abstraction layer which gives programs the impression of communicating directly. At this layer, every socket type is associated with a protocol. For example, the PF_INET is associated with the TCP/IP protocol. By introducing all these layers, the stack remains very flexible and can accommodate features with reduced effort; nevertheless this flexibility is in part responsible for performance losses.

2) User space stacks: Besides adding complexity by re-implementing the network stack, high-performance IO frameworks are key-enablers for Containerized Network Functions (CNF) [14]. Sometimes, the variety of protocols and functions which the kernel's networking stack offers are not required. This makes it feasible to re-write the required portions of stack using a high-performance IO framework. DPDK [24] and PF_RING ZC [27] can lead to a nine-fold performance increase over the default kernel IO framework [2]. Figure 1 presents on the right the layers of DPDK networking stack for comparison. At the bottom we find the kernel driver, a minimal layer which loads and binds the ports to the poll-mode driver in user space. This is the only component which lies within the kernel space. In the user space we find the poll-mode driver. This is a key component for enabling high performance packet processing. In contrast to the kernel stack, which is interrupt-driven, the DPDK stack polls the NIC continuously. This in turns, consumes all available CPU cycles; in case no packets are available for processing, the CPU cycles are wasted (busy waiting). At the top we find the DPDK libraries, which provide utilities for high-performance packet processing applications. Introducing frameworks such as DPDK in container environments has already been accomplished and is a production ready approach. Examples are: Open vSwitch [25] and Tungsten Fabric [26]. The vRouter from the Tungsten Fabric project, for example, uses DPDK to efficiently route, encapsulate and decapsulate the packets.

B. Container Networks

We have seen the importance of container communication in microservices architectures. To materialize this communication, there are multiple options, each with its own drawbacks and advantages. In the following we differentiate between modes for inter-host communication, and services for intra-host communication. This classifications aims to support the evaluation process. Please note that these modes are not exclusive and usually they co-exist.

1) Modes for intra-host communication:

None Mode In this mode, a container is isolated into its own namespace. A namespace is a logical copy of the OS's network stack with only a loopback interface. Because of this, it cannot communicate with other containers. Achieving extreme isolation, it is suitable for offline computation such as batch processing or backup jobs.

Bridge Mode In this mode, the key component is a virtual bridge created inside a specified namespace. In addition, this mode uses linux's veth device drivers. Veth pairs serve as pipes between namespaces, as depicted in figure 2. Containers can then be launched including a pair of veth interfaces, where one of the pairs will be moved to the namespace of the bridge (and enslaved with it) and the other pair will remain in the container's namespace. Furthermore, the bridge can be enhanced with L3 communication by giving each veth interface an IP address within the bridge's network subnet. This mode allows star-like topologies but does not bring alone connectivity to external networks. For external connectivity other services, such as NAT or overlays must be configured.

Container Mode The container mode can be seen as an extension of the None mode. First, a container is spawned in None mode, thus creating a dedicated networking namespace. Subsequent containers are launched inside this namespace by providing the namespace's id as a launch parameter. What this effectively does is sharing a single namespace across containers. As a consequence, all containers share the access to the interfaces within this namespace as well as firewall rules and ip routes. The level of isolation is reduced but containers benefit from standard inter-process communication (IPC) and hence, suffer less overhead. This mode is often seen in Kubernetes environments under the name of pods. Pods are a group of containers that work together.

Host mode In this mode, containers share the host's OS networking stack. Consequently, all containers can communicate with each other through IPC. Additionally, the host can provide external connectivity while sharing the same IP addresses and port ranges. This is the lowest level of security and flexibility. Effectively, host mode is often used as performance baseline because its networking overhead is the smallest.

