Checking and Increasing the Accuracy of the Dns64perf++ Measurement Tool for Benchmarking DNS64 Servers

Gábor Lencse and Attila Pivoda

***Abstract*—Our DNS64 benchmarking program, dns64perf++, is the world's first standard DNS64 benchmarking tool, which complies with the requirements of RFC 8219 (Benchmarking methodology for IPv6 transition technologies) including DNS64. The aim of our current effort is to check and ensure its accuracy. In this paper, we disclose our measurement method and results. We have found inaccuracies at higher rates, which were caused by the self-correcting timing algorithm. We have replaced the timing algorithm by a simpler one, which resulted in accurate results at any tested rates. We have also tested the corrected version during real measurements: we compared the quality of the measurements results produced by the original and the corrected version.**

***Keywords*—benchmarking, DNS64, IPv6 transition technology, performance analysis.**

I. Introduction

DNS64 [1] and NAT64 [2] are important IPv6 transition technologies enabling IPv6-only clients to communicate with IPv4-only servers. There are several DNS64 implementations, and their performance is an important factor when network operators have to select from among them. To that end, we have developed a benchmarking methodology for DNS64 servers [3], which is also a part of the relevant RFC on benchmarking methodology for IPv6 transition technologies [4]. The compulsory requirements of the RFC for benchmarking DNS64 servers were satisfied by the `dns64perf++` measurement program [5], which was documented in [6]. Later, the optional feature of testing the efficiency of the caching performance of DNS64 servers was also added [7].

The `dns64perf++` benchmarking tool was successfully used in various measurements, which required only moderate rates, below 35,000qps (queries per second), see [3] and [8] for details. The program can also be used for testing the performance of DNS servers, and we used it for measuring the performances of several different authoritative DNS servers, in order to find out, which one would be the best choice to be used as authoritative DNS server for DNS64 benchmarking tests. When the testing rates were above 50,000qps, we experienced scattered measurement results. (RFC 8219 [4] requires at least 20 tests, which means that the binary search for the highest possible rate, at which the DNS64 server can serve AAAA record requests, should be executed at least 20 times. We experienced significant differences between the results of the 20 tests.) We were looking for the reason of the scattered results, and we have systematically checked the accuracy of `dns64perf++`.

The aim of our current paper is to document the accuracy measurements, analyze their results, patch the bug, and assure the accuracy of `dns64perf++` at high rates.

The remainder of this paper is organized as follows. Section II recalls the operation of the `dns64perf++` program in a nutshell. Section III presents our accuracy measurement method and the results, as well as the analysis of the results and the identification of the cause of the inaccuracies at high rates. Section IV discloses our solution for the problem and the test results of the corrected timing algorithm. Section V considers the limitations of corrected program. Section VI is a case study: both the original and the corrected versions of `dns64perf++` are used for real measurements and the accuracies of their results are compared. Section VII points out the significance of the proper timing algorithm in a larger context. Section VIII gives our conclusions.

II. Operation of Dns64perf++ in a Nutshell

A detailed description of the design and operation of the `dns64perf++` program can be found in our open access paper [6], now we give a short summary[1] of it including only the parts relevant to our topic. Fig. 1 shows the test setup for DNS64 measurements. It contains three devices: the client, the DNS64 server and the authoritative DNS server. When `dns64perf++` is used for benchmarking authoritative DNS servers, then the DNS64 server is removed and the remaining two devices are directly connected to each other. In this case, the authoritative DNS server is configured to serve AAAA records (IPv6 addresses), because `dns64perf++` always requests AAAA records.

The `dns64perf++` program executes in two threads: one of them sends queries for AAAA records of different domain names at a specified rate and the other one receives the answers and decides about every single answer if it is arrived in time (within a given timeout) and if it contains an AAAA record. If both conditions are met then the program qualifies the answer as "valid".

For being able to perform these tasks, the sending thread stores a nanosecond precision timestamp of the sending time

Manuscript received: March 8, 2018; revised: April 10, 2018.

Gábor Lencse and Attila Pivoda are with the Department of Telecommunications, Széchenyi István University, Egyetem tér 1, H-9026 Győr, Hungary.
a) E-mail: lencse@sze.hu
b) E-mail: pivodaattila@gmail.com

[1]Some of the text of this summary is taken verbatim from [6].

