IEICE TRANS. COMMUN., VOL.E104–B, NO.2 FEBRUARY 2021

POSITION PAPER

Design and Implementation of a Software Tester for Benchmarking Stateless NAT64 Gateways

Gábor LENCSE[†a)], ***Member***

SUMMARY The Benchmarking Working Group of IETF has defined a benchmarking methodology for IPv6 transition technologies including stateless NAT64 (also called SIIT) in RFC 8219. The aim of our effort is to design and implement a test program for SIIT gateways, which complies with RFC 8219, and thus to create the world's first standard free software SIIT benchmarking tool. In this paper, we overview the requirements for the tester on the basis of RFC 8219, and make scope decisions: throughput, frame loss rate, latency and packet delay variation (PDV) tests are implemented. We fully disclose our design considerations and the most important implementation decisions. Our tester, `siitperf`, is written in C++ and it uses the Intel Data Plane Development Kit (DPDK). We also document its functional tests and its initial performance estimation. Our tester is distributed as free software under GPLv3 license for the benefit of the research, benchmarking and networking communities.

key words: *benchmarking, IPv6 transition technology, NAT64, performance analysis, SIIT*

1. Introduction

Several IPv6 transition technologies have been invented to facilitate communication in various communication scenarios despite the incompatibility of IPv4 and IPv6 [1][*]. As prior benchmarking RFCs did not cover IPv6 transition technologies, RFC 8219 [2] was published in 2017 to define a comprehensive benchmarking methodology for IPv6 transition technologies. To that end, it classified the high number of IPv6 transition technologies into a small number of categories: dual stack, single translation, double translation and encapsulation. (Plus DNS64 [3], which did not fit into any of the categories.) Dual stack means that both IPv4 and IPv6 are present and thus benchmarking of network interconnect devices is possible with the existing RFC 2544 [4] and RFC 5180 [5] compliant measurement tools.

The elements of the double translation solutions as well as the encapsulation solutions can also be benchmarked according to the *Dual DUT Setup* [2] in pairs (e.g. NAT46 + NAT64, or encapsulation + de-encapsulation) using the existing measurement tools, too.

However, the *Dual DUT Setup* is unable to reflect the asymmetric behavior, e.g. 464XLAT [6] is a combination of stateless NAT (in CLAT) and stateful NAT (in PLAT). Therefore, they should also be tested separately using the *Single DUT Setup* [2]. Single translation technologies may only be benchmarked according to the Single DUT Setup.

Existing measurement tools assume that IP version does not change, when a packet traverses a network interconnect device, however, this condition is not satisfied in measurements according to the Single DUT Setup. Therefore, new measurement tools are needed.

Due to the depletion of the public IPv4 address pool, DNS64 [3] and stateful NAT64 [7] IPv6 transition technologies have a high importance, because they enable IPv6-only clients to communicate with IPv4-only servers. For DNS64, Dániel Bakai has already created an RFC 8219 compliant benchmarking tool, `dns64perf++` [8]. 464XLAT [6] is also very important as it is widely used in IPv6-only mobile networks to support legacy IPv4-only applications, thus providing IPv4aaS (IPv4 as a Service) [1].

The aim of our current effort is to create a Tester for stateless NAT64 gateways (including NAT46 operations, too). The scope of the tests is the most important ones from among the measurements described in Sect. 7 of RFC 8219, namely: throughput, latency, PDV (packet delay variation), and frame loss rate tests. We note that measurement procedures for stateful NAT64 implementations include the stateless tests plus two further ones, please refer to Sect. 8 of RFC 8219 for more details.

Our new test program, `siitperf`, is a free software for the benefit of the research, benchmarking and networking communities and it is available under the GPLv3 license from GitHub [9].

The remainder of this paper is organized as follows. Section 2 contains the basic operation requirements for the Tester based on RFC 8219. Section 3 discloses our most important design considerations and implementation decisions. Section 4 presents our functional and performance tests and their results. Section 5 highlights our plans for further tests, development, performance optimization and research on benchmarking methodology issues. Section 6 gives our conclusions.

2. Operation Requirements and Scope Decisions

Now, we give a high-level overview of the requirements for the tester, and disclose our considerations behind the scope decisions.

Manuscript received November 28, 2019.
Manuscript revised April 27, 2020.
Manuscript publicized August 6, 2020.

[†]The author is with the Department of Networked Systems and Services, Budapest University of Technology and Economics, Magyar tudósok körútja 2, Budapest, H-1117, Hungary.
a) E-mail: lencse@hit.bme.hu

DOI: 10.1587/transcom.2019EBN0010

[*]Please refer to our paper [1] for a brief description of the IPv6 transition technologies mentioned throughout this paper.

Copyright © 2021 The Institute of Electronics, Information and Communication Engineers

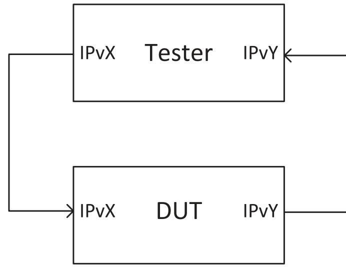

Fig. 1 Single DUT test setup [2].

2.1 Test and Traffic Setup

Section 4.1 of RFC 8219 [2] describes the *single DUT setup* (see Fig. 1), which is similar to the test setup of RFC 2544 [4], but here the IP versions of the left and right side interfaces are different (IPvX and IPvY, where $X \neq Y$, and $X, Y \in \{4, 6\}$). In both RFCs, unidirectional arrows are used, but they mean bidirectional traffic. In one direction, the Tester needs to be able to send IPvX packets to the DUT and receive IPvY traffic from the DUT, whereas in the other direction, it needs to send IPvY packets to the DUT and receive IPvX packets from the DUT at the same time.

We note that whereas bidirectional testing is required, unidirectional tests may also be used.

Although RFC 8219 mentions other media types, it relies on Ethernet and we deal exclusively with Ethernet.

RFC 2544 specifies frame sizes to be used over Ethernet. RFC 8219 explains that they have to be modified, e.g. 84 byte frames should be used on the IPv6 side to achieve 64 bytes on the IPv4 side, since the size of an IPv6 header is 40 bytes, whereas that of IPv4 is 20 bytes.

As for transport layer protocol, UDP should be used.

RFC 8219 also requires that besides the traffic that is translated, tests should also use non-translated traffic (we call it “background traffic”), and different proportions of the two types of traffic have to be used.

2.2 Scope of Measurements

RFC 8219 requires the measurement of different quantities. In practice, some of them are actually measured when benchmarking tests are performed with RFC 2544 testers, and some of them are omitted or rarely used. We intend to support those that we find important. The measurement procedures selected to be supported result in various requirements for the Tester. Now, we overview the procedures and their requirements.

2.2.1 Throughput

Measuring throughput is unavoidable both because it is important for the users and because it is needed for several other measurement procedures. Throughput is defined as the highest frame rate at which the number of frames received from the DUT by the Tester equals the number of frames sent to the DUT by the Tester, that is, no frame loss occurs. This implies that the Tester must be able to send frames at any constant rate for a given time period and count the sent and received frames. (In practice, binary search is used to find the highest rate.)

