Survey on Performance Models of Container Networks

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

***Abstract*—Reliability, scalability, and flexibility are some of the benefits of cloud architectures. 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. 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 bottlenecks of the available solutions and technologies is a tedious but crucial process. This paper gives an overview of different container solutions, performance benchmarks and performance models. Moreover, the major factors influencing container networks are conceptually introduced. We have the following findings: 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 useful insights.**

I. Introduction

Cloud architectures are motivated by both economies of scale and resources optimization and are enabled by virtualization technologies. The cloud promises elasticity, scalability, reliability, availability, and increased operability. However, performance degradation and reduced portability are often the price of virtualization. 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.

Microservices architecture exploits containers even further. 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. This function decoupling means that components can be updated or replaced without affecting the remaining components. Moreover, applications can be granularly scaled by load-balancing a function into several instances of the same container. If we take a step further and make individual container instances stateless, containers can be re-spawned, if necessary, without impacting the supported application. 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.

We can derive that network connectivity among containers is paramount for their correct functioning. However, it is not free of challenges. A microservice-populated cloud changes constantly within seconds and containers can be considered ubiquitous. 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 element 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 important factor is that caused by the processing of the headers of overlay networks. To enable the communication of containers in constant move, complex overlay networks are required. But again, the host's OS where not conceived to process efficiently headers at this scale. To overcome the performance challenge, operators have conceived sophisticated frameworks for high-performance packet IO.

Due to the immense number of choices for container networking solutions, an evaluation of the available solutions is of vital importance. To be thorough, this evaluation requires of several steps. A mandatory first step is to identify the requirements of the system to be *cloudified*. Further, consulting benchmarks and models can give an indication of the bottlenecks and viability. In the last stages, a high-fidelity model can precisely simulate the results that different configurations and 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 through which packets must 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:

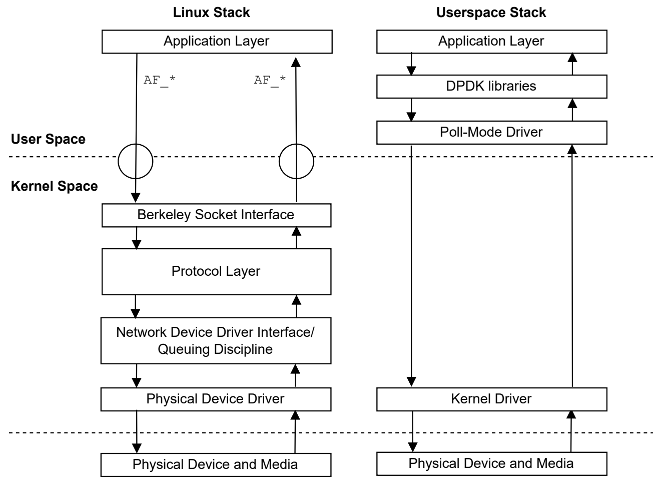

Fig. 1. Kernel's network stack [28] and DPDK stack.

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 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; however, this flexibility is in part responsible for performance losses.

2) User space stacks: Although adding complexity by re-implementing the network stack, high-performance IO frameworks are key-enablers for Containerized Network Functions (CNF) [14]. In some occasions the wide variety of protocols and rich 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. 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 and 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 all the elements needed for high-performance packet processing applications. Please refer to [24] for a detailed view of the libraries.
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, leverages 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 drivers 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) Drivers for intra-host communication:
None Mode In this mode a container is isolated into its own namespace. This 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 to 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. Therefore, 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 group of containers belonging to the same user and 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.

