Enabling Dns64perf++ for Benchmarking the Caching Performance of DNS64 Servers

Gábor Lencse[1]

[1] Department of Networked Systems and Services, Budapest University of Technology and Economics, Budapest, Hungary

The dns64perf++ DNS64 benchmarking program is the world's first standard DNS64 benchmarking tool, which complies with the compulsory requirements of RFC 8219 on benchmarking methodology for IPv6 transition technologies including DNS64. The aim of our current effort is to enable dns64perf++ for benchmarking the caching performance of DNS64 servers, which was qualified as optional by the RFC, but can be important in practice, and thus make dns64perf++ the world's first standard DNS64 benchmarking tool that provides all the features described in the RFC. In this paper, we disclose our goals, design considerations as well as implementation decisions. We also provide a simple case study to demonstrate the operability of the new feature.

ACM CCS (2012) Classification: Software and its engineering → Software creation and management → Designing software → Software design engineering; Networks → Network services → Naming and addressing;
Networks → Network performance evaluation → Network performance analysis

Keywords: DNS64, Internet, IPv6 deployment, IPv6 transition solutions, performance analysis

1. Introduction

DNS64 [1] and NAT64 [2] are important IPv6 transition technologies, which can be used by network operators for enabling IPv6-only clients to communicate with IPv4-only servers. Performance is an important factor when selecting the implementations to be used and there is a new RFC on benchmarking methodology for IPv6 transition technologies including DNS64 servers [3]. The compulsory requirements of RFC 8219 for benchmarking DNS64 servers were satisfied by the **`dns64perf++`** measurement program [4], but the optional feature of being able to test the efficiency of the caching performance of DNS64 servers was not included [5].

As caching may significantly improve the performance of a DNS64 server, their caching performance is worth measuring. The aim of our current effort is to extend **`dns64perf++`** to be able to measure the caching performance of DNS64 servers and thus comply with all the features of RFC 8219 and therefore be the world's first standard full featured DNS64 benchmarking tool. In this paper, we disclose our goals, design considerations and implementation decisions for the extension of the test program.

We contend that **`dns64perf++`** can be a useful tool for several class of people. Researchers may use it to compare the performances of different DNS64 implementations, and investigate, how their performance scales up in the function of the number of CPU cores (as it was done in [6]). Developers of DNS64 servers may use it to check how the performance of their product improved. Network operators may compare the performance of different DNS64 implementations in order to find out, which suits their needs the best.

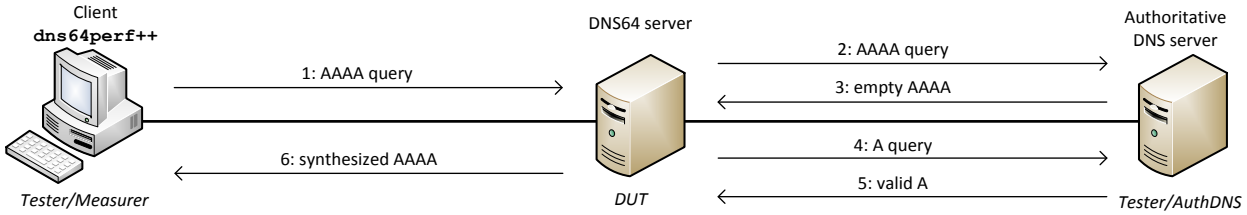

Figure 1. Test and traffic setup for benchmarking DNS64 servers [5].

The remainder of this paper is organized as follows. Section 2 contains the requirements for testing the caching performance of DNS64 servers based on RFC 8219. Section 3 recalls the operation of the **dns64perf++** program in a nutshell. Section 4 summarizes our goals and restrictions for the possible modifications of the program. Section 5 discloses our most important design considerations. Section 6 presents our implementation decisions. Section 7 considers the limitations of the extended program. Section 8 is a case study that demonstrates the operability of the new feature. Section 9 gives our conclusions.

2. Requirements for Testing Caching

2.1. Test and Traffic Setup

A detailed description of the test and traffic setup of DNS64 performance measurements was given in [5], which is open access, therefore, now we give only a short summary of it. Figure 1 shows three devices: the client, the DNS64 server and the authoritative DNS server. When neither a cache hit occurs nor a AAAA record exists, then all the following six messages are used.

