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

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

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

1. Introduction

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

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

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

The remainder of this paper is organized as follows. Section 2 recalls the operation of the dns64perf++ program in a nutshell. Section 3 presents our accuracy measurement method and the results, as well as the analysis of the results and the identification of the cause of the inaccuracies at high rates. Section 4 discloses our solution for the problem and the test results of the corrected timing algorithm. Section 5 considers the limitations of corrected program. Section 6 is a case study: both the original and the corrected versions of dns64perf++ are used for real measurements and the accuracies of their results are compared. Section 7 gives our conclusions.

2. Operation of Dns64perf++ in a Nutshell

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

---

1 The text of [6] is reused throughout the summary.
The *dns64perf++* program executes in two threads: one of them sends queries\(^2\) for AAAA records of different domain names at a specified rate and the other one receives the answers and decides about every single answer if it is arrived in time (within a given timeout) and if it contains an AAAA record. If both conditions are met then the program qualifies the answer as “valid”. For being able to perform these tasks, the sending thread stores a nanosecond precision timestamp of the sending time of each query and, similarly, the receiving thread stores a nanosecond precision timestamp of the receiving time of the answers.

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

\[
T_w(n+1) = \frac{N \cdot T - (t_g(n)-t_g(0))}{N-n} - T_g(n) \tag{1}
\]

where \(N\) is the total number of requests to be sent, \(t_g(n)\) denotes the timestamp when the preparation of the \(n\)-th request started and \(T_g(n)\) denotes the time it took to prepare and send the \(n\)-th request \((n\) takes the values from 0 to \(N-1\)). This way, the timing is self-correcting. We note that this method guarantees only the “global” accuracy of timing. There may be “local” inaccuracies, and they will surely occur if the request rate is high enough. Modern computer hardware support the efficiency of program execution by several solutions such as caching, branch prediction or prefetching data/instructions. Some high request rates can only be achieved after these solutions provide full benefits (program code and data are loaded into the cache, the branch predictors have already learnt the behavior of the program, etc.). Thus, a given number of requests may be sent somewhat late in the beginning of the test.

### 3. Measurements, Results and Problem Identification

The fact that *dns64perf++* dumps all its results (including all the nanosecond precision timestamps) in CSV format into the *dns64perf.csv* file, enabled us to test its accuracy without the need for purchasing highly expensive measurement devices. We have performed 60s long tests (to comply with the Internet Daft [4]) at various speeds from 50,000qps to 250,000qps with the increase of 50,000qps. We focused on the sending timestamps only. To make the huge number of results digestible, we have used a short script to count how many timestamps fall into each 100ms time window from 0s to 60s. For the repeatability of the measurements we present the most important parameters of the computer used for testing. It was a Huawei CH140 V3 compute node with Intel Xeon E5-2670 v3 2.30GHz CPUs, 8x 16GB 2133MHz DDR4 SDRAM and Ubuntu 16.04.2 LTS GNU/Linux operating system with 4.4.0-45-generic x86_64 kernel was used.

We note that in this case, *dns64perf++* was not used in a real measurement situation, but rather the timing accuracy of its AAAA record request generation was tested. Section 6 contains a case study where *dns64perf++* is used in benchmarking measurements. The result are shown in Fig. 2. Whereas the result of the 50,000qps test seem to be correct, all the others are visibly differ from the expected one. The behavior of the self-correcting timing algorithm can be very well observed on the graph, which belongs to the test at 250,000qps. For some reason, which we will soon determine, the curve starts at 24,450queries/100ms, which corresponds to 244,500qps, and the compensation is visibly too low, therefore the algorithm has to compensate too much at the end. But finally, the required 250,000qps was achieved.

### Table 1

<table>
<thead>
<tr>
<th></th>
<th>Validated of our error model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required query rate (qps)</td>
<td>50,000 100,000 150,000 200,000 250,000</td>
</tr>
<tr>
<td>Required cycle time (ms)</td>
<td>20,000 10,000 6,667 5,000 4,000</td>
</tr>
<tr>
<td>Cyc. t. in our model (ms)</td>
<td>20,090 10,090 6,757 5,090 4,090</td>
</tr>
<tr>
<td>Computed queries/100ms</td>
<td>4,978 9,911 14,799 19,646 24,450</td>
</tr>
<tr>
<td>Counted queries/100ms</td>
<td>4,979 9,913 14,805 19,651 24,450</td>
</tr>
</tbody>
</table>
should have send a request at every 4000ns. The achieved 244,500qps rate corresponds to 4090ns cycle time. It means that the program spent about 90ns more with each message than it should. We can easily check the validity of this model. Let us check two calculations. (Table 1 shows all of them.) The cycle time should be 5000ns at 200,000qps rate, and 5090ns results in 19,646 queries/100ms, which is very close to what we measured (19651). The cycle time should be 20,000ns at 50,000qps rate, and 20,090ns results in 4,975 queries/100ms, which is very close to what we measured (4,979).

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

4. Correction of the Timing Algorithm

As for the timing algorithm, it became evident that we should not try to distribute the compensation of the possible latency for the remaining testing time, but we should rather use a simple solution, where we attempt to compensate all the global latency at the current step. We calculate the waiting time before starting to prepare the \((n+1)\)-th request as follows:

\[ T_w(n+1) = t_1 + n \cdot T - t_s(n) \]  \hspace{1cm} (1)

where \(t_1\) denotes the timestamp when the preparation of the first request started and \(t_s(n)\) denotes the timestamp when the \(n\)-th request was sent. In this way, the timing error will not cumulate.