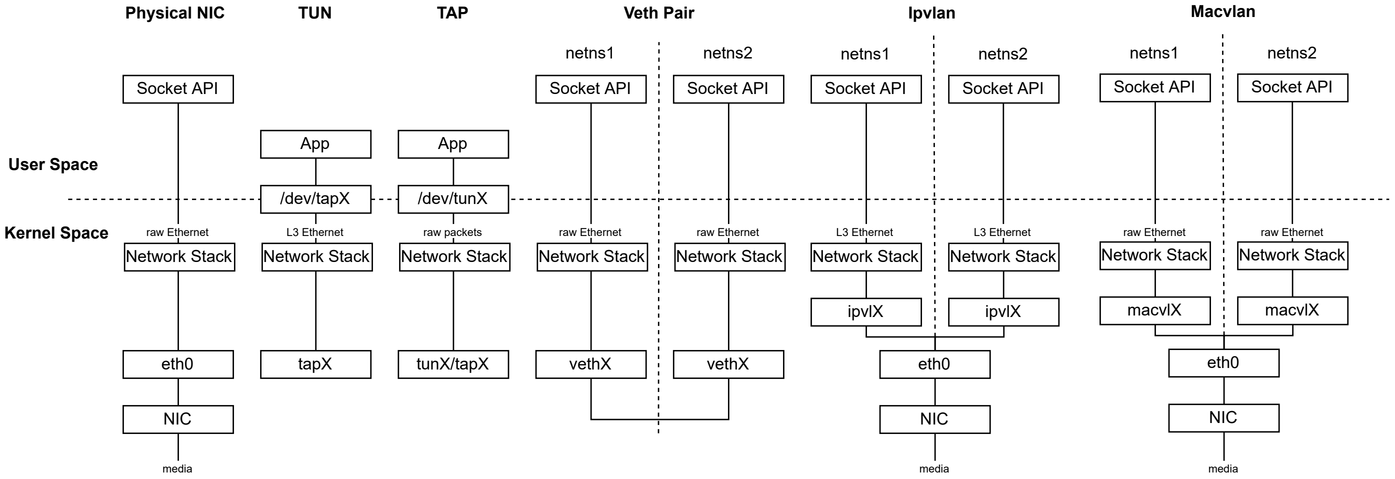

Fig. 2. Available kernel device drivers for intra-host communications.

Effectively, host mode is often used as performance baseline because its networking overhead is the smallest.

Macvlan Like VLAN tags, which allow to logically partition a HW interface, Macvlan allows the creation of multiple L2 interfaces with individual MAC addresses on top of a single HW interface. Using Macvlan, a MAC address can be assigned to a container, making it appear as a physical device on the network. To realize this, the Macvlan kernel module enslaves the driver of the NIC in kernel space. The enslaved interfaces now share the same broadcast domain, although whether communication between the interfaces is allowed depends on the configured Macvlan type.

Ipvlan Like Macvlan, Ipvlan enslaves the driver of the NIC in kernel space; with the difference that all interfaces share the same MAC address. Subsequently, forwarding to the right interface is done based on the L3 address only. According to [22] there are 2 modes of operation: L2 and L3. In layer 2 mode, the packet processing happens on the networking stack of the attached namespace while in L3, L2 processing will take place in the namespace of the master device. L2 mode is effectively the same as using Macvlan in bridge mode. Moreover, in L3 mode the master interface acts as a router, enabling routed communication between containers.

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 for each container a rule to the NAT table. The rule then maps a port number to a container private IP (bridge's subnet). When a container sends a packet, the bridge remaps the source (private) IP to the host's public IP and header changes. Upon reception of packets, the host checks the destination port and the NAT table and performs the corresponding address translation and changes the header. Although this approach is simple, it incurs a significant processing overhead which leads to 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 encapsulated inside 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. In addition, the overlay is resilient to changes in the underlay providing additional flexibility. Being the implementation different, most of the overlays work similarly. The choice usually depends on whether L2(VXLAN) or L3(IPIP, MPLSoGRE and MPLSoUDP) 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. It is important to note that using this technique increases the critical path of the host's datapath and has thus, performance implication.

Routing Yet another option to provide inter-host communication is to leverage 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 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. In table II, we summarize the characteristics of the studies focusing on standard container frameworks. We observe that most studies focus on a single pair of communicating container pairs. The exceptions are [8] and [13] which study intra- and inter-host pairs of containers. This use-case is of relevance, as it reflects the traffic patterns expected in microservice architectures. In both studies, the CPU is identified as bottleneck for the packet rate performance. And in table 2, we compare the characteristics of 3 studies using alternative means for achieving higher performance than using standard means. It stands out, that to achieve higher performance all three frameworks make use of either DPDK, XDP/eBPF or a custom driver. In all cases, the performance increase is due to the reduction of the processing overhead by different means. However, the bottleneck remains to be the CPU.

