



Optimizing Network Latency: Unveiling the Impact of Reflection Server Tuning

Jan Marius Evang^{1,2}  and Thomas Dreibholz¹ 

Abstract This study investigates the dynamics of network latency optimizations, with a focus on the role of reflection server tuning. In an era marked by the demand for precise and low-latency network measurements, our exploration unveils the interplay of diverse parameters in achieving optimal performance. Notably, the implementation of a tuned profile on Linux emerges as a standout strategy, showcasing significant rewards in network efficiency. We highlight the importance of early acceptance of latency-critical traffic in the firewall chain and emphasize the cumulative impact of various optimizations. These findings have practical implications for network administrators and system architects, providing valuable insights for the deployment of efficient and low-latency network infrastructures, essential in the landscape of emerging technologies such as 5G networks and edge computing solutions.

1 Introduction

Effective management of network latency is increasingly important, especially in our era of technologies such as 5G networks, cloud-based Radio Access Networks (RAN) and low-latency edge computing. HiPerConTracer [4, 5] is a valuable tool for high-precision short- and long-term network latency measurements through the utilization of hardware timestamping.

Jan Marius Evang

Simula Metropolitan Centre for Digital Engineering, Pilestredet 52, 0167 Oslo, Norway, and Oslo Metropolitan University, Postboks 4 St. Olavs plass, 0130 Oslo, Norway, e-mail: marius@simula.no, web: <https://www.simula.no/people/marius>

Thomas Dreibholz

Simula Metropolitan Centre for Digital Engineering, Pilestredet 52, 0167 Oslo, Norway, e-mail: dreibh@simula.no, web: <https://www.simula.no/people/dreibh>

Traditional latency measurements involve the transmission of Internet Control Message Protocol (ICMP) [3, 10] Echo Request packets across network paths. Upon reception of an ICMP Echo Request, the receiver’s network stack, i.e. usually the operating system kernel, replies with an ICMP Echo Reply (by default, unless deactivated or firewalled). The time between sending an Echo Request and receiving the corresponding Echo Reply is the Round-Trip Time (RTT). ICMP therefore provides relatively simple and reliable means of measuring latency, without any need for an additional service at the remote site.

While ICMP is convenient for measuring, it may be subject to varying treatments by networking and security devices. In contrast, the User Datagram Protocol (UDP), though less accurate, is the preferred protocol for quality measurements in modern network environments, because the high-demanding services often use UDP payloads [1]. Similar to ICMP, the UDP Echo protocol [11] can be used for UDP-based RTT measurements. However, unlike for ICMP, a UDP Echo service has to be deployed at the remote endpoint, replying to incoming UDP packets. That is, it requires explicit support by the remote endpoint, either by a UDP Echo service in user-space, or by a special router setup for a reply in hardware. This is denoted as “reflector”, since it simply echoes the UDP packet back to the sender.

This paper uses HiPerConTracer [4, 5] to measure round-trip time using reflectors with varying configurations, with the aim to optimize reflectors and to provide recommendations for configuring servers and network devices. By addressing this trade-off between measurement ease and accuracy associated with ICMP and UDP, our research contributes to enhancing latency- and jitter-sensitive production services in contemporary networks.

While recent efforts have been dedicated to achieving low latency by modifying the Linux kernel [2, 7, 9], our approach in this paper is distinct, as we only leverage standard Linux kernel tuning. Focusing on the lightweight UDP reflector service, involving minimal CPU processing, allows for a detailed examination of network performance factors induced by underlying hardware, drivers, and the kernel.

A comprehensive investigation into the underlying mechanisms contributing to the observed result variations is deferred to future research.

2 Methodology: The HiPerConTracer Framework

The measurements in this paper utilized the High-Performance Connectivity Tracer (HiPerConTracer) framework¹ [4, 5]. The HiPerConTracer architecture comprises measurement vantage points (clients) sending ICMP and/or

¹ HiPerConTracer: <https://www.nntb.no/~dreibh/hipercontracer/>.

