Can We Containerize Internet Measurements?

Chris Misa[†], Sudarsun Kannan[*], Ramakrishnan Durairajan[†]
[†]University of Oregon, [*]Rutgers University

ABSTRACT

Container systems (e.g., Docker) provide a well-defined, lightweight, and versatile foundation to streamline the process of tool deployment, to provide a consistent and repeatable experimental interface, and to leverage data centers in the global cloud infrastructure as measurement vantage points. However, the virtual network devices commonly used to connect containers to the Internet are known to impose latency overheads which distort the values reported by measurement tools running inside containers. In this study, we develop a tool called MACE to measure and remove the latency overhead of virtual network devices as used by Docker containers. A key insight of MACE is the fact that container functions all execute in the same kernel. Based on this insight, MACE is implemented as a Linux kernel module using the trace event subsystem to measure latency along the network stack code path. Using CloudLab, we evaluate MACE by comparing the ping measurements emitted from a slim-ping container to the ones emitted using the same tool running in the bare metal machine under varying traffic loads. Our evaluation shows that the MACE-adjusted RTT measurements are within 20 μs of the bare metal ping RTTs on average while incurring less than 25 μs RTT perturbation. We also compare RTT perturbation incurred by MACE with perturbation incurred by the built-in `ftrace` kernel tracing system and find that MACE incures less perturbation.

1 INTRODUCTION

Container technology directly benefits Internet research by streamlining the tool deployment process and exposing new vantage points at cloud-oriented data centers around the world [1, 4, 5]. The executables, scripts, and libraries needed for running an experiment remotely can be packaged into container images which are then deployed in standardized platforms (e.g., Docker [14]) and orchestration systems (e.g., Kubernetes [7]), eliminating the need for installing dependencies, compiling tools and drastically reducing the deployment footprint when compared with virtual machines (VMs). In terms of vantage points, the global data centers of the top three cloud service providers Microsoft Azure, Google, and Amazon contribute 87 new locations for containerized deployment [1, 4, 5]. In the research community, the PlanetLab consortium officially initiated the adoption of LXC containers as their node provisioning mechanism in 2012, contributing over 700 geographically diverse container-ready vantage points [12].

The building momentum around container technology in research efforts raises the question: *can we accurately containerize Internet measurements?* While containers do provide better performance than VMs, what are the container networking overheads and how do these overheads bias containerized Internet measurements? To answer these questions, the current work looks in particular at the round-trip latency overhead imposed on the ping tool and begins the development of a generic container latency-monitoring system. In resting systems under Docker's default bridged networking mode, ping round trip time (RTT) is inflated on the order of 50 μs [19]. As discussed in section 4, when other containers in the same system generate traffic, *the inflation can increase to over 300 μs depending on traffic volume.* Accurate measurement of this RTT inflation must be available to the measurement container at arbitrary times in order to make sense of the observed RTTs.

Estimating network stack latency is nearly impossible for tools running inside containers because, under the typical bridged and overlay networking modes[1], containers only observe an isolated slice of the host OS's network stack. Timestamps collected in the kernel and directly on hardware network interface cards (NICs) are not available to containerized tools as these timestamps are reset by virtual network devices. To continue leveraging the benefits of bridged and overlay networking while simultaneously accounting for latency, we propose the adoption of a mechanism running in the host's kernel which directly measures network stack latency and reports the results to containers on a need to know basis.

ANRW '19, July 22, 2019, Montreal, QC, Canada

ACM ISBN 978-1-4503-6848-3/19/07…$15.00
https://doi.org/10.1145/3340301.3341130

[1]https://docs.docker.com/network/

In this work we develop MACE, a trace-based method [17, 27, 31], to directly measure the RTT inflation introduced by the kernel network stack. Building on the insight that container functions all execute in the same kernel, we leverage the kernel's perspective to observe the network stack latencies of packets ingressing and egressing container environments. MACE runs in the background on container hosts used for network studies, incurring minimal perturbation on the host while allowing vast improvement in the accuracy of measurements made from containers. We implement MACE as a kernel module interfacing with the Linux trace event subsystem. From trace probes placed on key events in the network stack, MACE gathers per-packet ingress and egress latencies and forwards this information to userspace via per-namespace device files. The key contributions of this work are:

- Preliminary assessment of the challenges involved in using containers to obtain accurate delay measurements on the Internet;
- Development of a trace-based method called MACE to account for the container system's overhead by interfacing with kernel trace events;
- Implementation and evaluation of our method by applying MACE to ping measurements in a realistic deployment. To facilitate independent validation of our results, source code of MACE is publicly available under the GPL v3.0 license[2].