1. Query for the AAAA record of a domain name
2. Query for the AAAA record of the same domain name
3. Empty AAAA record answer
4. Query for the A record of the same domain name
5. Valid A record answer
6. Synthesized AAAA record answer [3]

When there is a cache hit at the DNS64 server, then message 1 is followed by message 6 and no other DNS messages are used [3].

2.2. Requirements for the Tester

RFC 8219 requires that first, different domain names MUST[1] be used and then measurements MAY be done with domain names, 20%, 40%, 60%, 80% and 100% of which are cached. It is noted in the RFC that "ensuring a record being cached requires repeating it both *late enough* after the first query to be already resolved and be present in the cache and *early enough* to be still present in the cache" [3].

3. Operation of Dns64perf++ in a Nutshell

A detailed description of the operation of the **dns64perf++** program can be found in [5], now we give a short summary[2] of it including only the parts relevant to our topic. The program executes in two threads: one of them sends queries for AAAA records of different domain names at a specified rate and the other one receives the answers and decides about every single answer if it is arrived in time (within a given timeout) and if it contains a AAAA record. If both conditions are met, then the program qualifies the answer as "valid".

[1] In this document, the key words "MUST" and "MAY", are to be interpreted as described in [7].

[2] The text of [5] is reused throughout the summary.

For being able to perform these tasks, the sending thread stores a nanosecond precision timestamp of the sending time of each queries and, similarly, the receiving thread stores a nanosecond precision timestamp of the receiving time of the answers. The program uses a special method for matching the queries and the answers. It is done so because DNS clients use the *Transaction ID* to identify the reply[3] a of DNS server [8] and it is enough for them, but during benchmarking of DNS or DNS64 servers the query rates may be so high that the same Transaction ID is repeated within timeout time, as the Transaction ID is only 16 bits long. Therefore, **dns64perf++** uses a different solution for the identification of the replies. To understand this method, we need to dig somewhat deeper into the operation of the program. It is designed to be able to use the following potential name space:

{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* are in [000, 255].

This is an independent namespace, which is resolved to IPv4 by a local authoritative DNS server. During a particular execution of the test program, the required part of this namespace is 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 10.0.0.0/10 range means the range with 2^{22} number of elements, which can also be described as:

010-{000..063}-{000..255}-{000..255}.dns64perf.test.

We note 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.

The sent AAAA record requests, which refer to all different domain names during the compulsory DNS64 test of RFC 8219, can be unambiguously identified by the first label of the contained domain name. When a reply is received, it contains the request in the "Question" section (see [8]). The first label of the domain name is read from it, and it is used to find the corresponding query.

As for implementations details, during the generation of the queries, a *counter* is used: its value is increased from 0 to the number of queries to be sent minus one. The bits of the counter are appended to the *common prefix* of all the queries. For example if the before mentioned range of 10.0.0.0/10 is used, then the common prefix of all queries is the binary sequence of 0000101000 (encoding the decimal number 10 by the first 8 bits followed by two 0 bits) and counter may take the values from 0 up to maximum 2^{22}-1. (In practice, less elements are used, their number is specified by the user.) The counter is also used for indexing the *array of queries*, where the sending and receiving timestamps and validation information are stored. Later we will refer to it as **counter**.

4. Goals and Constraints

The aim of our current effort is to enable **dns64perf++** for benchmarking the caching performance of DNS64 servers.

However, we have another, long term goal, which results in *several constraints* for our current design. It was shown in [5] that **dns64perf++** can be used for benchmarking DNS64 servers up to 200,000 queries per second. We aim to increase its performance about one order of magnitude. We have set this goal because we expect that this would be the performance requirement of the Testers testing high performance DNS64 servers. For example, Google Public DNS server served 70 billon requests per day in 2012 [9], which is about 810,000 requests per second on average. This number is likely growing, and RFC 8219 requires about 220% query rate for the self-test of the tester [3], thus our goal is to achieve a