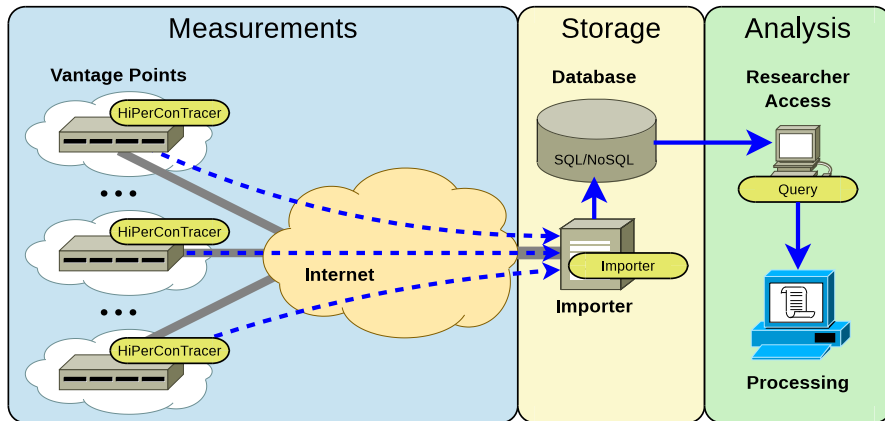


Fig. 1 An Overview of the Architecture of HiPerConTracer.

UDP packets to echo-servers (reflectors), alongside the associated storage and analysis devices, as shown in Fig. 2.

HiPerConTracer’s precise timestamping [5] enables both short- and long-term latency measurements. Measurement vantage points act as clients towards echo-servers (reflectors) to perform latency measurements. The modular architecture of the framework enables flexibility in configuring measurement scenarios, making it well-suited for our investigation into round-trip time optimization.

3 Measurements Infrastructure

For our experiments, we employed a single HiPerConTracer client composed of a server equipped with a 24-thread 3.2 GHz Intel i9-12900K CPU, 32 GiB RAM, and an Intel I225-V Ethernet card. The reflector server featured a 4-core 3.4 GHz Intel Xeon E-2224 CPU and was equipped with five different Ethernet cards. Both servers operated on Ubuntu 22.04. For some tests the the setup was configured as in Fig. 2 with a Linux kernel firewall on the reflector server. For other tests, a Juniper MX80 was configured as a routing firewall as illustrated in Fig. 3, addressing typical real-world scenarios.

Each test consisted of 20 iterations, with packets sent in bursts of 1, 10, and 50 as rapidly as possible. The measurement client remained static, while the configuration of the reflector server varied as described below.

As a measure of latency, HiPerConTracer records the RTT, which is used in this paper. As a measure of jitter we use the Inter-Quartile Range (IQR), defined as the difference between the 75th percentile (Q3) and the 25th percentile (Q1) of the RTT values for all packets in a test.

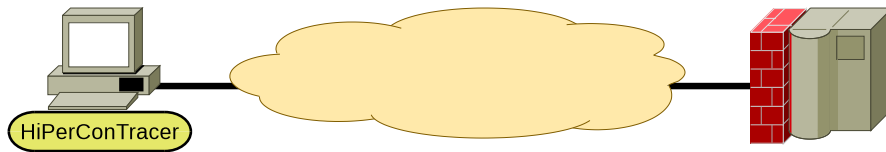


Fig. 2 Scenario 1: Direct Connection with Firewalled Server

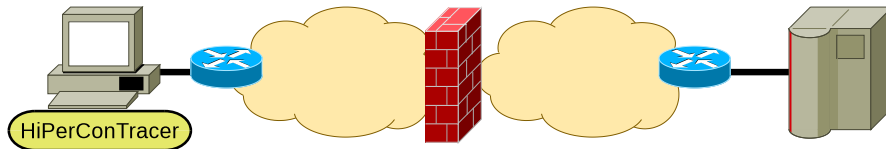


Fig. 3 Scenario 2: Routed Connection with Firewall in the Network

Most of the tests were performed with a user-space process running on the Linux server as reflector. The Linux kernel has a multitude of configuration parameters that may be adjusted at runtime using the `sysctl` interface. Many of the parameters affect the performance of our UDP reflector process, and are adjusted by various tools to test their effect as described below.

