Benchmarking Methodology for DNS64 Servers

Gábor Lencse[a,*], Marius Georgescu[b], Youki Kadobayashi[c]

[a]*Department of Networked Systems and Services, Budapest University of Technology and Economics, Magyar tudósok körútja 2, Budapest, H-1117, Hungary*
[b]*IP/MPLS Backbone Department of RCS&RDS, Str. Dr. Nicolae D. Staicovici 71-75, Bucharest 030167, Romania*
[c]*Internet Engineering Laboratory of Nara Institute of Science and Technology, Takayama-cho, 8916-5, Nara, 630-0192 Japan*

Abstract

DNS64 is an important IPv6 transition technology used in convergence with NAT64 to enable IPv6-only clients to communicate with IPv4-only servers. Several DNS64 implementations have been proposed as a solution. Their performance is an important decision factor for network operators with regard to choosing the most appropriate one among them. To that end, this article proposes a methodology for measuring their performance. The number of resolved queries per second is proposed as performance metric and a step by step procedure is given for its measurement. The design considerations behind the method are also disclosed and the performance requirements for the tester device are specified. The feasibility of our method is proven and its execution is demonstrated in two case studies, which include an empirical analysis of the tester as well as that of three open-source DNS64 implementations. The influence of the rate of existing AAAA records on the performance of the DNS64 server, as well as the influence of the cache hit rate of the DNS64 server on the performance of the DNS64 server are also measured and modeled. Our results and their precision may serve as a reference for further tests.

Keywords: benchmarking, DNS64, Internet, IPv6, IPv6 transition, performance analysis

1. Introduction

DNS64 [1] servers together with NAT64 [2] gateways play an important role in the IPv6 transition by enabling an IPv6-only client to communicate with an IPv4-only server. We expect this scenario to be very common in the upcoming years because the ISPs (Internet Service Providers) cannot provide public IPv4 addresses to their ever increasing number of new clients, due to the depletion of the public IPv4 address pool. They could distribute private IPv4 addresses and use CGN (Carrier Grade NAT), but the forward-looking procedure is to deploy global IPv6 addresses to the new clients. However, the majority of the servers on the Internet still have IPv4 addresses only. We believe that the NAT64/DNS64 tool suite [3] is one of the best solutions for this problem. NAT64 is mentioned as the only "feasible stateful translation mechanism" in [4]. Reference [5] gives an up to date survey of the IPv4 address sharing methods, and concludes that: "The only actual address sharing mechanism that really pushes forward the transition to IPv6 is Stateful NAT64 (Class 4). All other (classes of) mechanisms are more tolerant to IPv4."

Several implementations exist for both DNS64 and NAT64. When selecting from among them, performance is a decisive factor for network operators. Having performance data produced by using standardized benchmarking methods enables network operators to compare different implementations. RFC 2544 [6] aims to define such methods. IPv6 specificities were later addressed in [7], but this document explicitly excluded IPv6 transition mechanisms from its scope. The internet draft [8] aims to cover them. There are several IPv6 transition methods and the draft attempts to be general enough to cover most of them. To that end, several categories were defined (e.g. encapsulation, single or double translation) and a specific benchmarking setup is recommended for each category. DNS64 is a solution which does not fit in these categories, and therefore requires "individual attention".

In this article, we focus on the methodology for benchmarking DNS64 servers. Our aim is threefold. We would like to give an insight into our considerations which resulted in the method specified in [8], section 9. We also provide a detailed example of how to carry out the measurement procedure described in the draft. And last but not least we would like to receive feedback from the scientific community about the proposed benchmarking method.

The remainder of this paper is organized as follows. In section 2, the relevance of the DNS64 performance is stated and a brief introduction to the operation of the DNS64 plus NAT64 IPv6 transition solution is given. In section 3, a short survey of other methodologies for the performance analysis of DNS64 servers is presented. In section

*Corresponding author
Email addresses: lencse@hit.bme.hu (Gábor Lencse), marius.georgescu@rcs-rds.ro (Marius Georgescu), youki-k@is.aist-nara.ac.jp (Youki Kadobayashi)

4, the proposed benchmarking methodology is described. In section 5, performance requirements for the tester device are formulated. Section 6 is a general case study for demonstrating how to carry out the proposed tests and giving a deeper insight into the methods, as well as providing a reference concerning the expected accuracy of the results. Section 7 is a supplementary case study for examining different test and traffic setups. In section 8, our plans for future research are outlined. Finally, in section 9, the conclusions are stated.

2. Background information: relevance of DNS64

We examine the relevance of the DNS64 performance in the first subsection, and for those not familiar with the operation of DNS64 and NAT64, we present the operation of these important IPv6 transition solutions in the second subsection.

2.1. Relevance of DNS64 performance

A large ISP needs to resolve several hundred thousands of DNS requests per second. For example, RCS&RDS, the current employer of the second author, does about 300,000 queries per second, whereas Google Public DNS did a daily average of 810,000 queries per second in 2012 [9].

As for DNS64, it is used only by the IPv6-only clients. Their number is usually low in the beginning at all ISPs, but it is expected to rise due to the depletion of the public IPv4 address pool. We cannot see into the future, but if the transition to IPv6 will use mainly the DNS64+NAT64 technology and there will be a time when the majority of the clients will be already IPv6-only and they still need to be able to connect to IPv4-only servers, then the DNS64 servers will be faced with a load of the above mentioned magnitude. Practically it means that a delay in the DNS64 resolution will have an immediate negative effect on the user experience of the high number of IPv6-only clients.

We believe that the science of computer communication needs a proper benchmarking methodology for DNS64 servers so that the performance of the different DNS64 implementations may be accurately measured and compared by using standardized performance metrics and researchers may adequately qualify the different DNS64 implementations by obtaining reasonable and comparable performance characteristics.

2.2. Operation of DNS64 and NAT64

We demonstrate the operation of DNS64 and NAT64 on the example of an IPv6-only client and an IPv4-only web server taken verbatim from our conference paper [10]. Fig. 1 shows a scenario where an IPv6-only client communicates with an IPv4-only web server. The DNS64 server uses the 64:ff9b::/96 *NAT64 Well-Known Prefix* [11] for generating *IPv4-embedded IPv6 address*es [11]. There are two prerequisites for the proper operation:

1. A DNS64 server should be set as the DNS server of the IPv6-only client.
2. Packets towards the 64:ff9b::/96 network are routed to the NAT64 gateway (routing must be configured that way).

Let us follow the steps of the communication:

1. The client asks its DNS server (which one is actually a DNS64 server) about the IPv6 address of the `www.hit.bme.hu` web server.
2. The DNS64 server asks the DNS system about the IPv6 address of `www.hit.bme.hu`.
3. No IPv6 address is returned.
4. The DNS64 server then asks the DNS system for the IPv4 address of `www.hit.bme.hu`.
5. The 152.66.148.44 IPv4 address is returned.
6. The DNS64 server synthesizes an *IPv4-embedded IPv6 address* by placing the 32 bits of the received 152.66.148.44 IPv4 address after the 64:ff9b::/96 prefix and sends the result back to the client.
7. The IPv6 only client sends a TCP SYN segment using the received 64:ff9b::9842:f82c IPv6 address and it arrives to the IPv6 interface of the NAT64 gateway (since the route towards the 64ff9b::/96 network is set so in all the routers along the path).
8. The NAT64 gateway constructs an IPv4 packet using the last 32 bits (0x9842f82c) of the destination IPv6 address as the destination IPv4 address (this is exactly 152.66.248.44), its own public IPv4 address (198.51.100.10) as the source IPv4 address and some other fields from the IPv6 packet plus the payload of the IPv6 packet. It also registers the connection into its connection tracking table (and replaces the source port number by a unique one if necessary). Finally it sends out the IPv4 packet to the IPv4 only server.
9. The server receives the TCP SYN segment and sends a SYN ACK reply back to the public IPv4 address of the NAT64 gateway.
10. The NAT64 gateway receives the IPv4 reply packet. It constructs an appropriate IPv6 packet using the necessary information from its state table. It sends the IPv6 packet back to the IPv6 only client.

The communication may continue. It seems to the client that it communicates to an IPv6 server. Similarly, the server "can see" an IPv4 client. If it logs the IP addresses of the clients than it will log the public IPv4 address of the NAT64 gateway.

Most client-server applications can work well with the DNS64+NAT64 solution. See more information about the application compatibility in [12, 13, 14].

In practice, the word wide usage of the *NAT64 Well-Known Prefix* has several hindrances, see sections 3.1 and 3.2 of [11]. Therefore the network operators allocate a subnet from their own network for this purpose. It is called *Network Specific Prefix* [11].