2.2.2 Latency

Latency is an important characteristic of a NAT64 gateway, thus its measurement must be supported. Its measurement procedure is redefined in RFC 8219 as follows. The Tester must send a stream that is at least 120s long, mark at least 500 frames after 60s and measure the time elapsed from their sending by the Tester to their receiving by the Tester. (Although it is not specified how the marked frames should be distributed, we suppose that an even distribution is desirable.) Then two quantities are calculated, Typical Latency (TL) is their median and Worst Case Latency (WCL) is their 99.9th percentile. This test must be performed at least 20 times, and the final results are the medians of the 20 values for both TL and WCL.

2.2.3 PDV

PDV (Packet Delay Variation) and IPDV (Inter Packet Delay Variation) as defined in RFC 5481 [10] play an important role in the quality of real-time applications. As PDV is recommended and IPDV is optional in RFC 8219, we included only PDV, however, our measurement program may be easily extended to be able to measure also IPDV, as the core of their measurement procedure is the same, and only their calculation is done differently. Their measurement requires to measure the one-way delay for all frames in an at least 60s long stream. Unlike the latency measurement, which requires to store only a small number of timestamps (500 is enough), this measurement may be challenging by means of the storage capacity required and also the CPU performance required to handle two time stamps for each frame.

2.2.4 Frame Loss Rate

Because of the strict definition of the throughput (no frame loss is allowed) and the fact that there are many software based NAT64 implementations, which probably loose packets, when their performance limits are approached, we consider that measuring frame loss is very important. For example, if we determine the maximum lossless frame rate by the throughput test as r, it makes a significant difference whether the frame loss rate is 0.01% or 50% at $2 * r$ rate, as the first one can be used for communication unlike the second one.

The elementary step of the frame loss rate measurement is the same as that of the throughput measurement: send frames at a given rate and count the number of sent and received frames. The frame loss rate is defined by (1).

$$\frac{sent - received}{sent} * 100\% \quad (1)$$

The difference from the throughput measurement is that here, the first frame rate to be used is the maximum frame rate of the media, and then the frame rate is decreased to 90%, 80%, 70%, etc. of the maximum frame rate. The measurement can be finished, when no frame loss occurs during two consecutive measurements. We note that depending on the performance of the available hardware, a software tester my not be able to transmit frames at the maximum frame rate of the media. In such cases, the Tester still can be used for measurements in the range it supports, but then some frame loss rates will be missing. In this case, the supported ranges should be preliminary determined by a self-test, please refer to Sect. 4.2 for details.

2.2.5 Not Supported Measurements

We decided not to implement the remaining three tests, namely back-to-back frames, system recovery and reset. Our primary argument is that they are rarely used. Besides that, the first two would require the Tester to be able to transmit at the maximum frame rate of the media, which is not necessarily met by various devices the users would like to use for executing our test program. The third one would need the ability to cause (or the ability to sense) a reset of the DUT, which would also require additional hardware.

2.2.6 Number of Flows

Section 12 of RFC 2544 requires that first, the tests are performed using a single source and destination address pair and then the destination addresses should be random and uniformly distributed over a range of 256 networks.

3. Design and Implementation of the Tester

3.1 General Design Considerations

3.1.1 Performance Deliberation

`Dns64perf++`, our benchmarking program for DNS64 servers, uses standard socket API and it can send or receive about 250,000 packets per second per CPU core [8]. We considered this performance unsatisfactory on the basis of our previous benchmarking experience. Using the same old 800MHz Pentium III computer as DUT, our DNS64 performance measurement results were under 500 resolved queries per second [11], whereas our stateful NAT64 test results exceeded 21,000 packets per second [12]. Therefore, we decided to use DPDK [13] to ensure the highest possible performance.

3.1.2 Integration or Separation

A fully integrated Tester, which automatically performs all measurements, may be an attractive solution if we need a commodity Tester for routine tests. However, our tester is designed primarily for research purposes. Even the benchmarking methodology described in RFC 8219 is subject to research, because the described measurement procedures have not yet been validated by real measurements due to lack of compliant Testers. Therefore, we decided to develop a flexible tool, which enables the user to access all intermediate results, and experiment easily by executing only certain sub-functions, when required. To that end, we use high performance programs for the elementary functions, which are made flexible by using input parameters instead of built in constants even if RFC 8219 would allow using a constant (e.g. 60s duration or 500 timestamps, etc.) and by using easy to modify bash shell scripts to execute these programs.

3.2 High-Level Implementation Decisions

3.2.1 Software Architecture and Hardware Requirements

In the general case, bidirectional traffic is required by RFC 8219. To achieve a clear program structure and high enough performance, we use one thread pair for the forward[†] direction (one thread for sending and one thread for receiving) and another thread pair for the reverse direction. Each thread is executed by its own CPU core, thus, in the general case, four CPU cores are required to be reserved for their execution (in addition to the core, where the program is started). We note that either of the two directions may be inactive.

Both the Tester and the DUT need two NICs each for testing purposes and a third one for network communication (unless the user wants to work locally on console).

Please refer to Sect. 3.6.1 for further hardware requirements.

3.2.2 Input and Output

The above decision for separation also means that the shell script executes the programs multiple times. Those parameters that change from time to time (e.g. frame size, frame rate, etc.), can be easily supplied as command line arguments. Those parameters that do not change (e.g. IP addresses, MAC addresses, etc.) may be comfortably supplied in a configuration file.

Those results that are to be used by the script for making decisions (e.g. number of frames sent, number of frames received, etc.) are printed to the standard output using a simple format (separate line for each result and unambiguous identification string) so that they can be easily extracted for processing. Those results that are longer and not to be further processed by the script might be written into a result file. (We did not use this solution, because we believed that it did not worth the effort, but it can be easily added later, if needed.)

[†]Following the reading direction of English texts, we call the left to right direction through the DUT in Fig. 1 as “forward” and the right to left direction as “reverse” direction.

3.3 Implementation of the Tests

3.3.1 General Considerations and Input Parameters

The four supported measurements are implemented by three programs (with some overlaps). The first one of them, `siitperf-tp` measures throughput and frame loss rate. The second one `siitperf-lat` measures latency. The third one, `siitperf-pdv` measures PDV, whereas it can also be used for the throughput and the frame loss rate measurements according to more elaborated criteria, which are currently not required by RFC 8219, but are recommended by our paper [14]. For the differences, please refer to Sect. 3.3.2 and Sect. 3.3.4. All three programs use positional command line parameters. The common ones are to be specified in the following order:

- **IPv6 frame size** (in bytes, 84-1518), IPv4 frames are automatically 20 bytes shorter (please refer to Sect. 3.5.2 for the extension of the range to 84-1538)
- **frame rate** (in frames per second)
- **duration of testing** (in seconds, 1-3600)
- **global timeout** (in milliseconds), the tester stops receiving, when this global timeout elapsed after sending has finished
- **n** and **m**, two relative prime numbers for specifying the proportion of foreground and background traffic (see below).

Traffic proportion is expressed by two relative prime numbers n and m, where m packets form every n packets belong to the foreground traffic and the rest $(n-m)$ packets belong to the background traffic. Please refer to Sect. 3.4.2 for the details.

Besides the parameters above, which are common for all tester programs, `siitperf-lat` uses two further ones:

- **delay** before the first frame with timestamp is sent (in seconds, 0-3600)
- **number of frames** with timestamp (1-50,000)

Besides the common ones, siitperf-pdv uses one further parameter:

- **frame timeout** (in milliseconds), if the value of this parameter is higher than zero, then the tester checks this timeout for each frame individually. Please refer to Sect. 3.3.4 for more details.

