

High-Precision Round-Trip Time Measurements in the Internet with HIPERCONTRACER

Thomas Dreibholz¹

SimulaMet – Simula Metropolitan Centre for Digital Engineering
Centre for Resilient Networks and Applications (CRNA)
Pilestredet 52, 0167 Oslo, Norway
dreibh@simula.no

Abstract—Accurately measuring Round-Trip Times (RTT) for Internet communications is important for various research topics, ranging from protocol performance and congestion control to routing and network security. Unix systems, particularly Linux and FreeBSD, provide some features to obtain network packet timing information, but there is a lack of documentation for these. With High-Performance Connectivity Tracer (HIPERCONTRACER), there is already an open source tool for running large-scale, long-running and high-frequency ICMP Ping and Traceroute measurements. However, it lacks support of high-precision timing.

As part of this paper, first the network packet timestamping features of Unix systems are analysed and introduced, to provide the reader with a detailed overview over the available methods, their usage, as well as their limitations. Then, enhancements to HIPERCONTRACER are presented for adding high-precision timestamping support, as well as a UDP module to also perform UDP Ping and Traceroute measurements. Finally, the newly added features are demonstrated in a proof-of-concept analysis.

Keywords: Internet, Network, Round-Trip Time, Packet Timestamping, Measurement, Tools, HIPERCONTRACER

I. INTRODUCTION

The reliable transport of payload data between applications over interconnected networks is a complex operation. To tackle this challenge, the task is separated into layers, each providing specific functionalities (like routing, ordered transport, reliable transport, server pooling, etc.). Commonly used models for these layers are the 4-layer TCP/IP model, or the 7-layer Open Systems Interconnection (OSI) model. For this paper, the latter one is suited best. Since these models are well-known, it is referred to literature like [1]. Figure 1 shows the network communication in this model. Data flows through

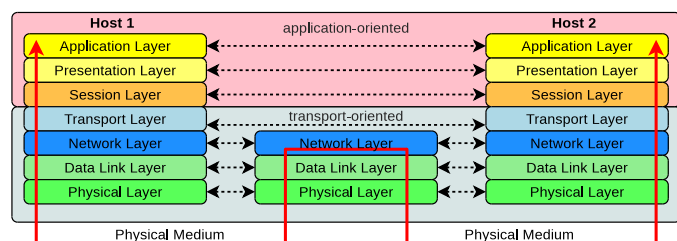


Figure 1. Application Communication over a Network in the OSI Model

the various layers via interfaces. Protocols realise the peer-to-peer communication of each layer. While end-systems obviously need all layers, routers may only realise the lower ones. But layering comes with costs, e.g. encapsulating and decapsulating Protocol Data Units (PDU), copying data, etc., not only for computation but also in form of latency. This particularly applies to end-systems, where most layers are realised in software. The latency reduces interactivity of real-time communications, but also affects protocol performances. So, for analysis and research, it is interesting and important to get *accurate* information about latencies in the network.

This paper focuses on the *high-precision* measurement of Round-Trip Times (RTT) between applications on Unix systems. The overall motivation is to run long-term measurements to observe RTTs between different end-systems, in order to support further analysis on performance and security [2].

In Section II, the basic tools Ping, Traceroute and HIPERCONTRACER [3] are introduced. Then, Section III provides a detailed overview over the possibilities of obtaining packet timestamping information on Unix (Linux, FreeBSD) systems. Particularly, this is the result of a detailed analysis of kernel sources and testing, to cope with the sparsely documented timing features. Section IV presents the enhancements made to obtain high-precision RTT measurements. This is followed by a proof-of-concept analysis in Section V. Finally, Section VI presents conclusions and future work.

II. PING, TRACEROUTE AND HIPERCONTRACER

Two basic features for network connectivity testing are usually provided by the operating system: Ping and Traceroute. They are briefly introduced in the following, due to their relevance for this work.

Ping [1] uses the Internet Control Message Protocol (ICMP), i.e. ICMPv4 [4] for IPv4 [5] or ICMPv6 [6] for IPv6 [7], for sending simple test messages. That is, an ICMP “Echo Request” is sent to a remote system, to which it should respond with an ICMP “Echo Response”. The response contains a copy of the request’s payload (hence the name “echo”), i.e. using the payload for storing a timestamp with the system time before the transmission, the sender of a request can use the response’s contained timestamp to compute the RTT between both systems.

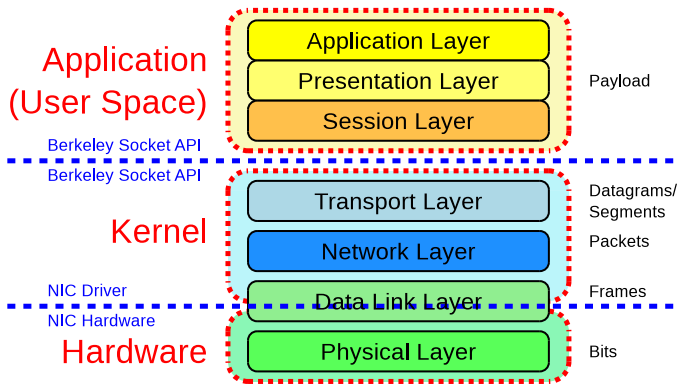


Figure 2. Networking API Layers

Traceroute [1] extends this idea by sequentially sending messages with Time-to-Live (via IPv4) or Hop Limit field (via IPv6) incremented from one. Obviously, if the setting is too low for reaching the destination, a router sends an ICMP “Time Exceeded” error message to the sender, thereby revealing the router’s IP address. This is utilised for collecting the router addresses on the path, including the RTTs between the sender and these routers.