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

few times a million requests per second. Since **dns64perf++** uses only two threads, one for sending queries and another one for receiving the replies, we expect that this goal can be achieved easily by using *n thread pairs*. (For example 10 thread pairs would achieve 10 times higher performance than that of a single thread pair and would use the computing power of 20 cores of a 24-core CPU leaving 4 cores free for the operating system.) According to our planned high-level design, each thread pair should work independently from the other thread pairs so that our solution can scale up well. Independence requires that the data structures are multiplied: each thread pair must have their own *array of queries* to avoid locking issues, as well as each thread pair must use their own socket (bound to their own UDP port). Therefore, the restriction is, that all the changes of the source code of **dns64perf++** made for the interest of enabling it for benchmarking the caching performance of DNS64 servers, should be carefully examined, whether they hinder the parallelization of the program. We also plan to keep the original structure of the program and limit the changes to as few files as possible.

It is also one of our goals, that the test program be fine tunable, e.g. it should be able to perform measurements not only at the required levels of 0%, 20%, 40%, 60%, 80% and 100% cache hit ratios, but e.g. at 10%, 90% or 99%, too.

Finally, the program must keep its high performance, which is especially critical when it is used at high cache hit rates (resulting in high DNS64 performance).

5. Design Considerations

5.1. General Considerations

The actually achieved cache hit rate of a real life DNS64 server depends on different factors such as the repetition pattern of user requests, the cache size and the cache control algorithm of the DNS64 server. All these questions may be important when one examines the gain of caching, but they are out of scope from the viewpoint of RFC 8219, which recommends only the testing of the efficiency of caching at given cache hit rates from 20% to 100%. Therefore, the task of the benchmarking program is to ensure the required cache hit rate regardless of the internal parameters and/or behavior of the tested DNS64 server (e.g. cache size, cache control algorithm, etc.) handling the DUT as a black box.

5.2. What and How to Repeat to Achieve Cache Hits?

As RFC 8219 does not say anything about how many different domain names have to be repeated, we decided to repeat only a single one. This choice has two advantages:

- Simplicity. Both when the repeated queries are generated and when they have to be recognized. The latter will be very important is section 6.2.
- Ensures cache hits even if the cache size is very small.

If a single domain name is repeated *frequently enough* then it will be still present in the cache of the DNS64 server at any low but realistic cache size, thus the “early enough” condition can be easily satisfied. (The lowest non-zero cache hit rate to be tested is 20%, which means that every fifth domain names should be the one that is being repeated.) To satisfy the “late enough” condition, we decided to use a preliminary measurement step. It can be done by either the standard **host** Linux command or by using the **dns64perf++** program for sending a single request for the domain name intended to be loaded into the cache of the DNS64 server.

5.3. How to Identify the Replies?

Repeating domain names in queries, which is absolutely necessary to achieve cache hits, destroys the operation of the original method designed for the unambiguous identification of requests and replies. The replies of queries

```
uint32_t ip = ip_ | num_sent_;  // old code: ip_ is the common prefix, num_sent_ is the counter
// modification for testing caching begins here
if ( threshold_ && ip % modulo_ < threshold_ ) { // threshold_ is the threshold
          ip = ip_;             // use the common prefix to achieve a cache hit
}
// modification for testing caching ends here
snprintf(label, sizeof(label), dns64_addr_format_string, (ip >> 24) & 0xff, (ip >> 16) & 0xff, \
          (ip >> 8) & 0xff, ip & 0xff); // old code: the first label is generated in this way.
```

Figure 2. Code fragment: the modification of the query generation in function **`DnsTester::test()`**, source file: **`dnstester.cpp`**.

containing the same domain name can only be distinguished by their Transaction IDs (when they are different).

We decided to keep the original identification method for the non-repeated domain names, and "fall back" to the usage of Transaction IDs for the repeated ones. Though it is not trivial, the two methods for identification can be used together. We present the details among the implementation decisions (in subsection 6.2), because the knowledge of some implementation details are needed for its understanding.