The rest of this work is organized as follows: §2 surveys previous work and discusses the issues faced in measuring the container overhead; §3 describes MACE's design and implementation; §4 offers evaluation of MACE's accuracy and perturbation; and §5 discusses directions for improving MACE based on our evaluation results.

2 RELATED WORK

In this section, we address related work on measuring container system overheads and highlight key challenges in accounting for these overheads in Internet measurement. We also provide a brief look at efforts using trace for latency monitoring to motivate our development of MACE.

Static Approaches: Prior efforts analyze container overheads in comparison with virtual machines (VMs) in wide-ranging performance surveys. The static analysis provided by these works offer insights into the advantages of containers over VMs, but the results are limited to the particular hardware and software configurations under test. Xavier et al. [32] and Felter et al. [19] use various benchmarks to compare the CPU, memory, disk-IO, and network performance of VMs, containers, and native installations showing a container RTT latency overhead between 10 to 30 μs under the default Linux-bridge network configuration. Perhaps the most focused discussion of container networking can be found in Zhao et al. [33], which provides a systematic evaluation of different virtual networking topologies and looks at the effect of NUMA node affinity and number of flows on network performance.

Dynamic Approaches: In realistic environments, the network traffic from other containers running on the same host has a significant impact on the latency overhead of the container system, as does the particular hardware and software configuration (see section 4). In light of this, container overhead measurement must be recast as a dynamic measurement problem. Some container measurement efforts do consider the possibility of dynamic monitoring: [30] suggests some techniques for monitoring the end-to-end latency of container-based VNF chains, [16] explores and evaluates existing performance analysis tools for Docker, and [20] integrates the Faban bench-marking system [3] in a containerized environment. None of these studies provide adequate solutions to account for overheads in containerized Internet measurement.

Mitigating Virtual Network Latencies: Several recent works propose redesigning the network stack to improve the performance of container networking. For example [34] proposes a flow-based virtualization scheme and [24] proposes virtualized RDMA. These efforts offer increased performance in terms of throughput for data center applications, but do not yield any benefits for containerized Internet measurement. For instance, measurement tools such as ping and traceroute generate large numbers of packets with no particular flow-level semantics while the hardware requirements for RDMA make is a non-starter. We leave restructuring of the network stack to support both high-performance container networking *and* Internet measurement tools as an open problem, focusing in this work on accurately measuring the container latency overhead as a first step.

3 MACE DESIGN & IMPLEMENTATION

To address container network latency as a dynamic measurement problem, we develop MACE as a Linux kernel module which employs the trace-event subsystem to directly measure network stack latencies. An application seeking to use MACE to account for network stack latency must only request that MACE be activated for its network namespace and then read from the file which exposes the latency ring buffer created for that namespace. In this section, we detail the architecture of MACE, describe MACE's userspace interface, and discuss some issues of implementation.

[2]http://github.com/chris-misa/mace

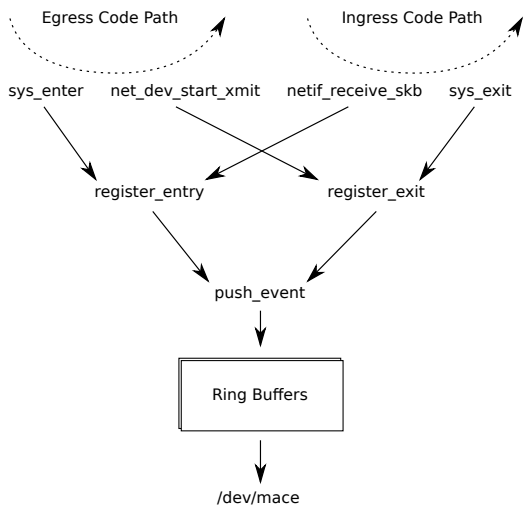

Figure 1: MACE architecture.