As for output, if the string “Input Error:” occurs in the standard output, it means that no test was performed due to one or more error in the input (including the command line arguments and the input file, too).

3.3.2 Throughput and Frame Loss Rate Measurements

The `siitperf-tp` program transmits the frames for the required duration and continues receiving until the global timeout time expires after the completion of sending (see more details below). It reports the number of the transmitted frames and the received frames for the active directions (one direction may be missing):

```
Forward frames sent:
Forward frames received:
Reverse frames sent:
Reverse frames received:
```

The pass condition for the throughput test is that the number of received frames equals the number of sent frames for the active directions. Our Tester simply reports these numbers, the pass or failure of the test as well as the actions to be taken must be decided by the shell script, which calls the Tester.

The shell script for frame loss rate test calculates the frame loss rate for the active directions using the output of the program.

We note that `siitperf-tp` uses the global timeout parameter only to determine when to stop receiving. This operation complies with the relevant RFCs, as RFC 8219 has taken the throughput test verbatim from RFC 2544, which requires such operation and sets this type of global timeout to 2 seconds in its Sect. 23 about the general trial description. It says that after running a particular test trial, one should “wait for two seconds for any residual frames to be received”. We have challenged this approach and recommended the checking of the timeout individually for each frame in our paper [14]. We have implemented our recommended solution in `siitperf-pdv` as described in Sect. 3.3.4.

3.3.3 Latency Measurements

First, `siitperf-lat` transmits the frames without inserting identifying tags until the specified $delay$ elapses, then for the remaining test time (that is: $duration - delay$) it inserts the required number of identifying tags using uniform time distribution and it continues receiving until the global timeout time expires after the completion of sending. As RFC 8219 requires, it records sending and receiving time of the tagged frames. (More precisely, the actual time right after the sending of the tagged frames finished and the actual time right after the receiving of the tagged frames finished.) It reports the TL (typical latency) and WCL (worst-case latency) values for the active directions using the following strings for their identification:

```
Forward TL:
Forward WCL:
Reverse TL:
Reverse WCL:
```

The above values are displayed in milliseconds.

Frames with identifying tags are also subject to frame loss. The latency of a lost frame is set to the highest possible value, that is: $duration - delay + global\ timeout$.

3.3.4 PDV

The `siitperf-pdv` program transmits the frames with

unique identifiers for the required duration and continues receiving until the global timeout time expires after the completion of sending. It records the sending time and receiving time of all frames. If frame loss occurs, post processing sets the delay value of the lost frame to $duration - delay + global\ timeout$.

In fact, `siitperf-pdv` is a two in one tester, as its behavior during post processing depends on the value of the frame timeout.

If the value of the frame timeout is 0, then it calculates and reports the PDV values (in milliseconds) for the active directions as required by RFC 8219.

If the value of the frame timeout is higher than 0, then no PDV calculation is done, rather the Tester checks the delay for every single frame during post processing, and if the delay of a frame exceeds the specified frame timeout, then the frame is re-qualified as "lost" for the report of the number of received frames. Thus, it can be used as a very precise throughput and frame loss rate measurement program, which complies with our recommendation in [14]. Its price is the performance penalty. We are aware that the handling of timestamps may cause higher memory consumption and some extra CPU load (similarly to the usage of individual identifiers for every single frame). Therefore, `siitperf-pdv`, is expected to perform up to lower rates than `siitperf-tp` on the same hardware.

3.4 Measurement Traffic

In this section, we examine how the measurement traffic required by RFC 8219 can be provided. These considerations are essential for the design of the traffic generation of the Tester.

3.4.1 Traffic for Stateless NAT64 Translation

Figure 2 shows a test and traffic setup for stateless NAT64 measurements. IPv6 is used on the left side of the Tester and of the DUT, which is actually a stateless NAT64 gateway, and IPv4 is used on their right sides.

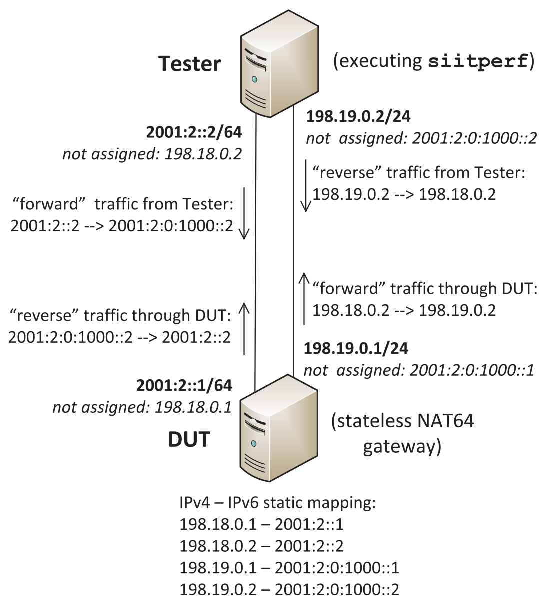

Fig. 2 Traffic for benchmarking stateless NAT64 gateways.

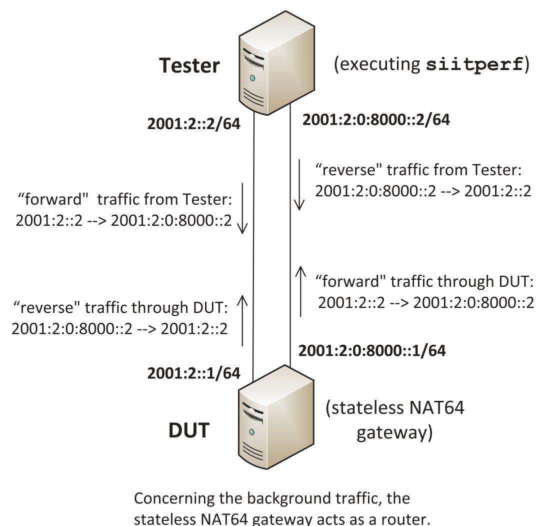

Fig. 3 Background traffic for benchmarking stateless NAT64 gateways.

Now, let us examine how the nodes from one address family (IPv4 or IPv6) can be identified in the other domain (IPv6 or IPv4). When stateful NAT64 is used, IPv4 nodes are identified in the IPv6 network by using IPv4-embedded IPv6 addresses. For stateless NAT64, explicit address mapping can also be used, and this is what we have chosen now. Please refer to the static mapping table of the DUT at the bottom of Fig. 2. Thus, in the forward (left to right) direction, the Tester sends an IPv6 packet with its own IPv6 address 2001:2::2, as source address and the destination address will be 2001:2:0:1000::2, which is mapped to 198.19.0.2.

Let us consider what IPv4 addresses should be used after the stateless NAT64 translation. The destination address will be simply 198.19.0.2. Unlike in the case of stateful NAT64, when the NAT64 gateway uses its own IPv4 address as source address (many to one mapping), the stateless NAT64 gateway uses one to one mapping. The mapping is described by the before mentioned static mapping rules. Thus, in our case, 2001:2::2 is mapped to 198.18.0.2.

In the reverse direction (traffic from right to left), the source and destination IP addresses are simply swapped in the "reverse traffic" compared to the "forward traffic".