6. Implementation Decisions

6.1. Program Arguments and Generation of the Queries

Several solutions are possible to inform the test program about the required proportion of the cached domain names, e.g. their proportion can be given using an additional parameter. It could be 0, 1, 2, 3, 4 and 5 to express 0%, 20%, 40%, 60%, 80% and 100% cache hit ratio, but we aimed to be able to fine tune the testing. It could also be a floating point value e.g. 0.2 for 20% and 0.99 for 99% but wanted to avoid additional floating point operations in the sending a receiving cycles. Instead, we have chosen to use two parameters because we considered that this solution better fits to our goals. These are **`modulo`** and **`threshold`**. If the value of **`threshold`** is zero then no domain names are repeated. Otherwise if condition (1) is met then instead of the value of the counter, only the appropriate number of zero bits are appended to the common prefix.

`counter % modulo < threshold` (1)

See the code fragment containing the modification in Figure 2.

We note that it is the responsibility of the user to specify relative prime numbers e.g. 5 as modulo and 1 as threshold to achieve 20% cache hit ratio (instead of using 100 and 20) in order to achieve the best possible interleaving of the cached and non-cached queries.

6.2. New Method for Matching the Replies

First, we introduce the operation of the identification method based on Transaction IDs. For simplicity, let us consider the case when 100% of the domain names are cached, thus this method can be used exclusively for all the replies. Due to the method used for generating the requests, the Transaction ID always takes the low order 16 bits of the **`counter`**. Thus, the Transaction ID could be used for indexing the array of queries if we had no more than 64k number of messages. However, the number of messages is significantly larger than that.

We have considered the usage of multiple UDP ports and sending maximum 65,535 queries pert port. This solution would require that multiple ports be kept open simultaneously and the receiver should check them in a round robin manner (using non-blocking receive function) until all the replies are received or the timeout for the lastly sent request elapsed. (As we have mentioned before, the usage of multiple threads had been reserved for increasing the performance of the benchmarking program, thus it is not an option here to start a

separate thread for each port.) We have identified several potential issues of this approach:

1. The opening of several sockets during the measurements may take unknown time and thus may cause undesirable delay between some consecutive queries.
2. The opening of all the sockets before the measurements might result in undesirable limitations regarding the maximum number of queries sent. (The namespace allows maximum 2^{32} number of queries, 2^{16} number of queries per socket can be sent, but the operating system would not let open 2^{16} number of sockets simultaneously. Although the number of queries seems to be abundant, significantly longer than 60s test at high rates may require all of them.)
3. Let us estimate the magnitude of the number of concurrently active sockets, which are to be polled by the receiver. Although the practically used timeout value is 1 second, the program should work with any reasonable timeout value, e.g. 10 seconds. If both the timeout value and the query rate are high enough it may happen that a receiving thread of the benchmarking program have to test hundreds of sockets, of which the majority of them is not resulting in receiving of packets (only still open due to large timeout value). Therefore, the implementation of the receiver may come inefficient.

In addition to the above, our final argument against this approach (and any other different solutions) is that the operation of the **`dns64perf++`** program is based on the array of queries. We contend that this data structure is fundamental for keeping the high performance of the test program, because it facilitates that only very little work has to be done during the test. Only the sending and timestamps plus two flags signaling whether there was a reply and if it contained a valid answer are stored during the test. All the processing and reporting functions are performed after the test. Therefore, we decided to keep the concept of the program. Conforming to our before mentioned constraints, we intended to make only as little changes in the source code as it was possible.

To address the 64k problem, we have introduced the array of **`counters`** (containing 64k number of elements), which is initialized in the way that the value of its *i*-th element is set to *i*. The value of the *i*-th element of the **`counters`** array is used to find the position in the array of queries, where the timestamps belonging to the cached domain name with the Transaction ID *i* are stored. Whenever an element of the **`counters`** array is used, its value is increased by 64k, thus it is ready for the next usage.