3.1 Architecture

As shown in figure 1, MACE works by filtering the raw stream of Linux trace events into per-packet egress and ingress latencies for each network namespace. MACE's `register_entry` and `register_exit` functions manage egress and ingress hash tables to track packets between trace probes while MACE's `push_event` function gathers latencies computed from these tables into a per-namespace ring buffer. Finally, userspace applications obtain latencies by reading the device special file `/dev/mace` which pulls entries from the ring buffer for the reading process's namespace and sysfs files which return aggregated statistics. This multi-layered, in-kernel filtering approach offers far less perturbation and than a generic tracing utility such as `ftrace` and greatly reduces the amount of potentially sensitive trace data forwarded to userspace.

3.2 Implementation

MACE is implemented as a Linux kernel module in ~1k lines of C code, utilizing Linux kernel hash functions, radix trees, traceprobes, and the device file framework. To deal with concurrency in the network stack, MACE employs spin locks and atomic integers, ensuring the accuracy of computed latencies. The compiled binary is ~30kB and expands on insertion to ~40kB.

4 EVALUATION OF MACE

We conduct initial microbenchmark experiments to gain insight on the container networking latency overhead and to evaluate the performance of MACE. Our results show that MACE is able to accurately account for the latency induced by the Linux bridge used in docker's default networking mode in a busy environment with multiple containers vying for network resources. We also show that MACE incurs significantly less perturbation than the built-in `ftrace` system.

4.1 Setup and Methodology

Our CloudLab [26] testbed consists of two m510 nodes[3] on the same experimental network. Both nodes run Ubuntu 18.04 with Linux kernel 4.15.0 and docker-ce 18.09.3. Our test container runs a fork of the iputils ping utility, modified to use gettimeofday for receive timing to facilitate accounting for network stack overheads. This modification allows us to reason directly about the network stack latency and is representative of many other latency measurement utilities (e.g. owamp[4]). To attain ground-truth measurement of the network RTT, we develop another fork of the iputils ping code which uses hardware timestamping to calculate RTT from the host's NIC[5].

In order to simulate worst-case cloud networking conditions, we add background ping containers executing flood ping against the target while the probe measurements are taken. To reduce scheduler noise, these background containers are sequentially pinned to the 16 logical CPUs of the m510 nodes, starting back at CPU 0 after the 16th container is added.

4.2 Effects of Packet Size

To gain initial insight into the nature of the container latency overhead, we observe the flood ping RTT in a resting system with different packet sizes (i.e., 16 B, 56 B, 120 B, 504 B, and 1472 B) (see Fig. 2). These packet sizes reflect a wide range of applications between the minimum and maximum transmission unit.

Fig. 2 shows the difference in mean RTT for each of 5 runs on 5 different cloudlab node pairs. There does not seem to be any strong correlation between packet size and the container latency overhead. *We conclude that the container induced portion of this bias is not affected by packet size.* Building on this conclusion, we execute the following experiments for one particular packet size and assume the results will hold for other packet sizes as well.

4.3 Accuracy and Coverage

In this section we present results from our experiments aimed at verifying the accuracy of MACE's reported latencies. We run 2k-packet flood pings from native and container environments with 1472 B payloads, repeating the process 5 times after 5 s intervals to mitigate the effect of temporal network

[3]Each node has an eight-core Intel Xeon D-1548 processor at 2.0 GHz with 64GB DDR4-2133 RAM and a dual-port Mellanox ConnectX-3 10 GB NIC.
[4]http://software.internet2.edu/sources/owamp/owamp-3.4-10.tar.gz
[5]The mean hardware latency is around 20μs.

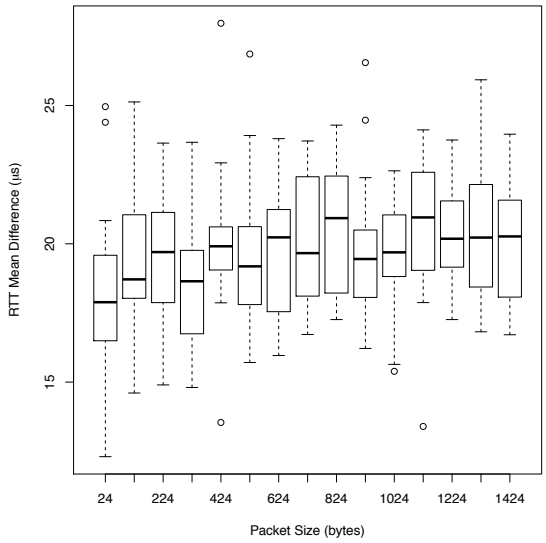

Figure 2: Difference between mean container RTT and mean native RTT in a resting system. There is no significant impact of packet size on the container latency overhead.