3.4.2 Background Traffic

RFC 8219 requires that the tests be performed using different proportions of the traffic to be translated (we call it foreground traffic) and some background traffic, which is not translated, but only routed. The background traffic is native IPv6 traffic. To be able to implement background traffic, we need to assign IPv6 addresses to the right side ports of the Tester and of the DUT. Figure 3 shows the background

traffic.

The Tester must be able to provide both kinds of traffics simultaneously.

RFC 8219 recommends various test cases using different proportions of foreground and background traffic. Please refer to Sect. 3.6.2 for the details of the required proportions and how it is implemented.

3.5 Further Design Considerations

3.5.1 Generalization

So far, we have considered, what is required to satisfy the requirements of RFC 8219. Following the requirements above as our design specifications would result in an asymmetric design: for example, the left side sender would send IPv6 traffic as foreground traffic, and the right side sender would send IPv4 traffic as foreground traffic. Writing two similar but different sender functions would not be a very efficient solution regarding coding efforts. Therefore, we decided to design only one general sending function, which can be parametrized to be able to perform both as left side and as right side sender function. The same considerations apply to the receiver functions, too.

3.5.2 Support for Legacy Tests

We wanted to be able to calibrate our test program in the way that we use it for RFC 2544/RFC 5180 measurements and benchmark the same DUT with both our test program and a legacy RFC 2544/RFC 5180 Tester. Setting 100% background traffic results in pure IPv6 traffic, but we also wanted to be able to provide bidirectional pure IPv4 traffic. Therefore, we decided to assign IPv4 addresses to both sides of the Tester and of the DUT. To support all possible frame sizes in the IPv4 only tests, it was necessary to extend the IPv6 frame size range from 84-1518 to 84-1538, which means 64-1518 for IPv4 frame sizes.

It also means that it is the responsibility of the user to control what traffic should be actually used during a given test. Out test program provides a handy way for it in the configuration file (see in Sect. 3.5.3).

3.5.3 Parameters Specified in the Configuration File

The Tester requires a high number of parameters that do not change during consecutive executions. Therefore, they are placed into the configuration file. We have designed an easy to use orthogonal format, which we demonstrate by the following example specifying the same test setup used in Fig. 2 and Fig. 3.

```
IP-L-Vers 6 # Left Sender foreground IP version
IP-R-Vers 4 # Right Sender foreground IP version

IPv6-L-Real 2001:2::2
IPv6-L-Virt :: # currently not used
IPv6-R-Real 2001:2::0:8000::2
IPv6-R-Virt 2001:2:0:1000::2
IPv4-L-Real 0.0.0.0 # currently unused
IPv4-L-Virt 192.18.0.2
IPv4-R-Real 192.19.0.2
IPv4-R-Virt 0.0.0.0 # currently unused

MAC-L-Tester a0:36:9f:c5:fa:1c
MAC-R-Tester a0:36:9f:c5:fa:1e
MAC-L-DUT a0:36:9f:c5:e6:58
MAC-R-DUT a0:36:9f:c5:e6:5a

Forward 1 # Left to Right direction is active
Reverse 1 # Right to Left direction is active
```

The first two lines control the IP versions of the foreground traffic. Any combination of 4 and 6 is acceptable, but if the IP versions of both sides are 6, then there is no difference between the foreground traffic and the background traffic.

The next eight lines can be used to specify IPv4 or IPv6 addresses of the left (L) or right (R) side interfaces of the Tester. “Real” always specifies an address that is actually assigned to the Tester, whereas “Virt” stands for “virtual”, and it specifies an address, which is not assigned to the Tester, but represents an IP address from the other IP domain. For example, when the Tester benchmarks the NAT64 gateway shown in Fig. 2, its left side interface sends out an IPv6 packet with source address 2001:2::2 and destination address 2001:2:0:1000::2. (The packet is translated into an IPv4 packet by the NAT64 gateway, and then its source address and destination address will be 198.18.0.2 and 198.19.0.2, respectively.) Similarly, its right side interface sends out an IPv4 packet with source address 192.19.0.2 and with destination address 192.18.0.2. The background traffic uses the left side and right side “Real” IPv6 addresses as shown in Fig. 3. We note that three of the eight parameters are currently not needed and thus not used, but they would be needed if the IPv6 and IPv4 sides were interchanged (and then other three parameters would be unused).

The next four parameters specify the MAC addresses of the Tester and of the DUT.

The last two lines specify the active directions. (At least one of the directions has to be active, otherwise an “Input Error:” error message will be generated.)

We note that the receiver function was designed to be resilient: it can recognize the IP version from the Type field of the Ethernet frame and then handle the rest of the frame accordingly.

Although RFC 2544 requires to use fixed source and destination IP addresses first, and then 256 destination networks, we decided to let the number of the networks on left and right side to be set independently to any value from 1 to 256 to support experimentation (to be able to examine, how their number influences performance). The settings apply for both background and foreground traffic.

```
Num-L-Nets 1 # Number of Left side networks
Num-R-Nets 1 # Number of Right side networks
```

In the case of IPv4 addresses, the program counts the networks using the 8 bits from 16 to 23, like 198.18.*0*.1, 198.18.*1*.1, · · · , 198.18.*255*.1. In the case of IPv6 addresses, bits from 56 to 63 are used like 2001:2:0:*0*::1, 2001:2:0:*1*::1, · · · , 2001:2:0:*ff*::1.

The DPDK environment also needs to identify the CPU cores to be used. Please see the following example.

```
CPU-L-Send 2 # Left Sender
CPU-R-Recv 4 # Right Receiver
CPU-R-Send 6 # Right Sender
CPU-L-Recv 8 # Left Receiver
```

The specification of the number of memory channels is optional. If it is not specified, then the program sets it to 1. It can be specified by the following line:

```
MEM-Channels 2 # Number of Memory Channels
```

Finally, there is a kind of convenience setting:

```
Promisc 0 # use promiscuous mode if !0
```

This setting puts the NICs into promiscuous mode. It was used for testing, and it was kept for the convenient self-test of the tester (the user does not have to set the correct MAC addresses).

Throughout the configuration file, missing critical specifications, which would result in program crash, will result an “Input Error:” error message.

A “#” sign means that the rest of the line is to be ignored. (Empty lines are also ignored.)

3.6 Implementation Details

3.6.1 Time Handling

We have chosen TSC (Time Stamp Counter) for time measurement, because it is both high precision and computationally inexpensive. It is a 64-bit register, which is increased with the CPU clock and it can be read by a single CPU instruction, `RDTSC` [15].

The TSC of the logical cores (called “lcore”-s in DPDK terminology) of the same CPU is the same, but synchronization is not guaranteed among the TSCs of cores belonging to different physical CPUs. We expect that the four CPU cores used for the execution of the four threads of the program and the main core, on which the program is started, belong to the same physical CPU[†], and thus the local times of the four threads and the main program are synchronized.

Important warning: the user of the program is advised to check the `constant_tsc` flag in the output of `cat /proc/cpuinfo` command, otherwise `siitperf` may not work correctly.

All the input parameters specified in seconds or milliseconds are converted to TSC and all time related quantities are handled inside in TSC. For output, TSC is converted to seconds or milliseconds.

[†]It is the responsibility of the user to specify the cores for the sending and receiving threads so that this condition be satisfied, `siitperf` does not verify it.

Table 1 How different traffic proportions can be set by using n and m.