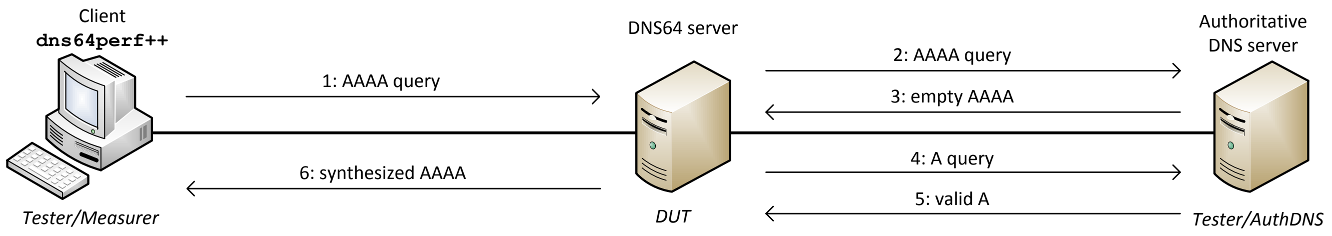

Fig. 1. Test setup for benchmarking DNS64 servers [6].

of each query and, similarly, the receiving thread stores a nanosecond precision timestamp of the receiving time of every single answer.

We have invested a lot of work into the design of the timing algorithm for sending the AAAA record requests. Instead of calculating the waiting time independently for each message, we always considered the remaining time until the end of the testing. We calculated the waiting time before starting to prepare the $(n+1)$-th request as follows:

$$T_W(n+1) = \frac{N*T - (t_B(n) - t_B(0))}{N-n} - T_R(n) \qquad (1)$$

where N is the total number of requests to be sent, T is the required time interval between the consecutive messages (that is, 1/frequency), $t_B(n)$ denotes the timestamp when the preparation of the n-th request started and $T_R(n)$ denotes the time it took to prepare and send the n-th request (n takes the values from 0 to N-1). This way, the timing is self-correcting.

We note that this method guarantees only the "global" accuracy of timing. There may be "local" inaccuracies, and they will surely occur if the request rate is high enough. Modern computer hardware support the efficiency of program execution by several solutions such as caching, branch prediction or prefetching data/instructions. Some high request rates can only be achieved after these solutions provide full benefits (program code and data are loaded into the cache; the branch predictors have already learnt the behavior of the program, etc.). Thus, a given number of requests may be sent somewhat late at the beginning of the test.

We also note that `dns64perf++` is still under development and its latest version has been enabled to use twice n threads (n threads for sending queries and n threads for receiving replies) to achieve higher rates, but this feature is not documented by a research paper yet.

In our current paper, we deal with the original version of `dns64perf++` documented in [6].

III. Measurements, Results and Problem Identification

The fact that `dns64perf++` dumps all its results (including all the nanosecond precision timestamps) in CSV format into the `dns64perf.csv` file, enabled us to test its accuracy without the need for purchasing expensive measurement devices.

We have performed 60s long tests (to comply with the requirements of RFC 8219 [4]) at various speeds from 50,000qps to 250,000qps with the increase of 50,000qps. We focused on the sending timestamps only. To make the huge number of results digestible, we have used a short script to count how many timestamps fall into each 100ms time window from 0s to 60s.

For the repeatability of our measurements, we present the most important parameters of the computer used for testing. It was a Huawei CH140 v3 compute node with Intel Xeon E5-2670 v3 2.30GHz CPUs, 8x 16GB 2133MHz DDR4 SDRAM and Ubuntu 16.04.2 LTS GNU/Linux operating system with 4.4.0-45-generic x86_64 kernel was used.

We note that in this case, `dns64perf++` was not used in a real measurement situation, but rather the timing accuracy of its AAAA record request generation was tested. Section VI contains a case study where `dns64perf++` is used in benchmarking measurements.

The results are shown in Fig. 2. Whereas the result of the 50,000qps test seem to be correct, all the other results are visibly differ from the expected one. The behavior of the self-correcting timing algorithm can be very well observed on the graph, which belongs to the test at 250,000qps. For some reason, which we will soon determine, the curve starts at 24,450queries/100ms, which corresponds to 244,500qps, and the compensation is visibly too low, therefore the algorithm has to compensate too much at the end. However, finally, the required 250,000*60=15,000,000 number of messages were successfully sent in 60 seconds, and thus the program reported that it could send all the messages during the required time.