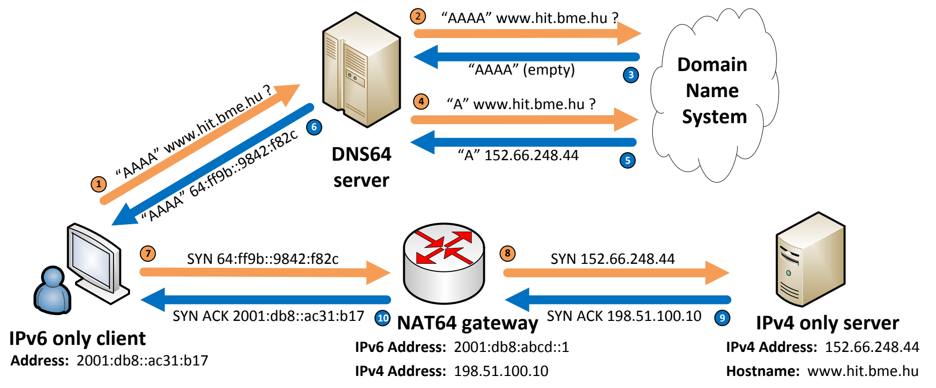


Figure 1: The operation of the DNS64+NAT64 solution: an IPv6-only client communicates with and IPv4-only server [10]

3. Short survey of methods for performance analysis of DNS64 servers

In existing literature, we could find only a few articles on this topic. Usually, articles other than ours do not deal with DNS64, but rather focus on NAT64, and when its performance is analyzed the performance of a given DNS64 implementation is also considered implicitly. For example, the performance of the TAYGA NAT64 implementation is compared to the performance of NAT44 in [15], whereas TOTD is used as a DNS64 server. The performance of the Ecdysis NAT64 implementation is compared to the performance of the authors own HTTP ALG in [16]. Ecdysis used its own DNS64 implementation. Similarly, the performance of the Ecdysis NAT64 implementation (using its own DNS64 implementation) is compared to the performance of both the NAT-PT and an HTTP ALG in [17].

The Ecdysis NAT64 implementation was used together with BIND as DNS64 implementation in [18]. In this paper, DNS64 was addressed without NAT64: the CPU usage of DNS64 was compared to that of the DNS. However, no methodology was provided for the performance analysis of DNS64 servers.

The first known method for the performance analysis of DNS64 servers was published in [19]. Its purpose was to test the stability of a DNS64 implementation, more specifically, to check if its behavior under heavy load complies with the *graceful degradation* [20] principle. The same method was used for stability testing and also performance comparison of two DNS64 implementations [21]. The basic idea of the method is to test the DNS64 servers by sending many AAAA record requests from a namespace which:

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

The testing method was implemented with bash shell scripts using the `host` Linux command. However, this command also requests an MX record by default, therefore its "`-t AAAA`" option was used later to query for the AAAA record only [22]. The usage of the bash shell scrips was rather inefficient, thus certain part (the inner for cycle) was replaced by a short C program in [23]. The program was used for the performance analysis and comparison of four DNS64 implementations executed by a quad-core computer. To address the performance problem, a short test program named `dns64perf` was written in C/C++ [24]. However, all these programs have very important limitations as they were designed for stability testing and performance comparison, but not for *benchmarking* DNS64 implementations. What is the difference?

The following technical challenges need to be solved in order to obtain a suitable DNS64 benchmarking environment.

- The benchmarking program should be able determine exactly how many AAAA record requests can the given DNS64 implementation service in a second.
- The performance results should not depend on the tester. In other words, if the performance of the tester is not enough to test a certain implementation, the user needs to be notified.

The above mentioned methods have not surpassed these challenges and therefore cannot be considered suitable for benchmarking. This document discusses how these challenges can be overcome and to what extent.

4. Proposed benchmarking methodology

4.1. Objectives

The challenge is to find a well-defined performance metric of DNS64 servers, which can be measured simply and efficiently. Moreover, we need a procedure which describes how to measure that performance metric. Our aim was to follow the simple test setup with a *Tester* and a *DUT* (Device Under Test) defined in section 6 of [6], or at least to use a similar and simple test setup. As for the procedure,

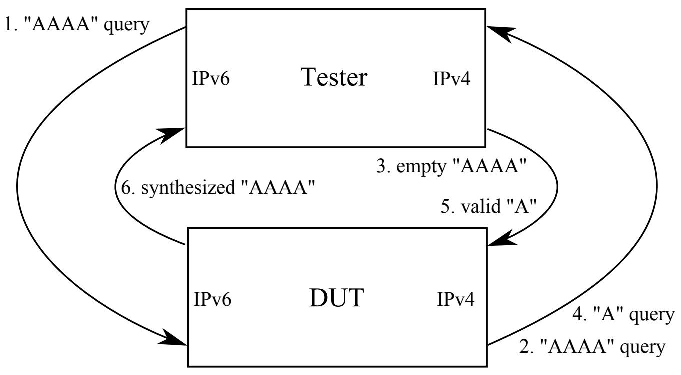


Figure 2: Test and traffic setup for benchmarking DNS64 servers

we intended to keep it simple and similar to other test procedures, whereas we intended to cover all the important circumstances or parameters which can significantly influence the performance of a DNS64 server.

4.2. Test and traffic setup

In our earlier DNS64 performance tests, there were three different roles:

- The clients sent the AAAA record requests for different domain names.
- A DNS64 implementation answered the requests and for doing so, it sent AAAA and A record requests to the authoritative DNS server.
- The authoritative DNS server answered the AAAA and A record queries of the DNS64 server.

To follow the above mentioned simple test setup with only two devices, we decided to give the *Tester* two roles, that of the client and that of the authoritative DNS server. Naturally, the *DUT* plays the role of the DNS64 server, see Fig. 2. (We discuss the potential advantages and disadvantages of this two-device setup compared to a three-device setup in section 6.11.) The operation of DNS64 is realized by the following messages:

1. Query for the AAAA record of a domain name (from client to DNS64 server)
2. Query for the AAAA record of the same domain name (from DNS64 server to authoritative DNS server)
3. Empty AAAA record answer (from authoritative DNS server to DNS64 server)
4. Query for the A record of the same domain name (from DNS64 server to authoritative DNS server)
5. Valid A record answer (from authoritative DNS server to DNS64 server)
6. Synthesized AAAA record answer (from DNS64 server to client)

We note that the above messages represent the scenario when no AAAA record exists for the given domain name, and the DNS64 server synthesizes the *IPv4-embedded IPv6 address* from the A record (IPv4 address) of the domain name. If an AAAA record for the given domain name exists, then it is returned to the DNS64 server in message 3 by the authoritative DNS server; message 4 and message 5 are left out and the DNS64 server returns the received AAA record to the client in message 6. Although the above order of the messages is logical, the DNS64 server may send message 4 before it receives message 3 to minimize delay (see section 5.1.8 of [1]). If the DNS64 server implements caching and the requested domain name is present in its cache memory, then message 1 is followed by message 6 and messages from 2 to 4 are omitted. Thus, besides the worst case scenario with all six messages, there are two other possible scenarios: the one when an AAAA record exists (with messages 1, 2, 3 and 6) and the one when there is a cache hit (with messages 1 and 6).

4.3. Performance metric and benchmarking procedure

4.3.1. Performance metric

We have chosen the number of successfully processed DNS requests per second as performance metric. Its exact definition is revealed by the measurement procedure.

4.3.2. Measurement procedure

The procedure was inspired by the throughput measurement recommended by Bradner et al. in [6]. The following steps are needed:

- Send a specific number of DNS queries at a specific rate to the DUT and then count the replies received in time (within a predefined timeout period from the sending time of the corresponding query, having the default value 1 second) from the DUT.
- If the count of sent queries is equal to the count of received replies, the rate of the queries is raised and the test is rerun. If fewer replies are received than queries were sent, the rate of the queries is reduced and the test is rerun.
- The maximum number of processed DNS queries per second is the fastest rate at which the count of DNS replies sent by the DUT is equal to the number of DNS queries sent to it by the test equipment.

We note that using a higher rate than the one measured with the above procedure might result in more successfully processed DNS record requests per second (and also non zero unanswered queries). However, we prudentially defined the performance metric with the above procedure for at least three reasons:

1. Our goal is a well-defined performance metric, which can be measured simply and efficiently. Allowing any packet loss would result in a need for scanning/trying a large range of rates to discover the highest rate of successfully processed DNS queries.

2. Even if users may tolerate a low loss rate (please note the DNS uses UDP with no guarantee for delivery), it cannot be arbitrarily high, thus, we could not avoid defining a limit. However, any other limits than zero percent would be hardly defensible.
3. Other benchmarking procedures use the same criteria of zero packet loss and this is the standard in IETF Benchmarking Methodology Working Group.

4.3.3. Parameters

We pointed out three different scenarios in section 4.2. We recommend the testing of the worst case scenario as compulsory and the others as optional. Its reason is trivial: if a DNS64 implementation performs well in the worst case scenario then it is a real guarantee for any other scenarios. Leaving out this one, would result in a possibility for gaming. The two other scenarios may give valuable insight, but we cannot tell how relevant they are. As for caching, the operating systems also do DNS caching, thus its relevance depends on the similarity of the domain names used by the members of the user community of a given DNS64 server. As for the existence of an AAAA record, its probability grows with the deployment of IPv6. Therefore, we consider the existence of AAAA records and caching as orthogonal parameters, and thus recommend various combinations of the two.

4.3.4. Reporting format

There may be random events, which influence the results. Consequently, the tests should be repeated multiple times and the final result should be calculated by using a particular summarizing function. Namely, the test should be repeated at least 20 times and the median should be used to summarize the results. To account for the variation of the results across the 20 repetitions, the 1st and 99th percentiles are recommended. We note that different definitions exist for percentile. We use the definition from section 11.3 of [25]. The rationale for using the median as summarizing function and the percentiles as index of dispersion is presented in section 6.10.