Unless explicitly specified, the tests in this paper were conducted with a 100/1000 Mbit/s Ethernet switch in the path. Test G deviated from this standard, utilizing a direct cable between the client and the server.

The tests were executed with both, `iptables` and `nftables` modules loaded, employing an “accept all” configuration, with a tuned profile of “balanced”. The Intel I350 card with ID 0.1 was used, unless stated otherwise in Section 4.

4 Results

We conducted a series of tests on the established infrastructure, exploring diverse aspects of potential tuning options for the reflector server system, as detailed in the following subsections. The used HiPerConTracer source branch is available via GitHub², the dataset of the following experiments is available as well [6].

² HiPerConTracer version 2.0.0~beta4 sources:

Git repository: <https://github.com/dreibh/hipercontracer>, branch “dreibh/udpping”.

4.1 Test A: Tuned Profile

Tuned³ is a tuning daemon designed to enhance the performance of the operating system under specific workloads by applying tuning profiles. It can dynamically respond to changes in CPU and network utilization, adjust `sysctl` settings to optimize performance for active devices and conserve power for inactive. We explored three distinct Tuned profiles:

1. `network-latency`: This server profile prioritizes reducing network latency, emphasizing performance over power savings. It configures `sysctl` parameters such as setting `intel_pstate`, `min_perf_pct=100`, disabling transparent huge pages, and turning off automatic Non-Uniform Memory Access (NUMA) balancing. Additionally, it utilizes `cpupower` to set the performance `cpufreq` governor, requests a `cpu_dma_latency` value of 1, and adjusts `busy_read`, `bus_poll` times to 50 μ s, and `tcp_fastopen` to 3.⁴
2. `balanced`: This profile strikes a balance between performance and power consumption, employing automatic scaling and tuning when feasible. However, it may result in slightly increased latency.
3. `powersave`: Geared towards maximizing power savings, this profile can minimize actual power consumption by throttling performance.

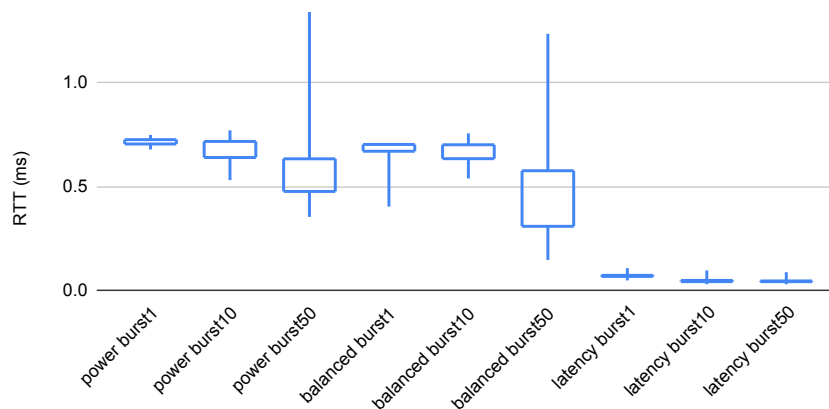


Fig. 4 Test A: Effects of Tuned profiles: Power-optimized, balanced and latency-optimized.

³ Tuned: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/performance_tuning_guide/chap-red_hat_enterprise_linux-performance_tuning_guide-tuned.

⁴ A description of the options can be found in <https://access.redhat.com/sites/default/files/attachments/201501-perf-brief-low-latency-tuning-rhel7-v2.1.pdf>.

As depicted in Fig. 4, employing the kernel Tuned profile for network-latency results in a notable enhancement, reducing latency by a factor of 7.0 to 15.9, and jitter by a factor of 4.0 to 44.8.

4.2 Test B: Firewall Rules

Firewalls can introduce variability to network latency. In this test, we measured the latencies imposed by the Linux kernel firewall running on the reflector server, utilizing the nftables module in iptables compatibility mode.