Ping and Traceroute tests are usually performed by the simple command-line tools `ping` and `traceroute`, which just print their results in text form to standard output. They are not suitable for long-term, high-frequency measurements, storing the data in a structured database (i.e. SQL or NoSQL). Furthermore, there are various different implementations, with mostly neither documentation on how timing information is obtained nor its accuracy. Finally, due to the sequential per-hop operation, `traceroute` is also quite slow. To overcome these limitations, particularly observed with long-term measurements for [8], the open source tool `HIPERCONTRACER`¹ [3] (High-Performance Connectivity Tracer) has been developed. It already offers high-performance, long-term, high-frequency `HIPERCONTRACER` Ping and Traceroute measurements, with a known method of how timing information is obtained. However, its timing accuracy is limited. Before explaining how `HIPERCONTRACER` has been extended to overcome this limitation (in Section IV), it is first necessary to provide an overview of how to measure packet timing in Unix systems.

III. HOW TO MEASURE PACKET TIMING

For accurate packet timing measurement, it is first necessary to understand how packet transmission and reception work in Unix systems. Therefore, this is explained first (Subsection III-A). Then, the timing is explained (Subsection III-B). This is followed by the actual measurement possibilities.

A. Overview

First, it is necessary to provide an overview over how the different OSI layers (see Figure 1) are actually realised in a Unix system. Therefore, Figure 2 provides an illustration: the

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

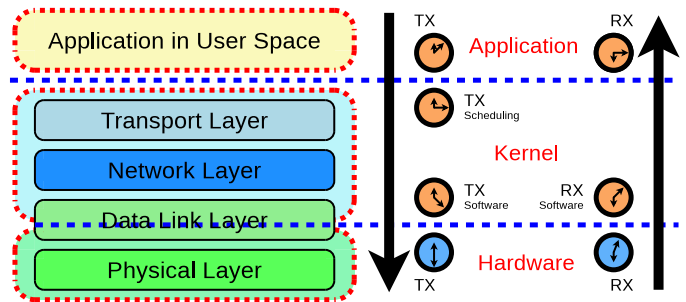


Figure 3. Network Timing in the API Layers

3 upper layers are part of the application itself, and realised in user space. Transport Layer (UDP, ICMP²), Network Layer (IPv4, IPv6) and the upper section of the Data Link Layer (generic part, as well as hardware-specific driver part) are realised in the operating system kernel (i.e. kernel space). The interface between user space and kernel space is the well-known Berkeley Sockets API [9], which provides system calls (syscalls) to the application for accessing the networking functionalities of the kernel. Below the kernel is the actual hardware (e.g. a certain Ethernet network interface). The interface between kernel and hardware is realised by the corresponding device’s driver implementation.

B. A Packet’s Journey Through the OSI Layers

Figure 3 illustrates the places where to obtain packet timing information. Clearly, the application itself can obtain the system time before sending (TX) and after receiving (RX) a packet. However, the application timing is influenced by the process scheduling, e.g. concurrent workload on the system, and context switches between kernel and user space.

In the kernel, it is interesting to distinguish between scheduling (i.e. the packet is enqueued) and transmission (i.e. the packet is provided by the driver to the underlying hardware) on transmission. Between scheduling and transmission, there may be queuing, e.g. due to busy network interfaces, traffic shaping, etc. On reception, datagrams (no reordering is necessary) should be passed without significant latency to the user space (i.e. the application gets notified to read the newly arrived datagram). The measurement accuracy in the kernel is higher than in user space, since the process scheduling and context switch inaccuracy is avoided. However, to obtain transmission timing, the driver-specific code needs to support it.

Clearly, the most accurate timing can be obtained from the hardware, e.g. an Ethernet interface, itself. The hardware is not influenced by effects of the application and operating system, but – of course – needs to support hardware timestamping for transmission (i.e. TX) and reception (i.e. RX). Hardware timestamping support is particularly used for the Precision Time Protocol (PTP) [10] to enable sub-microseconds clock synchronisation, which is e.g. necessary for mobile broadband

²From the perspective of using ICMP for “Echo Request”/“Echo Reply”.

base stations. That is, many recent network interfaces (mostly Ethernet) provide this feature.

Now, the interesting question is: *How to obtain these timestamps from the system?* Since the timestamping features are sparsely documented, detailed analysis of kernel sources, including extensive testing, had been conducted as part of this work, in order to solve this question. Table I summarises the results, showing the kernel features and their support for each type of timestamp: yes (✓), no (✗) or not applicable (-). The details are explained in the following.