Macvlan & Ipvlan Like VLAN tags, which allow to logically partition a HW interface, these drivers allow the creation of multiple L2/L3 interfaces with individual addressing on top of a single HW interface. By doing this, a MAC or IP address, depending on the use case, can be assigned to a container,

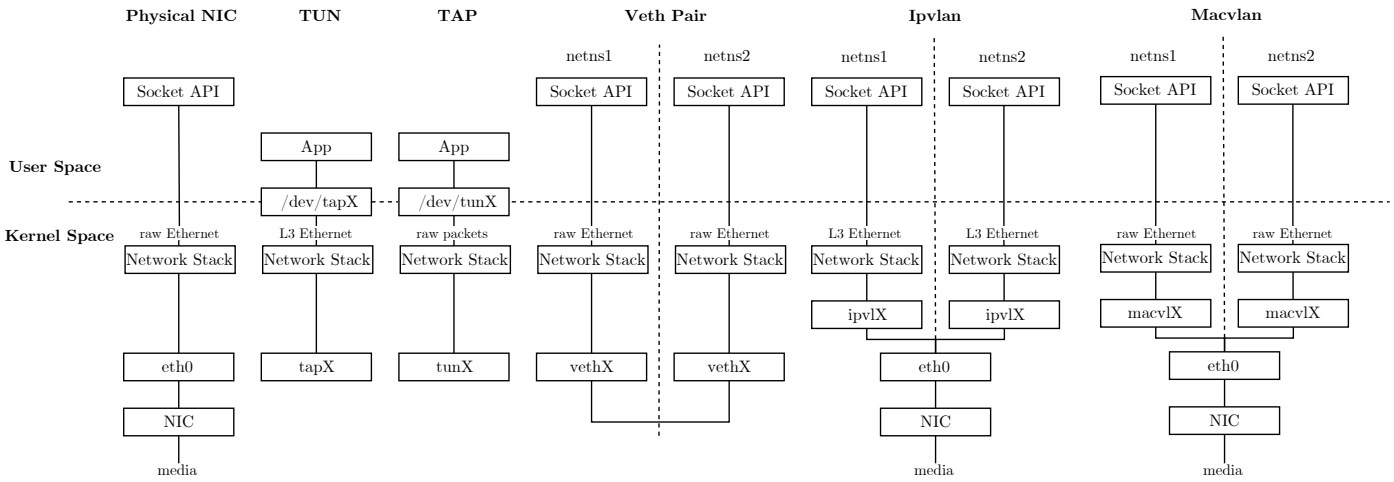

Fig. 2. Available kernel device drivers for intra-host communications.

making it appear as a physical device on the network. To implement this, both the Macvlan and Ipvlan kernel modules get enslaved with the driver of the NIC (master) in kernel space, as shown in figure 2. The enslaved interfaces now share the same broadcast domain, although whether communication between the interfaces is allowed depends on the configured Macvlan and Ipvlan type. The main difference between both is whether the processing of the packets will happen up to L2 (Macvlan) or up to L3 (Ipvlan) in the slave namespace stack. The respective configuration modes of each module determine what type of communication will be allowed between slave interfaces.

2) Network Services for inter-host communication:

Network Address Translation This service can be used on top of bridged mode to provide external connectivity. This approach adds a rule to the NAT table for each container. The rule then maps a port number to a container private IP. When a container sends a packet, the bridge remaps the source (private) IP to the host's (public) IP. Upon reception of packets, the host checks the destination port and the NAT table and performs the corresponding address translation. Although this approach is simple, it incurs a significant processing overhead which leads to a performance loss. In addition, because of the port range constraint, port-conflicts can arise in environments with short-lived containers. Nonetheless, NAT also provides some benefits. Thanks to the address translation, the container's network inside the host is decoupled from the external network. In other words, instead of having to allocate public addresses for each container, a single public IP is required and changes in the external network do not influence the hosts networks.

Overlay Networks An overlay network is a further level of abstraction. In this scenario an underlay network ensures network reachability between hosts. While the overlay, which is tunnelled through the underlay, provides connectivity between containers. Essentially, containers have the impression that they are directly connected to other containers. This abstraction allows great flexibility when deploying containers which need to communicate but cannot be launched within the same host, for instance because of resource limitations. Additionally, an overlay should be resilient to changes in the underlay, thus providing additional flexibility. The choice of the tunnelling protocol depends on whether L2 (VXLAN) or L3 (IPIP, MPLSoGRE, MPLSoUDP, etc.) connectivity is required and what the underlying infrastructure supports. To be able to create the tunnels, containers must share a mapping between their private address and their host's public address. This is usually done in the form of a key-value (KV) store available to all nodes. Overlays are usually based on the kernel's TUN/TAP device driver [23]. In essence, this device driver links the kernel's network stack and a program in user space. Instead of receiving packets from a physical interface, it receives them from a user space program and vice-versa. This is depicted in figure 2. While TUN devices are used with programs that read/write IP packets, TAP devices are used when programs handle Ethernet frames. For overlay networking, a TAP device is used to send the frames leaving a container to the program responsible for the encapsulation and decapsulation.