4.4. Timeout time and the duration of each test

Although the default timeout value of the Linux name resolver is 5 seconds [26], our personal experience under the Window 7 operating system is that the unanswered DNS queries are resent after 1 second to a different DNS server. We consider that the smallest timeout should be tested, therefore we recommend 1s (one second) as the default timeout value for DNS64 testing.

As for the duration of each test, we have chosen to use full length trials of 60 seconds on the basis of our empirical results (see in the case study). We recommend this because, if the DNS64 implementation is able to store the requests, then the result of a shorter test may be significantly biased due to the following effect. Let t_{Test} denote the duration of the test, that is, the interval while the Tester sends queries for AAAA records at rate r_T. Let t_{TO} denote the value of the timeout within the Tester accepts the replies from the DUT. To complete the test, the DUT must answer $t_{Test} * r_T$ number of queries within $t_{Test} + t_{TO}$ time. That is, it is enough[1] if the DUT can answer AAAA record queries at rate r_{DUT}, where:

$$r_{DUT} = \frac{t_{Test} * r_T}{t_{Test} + t_{TO}} = r_T \frac{1}{1 + \frac{t_{TO}}{t_{Test}}} \tag{1}$$

Now, it is evident that $t_{TO} \ll t_{Test}$ is necessary for a correct test result. Even if $t_{Test} = 60s$ and $t_{TO} = 1s$ then:

$$r_{DUT} = \frac{60}{61} r_T = 0.9836 r_T \tag{2}$$

5. Performance requirements for the Tester

First of all, the Tester must be able to provide AAAA queries at r_T rate.

Second of all, as the authoritative DNS server is a part of the Tester and each AAAA record query results in two queries (one of them is for an AAAA record and the other one is for an A record) thus the Tester must be able to reply DNS queries at $2r_T$ rate.

Thirdly, there should also be a timeout requirement laid down for the authoritative DNS server. It is clear that the faster the authoritative DNS server is, the more time will be left for the DUT. Theoretically, we could arbitrarily share the t_{TO} timeout between the DUT and the authoritative DNS server. Before any decision, let us consider the potential effects.

On one hand, the time budget of the DUT may be very important if it stores a high number of requests in a certain data structure and it needs to insert new elements into the data structure and find/delete old ones. If the time complexity of these operations of the data structure is logarithmic, then a little difference in the time budget results in a significant difference in the size of the data structure it may handle.

On the other hand, setting up too high requirements for the Tester would result in unnecessary high costs.

Therefore, we decided to halve the t_{TO} timeout between the two devices. In other words, the Tester must answer each query within $0.25t_{TO}$ time and thus $0.5t_{TO}$ remains for the DUT.

Before a Tester may be used for testing at rate r_T with the required timeout of t_{TO}, the Tester must perform a *self-test*[2] if it is able to answer its own queries at $2r_T(1+\delta)$ rate within $0.25t_{TO}$ timeout. The role of δ is to guarantee that the Tester itself does not limit the performance of the DUT. We recommend $\delta \geq 0.1$.

[1]If we omit the packet transmission times and propagation delays of both the request and the reply.

[2]For performing the self-test, the Tester must be looped back (omitting the DUT) and its authoritative DNS server subsystem must be configured to be able to resolve the AAAA queries.

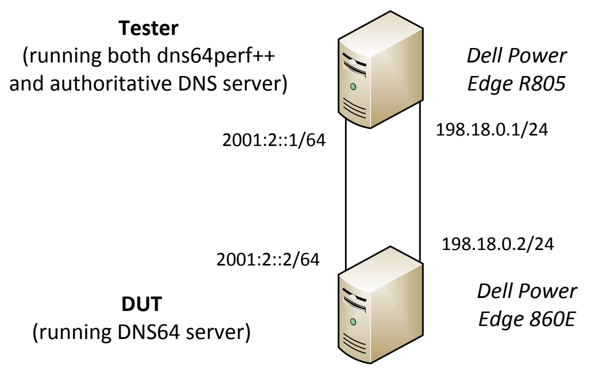


Figure 3: Test and traffic setup for benchmarking DNS64 servers

We note that a powerful Tester, which can reply AAAA and A queries within much less time than $0.25t_{TO}$ leaves more time for the DUT than a Tester which just passes the self-test. Therefore, the results may depend on the Tester as well.

6. General case study

In this case study, we demonstrate the feasibility of the required tests and provide a straightforward method for how to carry them out. In addition to that, we examine how some parameters can influence the results. The accuracy of our results may also serve as a reference for other test setups. A new test setup can be checked by testing one of the implementations we also tested, and the variation of the results can be compared to that of our results. We also give some hints about what circumstances can cause scattered results.

6.1. Hardware and software environment

We used the two-device setup[3] for our DNS64 measurements. The devices and their roles are shown in Fig. 3.

We provide the most important details of the used hardware and software for the repeatability of the measurements.

6.1.1. Dell PowerEdge R805

Two Quad-core AMD Opteron 2378 2400MHz CPUs, 32GB 800MHz DDR2 SDRAM, three Broadcom NetXtreme BCM5721 Gigabit Ethernet NICs (PCI Express).

Ubuntu 14.04 LTS Linux operating system with the 3.13.0-32-generic kernel version.

[3]By using the two-device setup, we demonstrate that it is feasible to use only a single computer as Tester. We argue that it is unnecessary to increase the number of machines for benchmarking purposes, as a single machine can reasonably ramp up the query rate to reveal the performance limits. Our goal is to facilitate benchmarking by everyone, under reasonable amount of resources, under reasonable amount of time. (We demonstrate the alternative, three-device setup in the complementary case study.)

6.1.2. Dell PowerEdge 860E

One Quad-core AMD Opteron 2378 1866MHz CPU, 8GB 667MHz DDR2 SDRAM, four Broadcom NetXtreme II BCM5708 Gigabit Ethernet NICs (PCI Express)

Ubuntu 14.04 LTS Linux operating system with the 3.13.0-32-generic kernel version.

6.1.3. DNS64 performance measurement program

As for measurement tool, `dns64perf++` 1.0.0 was used. The implementation was developed by Dániel Bakai for these purposes. The program is documented in [27] and it is available as a free software under the GNU GPLv2 license from Github [28].

6.1.4. Authoritative DNS server and DNS64 servers

BIND9.9.8-P3 (compiled from source to be single-threaded, in order to achieve more stable measurement results) was used as both authoritative DNS server and DNS64 server.

Two other DNS64 implementations were also tested: TOTD 1.5.3 with the patch documented in [22] available from Github [29] and mtd64-ng 1.0.0, available from Github [30].

6.2. Introduction of `dns64perf++` in a Nutshell

We give a short summary of the properties of the performance measurement tool on the basis of [27].

6.2.1. Namespace used

The test program can use the elements of the following independent namespace:

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

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

These domain names are to be mapped to the $k.l.m.n$ IPv4 address by the authoritative DNS server.

The actual namespace to be used, can be defined by the CIDR notation of the corresponding IPv4 address range.

6.2.2. Input parameters of the program

The program takes the following command line parameters:

1. IPv6 address of the DNS64 server
2. Port number of the DNS64 server program
3. Subnet of the required namespace
4. Number of AAAA record requests to be sent
5. Burst size – must be 1.
6. Delay between bursts (in nanoseconds) – must be the delay between the starting time of the sending of the consecutive requests, that is: $10^9/r_T$
7. Timeout – specifies the timeout value (in seconds) within which an answer is accepted.

6.2.3. Operation of the program

The execution of `dns64perf++` requires two CPU cores, because the program uses two threads: one of them sends AAAA record requests with proper timing (implemented by busy waiting, thus this thread utilizes all computing power of a CPU core), and the other thread receives the replies, it records the arrival time of all the received replies, as well as the fact that it contained an answer (an IPv6 address resource record). When the specified timeout value elapsed after the request sent last, the program evaluates its records and decides about every single request if it was replied. If yes, then the RTT (Round-Trip Time) is calculated. It also checks if the reply contained an answer. If yes, then it is considered as *valid*, if its RTT is less than or equal to the timeout value.

We note that the current version of the program cannot utilize the computing power of more than two CPU cores. This fact can be a limitation when it is used for benchmarking DNS64 server implementations executed by high-performance computers.

6.2.4. Output of the program

The program prints out the time required for sending the specified number of requests (in nanoseconds) so that the user can confirm its time accuracy.

It prints out the number of sent queries, the number of received replies, the number of valid answers as well as the average and standard deviation of the RTT of the received replies.

It also dumps its records in CSV format (making available the raw information to the user).

6.3. Shell script for binary search

We used a bash shell script which executed a binary search to determine the highest possible rate at which the number of valid answers was equal to the number of sent queries. The measurements were repeated 20 times. The timeout value and the duration of the tests were used as parameters and we have examined how their values can influence the results.

The shell script was also responsible for eliminating the effect of caching. This was done by restarting the given DNS64 implementation after each step of the binary search.

A possible alternative solution is to use different namespaces for each tests, that is, for each iteration steps of all 20 repetitions. Depending on the query rate and the duration of the tests, the authoritative DNS server may need a large amount of memory.