C. System Time

In an application, the most obvious way to generate a timestamp is to call the operating system’s function to obtain the current system time. That is, it can be obtained right before sending a datagram (transmission), or directly after receiving a datagram (reception). The Portable Operating System Interface (POSIX) standard for Unix offers two functions:

- 1) `gettimeofday()` returns the current system time as `timeval` structure in microseconds (μ s) granularity.
- 2) `clock_gettime()` returns the current system time as `timespec` structure in nanoseconds (ns) granularity.

Unix systems (including Linux and FreeBSD) offer underlying syscalls to be used by these functions.

However, syscalls may be expensive, since they involve context switches application \rightarrow kernel \rightarrow application. Both, Linux and FreeBSD, therefore apply a Virtual Dynamic Shared Object (vDSO) [11] to load a small shared library into the application’s address space. It provides wrappers for these functions. Instead of performing a syscall, the timing information is obtained via a shared, read-only memory block. So, the expensive syscall is avoided, and obtaining the timing information comes at the low cost of a regular function call.

D. SIOCGSTAMP and SIOCGSTAMPNS

More accurate packet reception timestamps are available from the Linux kernel via two `ioctl()` syscalls:

- 1) `SIOCGSTAMP` [12] returns the last packet’s reception time as `timeval` in microseconds (μ s) granularity.
- 2) `SIOCGSTAMPNS` [12] instead returns a `timespec` in nanoseconds (ns) granularity.

`SIOCGSTAMP/SIOCGSTAMPNS` is Linux-specific, i.e. not available under FreeBSD. This approach is simple to use, but `ioctl()` needs to perform a regular (i.e. expensive) syscall, which is necessary for every packet. However, since the reception timestamp is already recorded in the kernel upon reception, this does not influence the accuracy of the timing information.

E. SO_TIMESTAMP and Variants

A more platform-independent way to obtain packet reception timestamps is the `SO_TIMESTAMP` socket option (to be set once per socket via `setsockopt()` call). When enabled, it provides the timestamp as control data, together with the actual data of the packet, when making a `recvmsg()` syscall to request a received packet from the kernel. That is, both, data

and timing information, are obtained with a single syscall. `SO_TIMESTAMP` comes in different variants:

- `SO_TIMESTAMP` under Linux and FreeBSD (by default) provide a packet’s reception time as `timeval` in microseconds (μ s) granularity.
- On FreeBSD, it is possible to set the clock source using the socket option `SO_TS_CLOCK` [13]. If set to `SO_TS_REALTIME`, `SO_TIMESTAMP` provides a `timespec` in nanoseconds (ns) granularity instead.
- On Linux, `SO_TIMESTAMPNS` provides a `timespec` in nanoseconds (ns) granularity.

That is, unlike `SIOCGSTAMP` (see Subsection III-D), `SO_TIMESTAMP` – using compile-time conditions – provides kernel reception timestamping on different platforms.

F. SO_TIMESTAMPING

Packet transmission timestamping is (currently) only available on Linux, using the `SO_TIMESTAMPING` [14] socket option. This option controls both, the usage of software (i.e. kernel, which is denoted as “software” in the API) as well as hardware timestamping in both directions (see also Figure 3). As for the `SO_TIMESTAMP` variants (see Subsection III-E), the reception timestamping information (software and hardware, if available) is delivered by `recvmsg()` as control data together with the packet data.

For the transmission side, timing information about each outgoing packet (software and hardware, if available) is returned in the error queue of the transmitting socket. It is only necessary to once enable the socket option `RECVERR` (IPv4) or `IPV6_RECVERR` (IPv6) for the socket. Then, like for receiving incoming packets and their control data, the error queue can be read with `recvmsg()` using the `MSG_ERRQUEUE` flag. There, the data about the outgoing packets is provided in the same way as for the reception side. Since it would be inefficient to also return a copy of the transmitted packet data, `SO_TIMESTAMPING` allows to attach an internal ID (a sequence number, which is only stored in the kernel’s metadata structures) to each outgoing packet (e.g. on `sendmsg()` syscall), which is used in the error queue to only refer to a certain packet, without copying and delivering its contents again.

While software scheduling (outgoing) and reception (incoming) timestamping is always available, software transmission timestamping (outgoing packet is forwarded to the underlying hardware) requires the support of the network interface driver.