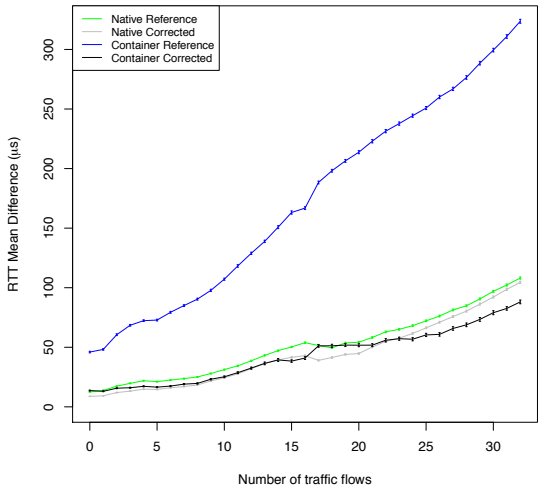

Figure 3: Mean error compared with hardware RTT ground-truth. The container corrected trace (black) using MACE to account for latency is comparable to the native reference trace (green).

anomalies. Each experiment is executed on 5 different pairs of CloudLab nodes to mitigate any bias introduced by the experiment network. In this manner, we gather 50 000 distinct RTT reports for each tested setting. In the following graphs, error bars show a 95% confidence interval over all RTT reports under the normal assumption.

Accuracy: To measure the accuracy of MACE, we take the RTT emitted from the container and subtract MACE's reported egress and ingress latencies to form container corrected RTT values, then compare these container corrected RTTs to the hardware base-line. We perform the same processes with the same ping version running in the native environment to attain native corrected RTTs. Additionally, we compare with a native reference RTT take by the original `SO_TIMESTAMP`-using verion of ping commonly found on Linux systems.

Fig. 3 shows the difference between each of these software-emitted RTTs and the hardware-emitted baseline as a function of the number of traffic-generating containers. From this plot we see that the container corrected and native corrected results are consistently within 20μs of the native reference result, even as the container overhead increases. This demonstrates that *MACE's reported latencies are accurate enough to compute corrected RTTs directly comparable with traditional native implementations of ping.* Another notable result visible in fig. 3 is that even traditional native RTT measurements can be shifted up to 100μs by traffic running from isolated containers on the same host.

Coverage: While MACE's design allows for the generation of per-packet latency information, collisions in the ingress and egress intra-kernal tracking tables can lead to dropped latencies for some packets. To understand the impact of this effect, we inspect the fraction of all RTTs emitted from our monitored container and monitored native ping runs for which MACE reported both egress and ingress per-packet latencies. The results across different background traffic settings are shown in fig. 4. We observe that *MACE reports latencies for nearly all packets,* though decline in coverage occurs much more rapidly for the monitored container processes than the monitored native processes with a sort of bottoming-out after adding the 17th traffic container.

4.4 Perturbation

In this section we compare the perturbation incurred by MACE with the perturbation incurred by the Linux-native `ftrace` kernel tracer. The `ftrace` system requires no modules or patching and is understood to allow visibility into kernel behavior while incurring minimal perturbation [22]. We access `ftrace` through the `trace-cmd` [28] utility and only activate tracing on the particular events used by MACE (see section 3). Flood ping probes are run in a manner consistent with the previous experiment under increasing background traffic loads and all trace and latency results are discarded to avoid storage issues. We calculate perturbation as the difference in mean reported RTT between monitored and un-monitored ping runs in a particular environment.