Routing Another option to provide inter-host communication is to use routing. If a software router is deployed within each host then routing protocols such as BGP can be used to exchange reachability information of containers running on different hosts and on different networks. Thanks to the flexibility of BGP, different virtual routing functions (VRFs) can be created to allow overlapping IP ranges between hosts. This solution is limited to the protocols supported by the software router as well as the routing table scale.

III. Benchmarks

In this section we present and classify a compendium of performance benchmarks for container networks available in the literature: [1], [3], [7], [9], [10], [11], [12], [13]. Thanks to this classification it is possible to identify the scenarios which have already been studied and which not.

First, we summarize in table I the characteristics of the studies limited to standard frameworks. We observe that most studies focus on a single pair of communicating container pairs. The exceptions are [8] and [13] which include inter-host communicating pairs — an expected traffic pattern in microservice architectures. Further analysis shows that most evaluations include libnetwork (docker's underlying networking library)

TABLE I. CLASSIFICATION OF STUDIES ANALYSING THE PERFORMANCE OF STANDARD CONTAINER NETWORKING FRAMEWORKS

Study	Year	Inter- or Intra-host	K8s-CNI/ Docker-plugins	Network mode	Comm. pairs	Measurement tools	Comments
[1]	2018	both	Calico, Weave, Flannel, libnetwork	Host, NAT, VXLAN, BGP	1	Netperf, Sockperf, Sparkyfish, OSU Benchmark	
[3]	2017	both	libnetwork	Veth, Macvlan	1	iperf3, pktgen	
[9]	2021	intra	libnetwork	Bridge, Macvlan, OvS	50	iperf, netcat, sockperf	
[10]	2018	inter	Calico, Weave, Flannel, libnetwork	VXLAN, BGP	1	iperf3	Public clouds: AWS, Azure, GCP
[12]	2018	both	Flannel	VXLAN, OvS	1	iperf	Openstack with Kuryr
[13]	2016	both	libnetwork	Veth, Ipvlan, Macvlan, OvS	1-128	iperf3	

TABLE II. CLASSIFICATION OF STUDIES ANALYSING THE PERFORMANCE OF SPECIAL CONTAINER NETWORKING FRAMEWORKS

Study	Year	Inter- or Intra-host	K8s-CNI/ Docker-plugins	Network mode	Comm. pairs	Measurement tools	Comments
[7]	2018	intra	Custom proto.	AF_Graft	50	iperf3, sockperf3	
[8]	2016	both	Weave and Custom proto.	MPI/DPDK vs. VXLAN	1	iperf	
[11]	2018	inter	Cilium	eBPF/XDP vs. DPDK	1	trex	Artifacts evaluated and reusable

because of its high adoption. However, with the deprecation of Docker as container runtime for Kubernetes, and the adoption of the CNI model, instead of CNM/libnetwork, these results might soon be of little interest. In relation to the network modes, the typical scenarios are all covered; particularly Macvlan and OvS are the most studied. Moreover, the measurement tools focus on L3/L4 protocols because of their prevalence; however CNF's would also benefit from L2 performance indicators. Finally, it is worth mentioning that [10] focuses on public clouds which are not otherwise evaluated.

Second, we compare in table II the studies which include alternative means for achieving higher performance. It stands out, that to achieve higher performance all three frameworks move away from the OS's default network path and use either DPDK, XDP/eBPF or a custom driver. This move effectively reduces the processing overhead; nevertheless, the CPU remains to be the bottleneck. Worth mentioning is the effort put by [11] to make their artefacts available to the community and thus, allowing further researchers to build on top.

IV. FACTORS ANALYSIS

In this section we break down the different factors that influence the performance of the presented container networks. This allows us to better analyse and reuse benchmark results.

Packet size Note that it is only slightly more expensive to send a 1.5KB packet than sending a 64B packet [6]. For this reason, it is useful to to provide the packet rate in packets per second, pps, and indicate additionally the packet size. The average packet size that a container needs to send, receive or manipulate has consequently a big impact on the throughput. Smaller packet sizes imply more packets, what in turn means more CPU cycles being consumed.