RFC 8219	foreground traffic	background traffic	n	m
i)	100%	0%	2	2
ii)	90%	10%	10	9
(missing)	75%	25%	4	3
iii)	50%	50%	2	1
(missing)	25%	75%	4	1
iv)	10%	90%	10	1
(missing)	0%	100%	2	0

3.6.2 Specification of Traffic Proportion

RFC 8219 recommends four test cases using different proportions of foreground and background traffic. The test cases could be simply identified e.g. by their ordinal numbers. However, we would like to enable the user to test any other proportions, too. Moreover, it is also important for us to implement this feature in a computationally cost efficient way and to interleave the foreground and background frames well enough. Therefore, we have chosen the same solution, which we originally designed for specifying the proportion of the domain names to be repeated, when we enabled `dns64perf++` for measuring the efficiency of caching of DNS64 servers [16]. Let N denote the ordinal number of the current packet, let n and m be relative prime numbers. The current packet belongs to the foreground traffic, if and only if:

$$N \,\%\, n < m \qquad (2)$$

(Otherwise, the current packet belongs to background traffic.)

Table 1 shows how different traffic proportions can be set by using n and m. Please refer to Sect. 6.1 of [16] for the advantages of this solution.

We note that 100% background traffic means that the NAT64 gateway is used as an IPv6 router.

3.6.3 Frame Format for Test Frames

We have followed the frame format for test frames defined in Appendix C.2.6.4 of RFC 2544. However, the value of the “identifying tags” to be used for marking at least 500 frames has not been specified in any of RFC 2544, RFC 5180 and RFC 8219.

To be able to distinguish our test frames from any other frames, which might appear in the test network, we use the 64-bit integer, which is encoded using the same 8 octets as the ASCII codes of the string “IDENTIFY”. It is placed at the beginning of the data field of the UDP datagram. When `siitperf-lat` tags a frame for latency measurement, then another 64-bit integer is used, which reflects “Identify”, and the next 16 bits contain the serial number starting from 0. All other frames, as well as all the frames generated by `siitperf-tp` are not numbered to speed up testing. However, the third program, `siitperf-pdv`, numbers all its test

```
// Main sending cycle
for ( sent_frames = 0; sent_frames < frames_to_send; sent_frames++ ){
  while ( rte_rdtsc() < start_tsc+sent_frames*hz/frame_rate ); // Beware: an "empty" loop!
  if ( sent_frames % n  < m )
    while ( !rte_eth_tx_burst(eth_id, 0, &fg_pkt_mbuf, 1) ); // send foreground frame
  else
    while ( !rte_eth_tx_burst(eth_id, 0, &bg_pkt_mbuf, 1) ); // send background frame
} // this is the end of the sending cycle
```

Fig. 4 Code fragment from the single flow sender of `siitperf-tp`.

frames using 64-bit integers.

The unused space of the UDP data is always filled up by increasing octets starting from 0 and repeated if required by the frame sizes (as required by Appendix C.2.6.4 of RFC 2544).

3.6.4 Object Oriented Design and Its Limitations

The C++ language was chosen to support code reuse, because the operation of the three test programs is fairly similar, but there are some deviations. We wanted to avoid writing three very similar programs, which would make our source code hard to maintain, therefore we used an object oriented design. The `Throughput` class served as the base class for the `Delay` and the `Pdv` classes. We wanted to implement the most important functionalities as member functions, but we were able to do it only partially, due to a limitation of DPDK. This limitation is that the `rte_eal_remote_launch()` function, which is used to start the sender and receiver functions on the appropriate cores, does not allow execution of non-static member functions.

As for the details, the `readConfigFile()` function, which reads the parameters from the `siitperf.conf` configuration file was defined in the base class, and it was not redefined in the derived classes. Although the `readCmdLine()` function, which reads the command line parameters, was redefined in the derived classes, but they call the `readCmdLine()` function of the base class, which does the lion's share of the work, and only a few further parameters are needed to be read. The implementation of the `init()` function, which initializes the DPDK EAL (Environment Abstraction Layer) as well as the hardware (the network interfaces), was even more successful: it is not redefined in the derived classes, and works properly due to using a virtual member function `senderPoolSize()` for the calculation of the appropriate sizes of the packet pools of the sender functions.

Unfortunately, the sender and receiver functions, which are not member functions, as well as the `measure()` member function, which starts them, are different for all three classes.

The `rte_eal_remote_launch()` function uses a `void *arg` pointer for its arguments, which are packed into an appropriate structure. We used classes and inheritance to reduce the programming work needed to pack the parameters.

3.6.5 Reuse of Test Frames

In order to increase the maximum achievable frame rate of the Tester, we were striving to reuse a few number of pre-generated test frames.

As for the single flow throughput test, each sender uses only two frames: one for the foreground traffic and another one for the background traffic. As for the multi flow throughput test, foreground and background frames are pre-generated for each destination network and are stored in two arrays: they are randomly chosen for sending during testing.

As for latency measurements, all the tagged frames are also pre-generated and stored in an array. It is also predetermined concerning each tagged frame, if the given frame belongs to the foreground or the background traffic, as well as its destination network, when multi flow test is performed. All the remaining non-tagged frames are handled in the same way as with the throughput test.

As for PDV measurements, our original plan was to use the same number of pre-generated test frames as with the throughput tests, and update them concerning the unique 64-bit serial number and the UDP checksum. However it turned out, that `rte_eth_tx_burst()` function reports the frames as sent, when they are still in the transmit buffer. (We have experienced that no frame 0 arrived, but two frames arrived with the highest serial number.) Therefore, we use N copies of each frame and an index from 0 to $N-1$ is used to select the actual one (for updating and sending). Our measurements using a *self-test* setup (it means that the Tester is looped back, please refer to Sect. 4.2 for details) shown some frame loss even with $N = 20$, thus we set N to 40, which completely eliminated frame loss.

3.6.6 Main Sending Cycle and Its Accuracy

Both the sending and the receiving functions were designed to be as fast (and simple) as possible. Figure 4 shows the main sending cycle of the throughput tester used for single flow testing. It waits until the sending time of the current frame arrives, then it makes a decision whether a foreground or background frame is to be sent, and (re)tries sending the frame, until DPDK reports that it was sent.

We note that using the current algorithm, some of the frames may be sent late, and it definitely occurs, when the frame rate is close to the limits of the hardware, because some mechanisms of the contemporary CPUs, like caching

or branch prediction, ensure their maximum speed only after the first (or first few) steps. Therefore, there is no guarantee for the minimum inter-frame time, and in the worst case, a few frames may be sent out close to full line rate (called back-to-back frames in RFC 2544 terminology).

The elapsed time during sending is checked and printed out as an "Info:" message after the sending cycle. According to current settings[†], 0.001% extra time is allowed to tolerate some very small random delay (e.g. due to an interrupt or anything else) even during the sending of the latest few frames. When the tolerated delay is exceeded, the sender exits with an error message, stating that the test is invalid.

3.6.7 Choice of Random Number Generator

We have chosen the 64-bit Mersenne Twister random number generator (`std::mt19937_64`) on the basis of the results of Oscar David Arbeláez [17].