For all our tests, we used the namespace which can be described by the 198.0.0.0/11 subnet. (It made possible testing up to 34,952 queries per second rate at 60 second duration, which was enough for our purposes.)

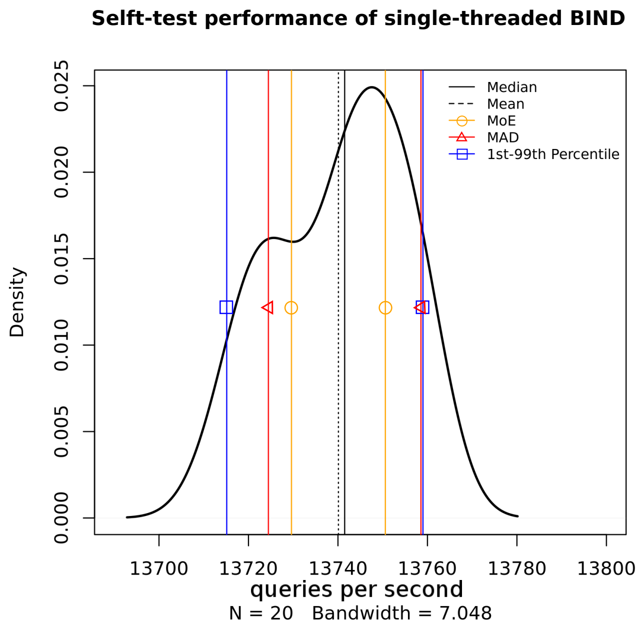


Figure 4: BIND9 authoritative DNS server performance results: number of successfully answered AAAA record requests per second (**single-threaded**, 800MHz CPU clock frequency)

6.4. Self-test for the Tester

We decided to measure the performance of the authoritative DNS server in the same way we tested the performance of the DNS64 server. We disclose both the details of the measurements and the results, as they offer important insights, as well as validate the proposed methodology.

6.4.1. Measurement method details

In order to ensure that the performance of the measurement tool will be enough to measure the performance of the authoritative DNS server, we used the following test setup. The clock frequencies of cores 0-3 of the Tester were set to 2400MHz using the `cpufreq-set` Linux command. The affinity of `dns64perf++` was set so that it could be executed by these cores only (using the `taskset` Linux command). Similarly, the clock frequencies of cores 4-7 of the Tester were set to 800MHz and the affinity of the single-threaded BIND (used as authoritative DNS server) was set so that it can be executed only by one of them.

The measurements were performed using different test duration and timeout values. Presently, we are focusing on the 60 seconds long one with 0.25 second timeout, because this one was needed for the DNS64 tests.

As `dns64perf++` can request only AAAA records, a zone file was generated for BIND which contained AAAA records for the namespace which can be described by the 198.0.0.0/11 subnet.

6.4.2. Results

The results were very stable, the minimum and the maximum of the successfully answered AAAA record requests

Table 1: Mtd64-ng DNS64 performance: number of answered queries per second, 1 worker thread, **1s timeout**

Duration (s)	5	10	30	60
Average	3448.1	3160.9	2970.8	2926.3
Median	3448	3161	2971	2926
1st Perc. (min)	3442	3157	2967	2923
99th Perc. (max)	3452	3164	2974	2929
Standard Deviat.	2.81	1.77	1.92	1.66
MoE 99.9	2.07	1.31	1.41	1.22
Median Abs. Dev.	2.97	1.48	1.48	1.48
Calculated rate	2873.3	2873.6	2875.2	2878.0

Table 2: Mtd64-ng DNS64 performance: number of answered queries per second, 1 worker thread, **0.5s timeout**

Duration (s)	5	10	30	60
Average	3168.1	3025.7	2932.2	2920.3
Median	3168	3026	2932	2920
1st Perc. (min)	3162	3022	2930	2917
99th Perc. (max)	3175	3031	2936	2923
Standard Deviat.	3.35	2.19	1.58	1.48
MoE 99.9	2.47	1.60	1.16	1.09
Median Abs. Dev.	2.97	1.48	1.48	1.48
Calculated rate	2880.0	2881.9	2883.9	2895.9

per second were 13714 and 13759, respectively. See further details in Fig. 4, which depicts the probability distribution of the results for the 20 measurements.

6.5. Parameters for the DNS64 tests

In order to explore how the duration of the tests and the timeout value influence the results, we used them as parameters. The test duration values were: 5s, 10s, 30s and 60s. The timeout values were: 0.5s, 1s and 5s. And we used the before mentioned three DNS64 implementations: BIND9, TOTD and mtd64-ng.

We do not include all results produced by using all three implementations using all possible parameter combinations because of their high number. We selected those that can be used for the demonstration of our most important observations.

Table 3: Mtd64-ng DNS64 performance: number of answered queries per second, 1 worker thread, **5s timeout**

Duration (s)	5	10	30	60
Average	5786.8	4338.7	3371.9	3135.9
Median	5787	4339.5	3372	3136
1st Perc. (min)	5780	4334	3368	3133
99th Perc. (max)	5792	4342	3375	3139
Standard Deviat.	3.19	2.20	1.53	1.62
MoE 99.9	2.35	1.62	1.13	1.19
Median Abs. Dev.	2.22	2.22	1.48	1.48
Calculated rate	2893.5	2893.0	2890.3	2894.8

6.6. DNS64 results of mtd64-ng

We start with mtd64-ng because it produced outstandingly stable results. We note that only one *worker thread* was used both for reducing its performance and making the results more stable. It meant that the program was running in two threads: one for receiving the requests and one for processing them. (In mtd64-ng terminology, the number of worker threads means the number of those threads processing the requests. The default value for the number of worker threads is 30, in order to be able to utilize the available computing power of multi-core CPUs.)

Table 1 shows the results with our proposed 1 second timeout value. The first line of the table specifies the test duration, and all the other values are given as the function of it. The average and median values are very close to each other at any test duration, and the largest difference between the minimum and the maximum values is 10 (occurred at 5 second test duration).

We note that the 1^{st} and 99^{th} percentiles correspond to the minimum and maximum values, respectively, because the number of repetitions is 20 (less than 100).

The number of answered queries per second shows a visibly decreasing tendency, which we explain by the phenomenon of storing requests and using up the timeout time for servicing them. To check our hypothesis, the last line of the table contains the values which were calculated according to (1) using the median, the 1s timeout and the test duration time values. These numbers predict very similar values for the real, infinitely sustainable service rate of the DUT between 2873 and 2878 queries per second.

In order to further validate our hypothesis, we have to observe the results with 0.5s and 5s timeout values in Table 2 and Table 3, respectively. They are also very stable and their prediction for the infinitely sustainable rate is also very similar. Mtd64-ng is a good example of a very stable DNS64 implementation, which can store the pending requests and thus it can efficiently perform some "gaming" in utilizing the timeout value for significantly increasing its measured response rate. To that end, an extreme example is, the median in Table 3 at 5s test duration and 5s timeout, which is 5787 queries per second, exactly the double of the calculated infinitely sustainable rate of 2893.5 queries per second.

6.7. DNS64 results of BIND9

The results of the single threaded BIND9 DNS64 server with 1s timeout can be seen in Table 4. They are also acceptably stable though the fact that the difference between the minimum (3124) and the maximum (3515) is 391 at 60s test duration, which is more than 11% of the median (3471), is a serious warning sign. Both average and median show some decrease as the test duration grows from 5s to 60s, but this decrease is much less than it was in the case of mtd64-ng. We included the calculated infinitely sustainable rate in the last line of the table, but these numbers

Table 4: **BIND9** DNS64 performance: number of answered queries per second, 1 worker thread, 1s timeout

Duration (s)	5	10	30	60
Average	3547.3	3505.7	3485.2	3455.0
Median	3552.5	3506	3480.5	3471
1st Perc. (min)	3514	3465	3464	3124
99th Perc. (max)	3595	3535	3515	3515
Std. Dev.	20.15	19.51	15.05	79.51
MoE 99.9	14.83	14.36	11.08	58.51
Median Abs. Dev.	17.79	16.31	11.86	13.34
Calc. r. (wrong)	2960.4	3187.3	3368.2	3414.1

Table 5: **TOTD** DNS64 performance: number of answered queries per second, 1 worker thread, 1s timeout

Duration (s)	5	10	30	60
Average	2577.6	2536.6	2426.9	2409
Median	2581	2532.5	2421.5	2412
1st Perc. (min)	2492	2453	2337	2332
99th Perc. (max)	2656	2617	2505	2502
Std. Dev.	41.02	48.63	51.76	54.56
MoE 99.9	30.18	35.78	38.09	40.15
Median Abs. Dev.	37.06	65.23	51.15	57.08
Calc. r. (wrong)	2150.8	2302.3	2343.4	2372.5

are visibly very different from each other, thus we can conclude that BIND9 cannot efficiently "game" with the rate by storing the requests and answering them later on.

6.8. DNS64 results of TOTD

The results of the TOTD DNS64 server with 1s timeout can be seen in Table 5. They are also acceptably stable. However, we consider it a warning sign that the difference between the minimum and the maximum is between 160 and 170 at any test duration which is about 6-7% of the median. Both average and median show some decrease as the test duration grows from 5s to 60s, but – similar to BIND9, – this decrease is much less than it was in the case of mtd64-ng. Again, we included the calculated infinitely sustainable rate in the last line of the table, but these numbers are significantly different from each other, thus we can conclude that TOTD as well is unable to efficiently "game" with the rate by storing the requests and answering them later.

