IEICE TRANS. COMMUN., VOL.E100–B, NO.6 JUNE 2017

POSITION PAPER

Design and Implementation of a Test Program for Benchmarking DNS64 Servers

Gábor LENCSE[†a)], ***Member and*** **Dániel BAKAI**[†b)], ***Nonmember***

SUMMARY A new Internet Draft on benchmarking methodologies for IPv6 transition technologies including DNS64 was adopted by the Benchmarking Working Group of IETF. The aim of our effort is to design and implement a test program that complies with the draft and thus to create the world's first standard DNS64 benchmarking tool. In this paper, we disclose our design considerations and high-level implementation decisions. The precision of our special timing method is tested and found to be excellent. Due to the prudent design, the performance of our test program is also excellent: it can send more than 200,000 AAAA record requests using a single core of a desktop computer with a 3.2 GHz Intel Core i5-4570 CPU. Its operation comprises all the functionalities required by the draft including checking the timeliness and validity of the answers of the tested DNS64 server. Our DNS64 benchmarking program, `dns64perf++`, is distributed as free software under GNU GPL v2 license for the benefit of the research, benchmarking and networking communities.

key words: *benchmarking, DNS64, IPv6 transition technology, performance analysis*

1. Introduction

DNS64 [1] and NAT64 [2] IPv6 transition technologies will play an important role in the following years by enabling IPv6-only clients to communicate with IPv4-only servers. There are several well-known DNS64 implementations, e.g. BIND, TOTD, Unbound and PowerDNS, which we have already tested to determine their stability and performance [3] and we have shown that their performances are significantly different for various reasons, e.g. some of them can benefit from a multi-core environment and some of them cannot. We have also created a small test program, `dns64perf`, which is suitable for examining the performances of DNS64 servers [4]. This program is feasible for both testing the stability of the DNS64 implementations (by themselves) and comparing their performances. However, this program is not suitable for *benchmarking* their performance. By "benchmarking" we mean accurately measuring some standardized performance characteristics and obtaining reasonable and comparable results. To be more specific, e.g. the benchmarking methodology for network interconnect devices described in [5] includes a throughput test that requires the Tester to be able to send packets (of given size) *at a predefined rate* and decide if the DUT (Device Under Test) can forward the packets at that rate or not. Unfortunately, `dns64perf` is not suitable for this type of measurement because it waits for the reply of its current query* before sending the next one. Although its query rate may be somewhat tuned by the appropriate setting of the number of threads, it is not able to send queries at a predefined rate. Another problem is that the speed of the Tester computer (used to execute `dns64perf`) also influences the results. This is not a problem if our aim is to compare multiple DNS64 implementations, but it is unacceptable if one would like to *benchmark* (that is to objectively quantify) a given DNS64 server.

As for the methodology of DNS64 server testing, a later RFC was prepared for addressing IPv6 specificities [6], but it explicitly states that IPv6 transition mechanisms are outside of its scope. There is a new Internet Draft to cover benchmarking methodology for IPv6 transition technologies including DNS64 servers [7]. The aim of our current efforts is to design and implement a test program which complies with the draft and thus to create the world's first standard DNS64 benchmarking tool. In this paper, we disclose the design considerations and implementation decisions of our test program.

Our new test program, `dns64perf++` is a free software for the benefit of the research, benchmarking and networking communities and it is available under the GNU GPL v2 license from GitHub [8].

The remainder of this paper is organized as follows. Section 2 contains the basic operation requirements based on the Internet Draft. Section 3 discloses our most important design considerations. Section 4 presents our high-level implementation decisions. Section 5 is a case study for the justification of our timing algorithm. Section 6 mentions a paper (in making) about our experience with `dns64perf++` and highlights our plans for its further development. Section 7 gives our conclusions.

2. Basic Operation Requirements

2.1 Test and Traffic Setup

The Internet Draft [7] adopted our basic operation step which we used in our previous works (including [3] and [4]) namely the test program should send queries for AAAA records of

Manuscript received May 24, 2016.
Manuscript revised October 23, 2016.
Manuscript publicized December 16, 2016.