The explanation of the curve is also very simple. To achieve the required 250,000qps rate, the program should have sent a request at every 4,000ns. The achieved 244,500qps rate corresponds to 4,090ns cycle time. It means that the program spent about 90ns more with each message than it should have spent. We can easily check the validity of this model. Let us check two calculations. (Table I shows all of them.) The cycle time should be 5,000ns at 200,000qps rate, and 5,090ns results in 19,646 queries/100ms, which is very close to what we measured (19,651). The cycle time should be 20,000ns at

TABLE I
Validation of our error model.

Required query rate (qps)	50,000	100,000	150,000	200,000	250,000
Required cycle time (ns)	20,000	10,000	6,667	5,000	4,000
Cycle t. in our model (ns)	20,090	10,090	6,757	5,090	4,090
Computed queries/100ms	4,978	9,911	14,799	19,646	24,450
Counted queries/100ms	4,979	9,913	14,805	19,651	24,450

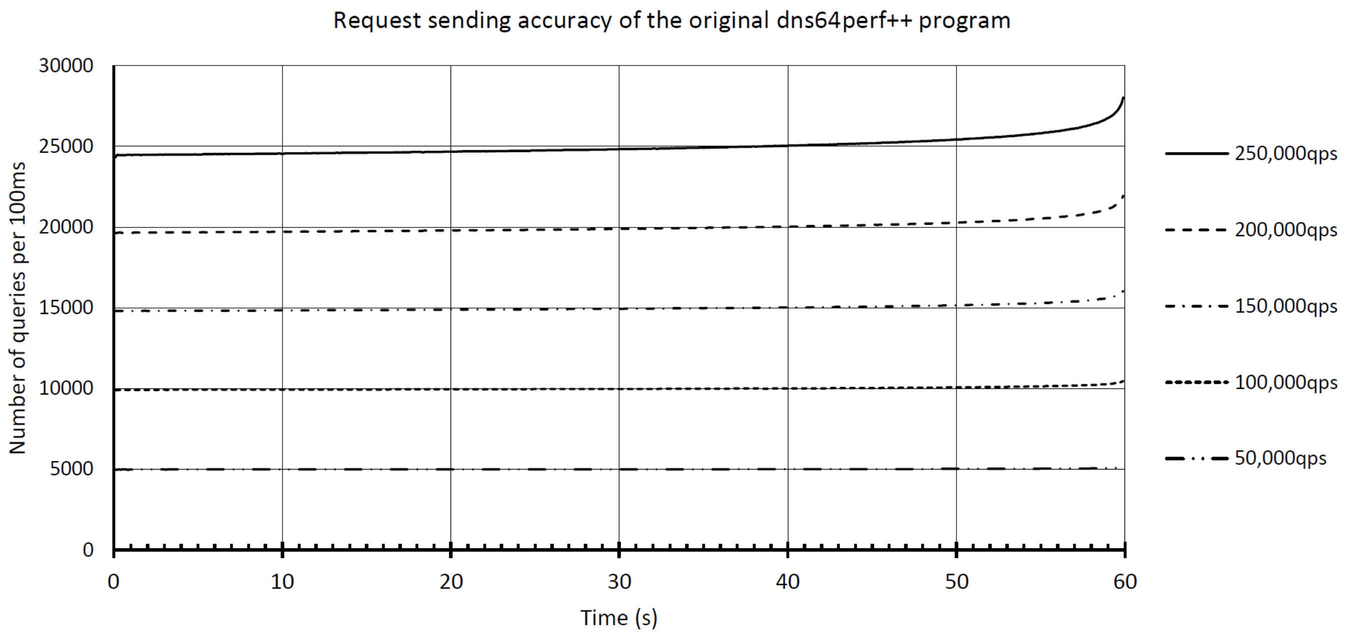

Fig. 2. Number of AAAA record queries sent in a 100ms long interval by the *original* `dns64perf++` program.

50,000qps rate, and 20,090ns results in 4,978queries/100ms, which is very close to what we measured (4,979).