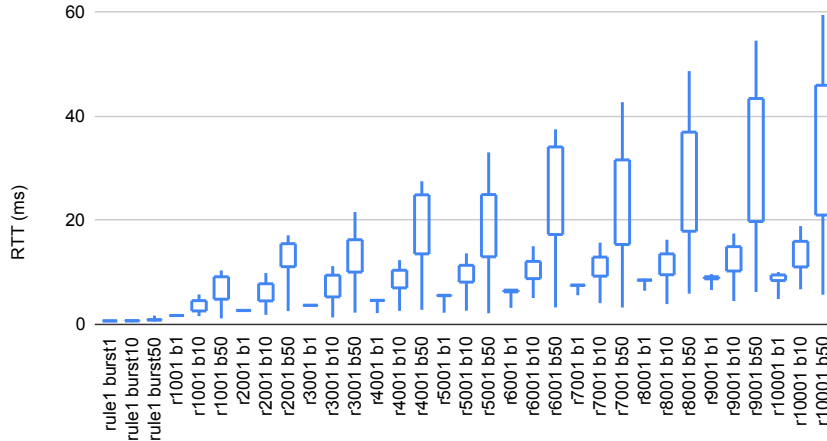


Fig. 5 Test B2: The Effect of a Large Linux Firewall Table

Fig. 5 illustrates the impact of incorporating firewall rules into the Linux kernel. For this experiment, the reflection server was equipped with 10001 firewall rules, each accepting a single UDP port. Subsequent tests were conducted, where the traffic matched the 1st, 1001st, \dots , 10001st firewall rule to gauge differences in processing time within the Linux firewall.

With just one firewall rule, we observed an almost 0.4 ms latency increase for a burst of 50 packets. Furthermore, we noticed a nearly linear latency escalation, with an additional latency of approximately $2 \mu\text{s}$ per firewall rule. It is worth noting that the measurement with 10001 rules, while perhaps unrealistically large, resulted in an almost 60 ms latency increase.

4.3 Test C: Network Interface Cards

The reflection server has three different network cards, with altogether five Ethernet interfaces. One Intel I350 with two interfaces identified with ID 0 and ID 0.1, one Broadcom (BCM) with interfaces ID 0 and ID 0.1, and one Intel 82574L with one interface. Four of the interfaces were tested to observe any differences.

Some notable differences were observed. Particularly intriguing was the variance between I350-0 and I350-0.1, where the ID0 interface exhibited significantly lower latency for burst50, while the ID0.1 interface demonstrated superior RTT albeit with a single high outlier displaying extended latency. There was no similar pattern for BCM, and the cause is unknown.

4.4 Test D: Coalesce

The network cards support network coalescing. This is a mechanism where the Linux kernel does not receive an interrupt (IRQ) for every packet. Instead, the network card queues up a number of packets before sending an interrupt to the kernel to handle a batch of packets. While this approach improves throughput, it introduces latency and jitter. The default coalescing setting for our server is 3 μ s.

Surprisingly, disabling coalescing has a modest impact on RTT and jitter. For burst1, a coalescing setting of 0 μ s shows a slight improvement. However, for bursts of 10 or 50 packets, the default 3 μ s setting has the best results.

4.5 Test E: Kernel Options

Ubuntu Linux offers various kernel options, including the standard kernel, the low-latency kernel, and the real-time kernel. Test E measures the impact of using the low-latency kernel, without modifying the used `udp-echo-server` program (which is the UDP Echo server provided by the HiPerConTracer framework). By enabling `CONFIG_PREEMPT`, the low-latency Ubuntu kernel disables preemption only at critical locations where the kernel must protect data from concurrent access. This means that the UDP reflector will send the response immediately, unless a higher-priority task is being executed. The real-time kernel imposes even stricter constraints but is generally not recommended for standard servers unless precise timing is a requirement.

The results of our tests show a latency improvement ranging from 0.02 ms to 0.2 ms with the low-latency kernel. However, this improvement comes at the cost of slightly increased jitter.

4.6 Test F: Process Affinity for the UDP Echo Server

Since we suspected that CPU scheduling might contribute to latency and jitter in `udp-echo-server`, we investigated the impact of CPU affinity. To explore this, we locked the `udp-echo-server` process to a specific CPU core using CPU affinity.