[†]The authors are with the Department of Networked Systems and Services, Budapest University of Technology and Economics, Magyar tudósok körútja 2, H-1117 Budapest, Hungary.
a) E-mail: lencse@hit.bme.hu
b) E-mail: bakaid@kszk.bme.hu
DOI: 10.1587/transcom.2016EBN0007

*The words *query* and *request*, as well as *reply* and *answer* are used with the same meaning throughout the paper.

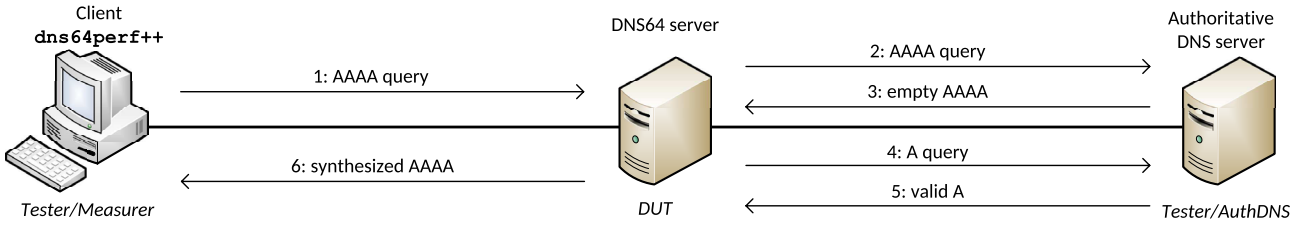

Fig. 1 Test setup for benchmarking DNS64 servers.

domain names which have actually no AAAA records but only A records in the DNS system.

Section 9.1 of the Internet Draft [7] describes the test and traffic setup. Although the draft follows the traditional two devices setup containing the *Tester* and the *DUT* (Device Under Test) only, now we use three devices for clarity: the two sub-functions of the Tester are realized by two physical devices, they are *Tester/Measurer* and *Tester/AuthDNS*, see Fig. 1. Let us follow there what happens during one testing step. First, the client sends a query for an AAAA record of a particular domain name to the DNS64 server. The DNS64 server has to use the DNS system to find out if an AAAA record for the given domain name exists. During the tests, the DNS64 server acts as a forwarder (and not a DNS recursor), thus it simply asks the authoritative DNS server (located on the right side of the figure) by sending message 2. Third, the authoritative DNS server sends and empty reply because no AAAA record exists for the queried domain name. Fourth, the DNS64 server asks the authoritative DNS server for an A record for the same domain name. Fifth, the authoritative DNS server sends a valid A record, which one is used by the DNS64 server to synthesize an IPv4-embedded IPv6 address. Sixth, the DNS64 server returns the synthesized AAAA record to the client.

2.2 Requirements for the Tester

Using the above detailed basic operation step, the draft requires to measure the number of successfully processed AAAA record requests per second by the DNS64 server. For this purpose, the measurement program MUST[†] be able to send the DNS queries at any predefined rate and decide if the DNS64 server is able to reply them in time or not.

First, different domain names MUST be used. (Either they are all different during the measurement, or at least they MUST NOT be repeated until the cache of the DNS64 server still contains them.) In addition to that, measurements MAY be done with domain names, 20%, 40%, 60%, etc. of which are cached. We note that if the information is cached, then messages from 2 to 5 are omitted.

Similarly, first, measurements MUST be done with domain names, which have no AAAA records, and then measurements MAY be done with domain names, 20%, 40%, 60%, etc. of which have AAAA records. We note that if a domain name has an AAAA record then message 3 is not empty but contains a valid answer, messages 4 and 5 are omitted and the native IPv6 address is returned in message 6.

The draft also mentions the timeout: if the test program does not receive a reply from the DNS64 server within a predefined time interval (default value is 1 second) then the request is considered to be lost and it is interpreted that the DNS64 server cannot serve the given rate of requests.