6.9. Overview of the results

6.9.1. Comparison with earlier results

Although the performance comparison of the different implementations is not our main goal, we have to mention that the performances of TOTD and BIND9 are now in a reverse order than they were found earlier in [21, 22, 23]. We see two possible reasons.

1. The version of BIND9 is now higher than it was in the earlier examinations and the DNS64 performance of the program could have been improved significantly.
2. The applied measurement methods are significantly different. Our benchmarking method considers late replies as lost, and it harshly penalizes lost replies: even if a single query is not answered from among several thousand ones, the test fails.

6.9.2. Sustainable rate calculation

Although the calculation for the sustainable rate was successfully demonstrated in the case of mtd64-ng, the same method proved to be unusable with BIND9 and TOTD. Consequently, the method cannot be used in general. However, the possibility of such gaming makes the usage of full 60 seconds long duration tests a must.

Of course, there is a trade-off between the accuracy of the results and speed of benchmarking. Using longer test duration will result in less chance for gaming. The maximum possible gain of gaming is under 2% according to (2) when using 60 seconds duration. By doubling the test duration, one could reduce the possible gain to less than 1% according to (3).

$$r_{DUT} = \frac{120}{121} r_T = 0.9917 r_T \tag{3}$$

However, we came to the conclusion that it is not worth the effort. When considering this question we should also bear in mind that the 60 or 120 seconds long test is executed in the core of several nested cycles. The most inner cycle is the binary search. Its number of necessary steps depend on the initial testing rate (e.g. in our experiments 16 steps were enough, but about 20 steps may be necessary if the maximum possible rate is 1,000,000 queries per second). The next cycle comes from the 20 repetitions of the experiment. If optional tests are also done for testing the effect of different proportion of existing AAAA records and/or cache hits, they are two orthogonal parameters, too. In addition to that, several other parameters may be tested, e.g. in the case of mtd64-ng the optimal number of working threads may be determined by a series of experiments. And of course, one may aim to test several DNS64 implementations to compare their performances.

6.10. Question of average versus median

Average (also called mean or arithmetic mean) and median are both often used as summarizing functions of the results of multiple experiments. In our earlier papers [21, 22, 23], we used average (together with standard deviation), but it had been chosen without any special considerations. Whereas average is more inclusive and less sensitive to noise, it is more sensitive to outliers. If the distribution is significantly skewed or multimodal then median is more representative. Reference [31] suggests the usage of mean if the difference between them is limited, and median otherwise. However, in our case it would be completely inadequate to use mean one time and median some other time. We have to choose one of them which can be used in all cases. It must be the one which is more sufficient if there is significant difference between them and

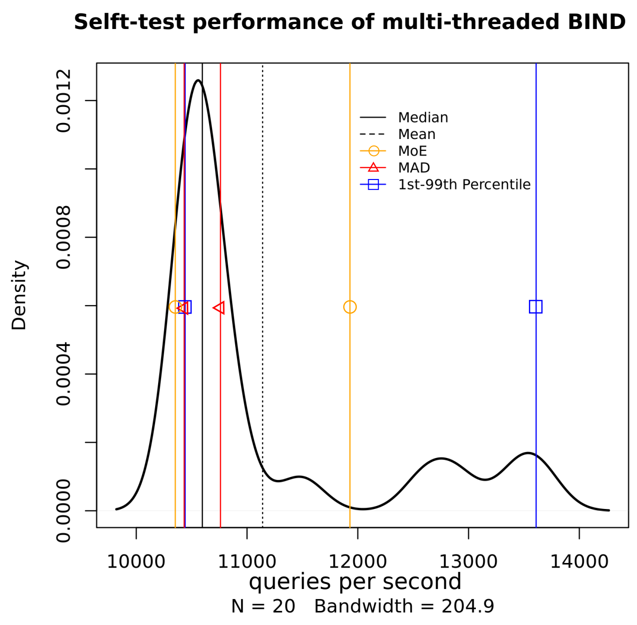


Figure 5: BIND9 authoritative DNS server performance results: number of successfully answered AAAA record requests per second (**multi-threaded**, 800MHz CPU clock frequency)

it is acceptable if they are close to each other. To address this question on the practical side, we show a case when the results are much more scattered then they were in the previous examples. Now BIND is used as multi-threaded. As observed from Fig. 5, the distribution of the results was quite skewed. In this context, the median is more representative as a summarized quantity. In terms of variation, the 1st and 99th percentiles showed to capture the most relevant part of the dataset distribution. We found a similar trend in most of the collected data (see also Fig. 4). Subsequently, we recommend the median for summarizing the results and the 1st and 99th percentiles as indices of dispersion.

6.11. Two-device setup versus three-device setup

Until now, we followed the two-device setup as shown in Fig. 2. This approach required the Tester to play two roles: to perform the task of an authoritative DNS server and to execute the performance test program. This approach did not cause serious limitations because of our special conditions:

- Our test program, *dns64perf++* could utilize only two CPU cores.
- Our authoritative DNS server, BIND9 was executed as single threaded.
- The memory of the Tester computer was more than enough because the DUT was restarted after each tests (instead of using a large zone file to provide independent namespace for each tests).

However, in other cases it may be recommendable using a three-device setup where the two functions of the tester are provided by two independent computers:

- Tester/AuthDNS plays the role of the authoritative DNS server
- Tester/Measurer executes the test program.

In our case study, that was not necessary. However, when benchmarking DNS64 implementations executed by modern high performance servers with e.g. 16 or 32 cores, the three-device setup will probably be necessary. (The three-device setup is demonstrated in section 7.)

6.12. Possible causes of scattered results

We have identified a number of reasons which may result in the high variation of the results. Some of them are reporting errors and some of them are just describing the unchangeable character of the DUT:

- CPU frequency scaling is enabled. We recommend to disable it in BIOS or to set it to a fixed value (using the `cpufrequtils` package under Linux).
- There are some missed interrupts. We recommend to use servers and not desktop PCs.
- Multi-threaded versions of some DNS64 implementations may also be the cause. We recommend to perform the tests also with a single threaded implementation.
- It may be a feature of the tested DNS64 implementation. We recommend to test also one of those implementations we tested. (If similarly stable results are produced with mtd64-ng then the measurement system is likely to work well.)

6.13. Checking the size of the name space

We have made some calculations to check the feasibility of testing a very high speed DNS64 server. If its maximum possible speed is 1Mqps (one million queries per second) then the rate of the self-test of the Tester should be 2.2Mqps and, as the duration is 60s, it requires 132M different domain names. The defined name space contains altogether $2^{32} = 4G$ different domain names, thus the name space is more than enough for a single test. And what about the memory requirements of storing this high number of DNS records? According to our experience, BIND used about 4GB RAM when we loaded a “/8” sized name space having $2^{24} = 16$M elements. Thus, about 33GB RAM would be needed for storing the 132M domain names. It is not a problem for a modern server having 64GB or more RAM. However, it would be too costly to store different names for a whole binary search (consisting about 22 steps during the self-test and “only” about 21 steps using each maximum 66M domain names during the DNS64 test), and practically impossible for storing them for all 20 repetitions of the binary search. Consequently, when testing at such high rates, we recommend to restart the DUT between each iteration step. (Naturally, if we can be sure that its cache is exhausted during a single test run then no restart is necessary; the name space can be reused.)

FakeDNS, which we mention at the end of subsection 8.4, also solves the memory problem of the authoritative DNS server.

6.14. Discussion

NAT64/DNS64 is one of the best choices in terms of IPv6 transition technologies. Measuring the performance of various implementations is a step forward towards better understanding which implementations to use in this context, or what are the pitfalls of using a certain implementation. In this article, we have proposed a benchmarking methodology which can achieve this task. To prove the validity of the proposed approach, we have used it to gather empirical data for three existing DNS64 implementations. However, the method is not without limitations.

One of the fundamental limitations is the performance of the Tester. As described previously, in order to test the performance of a DUT, the Tester needs to be at least twice as fast. To acknowledge and work around this fundamental limitation, we have proposed the concept of self-test which identifies when a Tester is unsuitable to measure the performance of a DUT.

A related limitation is the Tester setup. In the context where the DUT needs to be tested at higher rates, we recommend the two-machine Tester setup.

In the case study, one of the proved limitations is the ability to "game" with the rate by storing the DNS requests and answering them later on. However, this is a somewhat expected behavior for DNS implementations, as the timeout is not a design choice, but a necessity. More details are presented in subsection 8.1.

Other limitations and future work directions are presented in section 8.

7. Supplementary case study

In the general case study, we have addressed only the worst case scenario, when all six messages (shown in Fig. 2) had to be sent. The case when AAAA records exist for a given percentage of the domain names can be easily tested by using appropriate zone files. In addition, the case when a given percentage of the domain names is cached by the DNS64 server requires special considerations, which were not covered yet. The before mentioned three-device setup also seems to be worth testing.

In this supplementary case study, we consider the feasibility and possible methods for the cases when AAAA records exist for some of the domain names or when some of the domain names are cached. We conduct measurements for both cases using the three-device setup.