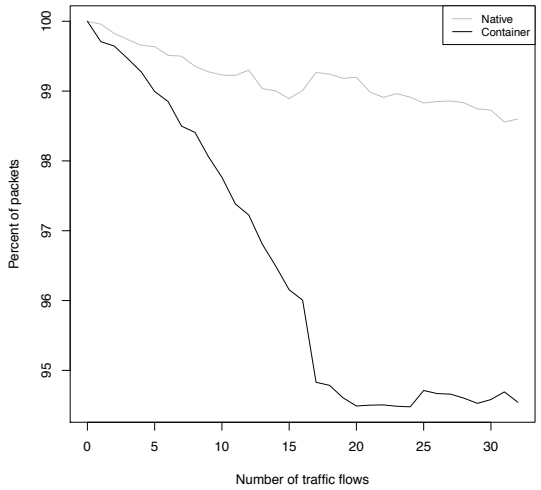

Figure 4: Coverage as the fraction of packets for which MACE successfully computed ingress and egress latencies. Higher egress and ingress times lead to more missed packet for the container environment.

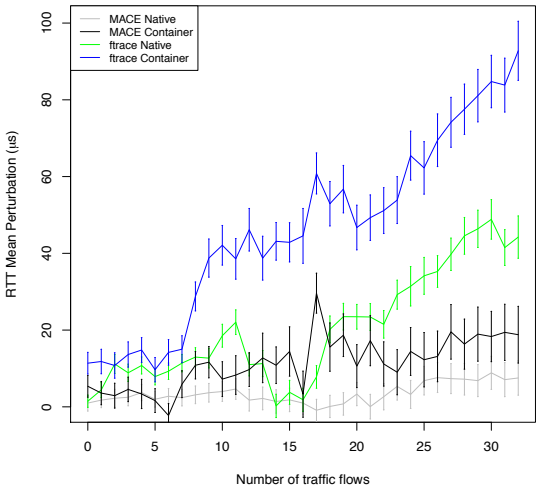

Figure 5: Comparison of RTT perturbations incurred by MACE and `ftrace` with the same trace events activated.

Fig. 5 compares perturbation calculated for MACE and `ftrace` in both native and container environments. These results show that *MACE consistently incurs lower perturbation than* `ftrace`, especially as background traffic increases, staying for the most part below 20μs and 10μs for the container and native environments respectively. Despite its optimized per-CPU ring buffering strategy, increasing the number of active CPUs still causes increase in `ftrace`'s perturbation. While [22] already identified issues with `ftrace`, alternative tracing systems (e.g. [18]) require additional kernel-instrumentation via module insertion or patching.

More problematically, generic tracing systems require large amounts of trace data to be stored and processed in userspace raising storage, privacy, and security concerns. For example previous iterations of MACE used `ftrace` to gather trace events and the resulting intermediate files could grow over 8 GB in size. By filtering trace events in the kernel, our current implementation of MACE only forwards requested latency data to userspace leading to 3.9 MB latency files for the 2k-packet ping runs described above.

Summary: Our testbed results demonstrate that MACE's calculated ingress and egress latencies are able to account for the network stack latency of docker's default bridged networking mode to with in 20μs for 95% of all packets. We also observe that MACE incurs significantly less perturbation than a comparable setting of `ftrace` (an advantage of ~90μs when the network is heavily loaded).

5 FUTURE WORK

In this section, we discuss the possibilities and challenges of extending MACE to other protocols and applications, a key focus of our ongoing efforts. We also discuss directions for future work to improve MACE's design and implementation toward the goals stated in section 1.

5.1 Broadening MACE's Applicability

The method developed here to account for network stack latencies potentially offers a more general approach for denoising Internet latency measurements. A key focus of our ongoing work is to develop MACE to support a wide variety of measurement tools and container standards such as CNI [2]. Tools such as traceroute [11], yarrp [15], and scamper [25] directly measure RTTs while tools in the OWAMP suite [29] such as `owping` measure separate out-bound and in-bound one-way delays. Since MACE already supports separate ingress and egress latency measurement, adapting to these tools is simply a matter of developing MACE to interpret the different system calls and protocols used. We also plan to explore the possibility of using MACE's latency reports to account for network throughput overheads as measured by tools such as iperf [6] and netperf [8]. Finally, adapting MACE to fit into standards such as CNI [2] would immediately enable MACE-corrected measurement in common container management systems such as Singularity [9] and Kubernetes [7].