Applying CPU affinity resulted in a slight reduction of jitter by 0.02 ms for the `burst1` tests. For the other cases, CPU affinity did not significantly impact the results.

4.7 Test G: Direct Connection

All previous tests were conducted with a single 1 Gbit/s switch between the HiPerConTracer client and the UDP Echo reflection server. In order to evaluate whether the switch significantly influenced our measurements, we conducted Test G with a direct cable connection.

The results show a reduction in RTT varying from 0.01 ms to 0.1 ms. Additionally, there is a decrease in jitter for `burst1`, while jitter increases for `burst10` and `burst50`.

4.8 Test H: Juniper MX80 Router as Reflector

While the standard measurement setup with HiPerConTracer's UDP Ping involves using a user-space UDP Echo server on a Linux server, certain router-firewalls like Juniper's MX80 offer a built-in system for latency measurement called Real-Time Performance Monitoring (RPM). In this test, we compare the performance of HiPerConTracer with the RPM probe-server against the Linux UDP Echo server.

The results depicted in Fig. 6 reveal that the RPM probe-server significantly lags behind the Linux UDP Echo server, causing RTT and jitter to increase by 16-120 times. It is noteworthy that in the RPM architecture, hardware timestamping is supported, but only on the client-side.

4.9 Test I: Inline MX80 Firewall

This test explores the impact of the configuration depicted in Fig. 3, where a dedicated routing firewall (Juniper MX80) is positioned between the HiPerConTracer client and the UDP Echo server. The firewall is specifically con-

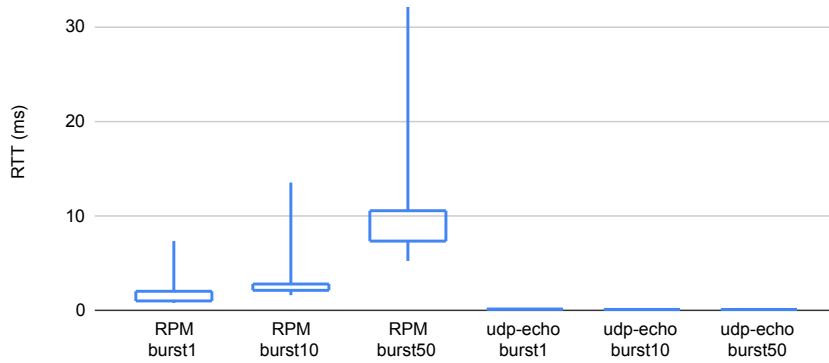


Fig. 6 Test H: Comparison Between RPM Probe-Server and UDP Echo Server.

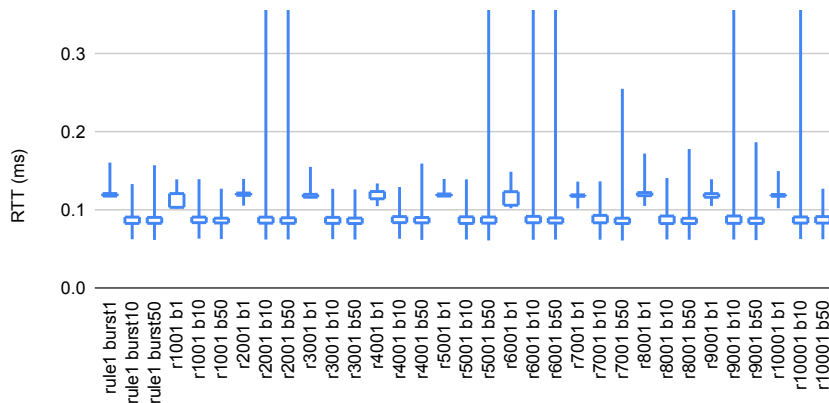


Fig. 7 Test I: The impact of an in-line MX80 firewall.

figured with 1 to 10001 firewall rules, and the corresponding results are visualized in Fig. 7.

The Juniper MX80 firewall is set up with a stateless firewall configuration. Comparing these results to those presented in Fig. 5 yields intriguing insights. The data indicates that the MX80 firewall is notably optimized for large firewall filters. Remarkably, aside from a few outliers (primarily the first packet in a burst), there is no discernible difference in performance, irrespective of whether a packet matches the first or the 10001st filter rule. This pattern is consistent across most tests and suggests that the MX80 firewall employs some form of connection tracking optimization, even in stateless mode.