We note that Tester/Measurer, that is, `dns64perf++` has no responsibility for messages other than number 1 and 6 during a DNS64 benchmarking measurement. Section 9.2.1 of the Internet Draft mentions some requirements for the Tester and describes a preliminary self-test. To summarize the self-test in a nutshell, a Tester (including both Measurer and AuthDNS subsystems) can be certified for benchmarking up to rate r with timeout t, if it is looped back (that is Measurer is connected immediately to AuthDNS leaving out the DUT) and the Tester can achieve $2 * (r + \delta)$ rate with timeout $t/4$, where $\delta \geq 0.1$. (See the Internet Draft for more details and explanation.) We note that this self-test gives a guarantee for the sending ability of the Measurer, the replying ability of AuthDNS, the receiving ability of the Measurer as well as the speed of the interconnection. These measurements may be performed using AAAA record requests, thus no further requirements arise.

3. Design Considerations

3.1 Choice of the Namespace

The most important requirements concerning the namespace were already defined in [10]. We need a namespace that:

- can be described systematically
- can be resolved to IPv4 only
- can be resolved without delay

Now, we add one more requirement: the test program should be able to generate the AAAA record requests in a computationally efficient way. We consider that the namespace used in [4] can be used with two modifications:

1. The numbers in it should always contain 3 digits.
2. Its potential size should be extended to 2^{32}.

Thus it will look as follows:

```
{000..255}-{000..255}-{000..255}-{000..255}.dns64perf.test.
```

[†]The key words written "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [9].

Or with a different notation:
k-l-m-n.dns64perf.test., where $k, l, m, n \in [000, 255]$

This is an independent namespace, which can be efficiently resolved to IPv4 by a local authoritative DNS server.

Why do we need the two modifications above? Writing the numbers always in three digits fixes the length of the first label to 15 (4 times 3 digits and 3 dashes) thus it will not be necessary to perform the special encoding (called *message compression*) applied in DNS messages (see [11] for details) for every single domain name.

The potential size of the namespace should be extended because we know from Carsten Strotmann[†] that a good DNS server executed by a modern hardware may resolve more than one million queries per second. To be future proof, we decided to allow potentially four billion different domain names and made it possible for the user to specify the required part of the namespace (see more details in Sect. 4.1).

3.2 Timing of the AAAA Requests

Perfect timing is necessary to ensure the predefined rate of AAAA record requests. In order to produce requests at frequency f, the program must send a request at every T interval, where $T = 1/f$. If the preparation and sending of a request last T_R long time then the program should wait for $T_W = T - T_R$ time before preparing and sending the next request. However, this approach would result in cumulative timing error for multiple reasons:

- the execution of the calculations require non zero time
- the solution of the timed waiting is imprecise
- the time measurement itself is imprecise

To keep the cumulative error marginal, we use an improved algorithm. Instead of calculating the waiting time independently for each message, we always consider the remaining time until the end of the testing. We calculate the waiting time before starting to prepare the $(n + 1)$-th request as follows:

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

where N is the total number of requests to be sent, $t_B(n)$ 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 in the beginning of the test.

3.3 Tracking the AAAA Requests

The program has to decide about every AAAA record request if it was answered within the timeout interval or not. Therefore, it must store every request (not the whole DNS message but enough information for the identification[††]) and the sending timestamp. When a DNS reply message is received then the program has to find the corresponding request and check the difference between the sending and receiving timestamps.

We note that our choice of the namespace makes it possible to identify the requests by using only 32 bits. See implementation details in Sect. 4.5.

3.4 Consideration of Having an AAAA Record – Trivial

This is not an issue for the test program. It can be easily solved at the authoritative DNS server. When the zone file is generated, first, it should contain A records only. After that 20% of the domain names should also have AAAA records (in a uniform distribution) and the measurements should be repeated. Later 40%, 60%, etc. of the domain names should have AAAA records.

3.5 Consideration of Caching – Not Supported

We view the DUT (Device Under Test) as a black box. The potential namespace is large enough to ensure that AAAA record requests are not repeated at all.