If the proportion of the cached domain names is less than 100% but higher than 0% then both identification methods must be used. How can we decide, which of them is to be used for a given reply? When a reply is received, the domain name is read from the question section of the reply. If it is not the cached domain name then the appropriate part of the corresponding IP address is used for indexing the array of queries as it was done in the original program. If it is the cached domain name then the Transaction ID is extracted from the reply. When a position in the array of queries is determined by the above described method using the Transaction ID and the array of **`counters`** then it must be checked that according to condition (1) the given position is used for storing a query with the cached or with a non-cached domain name. In the first case we have found it, thus the receiving timestamp and validation information are stored and the used element of the array of **`counters`** is increased by 64k. However, in the second case the given position of the array of queries stores the timestamps for a non-cached domain name having the same Transaction ID as the currently received query has. Therefore, the next position of the array of queries with the same Transaction ID should be checked, which is

```
// the following line is added to the variable declarations:
char cachedlabel[64]; // for testing caching: the first label of the query which is cached
//  then the cached label is produced from the common prefix (= base IP address):
snprintf(cachedlabel, sizeof(cachedlabel), dns64_addr_format_string, \
          (ip_ >> 24) & 0xff, (ip_ >> 16) & 0xff, (ip_ >> 8) & 0xff, ip_ & 0xff);
//
// several lines of the old code are unquoted here
//
// Due to testing caching, it is a bit more complicated to find the query in the array
// The old code was the following simple line:
// DnsQuery& query = tests_[(ip & (((uint64_t) 1 << (32-netmask_))-1))];
// new code begins here:
uint64_t index;
if ( !threshold_ || strcmp(label,cachedlabel) ) {
          // we are not testing caching OR NOT the critical label is found
          index = (ip & (((uint64_t) 1 << (32-netmask_))-1)); // index is from the label
}
else {    // we are testing caching AND the critical label is found
          // index should be prepared from Transaction ID and receving history
          uint16_t transactionID=answer.header_->id(); // called 'DNS Query identifier' in "dns.h"
          index = counters_[transactionID];
          while ( index % modulo_ >= threshold_ ) {
                    // this is NOT a place of a query which is cached
                    index += 65536 ; // try 64k further
          }
          counters_[transactionID] = index + 65536; // point to the next one
}
DnsQuery& query = tests_[index]; // this is the query
// this is the end of the new code
```

Figure 3 Code fragments: the most significant modifications of the processing of received queries query generation in function **`DnsTester::start()`**, source file **`dnstester.cpp`**.

located at 64k farther position. This search must be continued until condition (1) is satisfied. Then the receiving timestamp and validation information are stored and the used element of the **`counters`** array is set for the position of the next candidate with the same Transaction ID (that is the current position +64k). See the most relevant changes to the source code in Figure 3.

7. Discussion of the Limitations of our Solution

7.1. Correctness

The original method can unambiguously identify the replies of the DNS64 server, distinguishing them by the unique domain names included in the "Question" section. Testing caching inherently eliminates this solution. As Transaction IDs are only 16 bits long, they are repeated within timeout time (1 second), if the tested rate is higher than 65,536 queries per second, which may happen if a fast enough DNS64 server is being tested. Thus, it may happen that a DNS64 server does not answer a query with the cached domain name due to overload and the test program mistakenly accepts the answer of a later query for the cached domain name with the same Transaction ID arriving within timeout time. Although the reply will be falsely accounted in this case, but the reply of the later query will be missing, thus the test will fail. The other kind of slip is also possible: if a query with the cached domain name is answered after timeout, the late reply may be accepted as a valid reply of a later query for the cached domain name having the same Transaction ID. The test will still fail because the earlier query was not replied in time. Therefore, we can conclude that although some messages may be accounted mistakenly, which is the consequence of the fact that some messages are indistinguishable, the final decision will be still correct.

7.2. Performance

As for query generation, we have chosen the computationally inexpensive modulo operation

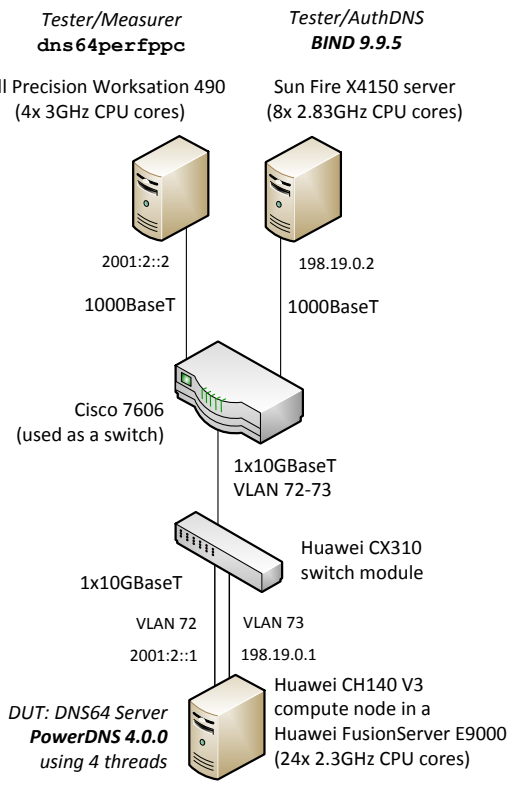

Figure 4. Topology of the test network for benchmarking the caching performance of the PowerDNS DNS64 server.