Support for hardware timestamping (both directions) requires support by the underlying hardware, as well as support by the corresponding driver. For each network interface, it is furthermore necessary to enable hardware transmission and reception support. Hardware may support both directions, only one direction, or none. Furthermore, hardware may support timestamping for arbitrary incoming packets, or only for incoming PTP [10] packets. For generic hardware timestamping (as needed for ICMP and UDP), the underlying network interface must be configured using the `ioctl()` syscall `SIOCShwtstamp` [14], with:

Table I
PACKET TIMESTAMPING POSSIBILITIES UNDER LINUX AND FREEBSD

Method	Type	Application		Kernel			Hardware		Accuracy	System Support
		Out	In	Scheduler	Out	In	Out	In		
<code>gettimeofday()</code>	System Call	✓	✓	–	–	–	–	–	μ s	POSIX Standard
<code>clock_gettime()</code>	System Call	✓	✓	–	–	–	–	–	ns	POSIX Standard
SIOCGSTAMP	<code>ioctl()</code> System Call	–	–	×	×	✓	×	×	μ s	Linux
SIOCGSTAMPNS	<code>ioctl()</code> System Call	–	–	×	×	✓	×	×	ns	Linux ($\geq 2.6.22$)
SO_TIMESTAMP	Socket Option	–	–	×	×	✓	×	×	μ s	Linux, FreeBSD
SO_TIMESTAMP+SO_TS_CLOCK	Socket Option	–	–	×	×	✓	×	×	ns	FreeBSD
SO_TIMESTAMPNS	Socket Option	–	–	×	×	✓	×	×	ns	Linux ($\geq 2.6.22$)
SO_TIMESTAMPING	Socket Option	–	–	✓	✓	✓	✓	✓	ns	Linux ($\geq 2.6.30$)

- 1) `tx_type = HWTSTAMP_TX_ON` enables hardware timestamping for outgoing packets.
- 2) `rx_filter = HWTSTAMP_FILTER_ALL` enables hardware timestamping for *arbitrary* incoming packets.

The hardware timestamping capabilities of a network interface can be checked³ with the `ethtool` tool. It is important to note that hardware timestamps are using the device’s clock (denoted as “raw”, see [14]), while all other timestamps (software, i.e. kernel, as well as the application) are using the system’s clock.

IV. HiPERCONTRACER ENHANCEMENTS

HiPERCONTRACER [3] in its current version 1.6.8 provides HiPERCONTRACER Ping and Traceroute measurements over ICMP. For the packet timestamping, it uses:

- `clock_gettime()` (see Subsection III-C) for the transmission timestamp (via `std::chrono::system_clock` in the standard C++ library, converted to microseconds),
- SIOCGSTAMP (see Subsection III-D) for the reception timestamp (on Linux) or `clock_gettime()` (else).

So, HiPERCONTRACER records RTTs with mediocre accuracy on Linux, with reduced accuracy otherwise, in microseconds granularity.

As part of this work, HiPERCONTRACER has been improved with the following features:

- 1) On Linux, SIOCGSTAMP usage has been replaced by SO_TIMESTAMPING with corresponding error queue processing. SO_TIMESTAMP+SO_TS_CLOCK (see Subsection III-E) is used on FreeBSD. All timestamps are now recorded with nanoseconds granularity.
- 2) With SO_TIMESTAMPING, all time sources from Figure 3 are recorded (when supported by the driver implementation and the hardware; see Subsection III-F).
- 3) The ICMP code has been separated into an ICMP module, to allow for implementing further protocols.
- 4) An UDP module has been added. So, it is possible to run UDP Ping and Traceroute measurements as well.

UDP Ping and Traceroute are a variation of the ICMP variant (see Section II): instead of sending a request by ICMP, they use UDP. Since ICMP is implemented as part of the

TCP/IP protocol stack, no additional software is needed for a system to support responding to an ICMP “Echo Request”. However, firewalls may block this kind of test traffic, or limit the data rate. UDP, on the other hand, needs a small server application to send a response: an Echo [15] service. Therefore, a corresponding implementation has been added to HiPERCONTRACER as well. This requires support by the remote side, but allows more flexibility. Measurement infrastructures like NORNET EDGE [16], [17] therefore run UDP Ping measurements, while infrastructures like NORNET CORE [18] instead use ICMP. The new UDP module now allows HiPERCONTRACER to support *both* variants, and even compare them, with the same program settings and timestamping features.

It should be noted that the UDP module uses raw sockets to generate the outgoing UDP packets, in order to obtain all incoming error messages (like ICMP “Time Exceeded”) for each message. This means that the full HiPERCONTRACER/UDP/IP header hierarchy is prepared by the application, including the Internet-16 checksum [19] computation for IPv4 header [5] (IPv6 has no checksum) and UDP datagram [20].






















The HiPERCONTRACER enhancements are available in the Git repository⁴, as branch “dreibh/udpping”⁵. They are going to be merged into the next major release 2.0.

V. PROOF-OF-CONCEPT ANALYSIS

For a proof-of-concept analysis, and to demonstrate the capabilities of the HiPERCONTRACER enhancements described in Section IV, measurements were conducted from a system at SimulaMet in Oslo, Norway (NO). The system is a Dell Precision 5820 running Ubuntu Linux 22.04.2 LTS. It is equipped with an Intel I219-LM Gigabit Ethernet interface. Of course, this interface and its driver support the software and hardware timestamping capabilities (see Subsection III-F). All measurements were made between May 20 and May 24, 2023. The Internet connection is using the Internet service provider (ISP) Uninett.

³Example: `sudo ethtool -I <INTERFACE>`.