7.1. Methods for testing with existing AAAA records and caching

On the one hand, the case when AAAA records exist for a given percentage of the domain names can be easily tested by constructing appropriate zone files, which contain AAAA records for the required proportion of the domain names. We consider that setups when 0%, 20%, 40%, 60%, 80% and 100% of the domain names have AAAA records should be tested.

On the other hand, to construct a method for the testing of caching may be a difficult problem if the DUT is handled as a black box, because one needs to know the size of its memory and its cache control algorithm to construct proper test data sequence which contains domain names, exactly 20%, 40%, 60%, etc. of which are cached. To understand why the problem is difficult, let us consider the following solution. Let n_i denote different domain names. A naive algorithm for generating a sequence of 10 domain names containing 2, 4 and 6 cache hits could produce the following sequences:

$n_1, n_1, n_1, n_4, n_5, n_6, n_7, n_8, n_9, n_{10}$;
$n_1, n_1, n_1, n_1, n_1, n_6, n_7, n_8, n_9, n_{10}$;
$n_1, n_1, n_1, n_1, n_1, n_1, n_1, n_8, n_9, n_{10}$;

Though these sequences appear correct for the first sight, unfortunately they will not result in 2, 4 or 6 cache hits if they are sent at a too high rate (because the information will not yet be cached). The difficulty is that the repetition(s) should not happen "too early" in the sequence after the first occurrence of a given domain name *to be already present in the cache*, whereas they should not happen "too late" in the sequence *to be still present in the cache*.

We recommend a simple solution which uses only a single domain name that is repeated and fills that domain name into the cache of the DNS64 server using a preliminary measurement step. This step can be done by using either `dns64perf++` or e.g. the `host` Linux command for sending a single AAAA record request for the domain name we want to be cached. After its being answered, it is cached (if our DNS64 implementation supports caching), and if we repeat it frequently enough (e.g. it is sent as every fifth domain name when 20% cache hit is needed) then it will remain present in the cache.

We must note that this kind of testing shows only that how much faster a DNS64 implementation can be when a given percentage of cache hit is achieved, but it does not provide information about the question that what cache hit rate will be achieved by a given DNS64 implementation in a real life situation. It can also be another direction of future research to work out a different method for addressing this question.

7.2. Theoretical model for results with existing AAAA records and caching

We propose a simple theoretical model for how the reply time of a DNS64 server may depend on the proportion of the existing AAAA records and cache hits. This model takes into consideration what reply time a small number of individual requests may expect. First, we introduce notations for the duration of some important steps:

T_Q "time of a Query": the duration while message 1 (see Fig. 2) is sent, received, queued and processed.

T_C "time of a Cache lookup": the duration while the cache of the DNS64 server is looked up for a given domain name. (We note that the work of refreshing the cache of the DNS64 server must also be considered.)

T_A "time of an Authoritative answer": the duration while message 2 is constructed, sent and received, message 3 is produced, sent, received, queued and processed by the DUT. It is considered to be equal with the time while message 4 is constructed, sent and received, message 5 is produced, sent, received, queued and processed by the DUT.

T_S "time of a Synthesis": the duration while the IPv4 embedded IPv6 address is synthesized.

T_R "time of a Reply: the duration while message 6 is assembled, sent, and received.

For the simplicity, we suppose that:

- message 2 is sent only after an unsuccessful cache lookup (not in parallel)
- message 4 is sent after receiving and empty message 3 (not sent in parallel with message 2)
- the assembly time of message 6 does not depend on where the AAAA record comes from.

Let p_{4A} and p_C denote the probability of the existence of the AAAA record and the probability of the cache hit, respectively.

When only the existence of AAAA records is considered, the reply time of a query may be calculated as:

$$T_{4A} = T_Q + T_C + T_A + (1 - p_{4A})(T_A + T_S) + T_R \quad (4)$$

When only the effect of the caching is considered the reply time of a query may be calculated as:

$$T_C = T_Q + T_C + (1 - p_C)(2T_A + T_S) + T_R \quad (5)$$

When both the existence of AAAA records and the effect of caching are considered, the reply time of a query may be calculated as:

$$\begin{aligned} T_{4A\&C} &= T_Q + T_C \\ &+ (1 - p_C)(T_A + (1 - p_{4A})(T_A + T_S)) + T_R \quad (6) \end{aligned}$$

Important notes:

- In formula (6), we considered that the existence of an AAAA record and the occurrence of a cache hit are independent events.

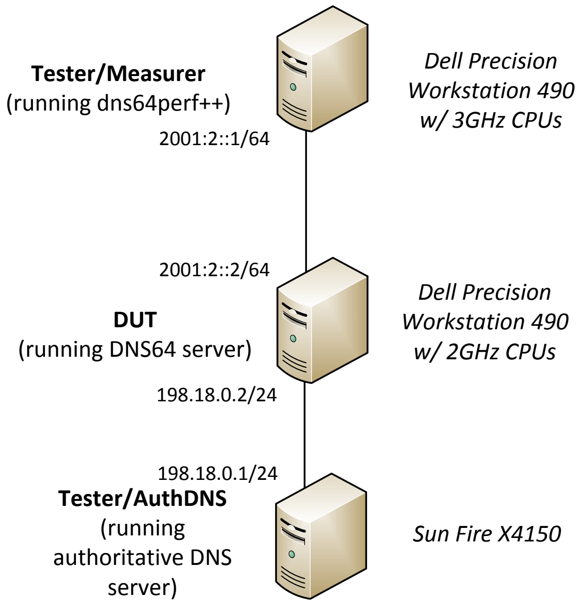


Figure 6: Test setup for DNS64 measurements using three devices

- Formulas (4)-(6), can be used only for a few individual messages. Otherwise the queuing times will be significantly influenced by the values of p_C and p_{4A}.

We may be able to check formulas (4) and (5) in the following two subsections.

7.3. Examining the existence of AAAA records

We used the so called three-device setup to take measurements with different rates of existing AAAA records. The three devices and their roles are shown in Fig. 6.

Below we provide the most important details of the used hardware and software for the repeatability of the measurements.

7.3.1. Tester/Measurer

Dell Precision Workstation with two dual-core Intel Xeon 5160 3GHz CPUs, 4x1GB 533MHz DDR2 SDRAM (accessed quad-channel), Intel PT Quad 1000 type four port Gigabit Ethernet controller (PCI Express).

7.3.2. DUT

Dell Precision Workstation with two dual-core Intel Xeon 5140 2.33GHz CPUs, but operated at 2GHz and only 2 cores were online, 4x1GB 667MHz DDR2 SDRAM (accessed quad-channel), Intel PT Quad 1000 type four port Gigabit Ethernet controller (PCI Express).

7.3.3. Tester/AuthDNS

SunFire X4150 server with two quad-core Intel Xeon E5440 2.83GHz CPUs, 4x2GB 667MHz DDR2 SDRAM, four integrated Intel 82571EB Gigabit Ethernet controllers.

7.3.4. Software

All four computers had Debian 8.6 GNU/Linux operating systems with 3.16.0-4-amd64 kernel version. The before mentioned `dns64perf++` 1.0.0 was used as measurement tool and BIND 9.9.5-9+deb8u7-Debian was used as both DNS64 server and authoritative DNS server.

7.3.5. Measurements

To be able to observe how the proportion of the existing AAAA records may influence the performance of a DNS64 server, 6 series of measurements were performed with domain names, 0%, 20%, 40%, 60%, 80% and 100% of which had AAAA records. The duration of each test was 60 seconds, the timeout was 1 second and the binary search was performed 20 times with each rate of AAAA records.

7.3.6. Results

The results of the measurements aimed at investigating the effect of the AAAA record rate on the performance of the DNS64 server are presented in Table 6. Increasing the rate of existing AAAA records from 0% to 100%, the median of the number of queries per second grows from 3,682 to 5,976. Considering the differences between the performances of the consecutive measurements, they are also growing from 298 to 685. This observation seems to be in a qualitative agreement with our theoretical model, where the reply time showed linear decrease in the function of the AAAA rate. Of course, there is no direct connection between the reply time and the maximum query rate, but if we omit the first and last members (that is T_Q and T_R) from the sum of (4), the remaining members somewhat reflect the amount of work of the DUT, thus the replied query rate of the DUT may be proportional to its reciprocal as follows:

$$R_{4A} = \alpha \frac{1}{T_C + T_A + (1 - p_{4A})(T_A + T_S)} \tag{7}$$

We would like to emphasize the role of T_C. Without it, if we neglect the time (amount of work) necessary for the synthesis of the IPv4 embedded IPv6 address, the approximation would suggest that the 100% AAAA record rate should double the performance of the DNS64 server compared to its performance at 0% AAAA record rate, what does not correspond to our measurement results. However this approach would entirely disregard the work and time used by the DNS64 server for caching. (Please recall, that BIND uses caching, even if we eliminated all its possible benefits by using all different domain names and restarting BIND between the steps of the binary search.)

As for the quality of the results, the difference between the maximum and minimum is always less than 2.5% of the median thus they can be qualified stable enough.