for making a decision whether the cached domain name is to be used for the current query, thus the query sending performance of the program expected to remain high.

As for receiving the queries, there is an additional string comparison of short (15 character long) strings, and a variable number of modulo and integer operations. Their number may be high, if the ratio of the required cache hits is very low, e.g. 1% or even less. As the smallest positive cache hit ratio recommended by RFC 8219 is 20%, in that case the modulo value is 5 and therefore no more than 5 executions of test (1) per reply is necessary, thus we forecast no performance problems.

We note that non-standard low cache hit rates (e.g. 10% and below) cause only small performance increase and thus are very likely out of interests. (Please refer to our measurement results at 0% and 20% cache hit rates in Table 1.) The testing of non-standard high cache hit rates (e.g. 90% and above) will not cause performance problems and they may be worth testing: the results at 80% and 100% cache hit rates are significantly differ. Someone may wish to test the performance of a DNS64 implementation e.g. at 90%, 95% or 99% cache hit rates.

8. Case Study: Demonstration of the Benchmarking of Caching Performance of DNS64 Servers

Although RFC 8219 follows the traditional benchmarking setup, which uses only two devices, the Tester and the DUT, it was elaborated in the relavant paper about benchmarking methodology for DNS64 servers [10] that the two functions of the Tester (Measurer and AuthDNS) may be implemented by two separate devices. This approach was followed in the setup of the test system. Its topology is shown in Figure 4, which also contains the CPU parameters of the computers to reflect their approximate performances. We note that the Huawei FusionServer E9000 resides in a different building than the two other computers and its compute nodes are available only through the CX310 internal switch module, the 10GBaseT port of which had to be connected to the two other computers having 1000BaseT ports, thus we had to use another element, which was actually a router used as a switch.

For the repeatability of our measurements, we briefly summarize the most important parameters of the computers.

Tester/Measurer: Dell Precision Workstation 490 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). Debian 8.6 GNU/Linux operating system with 3.16.0-4-amd64 kernel.

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. Debian 8.6 GNU/Linux operating

```
allow-from=::/0, 0.0.0.0/0
forward-zones=dns64perf.test=198.19.0.2
local-address=127.0.0.1,::1,2001:2::1
lua-dns-script=/etc/powerdns/dns64.lua
threads=4
```

Figure 5. Changes made to the **recursor.conf** configuration file of PowerDNS.

```
prefix = "2001:db8:ffff:ffff:ffff:ffff::"

function nodata ( dq )
  if dq.qtype ~= pdns.AAAA then
    return false
  end  --  only AAAA records

  dq.followupFunction = "getFakeAAAARecords"
  dq.followupPrefix = prefix
  dq.followupName = dq.qname
  return true
end
```

Figure 6. The contents of the **dns64.lua** file.

system with 3.16.0-4-amd64 kernel and BIND 9.9.5-9+deb8u7-Debian as authoritative DNS server.

DUT: Huawei FusionServer E9000, CH140 V3 compute node with two 12-core Intel Xeon E5-2670 v3 2.30GHz CPUs, 8x16GB 2133MHz DDR4 SDRAM, Two Intel Corporation 82599 10 Gigabit Dual Port Backplane Connection (rev 01). Ubuntu 16.04.2 LTS GNU/Linux operating system with 4.4.0-45-generic x86_64 kernel and PowerDNS 4.0.0-alpha2 as DNS64 server.

We used PowerDNS as DNS64 server program, because earlier experiments showed that PowerDNS scaled up better than BIND [6]. We present the changes made to its default configuration file named **recursor.conf** in Figure 5. The number of threads were limited to 4 in order to make the DUT the performance bottleneck and to avoid that the Authoritative DNS server be a performance bottleneck. The operation of the DNS64 function was described in the **dns64.lua** file as shown in Figure 6.