4.10 Test J: UDP vs. ICMP

Testing with ICMP is often simpler to implement than testing with UDP, as any device can be used to reflect ICMP packets. This method may offer higher accuracy, as packets are reflected in kernel-space. However, considering that latency-sensitive production traffic typically involves UDP, and ICMP packets may be treated differently by network devices, conducting tests with UDP provides more representative measurements.

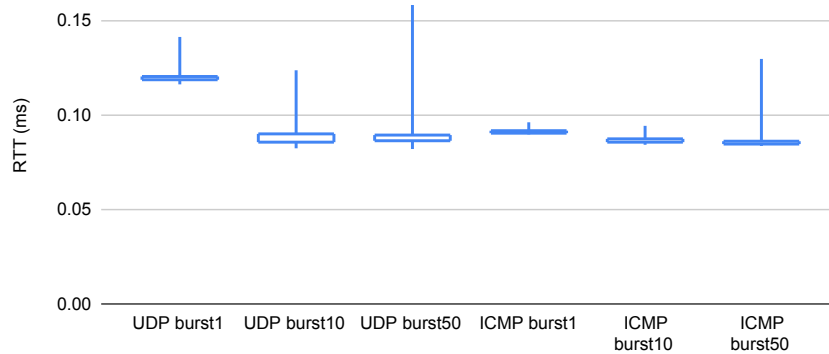


Fig. 8 Test J: Comparison Between UDP and ICMP.

In Fig. 8, the contrast between UDP and ICMP is examined using the Tuned (latency-optimized) Linux server as a reflector. The results indicate that, for burst1, ICMP exhibits significantly lower RTT. In all cases, ICMP demonstrates approximately half the jitter compared to UDP.

4.11 Test K: ICMP to Server vs. ICMP to Firewall

The Linux server responds to ICMP packets in kernel-space, but the Juniper MX80 offers multiple response layers, including a Modular Port Concentrator (MPC), two Flexible Physical Interface Controllers (PIC), a Trio [12] programmable chipset, a bare-metal Linux kernel and a virtualized FreeBSD kernel (Routing Engine), plus various user-space processes. Routers are often not optimized for responding to ICMP ping. This test aims to evaluate whether ICMP Ping to the router is superior or inferior to the Linux server.

Our measurements unsurprisingly reveal that the Juniper MX80 exhibits suboptimal performance in responding to ICMP Ping, introducing up to 18 ms of additional RTT and 1000 times the jitter compared to the Tuned Linux kernel. The reason for the low performance is that ICMP Echo handling is performed in software and is a low priority task.

4.12 Test L: Linux Kernel-Space Reflector

Since HiPerConTracer supports specifying the source UDP port, it is possible to use the Linux kernel's Network Address Translation (NAT) functionality to enable UDP reflection in kernel-space. Our tests show that this further reduces jitter for the bursty measurements by 40 % to 50 % down to the minimal attained $2.4 \mu\text{s}$ to $2.9 \mu\text{s}$ IQR.

4.13 Test Z: Multiple Optimizations

The preceding tests measured the individual variables affecting RTT and jitter. This test, however, examines the collective impact of applying the optimal values for all parameters: It was conducted using a Linux kernel with a latency-optimized Tuned profile, the first firewall rule matching packets, the optimal interface card (BCM-0), zero coalescing, a low-latency kernel, UDP tests, no inline firewall, and utilizing the UDP Echo server.

The improvements achieved through these optimizations are significant, median RTT was reduced from 0.66 ms to 0.04 ms and jitter was reduced by 99 %.

5 Conclusion

In this study, we delved into the intricate landscape of network latency optimizations, unveiling the critical importance of reflection server tuning in achieving precise and low-latency network measurements. Our findings highlight the synergistic effects of various parameters, emphasizing the need for a holistic approach to network optimization.