7.4. Examining the effect of caching

The original `dns64perf++` program does not support the testing of caching for the reason that no testing method was proposed at the time of its implementation. During the review process of this paper we have enabled `dns64perf++` to support the above described simple method, which can be used to test how a given cache hit rate influences the achievable performance of different DNS64 implementations.

We note that the implementation of the above suggested method was not as simple as expected. Although it was easy to modify the source code of the `dns64perf++` program to change some of the domain names to the one which had been cached, this change (without some other modifications) would have caused a serious problem in the operation of the `dns64perf++` program, which utilizes the fact that DNS answers contain the request and it reads the first label of the domain name in question, to identify the reply, because the Transaction ID of the DNS messages is only 16 bits long, which is far too short for the unambiguous identification of the replies (without which it cannot be decided which queries are answered in time) [27]. The details of the necessary modifications can be found in [32] and the modified source code is available from [33].

For testing the effect of caching, the same hardware and software environment was used which had been put together for testing the effect of the existence of AAAA records. To be able to observe how the proportion of the cache hit rate may influence the performance of a DNS64 server, 6 series of measurements were performed with domain names, 0%, 20%, 40%, 60%, 80% and 100% of which were the same as the one, which had previously been loaded into the cache of BIND by a `host` command. The duration of each test was 60 seconds, the timeout was 1 second and the binary search was performed 20 times with each cache hit rate. The results of the measurements are presented in Table 7. Increasing the cache hit rate from 0% to 100%, the median of the number of queries per second grows from 3,656 to 32,050, where the latter is 877% of the first value. The differences between the performances of the consecutive measurements are also increasing more and more steeply. The performance at 100% cache hit rate (32,050qps) is more than the double of the performance at 80% cache hit rate (13,157qps). These observations can be easily explained by our theoretical model if we approximate the caching reply query rate of the DUT in a similar way as we did when examining the effect of the existence of the AAAA records (by omitting T_Q and T_R):

$$R_C = \beta \frac{1}{T_C + (1 - p_C)(2T_A + T_S)} \tag{8}$$

Let us consider a simple numeric example. The choice of $T_C = 0.2(2T_A + T_S)$ would result in $R_C(100\%) = 2R_C(80\%)$.

As for the quality of the results, now the difference between the maximum and minimum queries per second rate at 100% cache hit rate is 1926, which is about 6% of the median. It can be considered acceptable.

Table 6: BIND9 DNS64 performance: number of answered queries per second **as a function of AAAA recored rate**, 2 working threads

AAAA record rate (%)	0	20	40	60	80	100
Median	3,682	3,980	4,338	4,771	5,291	5,976
1st Percentile (minimum)	3,641	3,935	4,279	4,733	5,239	5,935
99th Percentile (maximum)	3,733	4,001	4,385	4,809	5,325	6,017

Table 7: BIND9 DNS64 performance: number of answered queries per second **as a function of cache hit rate**, 2 working threads

Cache hit rate (%)	0	20	40	60	80	100
Median	3,656	4,466	5,684	7,782	13,157	32,050
1st Percentile (minimum)	3,615	4,411	5,629	7,679	12,185	30,463
99th Percentile (maximum)	3,745	4,515	5,761	7,937	13,313	32,389

8. Plans for future work

8.1. Problem of sustainable rate

We have shown a possible way of "gaming" with the timeout: how a DNS64 server can show higher performance during a finite duration test than it could sustain infinitely. We have demonstrated with the example of mtd64-ng that this kind of gaming is possible and we have given an upper bound for the difference between the measured and the infinitely sustainable rate. (It is less than 2% with 1s timeout and 60s test duration.)

However, it was only one of the possible reasons why the results of a finite test may not reflect the performance of a longer interval. Other issues are possible with DNS64 servers which are not typical with network interconnect devices. As an example, we mention memory leaking. If the DUT is seen as a black box, memory leaking cannot be discovered easily. The user might only experience a sudden server crash, which may happen after an hour, a week, or even a year, depending on the extent of the memory leaking. Our benchmarking method has no defense against such problems.

8.2. Can a fast Tester help a DNS64 implementation?

We have laid down the rule that a Tester must first successfully complete a self-test for being suitable for testing. However, our conditions specified only an upper time limit: for being able to test a DNS64 implementation at r_T rate with t_{TO} timeout, the tester must be able to answer authoritative DNS requests at $2r_T(1+\delta)$ rate within $0.25t_{TO}$ timeout. What if the Tester answers just within $0.25t_{TO}$ timeout or if it replies significantly faster than that? Can a fast Tester help a DNS64 implementation to achieve significantly higher rates than it could with a just suitable Tester? We intend to answer these questions in the future.

8.3. Further examining the effect of caching

To complement our suggested simple method, we also plan to design a method that will be suitable for the estimation of real life performance of DNS64 with caching. This task includes the invention of repetition patterns suitable for fair testing with different cache control algorithms and cache sizes.

8.4. Hints for authoritative DNS server program

We have chosen BIND as an authoritative DNS server program because we had experience with it. However, it is not necessarily the best choice. We know from Carsten Strotmann (through personal communication) that BIND did not scale up very well when he used it as a resolver (it could utilize only the performance of 6-8 cores of a 36 core computer due to locking issues) and Unbound significantly outperformed it (1,200,000 queries per second vs. 500,000 queries per second). But, unfortunately, Unbound is not an authoritative DNS server. We are currently testing the authoritative DNS server functionalities of several DNS implementations executed by a 12 cores computer and our preliminary results show that YADIFA has significantly higher performance than BIND.

As an alternative, a special-purpose program, called FakeDNS is being developed. This one is not a real DNS server, but it synthesizes the A or AAAA records using the numbers found in the first label of the particular domain name in the query. The program uses the code base of the mtd64-ng DNS64 server program and we expect that it will be significantly faster than any real DNS server.

8.5. RTT of the replies as a possible secondary metric

The RTT of the valid replies could be used as a secondary performance metric of DNS64 servers. Although `dns64perf++` calculates and outputs the RTT values in CSV format, the RTT values highly depend on the delay caused by the authoritative DNS server. The RTT values could be used as a complementary metric if the delay caused by the authoritative DNS server could be isolated. We consider using the following two approaches for solving this:

- The reply time of the authoritative DNS server could be logged and used during post-processing.
- Much smaller reply time could be required from the authoritative DNS server, which is negligible compared to t_{TO}.

The currently developed FakeDNS program might help in both of these issues.

8.6. *Investigation of the effect of CPU cache*

At certain low rates, the used portion of the zone file fits into the L2 or L3 cache of the CPU of the authoritative DNS server, which may result in faster replies from the authoritative DNS server. As discussed in subsection 8.2, it may influence the performance of the DNS64 server. FakeDNS may eliminate this problem, too.

Although the DUT is viewed as a black box, it is also a computer and its performance may be influenced by its CPU cache hierarchy. If the DNS64 implementation uses caching, it depends on the size of its DNS cache and the size and policy of the CPU cache, how effective the CPU cache can be when the elements of the DNS cache have to be reached.

8.7. *Production class Tester implementation*

Even the most current version of `dns64perf++`, which has been enabled for testing the caching performance of DNS64 servers, is only a *prototype*, which is appropriate for proving the correctness of our proposed methodology but may be of insufficient performance for testing high performance DNS64 servers. It is so, because it uses only one thread for sending queries and the main thread for receiving the answers. It was found that it can send about 200,000 queries per second, or at most 250,000qps if we can tolerate some inaccuracies [27]. For the self-test of a production class Tester, we need a program, which can send and receive at one order of magnitude higher rates (about 2,000,000qps rate may be needed for a self-test). It may perhaps be achieved by using 10 threads for sending and 10 threads for receiving executed by a 24 cores computer (to leave some cores free for the host operating system). Of course, we cannot foresee, how its performance will scale up, until our multi-threaded implementation is ready.

9. Conclusion

NAT64/DNS64 is one of the forward-looking IPv6 transition technologies. DNS64 is at the core of its function and various implementations have already been developed. The performance analysis of these implementations is a critical step in choosing the most appropriate one as well as learning how to improve them.

We have proposed a methodology for benchmarking DNS64 servers and also demonstrated its operation in two detailed case studies. In the process, we were able to identify possible pitfalls, such as the possibility of gaming with the results by taking advantage of the timeout value, or choosing the most suitable summarizing function. We have also demonstrated the operation of different test and traffic setups, e.g. the physical realization of the Tester with a single device or with two devices and the lack of or the existence of AAAA records and caching.

We have also discussed several problems to be addressed in the future.

Ultimately, this was done in an effort to build a methodology which can rise up to the expectations of the academic community as well as open standardization bodies, such as the IETF.

Acknowledgements

The first author would like to thank Adél Kiviharju-Turi for English-language proofreading of the paper.

References

[1] M. Bagnulo, A. Sullivan, P. Matthews, I. Beijnum, DNS64: DNS extensions for network address translation from IPv6 clients to IPv4 servers, IETF RFC 6147 (2011). `doi:10.17487/RFC6147`.

[2] M. Bagnulo, P. Matthews, I. Beijnum, Stateful NAT64: Network address and protocol translation from IPv6 clients to IPv4 servers, IETF RFC 6146 (2011). `doi:10.17487/RFC6146`.