The current study identifies the following implementation challenges in adapting MACE to a wider set of applications:

The System Call Challenge: While the ping utility employed as a sample application in this study makes consistent use of the `sendto` and `recvmsg` systems calls to send and receive network packets, one of the key challenges faced in applying MACE to other Internet measurement tools is the wide variety of network communication system calls. In Linux there are at least four different options to send and receive data on a connected socket (e.g. `send`, `sendto`, `sendmsg`, `write`) and each system call packages packet data in slightly different formats which will need to be identified and parsed by the `sys_enter` and `sys_exit` trace probes.
The Correlation Challenge: For each socket-api system call mentioned above, there are multiple layer 4 protocols used to encapsulate data. While MACE makes an effort to provide accurate timestamps on each reported latency, in our evaluation we found these timestamps did not provide a high enough level of precision to confidently correlate latencies with reported RTTs. The current implementation of MACE targets the ICMP protocol and includes ICMP sequence numbers with reported latencies for correlation at the application layer. Most common protocols feature such a per-packet sequence number which could be reported with per-packet latencies, but MACE must be implemented to explicitly identify the protocol and parse its header to extract the sequence number.

5.2 Improving MACE's Design

Hardware Timestamps: The `SO_TIMESTAMPING` interface used by our modified ping utility to measure ground-truth hardware RTT attaches timestamps to `sk_buffs` as they enter and exit the kernel from the driver layer. While accessing this data in the `net_dev_xmit` and `netif_receive_skb` trace probes is trivial, utilizing these timestamps to compute latency deltas requires synchronization between the NIC's hardware clock and the kernel's time keeping mechanisms. This method could allow MACE to include driver effects in its latency reports yielding accuracy comparable to the hardware baseline employed in this study as reference.
In-band Measurement: While our current implementation of MACE relies on a separate system to monitor packet latencies, we also intend to explore the benefits of an approach known as *in-band measurement* [13, 23]. In this model, timestamps could be injected into packet data segments either in the kernel, network driver, or Smart-NIC [10, 21] and measurement applications could directly determine the network stack latency by reading these timestamps on received packets. This approach offers a more general latency monitoring solution, but also poses challenges for clock synchronization and requires modification to application code or packet filtering on the receiving end.
Better Correlation Strategies: Evaluations of our current implementation of MACE show that copies between kernel and userspace needed for event correlation incur a substantial bottleneck on MACE's performance. In previous design iterations we had some success using simple heuristics to correlate trace events at the system call layer though such methods yield reduced accuracy. If MACE were implemented as a patch rather than as a module, the structures and functions probed in this study could be modified directly to include unique identifying information for each packet across layers eliminating the need to copy data for correlation.

Summary: While our current implementation of MACE is able to accurately account for latencies imposed by Linux bridged networking in the kernel, we find that more work is necessary to extend MACE to account for all latencies experienced by packets ingressing and egressing containers and to make MACE's results applicable to diverse measurement protocols. We hope to continue improving MACE toward the goals of accuracy, minimal perturbation, and wider applicability.

6 CONCLUSION

In this work, we developed MACE, a system for measuring the network stack latency of containerized applications with a focus on accounting for the latency overheads imposed on Internet measurement tools. MACE reports per-packet ingress and egress latencies for nearly all packets in a busy testbed system. We evaluated MACE's accuracy by subtracting its reported latencies from RTT emitted by ping running in a container and compared the results with ground-truth RTT computed using hardware timestamps and a reference version of ping running in the native context. MACE's reported latencies are able to correct the container RTT to within 20μs of the native reference version on average. Additionally we noted that MACE incurs significantly less overhead than other tracing solutions such as `ftrace` due to its ability to dynamically filter packets in the trace probe code. Finally, we detailed plans to improve MACE in terms of accuracy, perturbation, and applicability to other Internet measurement tools.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful feedback. This work is supported by NSF CNS 1850297 and a fellowship from the University of Oregon (UO) Office of the Vice President for Research and Innovation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF or UO.

REFERENCES

[1] Azure locations. https://azure.microsoft.com/en-us/global-infrastructure/locations/.

[2] The container network interface. https://github.com/containernetworking/cni.

[3] Faban - helping measure performance. http://faban.org/.

[4] Global cloud infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/.