The implementation of a Tuned profile on Linux emerged as a standout strategy, showcasing substantial rewards in network performance, in particular for a kernel-space reflector. Early acceptance of latency-critical traffic in the firewall chain proved pivotal. While the impact of individual factors might seem modest, the cumulative effect of optimizations became pronounced, especially in the context of stringent latency requirements, such as those dictated by emerging technologies like 5G networks.

Our study underscores the practical implications for system architects and network administrators, emphasizing the need to consider multiple factors collectively for optimal performance. As technology continues to advance, these insights contribute to the ongoing pursuit of efficient and low-latency network infrastructures, which are essential for the success of latency-sensitive applications and edge computing solutions.

Future research should probably delve more into changing the software, exploring socket options and further Linux kernel scheduling features, as well as TWAMP Light (Two-Way Active Measurement Protocol as defined in [8]).

Another topic for future research is the observation that the differences in firewall behaviour between the Linux firewall and the Juniper MX80 together with the achieved level of accuracy (jitter) has the potential to allow fingerprinting of firewalls based on similar network measurements.

References

1. Arouna, A., Bjørnstad, S., Ryan, S.J., Dreibholz, T., Rind, S., Elmokashfi, A.M.: Network Path Integrity Verification using Deterministic Delay Measurements. In: Proceedings of the 6th IEEE/IFIP Network Traffic Measurement and Analysis Conference (TMA). Enschede, Overijssel/Netherlands (2022)
2. Beifuß, A., Runge, T.M., Raumer, D., Emmerich, P., Wolfinger, B.E., Carle, G.: Building a Low Latency Linux Software Router. In: Proceedings of the 28th International Teletraffic Congress (ITC 28), vol. 01, pp. 35–43 (2016). DOI 10.1109/ITC-28.2016.114
3. Conta, A., Deering, S.E., Gupta, M.: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. Standards Track RFC 4443, IETF (2006). DOI 10.17487/RFC4443
4. Dreibholz, T.: HiPerConTracer - A Versatile Tool for IP Connectivity Tracing in Multi-Path Setups. In: Proceedings of the 28th IEEE International Conference on Software, Telecommunications and Computer Networks (SoftCOM), pp. 1–6. Hvar, Dalmacija/Croatia (2020). DOI 10.23919/SoftCOM50211.2020.9238278
5. Dreibholz, T.: High-Precision Round-Trip Time Measurements in the Internet with HiPerConTracer. In: Proceedings of the 31st International Conference on Software, Telecommunications and Computer Networks (SoftCOM). Split, Dalmacija/Croatia (2023). DOI 10.23919/SoftCOM58365.2023.10271612
6. Evang, J.M., Dreibholz, T.: Reflection Server Tuning Dataset. IEEE DataPort (2024). DOI 10.21227/8wbz-7e29
7. Fujimoto, K., Natori, K., Kaneko, M., Shiraga, A.: Energy-Efficient KBP: Kernel Enhancements for Low-Latency and Energy-Efficient Networking. IEICE Transactions on Communications Vol.E105-B (2022). DOI 10.1587/transcom.2021EBP3194
8. Hedayat, K., Krzanowski, R.M., Morton, A., Yum, K., Babiarz, J.Z.: A Two-Way Active Measurement Protocol (TWAMP). Standards Track RFC 5357, IETF (2008). DOI 10.17487/RFC5357
9. Heursch, A.C., Rzehak, H.: Rapid Reaction Linux: Linux with Low Latency and High Timing Accuracy. In: Proceedings of the 5th Annual Linux Showcase & Conference (ALS) (2001)
10. Postel, J.B.: Internet Control Message Protocol. RFC 792, IETF (1981). DOI 10.17487/RFC0792
11. Postel, J.B.: Echo Protocol. RFC 862, IETF (1983). DOI 10.17487/RFC0862
12. Yang, M., Baban, A., Kugel, V., Libby, J., Mackie, S., Kananda, S.S.R., Wu, C.H., Ghobadi, M.: Using Trio: Juniper Networks' Programmable Chipset - for Emerging in-Network Applications. In: Proceedings of the ACM SIGCOMM Conference. Association for Computing Machinery, New York/U.S.A. (2022). DOI 10.1145/3544216.3544262