To support the testing of the efficiency of caching, we would need to repeat some of the requests. We considered the trivial algorithm that for achieving 20%, 40%, 60%, etc. cache hit rates, we might send the first of every 10 requests three times, five times, seven times, etc. instead of sending the consecutive request(s). However, when requests are sent at a high enough rate, the DNS64 server does not yet have the answer for a request when it receives the next one. Therefore, the requests should be repeated not promptly but rather a "certain" amount of time later. However, if this delay is too high then the information may be deleted from the cache of the DNS64 server. The appropriate choice would require special knowledge of the tested DNS64 implementation. Using an inappropriate method would result in unfair

[†]He wrote us by e-mail: "For BIND 9, I currently see a peak of around 500,000 queries/second on modern hardware (limited to 6-8 cores due to BIND 9 cache locking issues with more cores), and Unbound up to 1,200,000 queries per second (using $cores - 2$ for Unbound, e.g. 34 cores on a 36 core machine)."

[††]It is possible theoretically that the program mistakenly accepts a reply from an earlier experiment. If different domain names are used in every execution of the program (to eliminate the effect of caching) then this mistake may not happen; otherwise it is the responsibility of the user to avoid it by using a long enough gap (or restarting the DNS64 server) between the consecutive executions of `dns64perf++`.

testing.

As the testing of the effect of caching is OPTIONAL in the draft, we do not include this feature in the first version of the test program, but we plan to include it in a later version.

4. High-Level Implementation Decisions

4.1 Specifying the Actual Namespace

The required part of the potential namespace can be easily identified by the specification of the corresponding IPv4 address range (to which it is mapped by the authoritative DNS server) using the CIDR notation. For example, the 10.0.0.0/12 range means the range with 2^{20} number of elements which can be described also as:
010-{000..015}-{000..255}-{000..255}.dns64perf.test.

In addition to that, it is not necessary to use all the elements of the given range, the user must specify the number of requests to send, which must be less than or equal with the size of the range.

4.2 Implementation of Timing

The test program is intended to be executed under Linux, where the scheduling frequency of the timer interrupt is one millisecond. If the calculated waiting time is significantly higher than 1ms, than it may be worth using the sleep[†] function to spare CPU capacity. Though its precision is rough, the timing error can be always be compensated at the next request. (For time measurement, nanosecond precision clocks are used.) If the waiting time is less, then busy waiting is used: the current time is tested in a loop.

Though this combined method produced good results, later we completely abandoned using the sleep function to achieve excellent precision timing, see Sect. 5 for details.

4.3 Threads and CPU Cores

The current implementation uses two threads. The main thread is responsible for receiving the replies and a thread is started for timely sending of the queries. Thus having at least two CPU cores is a prerequisite for the execution of the test program. It is planned that later versions will be able to utilize all available CPU cores.

4.4 Adding Burst Mode

The exact timing requires additional work, and thus the highest AAAA record request sending rate of our test program is less than it could be if the requests were sent without timing. On the one hand, this is natural (this is the price of exact timing), but on the other hand, sometimes it is desirable to be able to send requests at a higher rate than our hardware is able to do it with proper timing. (For example, if we want to compare two very good DNS64 implementations and we do not have a fast enough tester device compared to the DUT we use.) To implement this feature, we added the "burst size" parameter. Within a burst, the packets are sent as soon as possible without timing.

The burst size of 1 results in the original algorithm providing exact timing of all the requests. Higher burst sizes enable higher average rate of requests but with less precise timing. Of course, results produced in this way, are not to be considered as trustworthy benchmarking results but they can be useful as an estimation.

4.5 Storing Requests and Validating Answers

The sent AAAA record requests can be unambiguously identified by the prefix of the actual namespace and the number of the given request. The latter one is identical with the bits of the corresponding IPv4 address after the prefix. (That is the last 20 bits in the example of Sect. 4.1.) Therefore, we do not store this information at all, rather it is used for indexing in an array which contains important information (including the sending time) about each sent AAAA record requests.

When a reply is received, it contains the request in the "Question" section (see [11]). The first label of the domain name is read from it, the corresponding IPv4 address is calculated and its appropriate part is used for indexing the array. In the appropriate element of the array, the received flag is set to true and the receiving time is stored. It is also registered if the reply contained at least one answer.