Transport Protocol The linux kernel includes optimizations for TCP, such as GRO [29], that allows TCP to perform better than UDP [3]. This is a possible explanation to the important performance degradation observed in [5]. In this study, the authors observe that the UDP throughput is 3.5 times lower than the TCP throughput when using the ipvlan and macvlan drivers.

Latency Budget To reduce the number of interruptions and sustain high throughput rates when the incoming rate is high, packets are buffered before being sent to the network interface card [4]. [2] points out that larger buffer sizes not only increase the throughput but also the average latency. As a result, if the latency budget is limited by the application requirements, the achievable throughput must be reduced by means of reducing the buffer size.

Virtualization layers The number of virtualization layers causes significant overheads. When running containers inside VMs, the packet's data is copied from the hypervisor's kernel space to the user space, where the VM resides. Once in the VM, the virtualized kernel must again copy the data to the VM's user space. This additional copy actions as well as context swapping results in performance degradation. In other words, adding virtualization layers increases the packet's critical path. [1] reports a 42% loss in the TCP throughput when running the containers inside a VM.

Containers-Resources Ratio Another factor is the number of containers within one host which must compete for resources. Most of the container network options make a noticeable use of CPU resources for either NAT or overlay services. Therefore, an elevated number of containers competing for CPU resources interfere each other with multiple context changes. Furthermore, if the CPU becomes the bottleneck, then packets need to be queued incurring additional delays. To remedy this, CPUs can be pinned to specific containers so that they become exclusive. [9] shows this effect, namely that the average throughput decreases and the flow completion times increase with an increasing containers-to-core ratio.

Network driver The implementation of the different network drivers and services has also a relevant performance impact. [1] shows that containers in the same host using bridge mode incur 18% and 30% throughput loss in upload and download, respectively in comparison to the baseline (host). In [5], ipvlan and macvlan perform up to 3 times better than veth bridges. For this reason, linux bridges should be avoided when possible in favour of macvlan or ipvlan.

Network Services While improving flexibility, the encapsulation and decapsulation operations increase the critical path, and thus consume additional clock cycles. Furthermore, the additional headers reduce the payload size (the MTU of the underlay must be respected). Last, possessing the key-value mapping is a requisite for establishing the tunnel. Conse-

quently, the time required for their distribution limits the container's start.

Encryption In cloud environments with multiple tenants, packet encryption is recommended as an additional security layer. The lower the encryption is performed, the higher security degree is achieved. The cryptographic operations have a relevant performance impact which affects both the packet rate as well as the packet delay. The impact of using IPSec has been captured in [19], [20], and [21] for LINUX systems. These studies show a 25-33% performance degradation. Still, similar studies in container environments are missing to confirm the applicability of these results.

V. Models

Having a precise mathematical model whose parameters can be manipulated to observe possible reactions is a very powerful tool. However, modelling complex large-scale systems as a containers network is a complex task. The number of abstraction layers, concurrent processes and middleware makes modelling with high precision an intractable task. This does not mean that it shouldn't be done at all. With the appropriate simplifications and assumptions, a model can provide an approximate result saving a lot of simulation and/or experimentation time.

Gallenmuller et al. [2] survey various frameworks for high-performance packet IO and introduce a model to estimate and assess their performance. The model builds on top of [6] which claims that packet processing costs can be divided into per-byte and per-packet cost; for IO frameworks, per-packet costs dominate. Two assumptions follow: (1) per-packet costs are constant for high performance IO frameworks, (2) measurements are performed under the most demanding circumstances if the highest packet rate is chosen, i.e. 64B packets. Further analysis leads to

$$f^{CPU} = n \cdot (c_{IO} + c_{task} + c_{busy}) \tag{1}$$

where n represents the pps; f^{CPU} describes the available CPU cycles; c_{IO} represents the framework's costs for sending and receiving packets, which are constant by the first assumption; c_{task} are the costs of the application running on top of the framework, and depend of the complexity of the processing task; and c_{busy} which are the costs introduced by the busy wait i.e. polling the NIC. Recall that, in the case of container networks, overlay encapsulation or NAT would be included in c_{IO}, while the containers application logic is represented in c_{task}. The presented measurements prove the accuracy of the proposed model. Consequently, this model can be used to assess the number of containers which can run concurrently within a single host. This removes the possible performance degradation due to interference.