Table II
HiPERCONTRACER PING RTTs TO DIFFERENT TARGET SITES (IN μ S)

Target	Provider	IP	Samples	Q _{1%} .App	Q _{1%} .SW	Q _{1%} .HW	Mean.App	Mean.SW	Mean.HW
Bergen,  NO	BKK	IPv4	1452455	6459.97	6362.07	6306.38	6810.67	6672.72	6604.70
Bergen,  NO	BKK	IPv6	1462396	6285.53	6235.80	6203.00	6596.24	6517.30	6465.54
Bergen,  NO	Uninett	IPv4	1450837	6173.01	6099.72	6061.38	6431.39	6363.84	6328.97
Bergen,  NO	Uninett	IPv6	1461746	6339.03	6214.52	6149.12	6618.63	6487.25	6416.91
Gjøvik,  NO	Uninett	IPv4	1451237	14083.91	13950.44	13879.62	14360.74	14222.00	14146.22
Gjøvik,  NO	Uninett	IPv6	1462896	14187.44	14059.04	13987.00	14462.50	14329.15	14252.39
Karlstad,  SE	SUNET	IPv4	1450708	17298.92	17169.90	17100.00	17536.45	17397.74	17322.09
Kristiansand,  NO	PowerTech	IPv4	1450445	17112.00	17013.94	16965.25	17619.38	17544.81	17501.85
Kristiansand,  NO	Uninett	IPv4	1450878	4993.59	4865.45	4811.00	5287.41	5153.14	5093.88
Kristiansand,  NO	Uninett	IPv6	1462120	4941.10	4819.37	4765.50	5225.37	5096.89	5037.81
Narvik,  NO	Broadnet	IPv4	1451261	24546.38	24412.25	24339.12	27592.40	27462.12	27389.03
Narvik,  NO	PowerTech	IPv4	1448400	25914.21	25791.63	25722.50	28974.68	28841.99	28768.26
Narvik,  NO	PowerTech	IPv6	1460417	26014.87	25884.36	25808.12	29069.09	28934.54	28854.93
Narvik,  NO	Uninett	IPv4	1450879	19783.49	19650.63	19578.62	20078.29	19940.41	19864.31
Oslo (Simula),  NO	Uninett	IPv4	1450769	1833.12	923.83	892.62	1960.95	1279.53	1240.36
Oslo (UiO),  NO	Uninett	IPv4	1450950	1915.09	996.43	965.38	2073.75	1321.88	1277.05
Oslo (UiO),  NO	Uninett	IPv6	1465612	1463.92	957.06	925.12	1627.17	1248.67	1214.66
Stavanger,  NO	Uninett	IPv4	1450998	10332.18	10202.71	10133.88	10603.50	10466.64	10393.53
Tromsø,  NO	Uninett	IPv4	1450913	23164.79	23032.40	22959.88	23450.55	23312.30	23235.45
Trondheim,  NO	Uninett	IPv4	1451140	8899.60	8768.07	8700.25	9181.28	9044.08	8972.03
Trondheim,  NO	Uninett	IPv6	1463854	8829.04	8704.03	8637.62	9094.92	8963.60	8891.82

A. Round-Trip Times


To present an overview, HiPERCONTRACER performed Ping and Traceroute measurements to 10 target sites of the NORNET CORE [18] infrastructure in Norway as well as Sweden ( SE), over IPv4 as well as IPv6 (where available), using its ICMP module. Most of these sites block UDP Echo [15] traffic (firewalls), i.e. UDP vs. ICMP will be analysed separately in Subsection V-B. The interval for Ping is 1 s, Traceroute runs are made approximately every 5 min with 3 rounds per run. Sites can have multiple ISPs.

Table II presents the Ping results of the measurement: the application RTT (App; using system time, see Subsection III-C), as well as the software and hardware timestamping RTTs (SW and HW; using SO_TIMESTAMPING, see Subsection III-F) in form of 1% quantile (Q_{1%}; i.e. 1% of the samples are below this value) and mean (Mean) in μ s. Obviously, as expected, the RTT is lowest for hardware and highest for application timestamping. Interesting is the difference, which represents the cost of the corresponding processing step: approximately 30 to 70 μ s between software and hardware, significantly more between application and software (i.e. kernel). Note that the hardware results (i.e. from/to hardware interface) make the measurements independent of software effects on the *local* side. Of course, the remote side still involves software (for ICMP: kernel only) to send back an Echo Reply.

As part of future work, the precise HiPERCONTRACER RTT results may be used to detect anomalies, like detours for malicious interception, in network paths [2]: given the speed of light in fibre cables is approximately 2.054×10^8 m/s [21],

each μ s of RTT represents around 102.7 m⁶ of cable length. However, the latency contributions of the involved components (routers, remote end-system) need to be investigated further.