IV. 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. Further studies can follow if the result yielded by the model is not a show stopper.

Gallenmüller 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, and (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 f^{CPU} describes the available number of cycles provided by the CPU per second; c_{IO} represents the costs used by the framework for sending and receiving packets, and 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 posterior measurements prove the accuracy of the packet processing model. Consequently, this model can be used to assess the number of containers to run concurrently within a single host without leading to 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. Moreover, 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. Regarding network applications, Medel et al., consider two scenarios: (i) one pod is deployed and all containers are inside that pod; and (ii) several pods are deployed with exactly one container each. In both cases they only study intra-host container networking as all pods are scheduled on a single node. From the results, they observe that for applications with more than eight containers, the bandwidth per container is better when deployed in an isolated pod; and suggest, that deploying several pods with a few coupled containers is better than a single pod with a large number of containers. This statement however lacks of relevance without knowledge of the used container network interface, network drivers, and resource policy. Overall, using this modelling approach does not releases operators from the need of performing empirical experiments to characterize the network and thus, does not enable the prediction of the behaviour.

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 doing so, it could provide the relationship between the number of containers and the performance indicators such as mean number of containers in the system, blocking probability, and probability that a container will obtain immediate service, on the other. In the original work, Khazaei et al., propose a M/G/m/m+r queuing system with single task arrivals and a task buffer of finite capacity 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 response time and number of tasks in the system, probability of immediate service, blocking probability. The obtained data can be used to determine the size of the buffer needed to ensure the blocking probability remains below a defined threshold. To validate their results, they have used discrete-event simulation techniques. Khazaei et al. make two remarks which might be extrapolable to the container environment: (i) in cloud centres with diverse services, there

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	Host, NAT, VXLAN, BGP	1	Netperf, Sockperf, Sparkyfish, OSU Benchmark	
[3]	2017	both	libnetwork	Veth, MACVLAN1	1	iperf3, pktgen	
[9]	2021	intra	libnetwork	Bridge, MACVLAN, OvS	50	iperf, netcat, sockperf	
[10]	2018	inter	libnetwork, Flannel, Weave, Calico	VLAN, 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
[11]	2018	inter	Cilium	eBPF/XDP vs. DPDK	1	trex	Artifacts evaluated and reusable
[7]	2018	intra	Custom proto.	AF_Graft	50	iperf3, sockperf3	
[8]	2016	both	Weave and Custom proto.	MPI/DPDK vs. VXLAN	1	iperf	

may exist some tasks which have a relatively long response time than others; consequently, in those general-purpose cloud centres, some tasks may experience unexpectedly long delay or even get blocked; (ii) 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 backup tasks transferring large amounts of data could be specially delayed in non-specific container-based clouds; and that in hosts running heterogeneous container loads, it will be more difficult to ensure that latency is upper-bounded. Further work is required to validate these statements.

V. FACTORS ANALYSIS

To complete the network evaluation toolset, we break down in this section the different factors that influence the performance of the presented container networks.

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 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] also points that in high-performance IO frameworks larger buffer sizes increase the throughput but also the average latency. Therefore, 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 consume additional clock cycles. Moreover, 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. Consequently, 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. However, the encryption and decryption 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. However, similar studies in container environments are missing to confirm the applicability of this results.

VI. Conclusion

We have seen that there are many performance benchmarks available for container networking. Yet, the high degree of customization that containers environment and use-cases offer, reduces the chances of re-using the results of already available benchmarks. Is in the area of modelling were additional effort is required, because models can be adapted to account for specific scenarios. Moreover, we have seen that the current bottleneck for containers networking is the CPU resources. For this reason, modelling the consumption of CPU resources by the virtual networking infrastructure, as done by [2], can provides valuable indications 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. Gallenmüller, 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