Of course, this behavior is completely unacceptable from a measurement program, but the non-constant rate and especially the steep rise at the end of the measurement interval gave a good explanation for the scattered results of our authoritative DNS server measurements.

IV. Correction of the Timing Algorithm

As for the timing algorithm, it became evident that we should not try to distribute the compensation of the possible latency for the remaining testing time, but we should rather use a simple solution, where we attempt to compensate all the accumulated latency at the current step.

We calculate the waiting time before starting to prepare the $(n+1)$-th request as follows:

$$T_W(n+1) = t_1 + n * T - t_S(n) \tag{2}$$

where t_1 denotes the timestamp when the preparation of the first request started, T is the required time interval between the consecutive messages, and $t_S(n)$ denotes the timestamp when the n-th request was sent. In this way, the timing error will not cumulate.

As for the modification of the source code, we limited the change for a single line of a single file (line 49 of `timer.cpp`, as shown in Fig. 3).

Technical note: the expression "`(n_-n)`" of the source code corresponds to "n" in (2).

We note that after the above change, the calculation of the function execution time in line 42 is used only in debug mode, otherwise it is now calculated unnecessarily, thus one wants to optimize the code, may exchange lines 42 and 43 to execute the calculation within the `#ifdef DEBUG` macro. We did not include this change for simplicity.

Using the corrected timing algorithm, we have performed the same tests as before, and the results are displayed in Fig. 4. They show that the correction was successful, and the accuracy is now ensured at all tested query rates.

V. Discussion of the Accuracy

We would like to emphasize that the accuracy of `dns64perf++` is still limited. It is a software-based generator, which is executed by a modern computer hardware under the Linux operating system and uses socket interface API functions, thus the limitations mentioned in section 5.6 of [6] are still valid.

To fully disclose the accuracy of `dns64perf++`, we have prepared a plot using 10ms wide cells. Fig. 5 shows the result of the same measurements, as Fig. 4, the only difference is that narrower cells were used during the processing of the results. Several spikes can be observed on the graphs of both the 200,000qps and the 100,000qps tests. We have identified the first spike of the 200,000qps graph. Its downwards pointing part belongs to the 2.05s-2.06s time window, where only 1904 requests were sent instead of the required 2000 requests. (Its reason could be e.g. the handling of an interrupt, rescheduling the thread to a different CPU core, or some other thing mentioned in section 5.6 of [6].) The upwards pointing part of the spike belongs to the 2.06s-2.07s time window, where 2097 requests were sent. This example shows that the modified program attempts to compensate for any latency as soon as possible.

As the measurement method requires the usage of 1s timeout [3], we believe that these local inaccuracies, which are visible only in Fig. 5 having 10ms wide cells but are invisible in Fig. 4 having 100ms wide cells, are satisfactorily ironed out during the 1s timeout interval and thus `dns64perf++` may be used. However, we recommend the users of the

The sleep time calculation in the source file `timer.cpp` was replaced as follows:
Original code (lines 42-49):

```
42:  function_execution_time = std::chrono::duration_cast<std::chrono::nanoseconds>
         (std::chrono::high_resolution_clock::now() - before);
43:  #ifdef DEBUG
44:  if (function_execution_time > interval) {
45:                  std::cerr << "Can't keep up!" << std::endl;
46:  }
47:  #endif
48:  --n;
49: sleep_time = interval - function_execution_time; // ONLY THIS LINE WILL BE CHANGED!
```

New code (lines 42-49):

```
42:  function_execution_time = std::chrono::duration_cast<std::chrono::nanoseconds>
         (std::chrono::high_resolution_clock::now() - before);
43:  #ifdef DEBUG
44:  if (function_execution_time > interval) {
45:                  std::cerr << "Can't keep up!" << std::endl;
46:  }
#endif
47:  --n;
49: sleep_time = starttime + (n_-n)*interval_ - std::chrono::high_resolution_clock::now(); // NEW CODE
```

Fig. 3. Modifications to the `dns64perf++` program.