Comparing the hardware RTTs for IPv4 and IPv6 by the 1% quantile, it can be seen that the difference between the two IP versions is relatively small, with differences of just up to around 150 μ s. There is no clear advantage for one of the versions, e.g. IPv6 is slightly better with BKK in Bergen (6203 μ s vs. 6306 μ s), while IPv6 is slightly worse with Uninett at the same site (6149 μ s vs. 6061 μ s). Theoretically, IPv6 has slightly less overhead (constant header size, no header checksum to process). However, there are clear differences between the different ISPs. These are particularly visible for the targets in Narvik,  NO and Kristiansand,  NO. The differences indicate routing and access technology variations, so it is interesting to look at the cumulative distribution function (CDF) of the hardware RTTs in Figure 4: Most targets show a steep incline around the average RTT. But for some targets, this incline is smaller. The most interesting one is clearly the Narvik target. For further analysis of the underlying effects, it is therefore interesting to take a look at the *raw* RTT data the CDF is computed from.

Figure 5 presents a 3-hour segment of the data for the Narvik,  NO target, again split between IPv4 (left-hand side) and IPv6 (right-hand side), as well as ISP (Broadnet, PowerTech, Uninett). Only PowerTech offers IPv6 at this site. Each point shows a single hardware RTT measurement. The data is aggregated in 15-minute intervals, where a bright box indicates the absolute minimum/maximum of the interval, and a dark box represents the 1%/99% quantiles. A thick blue line displays the mean of each interval. Interesting here is a split of the RTTs between the three ISPs: While Uninett offers a high-

⁴HiPERCONTRACER master: <https://github.com/dreibh/hipercontracer/>.

⁵Branch: <https://github.com/dreibh/hipercontracer/tree/dreibh/udpding/>.

⁶Note, the RTT measures both, forward and backward direction.

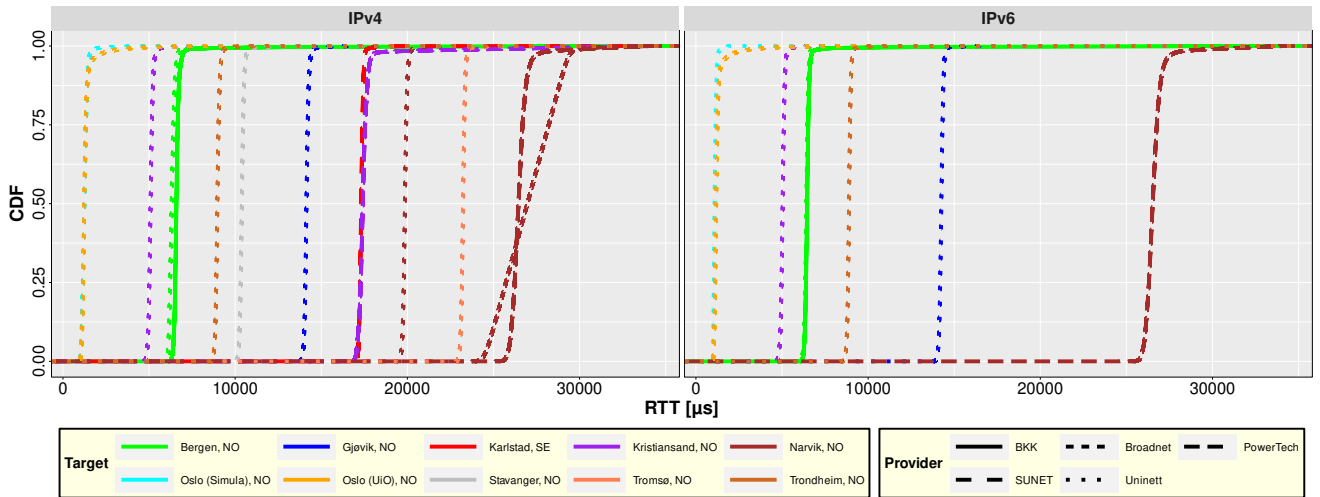


Figure 4. CDF Plot of HiPERCONTRACER Ping Hardware RTTs (in μs)