[3] M. Bagnulo, A. Garcia-Martinez, I. V. Beijnum, The NAT64/DNS64 tool suite for IPv6 transition, IEEE Commun. Magazine 50 (7) (2012) 177–183. `doi:10.1109/MCOM.2012.6231295`.

[4] P. Wu, Y. Cui, J. Wu, J. Liu, C. Metz, Transition from IPv4 to IPv6: A state-of-the-art survey, IEEE Commun. Surveys and Tutorials 15 (3) (2013) 1407–1424. `doi:10.1109/SURV.2012.110112.00200`.

[5] N. Skoberne, O. Maennel, I. Phillips, R. Bush, J. Zorz, M. Ciglaric, IPv4 address sharing mechanism classification and tradeoff analysis, IEEE/ACM Trans. Netw. 22 (2) (2014) 391–404. `doi:10.1109/TNET.2013.2256147`.

[6] S. Bradner, J. McQuaid, Benchmarking methodology for network interconnect devices, IETF RFC 2544 (1999). `doi:10.17487/RFC2544`.

[7] C. Popoviciu, A. Hamza, G. V. de Velde, D. Dugatkin, IPv6 benchmarking methodology for network interconnect devices, IETF RFC 5180 (2008). `doi:10.17487/RFC5180`.

[8] M. Georgescu, L. Pislaru, G. Lencse, Benchmarking methodology for IPv6 transition technologies, IETF BMWG Internet Draft (2017).
URL `https://tools.ietf.org/html/draft-ietf-bmwg-ipv6-tran-tech-benchmarking-07`

[9] J. K. Chen, Google public DNS: 70 billion requests a day and counting, Google Official Blog.
URL `https://googleblog.blogspot.hu/2012/02/google-public-dns-70-billion-requests.html`

[10] G. Lencse, A. G. Soós, Design of a tiny multi-threaded dns64 server, in: Proc. 38th Int. Conf. on Telecommunications and Signal Processing (TSP 2015), Prague, Czech Republic, 2015, pp. 27–32. `doi:10.1109/TSP.2015.7296218`.

[11] C. Bao, C. Huitema, M. Bagnulo, M. Boucadair, X. Li, IPv6 addressing of IPv4/IPv6 translators, IETF RFC 6052 (2010). `doi:10.17487/RFC6052`.

[12] N. Skoberne, M. Ciglaric, Practical evaluation of stateful NAT64/DNS64 translation, Advances in Electrical and Computer Engineering 11 (3) (2011) 49–54. `doi:10.4316/AECE.2011.03008`.

[13] V. Bajpai, N. Melnikov, A. Sehgal, J. Schonwalder, Flow-based identification of failures caused by IPv6 transition mechanisms, in: Proc. 6th IFIP WG 6.6 Internat. Conf. on Autonomous Infrastructure, Management, and Security (AIMS 2012), Luxembourg, Luxembourg, 2012, pp. 139–150. `doi:10.1007/978-3-642-30633-4_19`.

[14] S. Répás, T. Hajas, G. Lencse, Application compatibility of the NAT64 IPv6 transition technology, in: Proc. 37th Int. Conf. on Telecommunications and Signal Processing (TSP 2014), Berlin, Germany, 2014, pp. 49–55. `doi:10.1109/TSP.2015.7296383`.

[15] K. J. O. Llanto, W. E. S. Yu, Performance of NAT64 versus NAT44 in the context of IPv6 migration, in: Proc. Internat. Multiconf. of Engineers and Computer Scientists 2012 (IMECS 2012), Hong Kong, Hongkong, 2012, pp. 638–645.
URL `http://www.iaeng.org/publication/IMECS2012/IMECS2012_pp638-645.pdf`

[16] C. P. Monte, M. I. Robles, G. Mercado, C. Taffernaberry, M. Orbiscay, S. Tobar, R. Moralejo, S. Pérez, Implementation and evaluation of protocols translating methods for IPv4 to IPv6 transition, Journal of Computer Science & Technology 12 (2) (2012) 64–70.
URL `http://sedici.unlp.edu.ar/handle/10915/19702`

[17] S. Yu, B. E. Carpenter, Measuring IPv4 IPv6 translation techniques, Tech. Rep. 2012-001, Dept. of Computer Science, Univ. of Auckland, Auckland, New Zeeland.
URL `http://hdl.handle.net/2292/13586`

[18] E. Hodzic, S. Mrdovic, IPv4/IPv6 transition using DNS64/NAT64: Deployment issues, in: 2012 IX International Symposium on Telecommunications (BIHTEL), Sarajevo, Bosnia and Herzegovina, 2012. `doi:10.1109/BIHTEL.2012.6412066`.

[19] G. Lencse, G. Takács, Performance analysis of DNS64 and NAT64 solutions, Infocommunications Journal 4 (2) (2012) 29–36.
URL `http://www.infocommunications.hu/documents/169298/404123/InfocomJ_2012_2_Lencse.pdf`

[20] NTIA ITS, Definition of "graceful degradation".
URL `http://www.its.bldrdoc.gov/fs-1037/dir-017/_2479.htm`

[21] G. Lencse, S. Répás, Performance analysis and comparison of different DNS64 implementations for Linux, OpenBSD and FreeBSD, in: Proc. IEEE 27th Internat. Conf. on Advanced Information Networking and Applications (AINA 2013), Barcelona, Catalonia, Spain, 2013, pp. 877–884. `doi:10.1109/AINA.2013.80`.

[22] G. Lencse, S. Répás, Improving the performance and security of the TOTD DNS64 implementation, Journal of Computer Science & Technology 14 (1) (2014) 9–15.
URL `http://sedici.unlp.edu.ar/handle/10915/34537`

[23] G. Lencse, S. Répás, Performance analysis and comparison of four DNS64 implementations under different free operating systems, Telecommun. Syst. 63 (4) (2016) 557–577. `doi:10.1007/s11235-016-0142-x`.

[24] G. Lencse, Test program for the performance analysis of DNS64 servers, Int. J. of Adv. Telecom., Elect., Sign. Syst. 4 (3) (2015) 60–65. `doi:10.11601/ijates.v4i3.121`.

[25] V. Paxson, G. Almes, J. Mahdavi, M. Mathis, Framework for IP performance metrics, IETF RFC 2330 (1998). `doi:10.17487/RFC2330`.

[26] RESOLV.CONF(5), Linux Programmers Manual.
URL `http://man7.org/linux/man-pages/man5/resolv.conf.5.html`

[27] G. Lencse, D. Bakai, Design and implementation of a test program for benchmarking DNS64 servers, IEICE Trans. on Commun. E100-B (6) (2017) –. `doi:10.1587/transcom.2016EBN0007`.

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

[29] F. W. Dillema, DNS proxy and translator for IPv6 and IPv4, source code.
URL `https://github.com/fwdillema/totd/tree/1.5.3`

[30] D. Bakai, A lightweight multithreaded C++11 DNS64 server, source code.
URL `https://github.com/bakaid/mtd64-ng`

[31] R. d. Nijs, T. L. Klausen, On the expected difference between mean and median, Electr. J. of Applied Statistical Analysis 6 (1) (2013) 110–117. `doi:10.1285/i20705948v6n1p110`.

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

[33] G. Lencse, Modified source code of the dns64perfpp program, source code available.
URL `http://www.hit.bme.hu/~lencse/dns64perfppc`

About authors

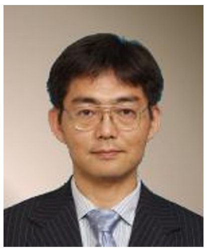
Gábor Lencse received MSc and PhD in computer science from the Budapest University of Technology and Economics, Budapest, Hungary in 1994 and 2001, respectively.

He has been working full time for the Department of Telecommunications, Széchenyi István University, Győr, Hungary since 1997. Now, he is an Associate Professor. He teaches computer networks, and the Linux operating system. He is a founding and also core member of the Multidisciplinary Doctoral School of Engineering Sciences, Széchenyi István University. He has been working part time for the Department of Networked Systems and Services, Budapest University of Technology and Economics, Budapest, Hungary since 2005. There he is a member of the Medianets Laboratory and he teaches computer architectures and computer networks. The area of his research includes discrete-event simulation methodology, performance analysis of computer networks and IPv6 transition technologies.

Dr. Lencse is a member of IEICE (Institute of Electronics, Information and Communication Engineers, Japan).

Marius Georgescu received his B.Sc. degree in computing and communication systems from the Faculty of Automation, Computers and Electronics, University of Craiova, Romania in 2006 and M.E. D.E. degrees from the Graduate School of Information Science, Nara Institute of Science and Technology (NAIST), Japan in 2014 and 2016 respectively.

He is currently an IP/MPLS Backbone Engineer at the Romanian ISP RCS&RDS. He is an active member of the Benchmarking Working Group (BMWG) in the IETF and his research interests include the transition from IPv4 to IPv6, Internet Engineering, and Delay Tolerant Networks.



Youki Kadobayashi received his Ph.D. degree in computer science from Osaka University, Japan, in 1997.

He is currently a Professor in the Graduate School of Information Science, Nara Institute of Science and Technology, Japan. Since 2013, he has also been working as the Rapporteur of ITU-T Q.4/17 for cybersecurity standardization. His research interests include cybersecurity, web security, and distributed systems.

Dr. Kadobayashi is a member of IEEE Communications society.