3.6.8 Correction of Negative Delay to Zero

As the delay of a frame is measured in the way that the current time is stored *after* its sending (and not before that), it might happen that an interrupt occurs *after* sending out the frame and *before* getting the current TSC by `rte_rdtsc()`. The processing time of the interrupt may be longer than the actual one-way delay of the frame. Thus, in this case the measured delay of the frame might be a negative number. It may more easily happen in a *self-test* setup, when the actual delay is very short. This phenomenon causes no problem, when latency tests are done, because only the typical latency (TL) and the worst-case latency (WCL) values are reported. However, the PDV measurement is more sensitive to this phenomenon, because it uses the minimum one-way delay for calculating the final result. PDV is defined by (3), where $Dmin$ is the minimum of the measured one-way delay values, and $D99.9^{th}Perc$ is their 99.9th percentile.

$$PDV = D99.9^{th}Perc - Dmin \tag{3}$$

We have mitigated the problem by correcting "negative" delays to 0. If such correction happens, the Tester prints out the number of corrections in a "Debug:" message at the end of post processing.

We note that this mitigation is not perfect, as it can handle only a negative delay value, and it can not help if the delay is only decreased somewhat, but remains positive.

The other possible case, when an interrupt falsifies the receiving timestamp, which may increase the measured delay. Thus, it can influence the final result through the 99.9th percentile (if it is frequent enough).

The good news is that this rare phenomenon always *increases* the PDV, thus one can be sure that the real value of PDV is surely not higher than the measurement result produced by `siitperf-pdv`.

[†]Please refer to the definition of `TOLERANCE` in `defines.h` as 1.00001.

4. Functional and Performance Tests

The aim of this section is to demonstrate the operation of siitperf and to make an initial performance assessment.

Measurements were carried out using the resources of the NICT StarBED, Japan. All used computers were Dell PowerEdge R430 servers with two 2.1GHz Intel Xeon E5-2683 v4 CPUs having 16 cores each, 384GB 2400MHz DDR4 RAM and Intel 10G dual port X540 network adapters.

Debian Linux 9.9 operating system with 4.9.0-8-amd64 kernel was used, and the DPDK version was 16.11.9-1+deb9u1.

Hyper-threading was switched off on all servers. The CPU clock frequency of the computers could vary from 1.2GHz to 3GHz, but power budget limited it to 2.6GHz, when all cores were loaded. Using the `cpufrequtils` package, the CPU frequency scaling governor was set to "performance" in all servers.

All three tester programs were compiled with g++ 6.3.0 using the -O3 flag.

4.1 Functional Tests

On the basis of our previous experience in [14], we have chosen the Jool 4.0.1 [18] SIIT implementation for testing.

For the functional tests, nodes p094 and p095 were used as Tester and DUT, respectively. Their 10Gbps Ethernet interfaces were interconnected by direct cabling.

As siitperf is currently not able to reply to ARP or ND requests, address resolution was done by manual addition of static entries.

The network interfaces and Jool was set up according to Fig. 2 and Fig. 3 for the single flow tests. As for the multi-flow tests, Jool was set up by the script shown in Fig. 5. The static ARP and Neighbor Table entries as well as the IP addresses for all the networks were set by scripts, too.

The content of the `siitperf.conf` file was shown in Sect. 3.5.3 as an example.

We have performed only a few tests as samples, and we note that performing all possible tests would have required a lot of time and the analysis of the results could be a subject of a complete paper.

4.1.1 Throughput Tests

The measurements were executed 20 times, then median, 1st and 99th percentiles were calculated. To reflect the consistent or scattered nature of the results, we have also calculated a further quantity, dispersion (Disp):

$$Disp = \frac{99^{th}\,percentile - 1^{st}\,percentile}{median} * 100\% \tag{4}$$

The throughput results of Jool using bidirectional traffic are shown in Table 2. We note that the results are to be interpreted that the same rates were used in both directions, thus the cumulative number of frames per second forwarded

```
/sbin/modprobe jool_siit
jool_siit instance add "benchmarking" --iptables
for (( i=0; i<256; i++ ))
do
  H=$(printf "%.2x" $i)
  jool_siit -i "benchmarking" eamt add 2001:2:0:$H::/120 198.18.$i.0/24
  jool_siit -i "benchmarking" eamt add 2001:2:0:10$H::/120 198.19.$i.0/24
done
jool_siit -i "benchmarking" eamt display
ip6tables -t mangle -A PREROUTING -s 2001:2::/120 -d 2001:2:0:1000::/56 -j JOOL_SIIT --instance "benchmarking"
iptables -t mangle -A PREROUTING -s 198.19.0.0/24 -d 198.18.0.0/16 -j JOOL_SIIT --instance "benchmarking"
```

Fig. 5 Bash shell script for setting up Jool with 256 networks for multi flow tests.

Table 2 Throughput of Jool using bidirectional traffic.

IPv6 frame size (bytes)	84	1518	84	1518
num. destination nets	1	1	256	256
median (fps)	426,576	415,741	898,559	812,784
1st percentile (fps)	412,499	399,999	896,653	812,780
99th percentile (fps)	428,138	417,191	900,004	812,787
dispersion (%)	3.67	4.14	0.37	0.00

by the DUT was the double of what is shown in the table.

As we expected, the change of the frame size made no significant difference in the case of the single flow tests (426,576 fps vs. 415,741 fps), this results complies with our earlier experience [14]. As for the multi flow test, the difference is higher (898,559 fps vs. 812,784 fps), because the frame rate of the 1518 bytes long frames achieved (and it was limited by) the maximum frame rate of the 10Gbps Ethernet.

Considering the results of tests with 84 bytes frames, the multi flow throughput (898,559 fps) is somewhat higher than the double of the single flow throughput (426,576 fps). We note that this growth is the resultant of two effects, which are working against each other. On the one hand, the handling of 256 networks requires more computation than the handling of a single network. On the other hand, the different destination addresses distributed the load nearly evenly among (the half of the) the CPU cores. As Jool works in kernel space, we could observe only the software interrupts using the `top` command. Cores 0-15 were nearly fully utilized by the software interrupts, and cores 16-31 were shown to be idle. The detailed analysis of the situation is beyond the limits of this paper.

4.1.2 Frame Loss Rate Test

Although RFC 2544 requires that frame loss rate tests should be performed for different frame rates starting from the maximum frame rate of the media, decreased in not more than 10% steps until two consecutive measurements show zero frame loss, we have shown in [14] that there was not much point in using such high rates, when the throughput was rather far from the maximum frame rate of the media. Following the same approach, we performed frame loss rate test for those frame rates, which we believed to provide meaningful results. Figure 6 shows the frame loss rate of Jool using bidirectional, single flow traffic, 84 bytes and 1518 bytes frame sizes for frames containing IPv6 datagrams (and 64 bytes and 1498 bytes frame sizes for IPv4). The color bars show the median values of the 20 measurements, whereas the error bars show the 1st and 99th percentiles.

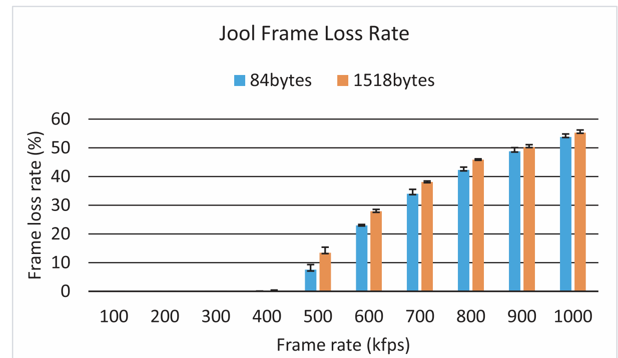

Fig. 6 Frame loss rate of Jool using bidirectional single flow traffic.