The measurement ends when the specified timeout expires for the lastly send request. Then the program processes the array containing information about the DNS requests and replies. If a query was answered then the RTT (Round Trip Time) is calculated by subtracting the sending time of the query from the receiving time of the reply. If the reply contained at least one answer then it is checked if the calculated RTT is not more than the timeout value. If yes, then the answer is qualified as valid.

4.6 Output of the Program

The output of the program contains the number of sent queries, the number of received answered, the number of valid answers as well as the average and standard deviation of the RTT of the received answers. In addition to that, all the results are dumped in CSV format (into the file `dns64perf.csv`).

4.7 Usage of Command Line Positional Parameters

Our test program is designed for being executed by a script several times with different parameters. (The script is likely to perform a binary search to determine the highest AAAA query rate at which a DNS64 implementation can answer.) We considered the application of positional command line arguments a natural and easy way of specifying the necessary parameters. See the accompanying documentation for

[†]Actually, the `std::this_thread::sleep_for()` function is called if waiting time is higher than 5ms.

Table 1 Accuracy of the individual waiting time calculation method.

Required frequency (req/s)		10	100	1000	10000	100000
Number of requests		10	100	1000	10000	100000
Sending time of the specified number of request (ms)	average	1001.04	1010.65	1000.11	1002.02	1022.72
	standard deviation	0.07	0.23	0.01	0.18	2.66
	minimum	1000.86	1010.02	1000.10	1001.73	1019.01
	maximum	1001.10	1010.86	1000.12	1002.28	1027.12
Real frequency (req/s)		9.99	98.95	999.89	9979.81	97778.13

Table 2 Accuracy of the waiting time calculation method using remaining time.

Required frequency (req/s)		10	100	1000	10000	100000
Number of requests		10	100	1000	10000	100000
Sending time of the specified number of request (ms)	average	1000.10	1000.11	1000.00	1000.00	1000.00
	standard deviation	0.01	0.00	0.00	0.00	0.00
	minimum	1000.08	1000.11	1000.00	1000.00	1000.00
	maximum	1000.11	1000.11	1000.00	1000.00	1000.00
Real frequency (req/s)		10.00	99.99	1000.00	10000.00	99999.99

Table 3 Accuracy of the final waiting time calculation method.

Required frequency (req/s)		10	100	1000	10000	100000
Number of requests		10	100	1000	10000	100000
Sending time of the specified number of request (ms)	average	1000.00	1000.00	1000.00	1000.00	1000.00
	standard deviation	0.00	0.00	0.00	0.00	0.00
	minimum	1000.00	1000.00	1000.00	1000.00	1000.00
	maximum	1000.00	1000.00	1000.00	1000.00	1000.00
Real frequency (req/s)		10.00	100.00	1000.00	10000.00	99999.99

details [8].

5. Investigation of the Precision of Timing Methods

In this section, first, we compare the precision of the two before mentioned timing methods: the one using independent waiting time calculation and the one using remaining time for waiting time calculation. Next, we modify the second one to achieve even higher accuracy. Then, we provide and initial performance estimation of `dns64perf++`. Finally, we consider the limitations of our test program.

5.1 Testing Method and Test Environment

We requested the test program to send AAAA record requests at different frequencies. The number of the requests was set so that their sending last exactly 1 second and we measured the actual time of their sending. All the experiments were executed 11 times, average, standard deviation, minimum and maximum values of the sending times were calculated. The actual frequency of the requests was calculated from the average time.

Unlike the old `dns64perf`, `dns64perf++` sends AAAA record requests independently from the responses. Therefore, now we have to specify only the parameters of the computer, which executed the `dns64perf++` test program: the rest of the test network is redundant.

A desktop computer was used with the following parameters: 3200 MHz Intel Core i5-4570 CPU (4 cores, 6 MB L3 cache), 16 GB 1600 MHz DDR3 SDRAM, 250 GB SSD; Debian GNU/Linux 8.2 operating system.