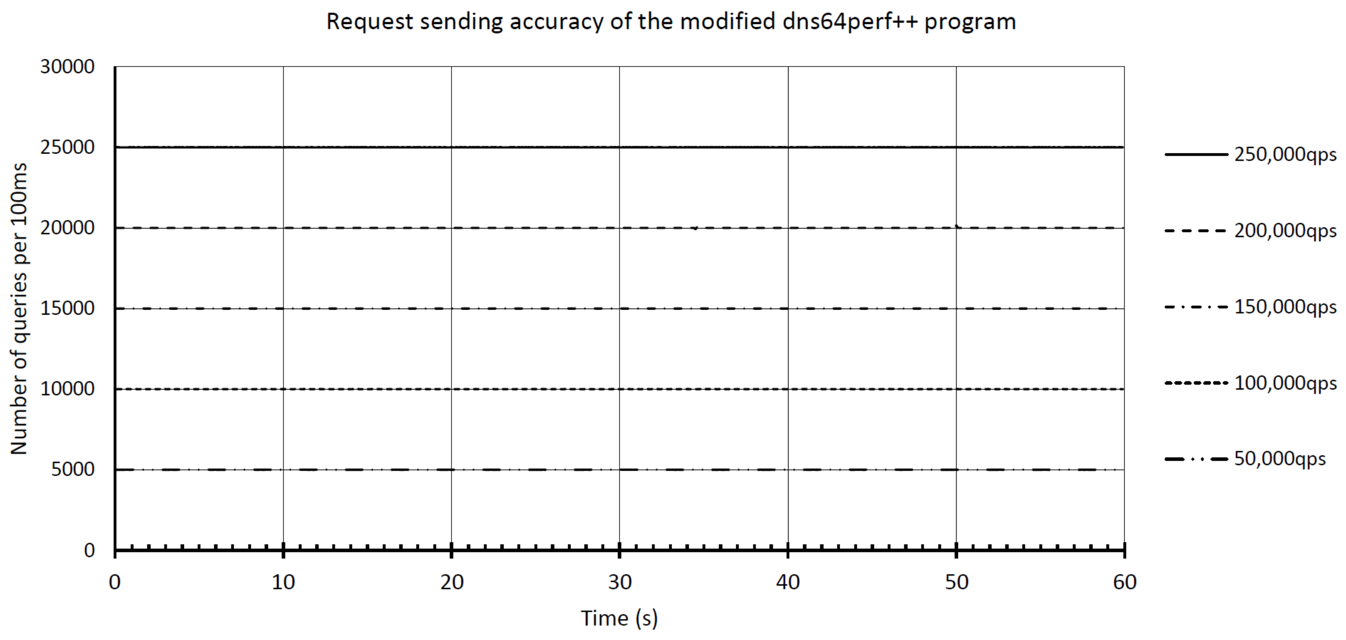

Fig. 4. Number of AAAA record queries sent in a 100ms long interval by the *modified* `dns64perf++` program.

program to check the nanosecond precision timestamps made available by the program in the `dns64perf.csv` file. Thus, by processing this file, the user may decide if the accuracy is acceptable for his/her purposes or not. (In the latter case, the measurement should be invalidated and repeated.)

We also note that higher rates and likely higher accuracy could be reached by using the DPDK (Intel Data Plane Development Kit) [9] instead of the socket interface API.

VI. Comparison during Real Measurements

In this case study, we demonstrate the real life effect of the correction of the program.

As for measurement environment, we used three Dell PowerEdge C6620 servers from the NICT StarBED, Japan. The servers were interconnected by 10G Ethernet direct cable links. The test setup is shown in Fig. 6. The three devices can be identified by their roles, which correspond to that of the three devices in Fig. 1. There is an important difference here: the Measurer and AuthDNS subsystems of the Tester are interconnected by a direct link, which will be used for the so-called “self-test” of the tester (see later).