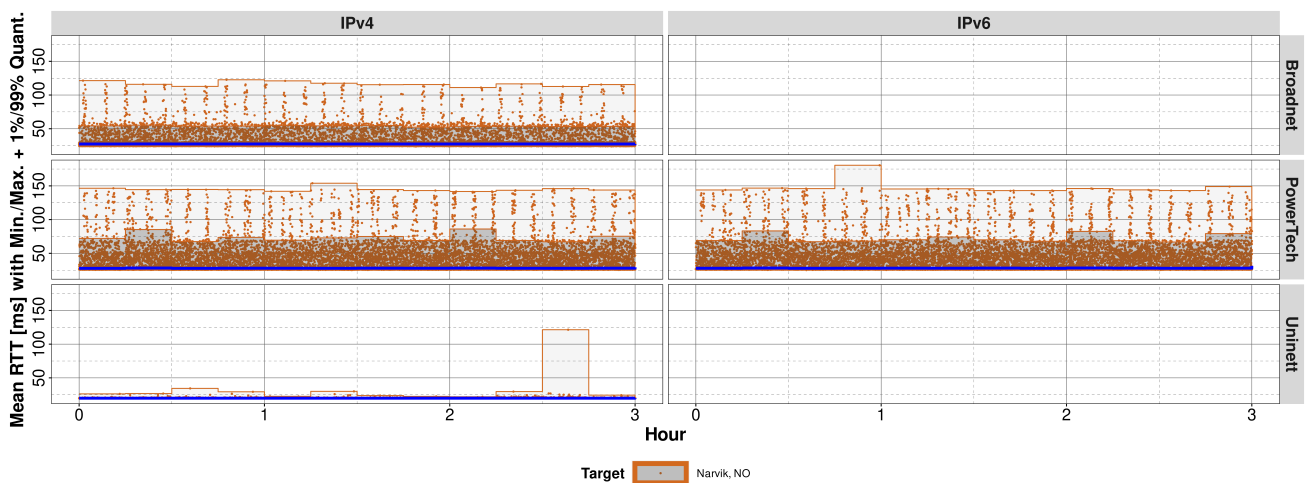


Figure 5. Time-Series Plot of HiPERCONTRACER Ping Hardware RTTs (in ms) for the Target in Narvik, NO

bandwidth fibre link, PowerTech and Broadnet are consumer-grade Asymmetric Digital Subscriber Lines (ASDL). That is, these ADSL lines not only have a significantly higher delay, but the low bandwidth also leads to more queuing. This results in a significant variation of the RTT values. Precise RTT measurement (see Table II) furthermore shows that Broadnet has an around $1400 \mu s$ better RTT than PowerTech ($24339 \mu s$ vs. $25723 \mu s$). Analysing network performance of different subscriptions and access technologies in detail, with HiPERCONTRACER as measurement tool, is subject of further work. For this, it can provide data for both, precise RTTs by Ping as well as routing information by Traceroute measurements.

B. ICMP versus UDP

Finally, RTT results for using the ICMP and UDP module (see Section IV) are compared. Since the NORNET CORE sites utilised for Subsection V-A block UDP Echo [15] traffic, a separate target server has been set up at Simula in Oslo, NO . That is, the HiPERCONTRACER machine and the target

server are located at different sites in Oslo, with the external network of the ISP Uninett in-between. For this experiment, 10 Pings/s have been sent via ICMP as well as UDP. Table III shows the RTT results, again by 1%-quantile and mean. For better readability, the results for software (SW) and application (App) are shown as differences: software to hardware (Diff.*.SW) and application to software (Diff.*.App).

The 1%-quantile for the hardware RTT provides an indicator for the latency of the transport using the 4 variants: ICMP or UDP over IPv4 or IPv6. Interestingly, IPv6 provides almost the same RTT for ICMP and UDP ($702 \mu s$ vs. $705 \mu s$). This clearly differs from the results for IPv4 ($866 \mu s$ vs. $731 \mu s$). Although it would be expected that processing ICMP at the remote side would be faster than for UDP (since it is handled only by the kernel, i.e. no context switches are necessary), UDP has a smaller RTT. Further investigation is necessary here, since this could be an effect of different handling in the network, like e.g. firewalls. Generally, the RTTs for IPv6 are

Table III
HiPERCONTRACER PING RTTs (IN μ S) FOR ICMP AND UDP TRANSPORT

IP	Protocol	Samples	Q _{1%} -HW	Diff.Q _{1%} -SW	Diff.Q _{1%} -App	Mean.HW	Diff.Mean.SW	Diff.Mean.App
IPv4	ICMP	1604231	866.12	↑ 74.38	↑ 129.30	1098.37	↑ 91.26	↑ 176.74
IPv4	UDP	1605151	730.88	↑ 77.49	↑ 159.15	1000.31	↑ 91.27	↑ 213.50
IPv6	ICMP	1604587	702.00	↑ 65.95	↑ 110.52	865.66	↑ 90.89	↑ 171.39
IPv6	UDP	1602995	705.50	↑ 70.85	↑ 149.06	914.96	↑ 91.23	↑ 221.98

lower than for IPv4. This is expected, since IPv6 does not need a header checksum to be computed and verified (i.e. reduced processing overhead).

But how is the software on the *local* side contributing to application and software RTT? On average, the packet processing between kernel and Ethernet interface (SW) very consistently adds around 91 μ s, regardless of IP version and protocol. This is not unexpected, since the Data Link Layer should not make a difference (unless offloading features are used, e.g. for TCP segmentation). For the application RTT, another 171 μ s to 222 μ s are added to the software RTT. The difference between the protocols and IP versions is expected, due to different code paths in HiPERCONTRACER. So, HiPERCONTRACER provides the accuracy to make these differences visible. Clearly, further analysis is needed. Therefore, as part of future work, it is intended to analyse the effects of end-systems and network components in more detail, particularly beginning with experiments in controlled environments.

VI. CONCLUSIONS

Obtaining high-precision RTT measurements for Internet communications is important for various research topics on computer networks. While Unix systems – like Linux and FreeBSD – already provide features for accurate network packet timestamping, they lack documentation. HiPERCONTRACER is an open source tool for high-performance, large-scale, high-frequency ICMP Ping and Traceroute measurements. But its RTT recording accuracy was not optimal.