[5] Google data centers. https://www.google.com/about/datacenters/inside/locations/index.html.

[6] iperf - the tcp, udp, and sctp network bandwidth measurement tool. https://iperf.fr/.

[7] Kubernetes. https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/.

[8] The netperf homepage. https://hewlettpackard.github.io/netperf/.

[9] Singularity. https://sylabs.io/guides/3.0/user-guide/networking.html.

[10] Towards converged SmartNIC architecture for bare metal public clouds. https://conferences.sigcomm.org/events/apnet2018/slides/larry.pdf. Accessed: Jan. 2019.

[11] traceroute(8) - Linux man page. https://linux.die.net/man/8/traceroute.

[12] PlanetLab migrating to LXC. https://planet-lab.org/node/263, September 2012.

[13] Data fields for in-situ oam. https://www.ietf.org/id/draft-ietf-ippm-ioam-data-05.txt, 2019.

[14] Docker. https://www.docker.com/, 2019.

[15] Robert Beverly. Yarrp'ing the Internet: Randomized High-Speed Active Topology Discovery. In *Proceedings of the Sixteenth ACM SIGCOMM/USENIX Internet Measurement Conference (IMC)*, November 2016.

[16] Emiliano Casalicchio and Vanessa Perciballi. Measuring docker performance: What a mess!!! *International Conference on Performance Engineering Companion*, pages 11–16, 2017.

[17] Shi-qiang Chen, Yi-fang Qin, Jun-feng Wang, and Xu Zhou. Hpap: High precision active probe for path roundtrip delay measurement. *Proceedings of the IEEE 14th International Conference on Communication Technology*, pages 82–87, 2012.

[18] Mathieu Desnoyers and Michel Dagenais. The lttng tracer : A low impact performance and behavior monitor for gnu / linux. 2006.

[19] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2015.

[20] Vincenzo Ferme and Cesare Pautasso. Integrating faban with docker for performance benchmarking. *International Conference on Performance Engineering*, pages 129–130, 2016.

[21] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, 2018. USENIX Association.

[22] Mohamad Gebai and Michel R. Dagenais. Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Computing Surveys*, 51(2), 2018.

[23] Changhoon Kim, Parag Bhide, Ed Doe, Hugh Holbrook, Anoop Ghanwani, Dan Daly, Mukesh Hira, and Bruce Davie. In-band network telemetry (INT). https://p4.org/assets/INT-current-spec.pdf, June 2016.

[24] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: software-based virtual rdma networking for containerized clouds. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, pages 113–125. USENIX Association, 2019.

[25] M. Luckie. Scamper: a scalable and extensible packet prober for active measurement of the internet. *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC)*, pages 239–245, November 2010.

[26] Robert Ricci, Eric Eide, and the CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *;login:*, 39(6):36–38, 2014.

[27] Steven Rostedt. Finding origins of latencies using ftrace. https://static.lwn.net/images/conf/rtlws11/papers/proc/p02.pdf, 2009.

[28] Steven Rostedt. trace-cmd: A front-end for ftrace. https://lwn.net/Articles/410200/, 2010.

[29] S. Shalunov, B. Teitelbaum, A. Karp, J. Boote, and M. Zekauskas. A one-way active measurement protocol (OWAMP). RFC 4656, September 2006.

[30] Amit Sheoran, Puneet Sharma, Sonia Fahmy, and Vinay Saxena. Contain-ed: And nfv micro-service system for containing e2e latency. *Proceedings of HotConNet '17*, pages 12–17, 2017.

[31] Benjamin Villain, Matthew Davis, Julien Ridoux, Darryl Veitch, and Nicolas Normand. Probing the latencies of software timestamping. *IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication Proceedings*, 2012.

[32] Miguel G. Xavier, Marcelo V. Neves, Fabio D. Rossi, Tiago C. Ferreto, Timoteo Lange, and Cesar A. F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240, 2013.

[33] Yang Zhao, Nai Xia, Chen Tian, Bo Li, Yizhou Tang, Yi Wang, Gong Zhang, Rui Li, and Alex X. Liu. Performance of container networking technologies. *Proceedings of HotConNet '17*, 2017.

[34] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim:OS kernel support for a low-overhead container overlay network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 331–344, 2019.