For the repeatability of our measurements, we briefly summarize the most important parameters of the servers used for our measurements. Each Dell PowerEdge C6620 server contained two Intel Xeon E5-2650 2GHz CPUs, having 8 cores each, 16x8GB 1333MHz DDR3 RAM, two Intel I350 Gigabit Ethernet NICs, two Intel 10G 2P X520 Adapters, with a 10G

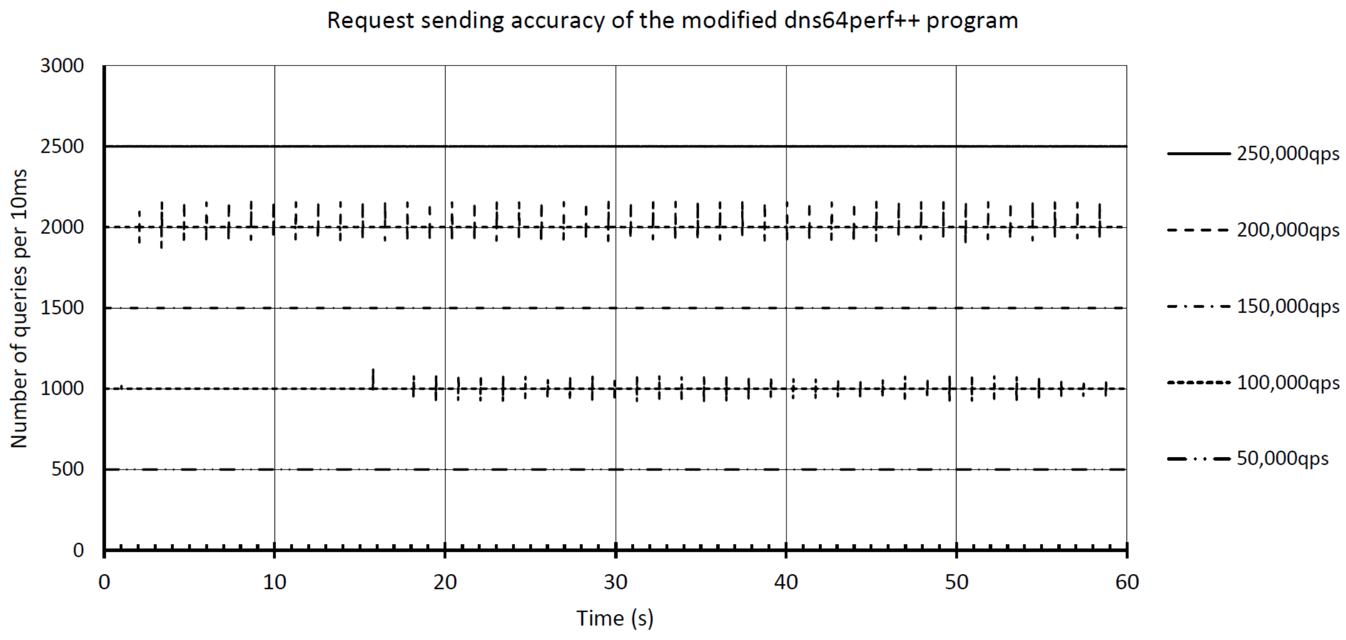

Fig. 5. Number of AAAA record queries sent in a **10ms** long interval by the *modified* `dns64perf++` program.

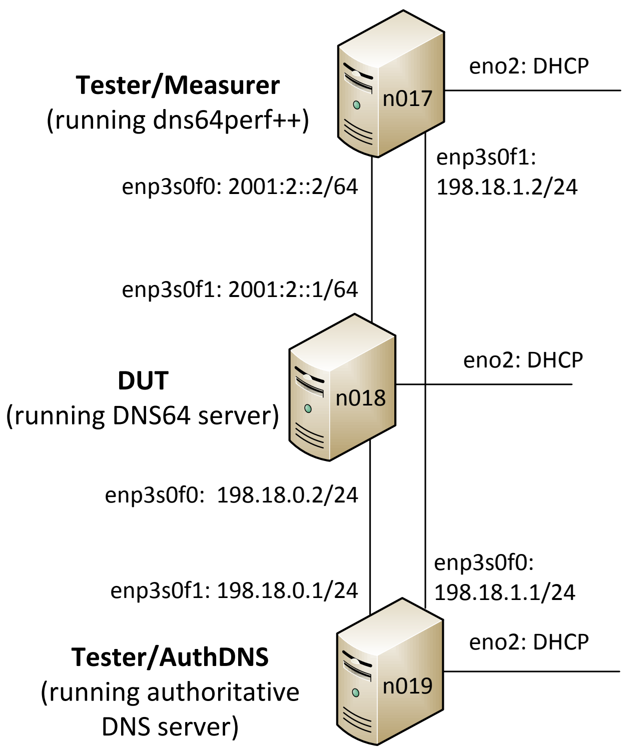

Fig. 6. Test setup for real DNS64 and tests.

interface module in each. Debian GNU/Linux 9.2 operating system with kernel version 4.9.0-4-amd64 was installed to all the computers.