Medel et al. [15] present a performance and resource management model of Kubernetes based on Object Nets [17] (a type of Petri Net [18]). Petri Nets, also known as a place/transition nets, are a formal modelling tool used to describe the behaviour of concurrent and distributed systems. A place represents the state of the system, while transitions represent the actions that can trigger a state change. Places and Transitions are connected by means of arcs. Further, places in a Petri Net may contain a discrete number of marks called tokens. The tokens represent resources which are available for the firing of a transition. Such transition may only be enabled, i.e. ready to be fired, if the required number of tokens are available in the Place. Medel et al. characterize their proposed model using real data from a Kubernetes deployment and suggest that it can be used to design scalable applications. Furthermore, Medel et al. consider the characterization of several overhead sources, including networking. They consider two networking scenarios: (i) one pod is deployed and all containers are inside that pod; and (ii) several pods are deployed with exactly one container each. Their results suggest that deploying several pods with a few coupled containers is better than a single pod with a large number of containers. This allows them to characterize the overhead of the placement of the containers. Further study for the characterization of the intra-host communication overhead is unfortunately not provided.

Khazaei et al. [16] describe an approximate analytical model for performance evaluation of cloud server farms and solve it to obtain accurate estimation of the complete probability distribution of the request response time and other important performance indicators. This analytical model, although conceived to represent dependencies between host in data-centres, could be adapted to characterize container environments due to the similarities between both cases. In the original work, Khazaei et al., propose a M/G/m/m+r queuing system to model the data-center. For the performance analysis they combine a transform-based analytical model and an approximate Markov chain model. This approach allows them to obtain a complete probability distribution of the performance indicators. Khazaei et al. make a remark which might be extrapolable to the container environment: the kurtosis values for two of their experiments show that in heterogeneous cloud centres (i.e., those for which the coefficient of variation of task service time is higher) it is more difficult to maintain Service Level Agreements (SLA) compared to homogeneous centres. In the container world we could expect that in hosts running heterogeneous container networks, meaning a variety of drivers and network services, it will be more difficult to ensure specific performance levels. Consequently, it might be sensible to group containers by driver type and services. Further work is required to validate these statements.

VI. Conclusion

In this paper I have analysed the characteristics of available networking stacks as well as of available container networks. This analysis, paired up with the application requirements shall allow the reader to select the most suitable container network solution. Furthermore, we have seen that many performance benchmarks exist. With consideration of the factors affecting the performance, these benchmarks are a good indication of the performance that can be expected. Yet, the customizability of container environments and the fast pace, at which container technologies evolve, reduce the chances of re-using, or even finding, applicable benchmark results. For this reason, parametrical models would be of high value to the field. Indeed, we have seen in this paper, that the modelling of the CPU usage — the current performance bottleneck— can provide valuable insights of the performance of container networks.

REFERENCES

[1] K. Suo, Y. Zhao, W. Chen and J. Rao, "An Analysis and Empirical Study of Container Networks," IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, 2018, pp. 189-197, doi: 10.1109/INFOCOM.2018.8485865.

[2] S. Gallenmuller, P. Emmerich, F. Wohlfart, D. Raumer and G. Carle, "Comparison of frameworks for high-performance packet IO," 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2015, pp. 29-38, doi: 10.1109/ANCS.2015.7110118.

[3] Yang Zhao, Nai Xia, Chen Tian, Bo Li, Yizhou Tang, Yi Wang, Gong Zhang, Rui Li, and Alex X. Liu. 2017. Performance of Container Networking Technologies. In Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems (HotConNet '17). Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3094405.3094406

[4] Charalampos Stylianopoulos, Magnus Almgren, Olaf Landsiedel, Marina Papatriantafilou, Trevor Neish, Linus Gillander, Bengt Johansson, and Staffan Bonnier. 2020. On the performance of commodity hardware for low latency and low jitter packet processing. In Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems (DEBS '20). Association for Computing Machinery, New York, NY, USA, 177–182. https://doi.org/10.1145/3401025.3403591

[5] J. Claassen, R. Koning and P. Grosso, "Linux containers networking: Performance and scalability of kernel modules," NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium, 2016, pp. 713-717, doi: 10.1109/NOMS.2016.7502883.

[6] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12). USENIX Association, USA, 9.