5.2 Results

The results produced by using individual waiting time calculation are presented in Table 1. We can observe that the inaccuracy is about 0.1% at 10 Hz, and it grows to about 1% at 100 Hz. It happens because the sleep function of the operating system is used (please recall the 1 ms resolution of the timer interrupt frequency). The inaccuracy is about 0.01% and 0.2% at 1,000 Hz and 10,000 Hz, respectively. We consider all these values acceptable. However, the inaccuracy is about 2.2% at 100,000 Hz. As `dns64perf++` is intended to be a standard benchmarking program for producing trustworthy results, we can no way tolerate an inaccuracy over 1%. Therefore, we decided to replace the individual waiting time calculation method by the one that uses the remaining time for the calculation of the waiting time before the generation of the next request. Table 2 shows the results produced by our improved algorithm. The results are convincing. The highest inaccuracy, which occurred ad 100 Hz, was only 0.01%.

5.3 Removal of the Sleep Function

We have completely removed the usage of the sleep function from the program to achieve always the best possible accuracy. Table 3 shows the results of the final version of `dns64perf++`. They are unexceptionable.

Thus, we have shown that `dns64perf++` is a precise measurement tool for benchmarking DNS64 servers.

Table 4 Performance limit estimation.

Required frequency (req/s)		200000	250000	300000
Number of requests		200000	250000	300000
Sending time of the specified number of request (ms)	average	1000.00	1000.56	1176.69
	standard deviation	0.00	0.94	2.50
	minimum	1000.00	1000.00	1172.25
	maximum	1000.00	1002.97	1181.83
Real frequency (req/s)		199999.99	249860.07	254951.59

5.4 Performance Testing

Although the performance analysis of `dns64perf++` is not an aim of this paper, the approximate performance of the program was checked. Table 4 shows that the accuracy of timing starts degrading at 250,000 requests per second. As for the performance of `dns64perf++`, its more than 200,000 AAAA requests per second performance is expected to be enough for testing DNS64 implementations executed by commonly used servers, as our experienced highest rate by which a DNS64 server (executed by a quad-core Sun server) could work was less than 8,000 requests/s [3].

5.5 Hardware Performance Warning

Please note that we tested only the sending performance of `dns64perf++`. To be suitable for benchmarking up to a given frequency, it is also necessary that the hardware used for the execution of `dns64perf++` be able to receive and process all the packets. Otherwise, e.g. missed interrupts may cause that the replies are lost and the queries are considered by `dns64perf++` to be unanswered. The program can no way find out why it did not receive a reply.

5.6 Considering the Jitter of the Queries

We have shown that the final timing algorithm provides high accuracy concerning the average frequency calculated as the overall number of queries per the overall testing time. However, there are certain unavoidable situations when timing will not be precise locally. This is a consequence of the fact that we designed a software-based generator executed by a computer under a given operating system, namely Linux. For example, it may happen that the CPU core executing the sender thread receives an interrupt during the busy waiting cycle of the timing algorithm and the processing of the interrupt lasts too long and therefore the next request will be sent too late. The timing algorithm is self-correcting in the sense that the average frequency will be precise, but there will be some jitter in the sequence of the queries. Another typical situation can happen when the required frequency is so high that the CPU can still cope with the sending task when the code is already loaded into its cache memory but it will be lagging behind the required timing in the beginning. Again, the self-correcting nature of the timing algorithm will set the average frequency precisely, but the inter-arrival time of the first queries will be significantly higher than it should be. (Similarly, the branch predictors of the CPU may make some wrong decisions until they learn the behavior of the program, etc.)

Although it is theoretically possible that the jitter of the requests impairs the performance of the DNS64 servers, we do not think that it can significantly influence the results, because the DNS64 servers should be able to store the requests to some extent to be able to serve the real life requests with bursts. Moreover, the draft [7] requires at least 60 seconds long test duration, which is long enough to iron out the consequences of the local timing inaccuracies.

Although we cannot eliminate the jitter, we provide all the jitter information to the user, as `dns64perf++` stores the nanosecond precision sending timestamps of the requests. It also writes the timestamps (with other raw data) into the `dns64perf.csv` file after finishing the measurements. Thus, by processing this file, the user may calculate the jitter values and decide if it is acceptable for his/her purposes or not. (In the latter case, the measurement may be invalidated and repeated.)