At the authoritative DNS server, a zone file was generated to resolve the queries for the 10.0.0.0/8 range. We included the generator script called **gen-zonefile-A.sh** in the directory of the modified source code of the **dns64perf++** program [11].

We note that an inaccuracy of the original timing algorithm of the **dns64perf++** program was discovered. The correction is only a single change (in line 49 of source file **timer.cpp**) as documented in [12]. We used the corrected version for our measurements.

We have tested all six cache hit rates recommended by RFC 8219. The duration of the measurements was 60 seconds and the timeout value was 1 second. The maximum number of processed DNS queries per second was determined by using binary search. The binary search script was executed 20 times for each cache hit rate, to receive reliable results. For the detailed explanation of the measurement method, please refer to [10]. These steps were performed by the **measure.sh** bash shell script, which is also included in [11].

Table 1. Caching performance of the PowerDNS DNS64 server as a function of cache hit rate, executed by a Huawei CH140 v3 compute node in 4 threads.

Cache hit rate (%)		**0**	**20**	**40**	**60**	**80**	**100**
Number of requests per second	median	14166	17445	22218	29997	45714	88090
	minimum	13807	17103	21759	27647	42991	87035
	maximum	14593	17689	22657	30913	49153	88677

The median as well as the minimum and maximum values were determined and they can be found in Table 1. Row 1 shows the cache hit ratio, whereas rows 2, 3 and 4 show the median, minimum and maximum values of the number of successfully serviced AAAA record requests per second (calculated from the 20 repetitions of the experiments for each cache hit ratio). The results show similar tendency to that of shown in Table 7 of [10], but now they are significantly higher due to several factors including the usage of a different DNS64 server program, higher number of working threads, faster CPU and faster memory. We plan to analyze how these factors influence the results, but this analysis is beyond the scope of our current paper. Now, our aim was to demonstrate that the modified test program works properly at higher than 65,536qps rates, in which we were successful.

9. Conclusions

We conclude that our efforts were successful in making the existing **`dns64perf++`** DNS64 benchmarking tool the world's first full functional DNS64 tester that provides all the features described in RFC 8219 including the testing of caching performance. We have demonstrated the operability of the new feature in a case study.

References

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

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

[3] M. Georgescu, L. Pislaru and G. Lencse, "Benchmarking methodology for IPv6 transition technologies", IETF RFC 8219. https://doi.org/10.17487/RFC8219

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

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

[6] G. Lencse and S. Répás, "Performance analysis and comparison of four DNS64 implementations under different free operating systems", Telecommunication Systems, vol 63, no 4, pp. 557-577. https://doi.org/10.1007/s11235-016-0142-x

[7] S. Bradner, "Key words for use in RFCs to indicate requirement levels", IETF RFC 2119, March 1997. https://doi.org/10.17487/RFC2119

[8] P. Mockapetris, "Domain names – implementation and specification", IETF RFC 1035, November 1987. https://doi.org/10.17487/RFC1035

[9] J. K. Chen, "Google public DNS: 70 billion requests a day and counting", Google Official Blog,

https://googleblog.blogspot.hu/2012/02/google-public-dns-70-billion-requests.html

[10] G. Lencse, M. Georgescu, and Y. Kadobayashi, "Benchmarking Methodology for DNS64 Servers", Computer Communications, vol. 109, no. 1, pp. 162-175, September 1, 2017. https://doi.org/10.1016/j.comcom.2017.06.004

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

[12] G. Lencse and A. Pivoda, "Checking and Increasing the Accuracy of the Dns64perf++ Measurement Tool for Benchmarking DNS64 Servers", submitted for review: *International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems*, review version is available: http://www.hit.bme.hu/~lencse/publications/IJATES2-2018-dns64perfpp-accuracy-for-review.pdf

Received: March 24, 2018

Contact address:
Gábor Lencse
Department of Networked Systems and Services
Budapest University of Technology and Economics
2 Magyar tudósok körútja
H-1117 Budapest
Hungary
e-mail: lencse@hit.bme.hu

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 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.