[7] Ryo Nakamura, Yuji Sekiya, and Hajime Tazaki. 2018. Grafting sockets for fast container networking. In Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems (ANCS '18). Association for Computing Machinery, New York, NY, USA, 15–27. https://doi.org/10.1145/3230718.3230723

[8] Tianlong Yu, Shadi Abdollahian Noghabi, Shachar Raindel, Hongqiang Liu, Jitu Padhye, and Vyas Sekar. 2016. FreeFlow: High Performance Container Networking. In Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16). Association for Computing Machinery, New York, NY, USA, 43–49. https://doi.org/10.1145/3005745.3005756

[9] C. Boeira, M. Neves, T. Ferreto and I. Haque, "Characterizing network performance of single-node large-scale container deployments," 2021 IEEE 10th International Conference on Cloud Networking (CloudNet), 2021, pp. 97-103, doi: 10.1109/CloudNet53349.2021.9657138.

[10] R. Bankston and J. Guo, "Performance of Container Network Technologies in Cloud Environments," 2018 IEEE International Conference on Electro/Information Technology (EIT), 2018, pp. 0277-0283, doi: 10.1109/EIT.2018.8500285.

[11] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress data path: fast programmable packet processing in the operating system kernel. In Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18). Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[12] Y. Park, H. Yang and Y. Kim, "Performance Analysis of CNI (Container Networking Interface) based Container Network," 2018 International Conference on Information and Communication Technology Convergence (ICTC), 2018, pp. 248-250, doi: 10.1109/ICTC.2018.8539382.

[13] J. Claassen, R. Koning and P. Grosso, "Linux containers networking: Performance and scalability of kernel modules," NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium, 2016, pp. 713-717, doi: 10.1109/NOMS.2016.7502883.

[14] Yang Hu, Mingcong Song, and Tao Li. 2017. Towards "Full Containerization" in Containerized Network Function Virtualization. SIGARCH Comput. Archit. News 45, 1 (March 2017), 467–481. https://doi.org/10.1145/3093337.3037713

[15] V. Medel, O. Rana, J. Á. Bañares and U. Arronategui, "Modelling Performance & Resource Management in Kubernetes," 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), 2016, pp. 257-262.

[16] H. Khazaei, J. Misic and V. B. Misic, "Performance Analysis of Cloud Computing Centers Using M/G/m/m+r Queuing Systems," in IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 5, pp. 936-943, May 2012, doi: 10.1109/TPDS.2011.199.

[17] R. Valk, "Object petri nets: Using the nets-within-nets paradigm advanced course on petri nets 2003" in , pp. 3098, 2003.

[18] T. Murata, "Petri nets: Properties analysis and applications", Proceedings of the IEEE, vol. 77, no. 4, pp. 541-580, 1989.

[19] Miltchev, S., Ioannidis, S. and Keromytis, A. (2002) A Study of the Relative Costs of Network Security Protocols. Computer Science at Columbia University. Available at: http://www.cs.columbia.edu/ angelos/Papers/ipsecspeed.pdf

[20] A. Ferrante, V. Piuri and J. Owen, "IPSec hardware resource requirements evaluation," Next Generation Internet Networks, 2005, 2005, pp. 240-246, doi: 10.1109/NGI.2005.1431672.

[21] N. Kazemi, A. L. Wijesinha and R. Karne, "Evaluation of IPsec overhead for VoIP using a bare PC," 2010 2nd International Conference on Computer Engineering and Technology, 2010, pp. V2-586-V2-589, doi: 10.1109/ICCET.2010.5485628.

[22] https://www.kernel.org/doc/html/v5.12/_sources/networking/ipvlan.rst.txt

[23] https://www.kernel.org/doc/Documentation/networking/tuntap.txt

[24] "Data Plane Development Kit: Programmer's Guide, Revision 22.07.0-rc1" Linux Foundation, 2022.

[25] https://www.intel.com/content/www/us/en/developer/articles/technical/using-docker-containers-with-open-vswitch-and-dpdk-on-ubuntu-1710.html

[26] https://www.dpdk.org/wp-content/uploads/sites/35/2018/12/ZhaoyanYipeng_Tungsten-Fabric-Optimization-by-DPDK.pdf

[27] https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/

[28] http://affix.sourceforge.net/affix-doc/c190.html

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