According to the recommendations of [3], we have switched off hyper-threading on all three computers, and set the clock frequency of nodes n018 and n019 to fixed 2GHz to avoid scattered results. (Turbo mode was enabled on node n017 to achieve the highest possible performance.)

The qualities of the results of the old and of the modified program were compared in two types of tests, namely DNS64 tests and self-tests of the Tester [4].

For the DNS64 tests, mtd64-ng 1.1.0 [8] and YADIFA 2.2.3-6237 were used as DNS64 server and authoritative DNS server, respectively. On the basis of our experience with DNS64 testing [10], we limited the number of CPU cores at the DUT to 8, and the number of working threads of mtd64-ng to 14 in order to achieve both high rate and good quality results.

As specified by RFC 8219 [4], 60s long tests with 1s timeout were used. The tests were executed 20 times and median as well as 1 and 99 percentiles were determined, where the latter two correspond to minimum and maximum as the number of tests were less than 100. Although it is not required by RFC 8219, we introduced another quantity for the characterization of the quality of the results in [10]. This is dispersion, calculated as the proportion of the range of the results and the median, expressed in percentage, as defined by (3).

$$dispersion = \frac{max - min}{median} * 100\% \qquad (3)$$

All the DNS64 performance measurement results are shown in Table II. As for the results of the original `dns64perf++`, the difference between the maximum (54,785qps) and minimum (51,199qps) is 3,586qps, which is about 6.63% of the median (54,057qps) thus, the accuracy of the measurement is questionable. The results of the corrected program are significantly better. The difference between the maximum (55,459qps) and minimum (54,223qps) is 1,236qps, which is about 2.25% of the median (54,853qps) thus, the accuracy of the measurement is acceptable, but we would not call it as good. Having no other test program, we are unable to tell at this point, where the observed 2.25% dispersion of the results produced by the corrected test program comes from. Either it may be still caused by the corrected test program or it may be an inherent property of mtd64-ng. Therefore, we needed to perform further tests.

Besides the DNS64 tests, we have also performed "self-tests". We had multiple arguments for doing so:

- From the viewpoint of `dns64perf++`, the two kinds of tests (DNS64 and self-test) are the same: `dns64perf++` has to send requests for AAAA records and receive the answers, thus any of them can be suitable for the determination of the accuracy of `dns64perf++`.
- RFC 8219 requires both kinds of tests.
- As for the achievable rates, they are higher for self-tests, and thus self-tests can better demonstrate the difference between the old and the new software.

As for measurement setup, we used the one called the "self-test" of the tester in section 9.2.1 of RFC 8219 [4]. (This test is described in more details in [3].) For this measurement, the Tester is looped back, that is the Measurer subsystem of the Tester is directly connected to the AuthDNS subsystem of the Tester, leaving out the DNS64 server. We have achieved this by an additional direct cable link (without the removal of the DUT), as mentioned before.

As for authoritative DNS server for the self-test measurements, we used NSD 4.1.14-1 with no "server-count" specification, thus using only a single CPU core to achieve non-scattered results. This was not a random choice, but we knew from our experience that good quality results with low dispersion can be expected in this case.

As specified in RFC 8219 [4], 60s long tests with 0.25s timeout were used, which were executed 20 times. The results are shown in Table III. The results produced by the original test program, are rather poor. The difference between the maximum (155,653qps) and minimum (130,815qps) is 24,838qps, which is about 16.95% of the median (146,511qps) thus, the accuracy of the measurement is very much questionable. The results of the corrected program are incomparably better. The difference between the maximum (177,169qps) and minimum (176,127qps) is 1,042qps, which is about 0.59% of the median (177,051qps), therefore, the accuracy of the measurement can be considered as good. Thus, we have shown that `dns64perf++` can be used as a good quality measurement tool.

