Survey on Performance Models of Container Networks

David de Andrés Hernández
Department of Electrical and Computer Engineering
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. This translates into performance degradation, security exposures or unmanageable complexity. 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. 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 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.**
This paper gives an overview of different container solutions, performance benchmarks and performance models. Finally, the major factors influencing container networks are conceptually introduced.

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. 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 appropriately 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 datacenters, 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. This frameworks are orthogonal to the network choice but can be leveraged to compensate some of the drawbacks of the selected network. Cloud networks must also remain extremely flexible as they must often support multi-tenancy while remaining secure. Therefore, flexibility and security are to be considered in the analysis.

Due to the immense number of choices for container networking solutions, evaluating the application needs as well as the benefits and bottlenecks of the available solutions is of vital importance. This survey gives an overview of different solutions for container networking and their underlying technologies. In addition, it presents a collection of available performance benchmarks and models as well as the environments studied so far. Finally, the major factors influencing container networks are conceptually introduced.

II. Background

A. Container Networks

We have seen the importance of container communication in microservices architectures. To materialize this communication, there are multiple options. We differentiate between

modes for intra-host and inter-host communication. Please note that these modes are not exclusive and usually they co-exist.

1) 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 a virtual bridge is created inside a specified namespace. Containers can be launched including a pair of veth ports, 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. The veth pair serves as a pipe between both namespaces. 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. In addition, containers can communicate through the loopback interface at the L4 layer.

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 although sharing the same IP addresses and port ranges. This is the lowest level of security and flexibility. Effectively, host mode can be used as performance baseline as its networking overhead is negligible.

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 containers, making it appear as a physical device on your 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. There are 5 types, namely: private, VEPA, bridge, passthru and source. Private mode does not allow communication between Macvlan instances, even if the external switch supports hairpin mode. In general, a switch will not return a frame to the same interface it has received it. In VEPA mode, the interface expects a VEPA/802.1Qbg capable switch. In this mode the switch acts like a hairpin so frames can be exchanged between the interfaces through the switch. In Bridge mode all enslaved interfaces can communicate with each other by means of the physical interface which works as a bridge. In passthru mode, the frames are just passed to the network. So, if a default L2 switch is attached, it works essentially as in private mode. Finally, in source mode, each Macvlan interface receives only frames whose source address has been whitelisted for this interface. This allows the creation of MAC-based VLAN associations. There is however an important drawback of using Macvlan interfaces, and this is that the slaved interface must be configured in promiscuous mode to allow forwarding of frames which do not include the slaved interface MAC address. Promiscuous mode in turn needs to be carefully considered in environments where multitenancy due to security concerns. If configured with Docker, the daemon routes traffic to the containers based on their MAC addresses. Other container orchestrators, such as k8s, might not implement Macvlan by default, although it can be implemented as a custom resource.

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 [6] 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. Note that the level of operation directly influences whether slave namespaces will handle BUM traffic or not. Ipvlan is useful when the connected switch limits the number of MAC addresses connected per port. Moreover, in L3 mode the master interface acts as a router. This avoids the need for setting an interface in promiscuous mode when the limits of the master MAC address table is reached. Another useful property of both Macvlan and Ipvlan is to efficiently share a single HW NIC between different namespaces. The virtual interfaces can be moved to the respective namespaces and incoming traffic will be forwarded accordingly to the corresponding namespace. The alternative is to use veth pairs together with a virtual bridge which will incur additional overhead.

2) 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 mapping 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 changes the packet header. 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, although this is provided by the underlay. This abstraction allows great flexibility when deploying containers that 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. As always, this flexibility comes with a price. In this case the encapsulation and decapsulations operations consume additional clock cycles. Also, the additional header reduces the payload size as the MTU of the underlay must be respected. Last, as the KV is a requisite for establishing the tunnel, the time required for their distribution becomes the bottleneck for the containers start.

Routing Yet another option to provide inter-host communication is to leverage routing. If a software router is deployed within each host, it might be a daemon container, then routing protocols such as BGP can be used to exchange reachability to 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. Moreover, packet routing also consumes CPU resources.

III. Current Solutions

IV. Benchmarks

V. Models

VI. Factors Analysis

To evaluate the applicability of the different container networks, it is important to identify the major factors of influence. We can group the factors into three categories: performance, mobility, and security.

When talking about networking performance it is not sufficient to talk about throughput. The throughput is often expressed as the product of the packet size, in bytes, and the number of packets processed per time unit, in packets per second. This dependency on the packet size makes vital that when comparing experiments results, the throughput expressed in bps refers in both cases to the same packet size. Note that it is easier to achieve a 1Gbps throughput using 1024 bytes packets than using 64 bytes. In the first case, $976,563$ packets are being sent. While in the second, $15,625,000$ have been transmitted. For this reason, it is useful to express throughput in packets per second, pps, instead of bps. Moreover, latency has to be considered when discussing performance. 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. Therefore, the buffer size directly influences the latency. It might be the case that if the latency budget is limited by the application, the achievable throughput is also reduced [4].

Regarding performance , 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. Another factor is the number of containers within one host which must compete for resources as shown in []. 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. It has also been observed, that because of the auto NUMA scheduling, containers in the same VM might have better performance than in the same bare metal host [3]. This effect is equivalent to the one produced by CPU pinning but causing virtualization overhead.

Next is the network election itself. [1] shows that Containers in the same host using bridge mode incurs 18% and 30% throughput loss in upload and download, respectively in comparison to the baseline (host). This loss is even greater when considering inter-host communication. [] quantifies to X% the degradation in comparison with From the delay perspective, launching

VII. Conclusion

The conclusion goes here.

Acknowledgement

The authors would like to thank...

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