Table 3 Latency of Jool using 84 bytes IPv6 frame size, single flow, bidirectional traffic, 426,576 fps frame rate.

	Fwd TL	Fwd WCL	Rev TL	Rev WCL
median (ms)	0.027	0.058	0.020	0.152
1st perc. (ms)	0.027	0.056	0.019	0.140
99th perc. (ms)	0.028	0.761	0.020	62,000

4.1.3 Latency Measurements

As our aim is not the investigation of Jool, but the demonstration of the operation of `siitperf-lat`, we have performed the latency measurements only with 84 byte IPv6 frame size, using bidirectional single flow traffic at 426,576 fps frame rate determined by the throughput test.

As for latency measurements, the duration of the tests was 120 seconds and 50,000 identifying tags were inserted after 60 seconds using uniform time distribution. The test was performed 20 times and the results are shown in Table 3.

There is a visible asymmetry between latency values of the IPv6 to IPv4 translation in the Forward direction and that of the IPv4 to IPv6 translation in the Reverse direction. The 62,000 ms 99th the percentile value of the Reverse direction worst case latency is the result of the loss of several tagged frames and the handling of the situation described in

Table 4 PDV of Jool using 84 bytes IPv6 frame size, single flow, bidirectional traffic, 426,576 fps frame rate.

	Forward PDV	Reverse PDV
median (ms)	0.058	2.727
1st percentile (ms)	0.057	0.159
99th percentile (ms)	0.068	10.044

Table 5 Maximum frame rate achieved by `siitperf-tp/pdv`, using multi flow test, bidirectional traffic, 84 bytes IPv6 frame size and 10 ms frame timeout for `siitperf-pdv`.

	`siitperf-tp`	`siitperf-pdv`
median (fps)	7,205,039	6,430,908
1st percentile (fps)	7,174,546	6,430,651
99th percentile (fps)	7,241,211	6,430,928
dispersion (%)	0.93	0.00

Sect. 3.3.3.

4.1.4 PDV Measurements

As for PDV measurements, the duration of the tests was 60s, the test was performed 20 times and the results are shown in Table 4.

4.2 Performance Estimation

For the self-test of the Tester, the p096 server was used and its two 10Gbps network interfaces were interconnected by a direct cable. We disclose the result of the multi flow tests, where `siitperf` had to generate random numbers for every single frame. Besides `siitperf-tp`, we have also tested `siitperf-pdv` using 10 ms frame timeout. Their results are shown in Table 5. They are definitely more than enough for benchmarking SIIT implementations, like Jool.

5. Plans for Future Research and Development

5.1 Comprehensive Testing

5.1.1 Checking the Accuracy of the Sending Algorithm

As we mentioned in Sect. 3.6.6, currently there is no guarantee for the minimum inter-frame time. As `siitperf-pdv` stores all sending and receiving timestamps, the uniformity of the inter-frame times can be easily examined.

As for how this inaccuracy may effect measurement results, please refer to Sect. 5.1.2.

As for possible mitigation, we have experimented with using e.g. 90% of the calculated inter-frame time as the allowed minimum inter-frame time, and found that this method significantly decreased the achievable maximum frame rate. Therefore, we plan to use this method only in the case, if it proves to be necessary.

5.1.2 Validation of Siitperf with a Standard Tester

We plan to validate `siitperf` by measuring the IPv4 routing performance of the Linux kernel with it and also with a legacy commercial RFC 2544 compliant Tester and then comparing their results.

If the results of `siitperf` will be lower than that of the commercial tester, it will probably indicate that the non-uniformity of the inter-frame times influences the results.

5.1.3 Complete Benchmarking of Jool

We plan to perform all possible benchmarking measurements with the Jool SIIT implementation.

By doing so, our primary aim is to test all the functionalities of `siitperf` thoroughly. As a byproduct, we also provide network operators with ready to use performance data of Jool.

5.2 Adding Further Functionalities

Our current aim was to create a working Tester as soon as possible. Later we plan to add further functionalities, including the following ones.

5.2.1 Support for Overload Scalability Tests

Section 10 of RFC 8219 explains the need for testing with and increasing number of network flows and to observe the resulting performance degradation. Currently, `siitperf supports` testing with up to 256 destination networks. The support for significantly higher number of network flows would need to use further bits than the currently used 8 bits (from 16 to 23) of the IPv4 addresses. Potential candidates are bits from 24 to 29 (as bits 30 and 31 are needed to express the proper ending of the IPv4 addresses, which are currently ".1" and ".2" for the DUT and for the Tester, respectively. If the further 6 bits prove not to be enough, then larger IPv4 address ranges are needed than those reserved for benchmarking (198.18.0.0/24 and 198.19.0.0/24).

5.2.2 Implementation of ARP and NDP

We plan to make the usage of `siitperf` more comfortable by adding the ability of replying ARP requests (for IPv4) and Neighbor Solicitation (for IPv6), thus eliminate the need for manual settings of the these mappings in the DUT.

5.2.3 Implementation of IPDV Measurements

We also plan to add this optional functionality. The source code for PDV can be easily extended to support IPDV measurements.

5.3 Performance Optimization

5.3.1 Parallel Post-Processing in PDV Measurements

Currently, the timestamps of the PDV measurements are post processed after the measurements by the main core, and if a bidirectional test is performed, then they are processed for

the Forward and Reverse directions sequentially. They could be executed in parallel, by two cores. Its price is to write the code that packs all necessary information to an appropriate structure for the `rte_eal_remote_launch()` function.

5.3.2 Using Multiple Senders

Currently, the maximum frame rate is limited by the performance of the senders. (Whereas the senders send the frames one by one, the receivers receive multiple frames by a single call of the `rte_eth_rx_burst()` function.)

The sending performance could be increased by using multiple senders. However, this solution has a practical problem. Whereas theoretically the two frame flows can be perfectly interleaved, in practice, the timing inaccuracy could result in improper inter-frames times, which could not even be corrected using the method mentioned in Sect. 5.1.1.

5.4 Developing the Benchmarking Methodology

5.4.1 Global Timeout vs. Frame Timeout

In [14], we aimed to check the viability of the RFC 8219 benchmarking measurements. We have pointed out different possible issues including the problem that throughput and frame loss rate measurements use a single global timeout, and we recommended the checking of the timeout individually for each frame. Now, this solution can be implemented by using `siitperf-pdv` for throughput and frame loss rate measurements. We plan to check, if the classic RFC 2544 measurement result and the results of our recommended tests are significantly different, and if so, then which one is closer to the users' experience.

5.4.2 Methodology for Benchmarking Stateful NAT64

RFC 8219 recommends the same measurements for stateless and stateful NAT64, plus some extra tests for the latter one. Namely, it recommends the measurement of "concurrent TCP connection capacity" and "maximum TCP connection establishment rate" in its Sect. 8.

Whereas we believe that they are important and meaningful, we surmise that further tests are needed. We are especially concerned, how the number of connections (that is the size of the state table) influences the performance. Many people think that stateful solutions do not scale up well. Others think that hashing reduces the lookup cost in the state table efficiently. In Sect. 3.2 of our Internet Draft [19], we promised to address this question by benchmarking measurements.

Our results may also lead to the amendment of RFC 8219 with further tests.