In this paper, the network packet timestamping features of Unix systems have been analysed and introduced first. This includes available methods, their usage and their limitations, to provide an overview over the various options. In the following, HiPERCONTRACER has been enhanced with support for the high-precision timestamping features of current Linux kernels, as well as with improved accuracy for FreeBSD, in order to provide highly accurate RTT measurements. Furthermore, a UDP module has been added to also support UDP Ping and Traceroute measurements. The new features have been demonstrated by a proof-of-concept analysis.

As part of ongoing and future work, all enhancements are going to be merged into the coming HiPERCONTRACER 2.0 release. Furthermore, it is intended to introduce HiPERCONTRACER 2.0 into large-scale measurement infrastructures, in order to continuously run long-term measurements in the Internet. This particularly includes the NORNET CORE [18] testbed (currently using HiPERCONTRACER 1.6.8), as well as the 4G/5G mobile broadband measurement infrastructure NORNET EDGE [16], [17].

REFERENCES

- [1] A. S. Tanenbaum, *Computer Networks*, 5th ed. Upper Saddle River, New Jersey/U.S.A.: Prentice Hall, Oct. 2010.
- [2] A. Arouna, S. Bjørnstad, S. J. Ryan, T. Dreibholz, S. Rind, and A. M. Elmokashfi, "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, Jun. 2022.
- [3] T. Dreibholz, "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)*, Hvar, Dalmacija/Croatia, Sep. 2020, pp. 1–6.
- [4] J. B. Postel, "Internet Control Message Protocol," IETF, RFC 792, Sep. 1981.
- [5] —, "Internet Protocol," IETF, RFC 791, Sep. 1981.
- [6] A. Conta, S. E. Deering, and M. Gupta, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," IETF, Standards Track RFC 4443, Mar. 2006.
- [7] S. E. Deering and R. M. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," IETF, Standards Track RFC 2460, Dec. 1998.
- [8] F. Golkar, T. Dreibholz, and A. Kvalbein, "Measuring and Comparing Internet Path Stability in IPv4 and IPv6," in *Proceedings of the 5th IEEE International Conference on the Network of the Future (NoF)*, Paris/France, Dec. 2014, pp. 1–5.
- [9] W. R. Stevens, B. Fenner, and A. M. Rudoff, *Unix Network Programming*. Addison-Wesley Professional, 2003.
- [10] H. Weibel, "Technology Update on IEEE 1588: The Second Edition of the High Precision Clock Synchronization Protocol," in *Proceedings of the Embedded World Conference and Exhibition*, Nürnberg, Bayern/Germany, Mar. 2009.
- [11] Linux Manual Pages, "vdso(7)," Mar. 2023, accessed: 2023-05-17. [Online]. Available: <https://man7.org/linux/man-pages/man7/vdso.7.html>
- [12] —, "socket(7)," Mar. 2023, accessed: 2023-05-17. [Online]. Available: <https://linux.die.net/man/7/socket>
- [13] FreeBSD Manual Pages, "setsockopt(2)," Mar. 2023, accessed: 2023-05-17. [Online]. Available: <https://man.freebsd.org/cgi/man.cgi?query=setsockopt>
- [14] Linux Kernel Documentation, "Timestamping," Mar. 2023, accessed: 2023-05-17. [Online]. Available: <https://docs.kernel.org/networking/timestamping.html>
- [15] J. B. Postel, "Echo Protocol," IETF, RFC 862, May 1983.
- [16] T. Čičić, A. Kvalbein, A. S. Al-Selwi, F. I. Michelinakis, and T. Dreibholz, "Norske mobilnett i 2022 – Tilstandsrapport fra Centre for Resilient Networks and Applications," Simula Metropolitan Center for Digital Engineering, Centre for Resilient Networks and Applications (CRNA), Oslo/Norway, Tech. Rep., Apr. 2023.
- [17] A. Kvalbein, D. Baltrūnas, K. R. Evensen, J. Xiang, A. M. Elmokashfi, and S. Ferlin, "The NorNet Edge Platform for Mobile Broadband Measurements," *Computer Networks, Special Issue on Future Internet Testbeds*, vol. 61, pp. 88–101, Mar. 2014.
- [18] E. G. Gran, T. Dreibholz, and A. Kvalbein, "NorNet Core – A Multi-Homed Research Testbed," *Computer Networks, Special Issue on Future Internet Testbeds*, vol. 61, pp. 75–87, Mar. 2014.
- [19] R. T. Braden, D. A. Borman, and C. Partridge, "Computing the Internet Checksum," IETF, RFC 1071, Sep. 1988.
- [20] J. B. Postel, "User Datagram Protocol," IETF, RFC 768, Aug. 1980.
- [21] N. M. Juma, A. D. Edwards, P. Chang, K. L. Corwin, B. R. Washburn, and N. S. Rebello, "Measuring the Speed of Light in an Optical Fiber – Integrating Experimentation and Instrumentation," Ann Arbor, Michigan/U.S.A., Jul. 2009, poster. [Online]. Available: <https://advlabs.aapt.org/tcal/files/JumaN9.pdf>