TABLE II
DNS64 PERFORMANCE RESULTS OF MTD64-NG PRODUCED BY THE ORIGINAL AND THE CORRECTED DNS64PERF++ MEASUREMENT PROGRAM.

dns64perf++ version		original	corrected
DNS64 performance (number of successfully processed queries per second)	median	54,057	54,853
	minimum	51,199	54,223
	maximum	54,785	55,459
	dispersion	6.63	2.25

TABLE III
AUTHORITATIVE DNS PERFORMANCE RESULTS OF NSD PRODUCED BY THE ORIGINAL AND THE CORRECTED DNS64PERF++ MEASUREMENT PROGRAM.

dns64perf++ version		original	corrected
Authoritative DNS performance (number of successfully processed queries per second)	median	146,511	177,051
	minimum	130,815	176,127
	maximum	155,653	177,169
	dispersion	16.95	0.59

VII. Future Work

RFC 8219 [4] contains benchmarking methodology for several other IPv6 transition technologies besides DNS64. For benchmarking double translation or encapsulation technologies, RFC 2544 testers may be used, when the dual DUT setup is followed (please refer to Section 4.2 of [4]). However, the usage of the single DUT setup is also recommended for them, and the single DUT setup is the only possible measurement setup for single translation technologies such as e.g. stateful NAT64 [3] or SIIT [11]. As far as we know, no RFC 8219 compliant testers are available for their benchmarking. The only tester mentioned in a research paper aims to address stateless IPv4/IPv6 translation implementations [12].

We contend that the significance of our correction of the timing algorithm of the `dns64perf++` program goes far beyond DNS64 bechmarking. As `dns64perf++` is a free software under GPLv2 license, its source code can be used as a starting point for implementing testers for different classes of IPv6 transition technologies. However, for doing so, it is a prerequisite to have a proper timing algorithm, what we have now ensured.

VIII. Conclusions

We conclude that our efforts were successful in measuring the accuracy of the `dns64perf++` program, finding the reason of its significant inaccuracy above 50,000qps and correcting it. Now, its accuracy is ensured by changing its malfunctioning self-correcting timing algorithm to a very simple one, which attempts to compensate the accumulated latency at each step. We have also demonstrated that the accuracy of the results produced by the corrected test program was significantly higher than the accuracy of the results produced by the original one and we concluded that `dns64perf++` can be used as a good quality measurement tool.

Acknowledgment

The measurements for section VI were carried out by using the resources of NICT StarBED, Japan.

Gábor Lencse thanks Masatoshi Enomoto for his help and support in NICT StarBED usage related issues.

References

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

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

[3] G. Lencse, M. Georgescu, and Y. Kadobayashi, "Benchmarking methodology for DNS64 servers", *Computer Communications*, vol. 109, no. 1, pp. 162–175, Sep. 1, 2017, DOI: 10.1016/j.comcom.2017.06.004

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

[5] D. Bakai, "A C++11 DNS64 performance tester", source code, https://github.com/bakaid/dns64perfpp/

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

[7] G. Lencse, "Enabling dns64perf++ for benchmarking the caching performance of DNS64 servers", unpublished, review version is available: http://www.hit.bme.hu/people/lencse/publications/

[8] G. Lencse and D. Bakai, "Design, implementation and performance estimation of mtd64-ng, a new tiny DNS64 proxy", *Journal of Computing and Information Technology* vol. 25, no, 2, pp. 91–102, Jun. 2017, DOI: 10.20532/cit.2017.1003419

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

[10] G. Lencse, Y. Kadobayashi, "Benchmarking DNS64 implementations: Theory and practice", unpublished, review version will be available: http://www.hit.bme.hu/people/lencse/publications/

[11] C. Bao, X. Li, F. Baker, T. Anderson, F. Gont, "IP/ICMP translation algorithm", RFC 7915, Jun. 2016, DOI: 10.17487/RFC7915

[12] 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, Jul. 2017, pp. 74–78, DOI: 10.1109/TSP.2017.8075940

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 an associate 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 of communication systems, parallel discrete event simulation methodology and IPv6 transition methods.

Attila Pivoda is a BSc student studying electrical engineering at the Széchenyi István University, Győr Hungary. He has experience in managing wireless ISP network with MikroTik, Ubiquiti and Cisco devices since 2010. In part time he manages Linux based web hosting servers and he is programming in HTML, PHP, MySQL. He is writing his degree thesis in benchmarking authoritative DNS servers.