There is one further issue that prevents `siitperf` from being (fully) usable for benchmarking stateful NAT64 gateways. Testing can be performed in the IPv6 to IPv4 direction, but it is impossible in the IPv4 to IPv6 direction. As discussed in Sect. 3.6.3, `siitperf` uses the frame format for test frames defined in Appendix C.2.6.4 of RFC 2544 including the hard-coded UDP source and destination port numbers. Benchmarking stateful NAT64 gateways will require the sending of some "pilot" frames (one frame per session) in the IPv6 to IPv4 direction, and then learning the port numbers from the received frames on the IPv4 side and using them in the IPv4 to IPv6 direction traffic. The fact that a stateful NAT64 gateway may use multiple public IPv4 addresses for the outgoing IPv4 packets, makes the problem somewhat more complex. Being `siitperf` a free software, anyone can implement this feature, and we also plan to do it in the upcoming years.

6. Conclusions

We conclude that our efforts were successful in creating the world's first standard free software stateless NAT64 benchmarking tool, `siitperf`. Our tests proved that it works correctly and it has high enough performance for benchmarking SIIT implementations. Our future plans include its comprehensive testing, adding further functionalities and its performance optimization. We also plan to use our new Tester for research in benchmarking methodology issues.

Acknowledgments

The development of `siitperf` and the measurements were carried out by remotely using the resources of NICT StarBED, 2-12 Asahidai, Nomi-City, Ishikawa 923-1211, Japan. The author would like to thank Shuuhei Takimoto for the possibility to use StarBED, as well as to Satoru Gonno for his help and advice in StarBED usage related issues.

The author thanks Keiichi Shima, Marius Georgescu, Tamás Budai and Alexandru Moise for their reading and commenting the manuscript.

Péter Bálint, a PhD student at the Széchenyi István University has reported the implementation of a stateless NAT64 tester in 2017 [20]. However, he told that its performance was unsatisfactory. For this reason, he re-implemented the tester using DPDK (and the C programming language) under the supervision of Gábor Lencse on the basis of the design described in an earlier version of this paper. Unfortunately, his program was unusable for measurements, and Gábor Lencse has corrected it to the extent that it could be used for throughput and frame loss rate measurements with single flow, and it was used for [14], but that program has not been publicly released. This is why we can state that `siitperf` is world's first standard free software stateless NAT64 benchmarking tool. On the one hand, `siitperf` is a completely new implementation from scratch in C++ to avoid copyright issues, but, on the other hand, we would like to acknowledge our learning from the C source code of Péter Bálint, especially concerning the DPDK functions used.

References

[1] G. Lencse and Y. Kadobayashi, "Comprehensive survey of IPv6

transition technologies: A subjective classification for security analysis," IEICE Trans. Commun., vol.E102-B, no.10, pp.2021–2035, 2019, DOI: 10.1587/transcom.2018EBR0002.

[2] M. Georgescu, L. Pislaru, and G. Lencse, "Benchmarking methodology for IPv6 transition technologies," IETF RFC 8219, Aug. 2017, DOI: 10.17487/RFC8219.

[3] M. Bagnulo, A. Sullivan, P. Matthews, and I. Beijnum, "DNS64: DNS extensions for network address translation from IPv6 clients to IPv4 servers," IETF RFC 6147, April 2011, DOI: 10.17487/RFC6147.

[4] S. Bradner and J. McQuaid, "Benchmarking methodology for network interconnect devices," RFC 2544, March 1999, DOI: 10.17487/RFC2544.

[5] C. Popoviciu, A. Hamza, G. Van de Velde, and D. Dugatkin, "IPv6 benchmarking methodology for network interconnect devices," RFC 5180, May 2008, DOI: 10.17487/RFC5180.

[6] M. Mawatari, M. Kawashima, and C. Byrne, "464XLAT: Combination of stateful and stateless translation," IETF RFC 6877, April 2013, DOI: 10.17487/RFC6877.

[7] M. Bagnulo, P. Matthews, and I. Beijnum, "Stateful NAT64: Network address and protocol translation from IPv6 clients to IPv4 servers," RFC 6146, April 2011, DOI: 10.17487/RFC6146.

[8] G. Lencse and D. Bakai, "Design and implementation of a test program for benchmarking DNS64 servers," IEICE Trans. Commun., vol.E100-B, no.6, pp.948–954, June 2017, DOI: 10.1587/transcom.2016EBN0007.

[9] G. Lencse, "Siitperf: An RFC 8219 compliant SIIT (stateless NAT64) tester written in C++ using DPDK," source code, https://github.com/lencsegabor/siitperf

[10] A. Morton and B. Claise, "Packet delay variation applicability statement," IETF RFC 5481, March 2009, DOI 10.17487/RFC5481.

[11] G. Lencse and S. Répás, "Performance analysis and comparison of different DNS64 implementations for Linux, OpenBSD and FreeBSD," Proc. IEEE 27th International Conference on Advanced Information Networking and Applications (AINA 2013), Barcelona, Spain, pp.877–884, March 2013, DOI: 10.1109/AINA.2013.80.

[12] G. Lencse and S. Répás, "Performance analysis and comparison of the TAYGA and of the PF NAT64 implementations," Proc. 36th International Conference on Telecommunications and Signal Processing (TSP 2013), Rome, Italy, pp.71–76, July 2013, DOI: 10.1109/TSP.2013.6613894

[13] D. Scholz, "A look at Intel's dataplane development kit," Proc. Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), Munich, Germany, pp.115–122, Aug. 2014, DOI: 10.2313/NET-2014-08-1_15.

[14] G. Lencse and K. Shima, "Performance analysis of SIIT implementations: Testing and improving the methodology," Comput. Commun., vol.156, no.1, pp.54–67, April 2020, DOI: 10.1016/j.comcom.2020.03.034.

[15] Intel, "Intel 64 and IA-32 architectures software developer's manual," Volume 2B: Instruction Set Reference, M-U, Order Number: 253667-060US, September 2016, https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf

[16] G. Lencse, "Enabling dns64perf++ for benchmarking the caching performance of DNS64 servers," J. Comput. Inform. Technol., vol.26, no.1, pp.19–28. July 2018, DOI: 10.20532/cit.2018.1004078.

[17] O.D. Arbeláez, "How competitive are C++ standard random number generators," https://medium.com/@odarbelaeze/how-competitive-are-c-standard-random-number-generators-f3de98d973f0

[18] NIC Mexico, "Jool: SIIT and NAT64," 2019, http://www.jool.mx/en/about.html

[19] G. Lencse, J. Palet Martinez, L. Howard, R. Patterson, and I. Farrer, "Pros and cons of IPv6 transition technologies for IPv4aaS," active Internet Draft, Jan. 2020, https://tools.ietf.org/html/draft-lmhp-v6ops-transition-comparison-04

[20] P. Bálint, "Test software design and implementation for benchmarking of stateless IPv4/IPv6 translation implementations," Proc. 40th International Conference on Telecommunications and Signal Processing (TSP 2017), Barcelona, Spain, pp.74–78, July 2017.

Gábor Lencse received his M.Sc. and Ph.D. degrees in computer science from the Budapest University of Technology and Economics, Budapest, Hungary in 1994 and 2001, respectively. He works for the Department of Telecommunications, Széchenyi István University, Győr, Hungary since 1997. Now, he is a professor. He is also a part time senior research fellow at the Department of Networked Systems and Services, Budapest University of Technology and Economics since 2005. His research interests include the performance analysis IPv6 transition technologies. He is a co-author of RFC 8219.