6. Experience and Future Plans

We are using `dns64perf++` in an ongoing project for a case study and demonstration of the DNS64 benchmarking method described in [7]. Our experiences and results will be published in [12].

As of our current implementation, `dns64perf++` uses two threads: one for sending the requests and one for receiving the replies. Thus, it can utilize two cores of a computer. Therefore it leaves several cores free if it is executed by a modern multi-core computer having four or more cores, which may be used to execute the authoritative DNS server if the test setup of the draft [7] is followed, which contains only two devices: the Tester and the DUT.

For testing very high performance DNS64 servers, it may be necessary to use three devices instead, by dedicating two high performance computers for Tester: one for the execution of the authoritative DNS server and one for executing `dns64perf++`. We plan to make `dns64perf++` multi-threaded to be able to utilize all the cores of a modern multi-core server.

We also plan to make the second version of `dns64perf++` capable of testing the effect of caching. However, currently we cannot see a suitable method yet. (We note that it is a general DNS server benchmarking issue, not DNS64 specific.) We welcome any experiences or suggestions concerning the benchmarking method for DNS caching from the research community.

We consider that our design principles may be used for the design of other benchmarking tools e.g. for testing NAT64 [2] implementations, too. Being `dns64perf++` a free software, our C++11 source code [8] may also be reused.

7. Conclusions

We conclude that our efforts were successful in creating the world's first standard DNS64 benchmarking tool, `dns64perf++`. Tests proved that it offers both high precision and high performance (concerning sending queries for different AAAA records).

Our further plans include: testing its limits for benchmarking (being executed by some particular hardware), adding the capability of utilizing the computing power of arbitrary number of CPU cores, and extending it for to test the effect of caching.

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, April 2011.
[2] M. Bagnulo, P. Matthews, and I. Beijnum, "Stateful NAT64: Network address and protocol translation from IPv6 clients to IPv4 servers," RFC 6146, April 2011.
[3] G. Lencse and S. Répás, "Performance analysis and comparison of four DNS64 implementations under different free operating systems," Telecommun. Syst., vol.63, no.4, pp.557–577, 2016. DOI: 10.1007/s11235-016-0142-x
[4] G. Lencse, "Test program for the performance analysis of DNS64 Servers," Internat. J. Advances in Telecommunications, Electrotechnics, Signals and Systems, vol.4, no.3, pp.60–65, Sept. 2015. DOI: 10.11601/ijates.v4i3.121
[5] S. Bradner and J. McQuaid, "Benchmarking methodology for network interconnect devices," RFC 2544, March 1999.
[6] C. Popoviciu, A. Hamza, G. Van de Velde, and D. Dugatkin, "IPv6 benchmarking methodology for network interconnect devices," RFC 5180, May 2008.
[7] M. Georgescu, L. Pislaru, and G. Lencse, "Benchmarking methodology for IPv6 transition technologies," IETF Benchmarking Working Group, Internet Draft, https://tools.ietf.org/html/draft-ietf-bmwg-ipv6-tran-tech-benchmarking-03
[8] D. Bakai, "A C++11 DNS64 performance tester," source code, https://github.com/bakaid/dns64perfpp/
[9] S. Bradner, "Key words for use in RFCs to indicate requirement levels," RFC 2119, March 1997.
[10] G. Lencse and G. Takács, "Performance analysis of DNS64 and NAT64 solutions," Infocommunications J., vol.4, no.2, pp.29–36, June 2012.
[11] P. Mockapetris, "Domain names – implementation and specification," RFC 1035, Nov. 1987.
[12] G. Lencse, M. Georgescu, and Y. Kadobayashi, "Benchmarking methodology for DNS64 servers," unpublished, review version will be available: http://www.hit.bme.hu/people/lencse/publications

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.

Dániel Bakai is a BSc student studying computer science at the Budapest University of Technology and Economics, Budapest, Hungary. He does project work for the Department of Networked Systems and Services, Budapest University of Technology and Economics since February 2015. He is also the author of the mtd64-ng DNS64 implementation, a successor of MTD64.