As for the modification of the source code, we limited the change for a single line of a single file (line 49 of timer.cpp, as shown in Fig. 3. We did not include this change for

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{Number of AAAA record queries sent in a 100ms long interval by the original dns64perf++ program.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{Modifications to the source code of dns64perf++ program}
\end{figure}

Line 49 of the source file "timer.cpp" was replaced as follows:

Original line 49:

```
sleep_time = interval - function_execution_time;
```

New line 49:

```
sleep_time = starttime + (n_-n)*interval_ - std::chrono::high_resolution_clock::now();
```

should not try to distribute the compensation of the possible latency for the remaining testing time, but we should rather use a simple solution, where we attempt to compensate all the global latency at the current step. We calculate the waiting time before starting to prepare the \((n+1)\)-th request as follows:

\[ T_w(n+1) = t_1 + n \cdot T - t_s(n) \]  \hspace{1cm} (1)

where \(t_1\) denotes the timestamp when the preparation of the first request started and \(t_s(n)\) denotes the timestamp when the \(n\)-th request was sent. In this way, the timing error will not cumulate.

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

We note that after the above change, the calculation of the function execution time in line 42 is used only in debug mode, otherwise it is now calculated unnecessarily, thus one wants to optimize the code, may change lines 42 and 43 to execute the calculation within the \#ifdef DEBUG macro. We did not include this change for
simplicity. Using the corrected timing algorithm, we have performed the same tests as before, and the results are displayed in Fig. 4. They show that the correction was successful, and the accuracy is now ensured at all tested query rates.

5. Discussion of the Accuracy

We would like to emphasize that the accuracy of dns64perf++ is still limited. It is a software-based generator, which is executed by a modern computer hardware under the Linux operating system and uses socket interface API functions, thus the limitations mentioned in section 5.6 of [6] are still valid.
To fully disclose the accuracy measurement results of dns64perf++, we have prepared a plot using 10ms wide cells. Fig. 5 shows the result of the same measurements, as Fig. 4, the only difference is that narrower cells were used during the processing of the results. Several spikes can be observed on the graphs of both the 200,000qps and the 100,000qps tests. We have identified the first spike of the 200,000qps graph. Its downwards pointing part belongs to the 2.05s-2.06s time window, where only 1904 requests were sent instead of the required 2000 requests. (Its reason could be e.g. the handling of an interrupt, rescheduling the thread to a different CPU core, or some other thing mentioned in section 5.6 of [6].) The upwards pointing part of the spike belongs to the 2.06s-2.07s time window, where 2097 requests were sent. This example shows that the modified program attempts to compensate for any latency as soon as possible.

As the measurement method requires the usage of 1s timeout [3], we believe that these local inaccuracies, which are visible only in Fig. 5 having 10ms wide cells but are invisible in Fig. 4 having 100ms wide cells, are satisfactorily ironed out during the 1s timeout interval and thus dns64perf++ may be used. However, we recommend the users of the program to check the nanosecond precision timestamps made available by the program in the dns64perf.csv file. Thus, by processing this file, the user may decide if the accuracy is acceptable for his/her purposes or not. (In the latter case, the measurement should be invalidated and repeated.)

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

6. Comparison during Real Measurements

In this case study, we demonstrate the real life effect of the correction of the program. As for measurement setup, we used the one called the “self-test of the tester” in RFC 8219 [4]. This test is described in more details in [3]. For this measurement, the Tester is looped back, that is the Tester/AuthDNS, is directly connected to the Tester/Measurer, leaving out the DNS64 server. This setup was chosen to be able to achieve higher rates.

The test setup is shown in Fig. 6. For the repeatability of our measurements, we briefly summarize the most important parameters of the computers used for measurements.

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

BIND was configured to resolve a full /8 size zone, that is the domain names in the dns64perf.test zone starting from 010-000-000-000 ending with 010-255-255-255 were used. When a DNS64 test is done, they are mapped to the corresponding IPv4 addresses from 10.0.0.0 to 10.255.255.255. Now they were mapped to IPv4 embedded IPv6 addresses from 2001:db8::10.0.0.0 to 2001:db8::10.255.255.255.

We note that the number of listeners and worker threads used by BIND may significantly influence its performance, therefore we also document that BIND used 8 UDP listeners and 8 worker threads.

As specified in RFC 8219 [4], 60s long tests with 0.25s timeout were used. The tests were executed 20 times and median as well as 1 and 99 percentiles were determined, where the latter two correspond to minimum and maximum as the number of tests were less than 100. Although it is not required, we have also calculated average and standard deviation, which may give further insight into the quality of the results. All the results are shown in Table 1. The results produced by the original test program, are rather poor. The difference between the maximum (53061qps) and minimum (44923qps) is 8138qps, which is about 15.91% of the median (51157qps) thus the accuracy of the measurement is questionable. Also the standard deviation is about 4.22% of the average. The results of the corrected program are significantly better. The difference between the maximum (53249qps) and minimum (50923qps) is 2326qps, which is about 4.41% of the median (52784qps) thus the accuracy of the measurement is acceptable. Also the standard deviation about 1.09% of the average.

![Fig. 6 Test setup for the self-test of the Tester.]
Having no other test program, we are unable to tell where the observed inaccuracy of the results produced by the corrected test program comes from. It may be caused by either the corrected test program or it may be an inherent property of the authoritative DNS server, which is multithreaded.

Anyway, our results in Fig. 2 and in Fig. 4 show that at higher test rates much higher differences can be expected between the quality of the results of the original and the corrected test program.

7. Conclusions

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

References