# Performance Analysis of MTD64, our Tiny Multi-Threaded DNS64 Server Implementation: Proof of Concept

Gábor Lencse

***Abstract*—In the current stage of IPv6 deployment, the combination of DNS64 and NAT64 is an important IPv6 transition technology, which can be used to enable IPv6 only clients to communicate with IPv4 only servers. In addition to the existing free software DNS64 implementations, we proposed a tiny multi-threaded one, MTD64.**

**In this paper, the performance of MTD64 is measured and compared to that of the industry standard BIND in order to check the correctness of the design concepts of MTD64, especially of the one that we use a new thread for each request. For the performance measurements, our earlier proposed `dns64perf` program is enhanced as `dns64perf2`, which one is also documented in this paper. We found that MTD64 seriously outperformed BIND and hence our design principles may be useful for the design of a high performance production class DNS64 server.**

**As an additional test, we have also examined the effect of dynamic CPU frequency scaling to the performance of the implementations.**

***Keywords*—IPv6 transition, DNS64, BIND, MTD64, Performance analysis.**

## I. Introduction

The combination of DNS64 [1] and NAT64 [2] is an appropriate solution for the problem that an IPv6 only client must communicate with an IPv4 only server. This situation occurs when an ISP (Internet Service Provider) decides to introduce IPv6 in a way that it distributes only IPv6 addresses to its subscribers.[1] However, the majority of the servers on the Internet still uses only the IPv4 protocol. Several free software [3] (also called open source [4]) DNS64 implementations exist, e.g. BIND, TOTD, Unbound, PowerDNS, and we have also contributed with a tiny multi-treaded DNS64 implementation named MTD64 [5], [6]. In our earlier papers, we have compared the performances of BIND and TOTD [7], and later BIND, TOTD, Unbound and PowerDNS [8]. The aim of our current work is to compare the performance of MTD64, our new DNS64 implementation to that of BIND, the most well-known and industry standard DNS64 implementation and to thus check if our design concepts were right.

The remainder of this paper is organized as follows. In section 2, we recall the most important performance relevant design concepts of MTD64. In section 3, we disclose our performance measurement method including the documentation of the second version of our `dns64perf` test program. In section 4, we present and discuss our results.

We note that we have given an introduction to the operation of DNS64 in our paper [6], which may be worth consulting for those not familiar with DNS64.

Manuscript received April 21, 2016, revised May 23.

G. Lencse is with the Department of Networked Systems and Services, Budapest University of Technology and Economics, Magyar tudósok körútja 2, Budapest, H-1117, e-mail: lencse@hit.bme.hu.

[1]This one can be a forward looking solution for the problem of the depletion of the public IPv4 address pool. Of course, there are several other possible solutions, e.g. the distribution of private IPv4 addresses to the clients together with the use of CGN (Carrier-grade NAT).

## II. Performance Relevant Design Decisions of MTD64

All our design considerations can be found in both [5] and [6] including e.g. to write free software. Now, we focus on those of them, which ones we consider performance relevant. The most important ones required that MTD64 should:

- be simple and therefore short (in source code)
- be fast (written in C, at most some parts in C++)
- not store the AAAA record (IPv6 address) requests in a common data structure, but start a new thread for each of them.

The first two ones are self explanatory, but the third one requires some explanation. A DNS64 server may receive many requests and because of the nature of the DNS64 service (the server must ask information from external source and thus it has to wait for the reply) it is deliberate that if we want a high performance DNS64 server then the processing of the consecutive AAAA record requests must overlap. A natural solution would have been to use e.g. three data structures:

- one for storing the new, unprocessed requests
- one for storing the requests, for which AAAA record requests were sent to the DNS system
- one for storing the requests, for which A record requests were sent to the DNS system.

The first one of them could have been a simple queue, as the incoming requests can be processed in the receiving order. However, the two other ones should support a searching method to find the matching request when a reply is received from the DNS system. Concerning the appropriate data structure, we considered in [5] that e.g. linked list, balanced or unbalanced trees could be used: “Their operations (insert, find, delete) involve programming complexity and the operations may involve significant time complexity if the data structure has high number of elements. Unfortunately there is a trade-off between the programming complexity and the speed. E.g. the operations of the linked list are simple but their time complexity is $O(n)$, where $n$ denotes the number of elements

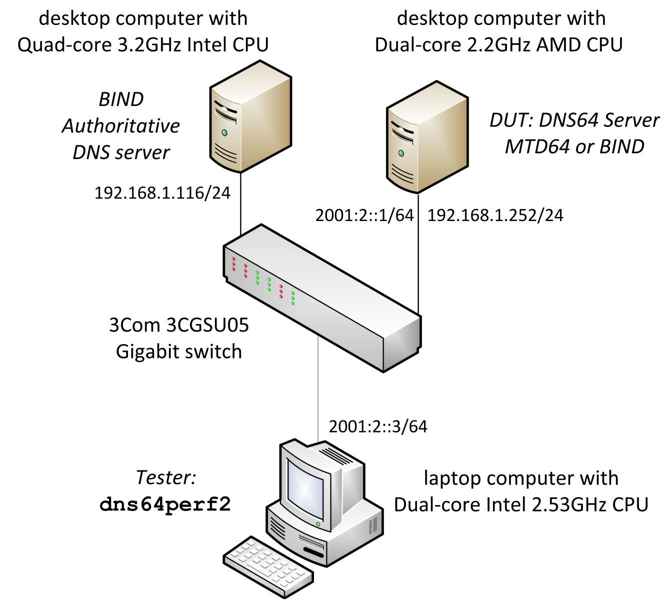

Fig. 1. DNS64 test network (See hardware and software details in subsection III-D.)

in the data structure. The time complexity of the operations of the balanced trees is *O(log n)*, but their operations require more programming work. For more information see [9] and its references." [5]

We decided not to store the requests explicitly but to start a new thread for each request. This was a risky decision and we new its potential positive and negative consequences. The positive consequences are:

- there is no need for implementing a data structure and its operations (programming simplicity, shorter code)
- all cores of the CPU can be utilized without the need of explicit programming efforts for it (may result in a good speed-up if executed by multi-core servers).

The negative consequences are:

- a multi-threaded code is much harder to debug
- starting a new thread for each request might result in too much computation costs (especially, because we used C++ for an easy thread handling)
- starting a new thread for each request makes MTD64 vulnerable to DoS attacks (attackers may exhaust the memory of the server by fake requests).

The aim of our current research is to examine the performance consequences of our design decisions.

## III. Performance Measurement Method

### A. Overview

We decided to use the DNS64 server performance measurement method that we had developed for our earlier papers [7], [8] for DNS64 server performance testing. That algorithm was later implemented in C/C++ by `dns64perf`, our DNS64 performance test program documented in [10]. Keeping the original algorithm made our current results comparable with our previous ones.

The logical topology of the test setup is shown in Fig. 1. As for *Tester*, we used the modified version[2] of `dns64perf`. The `dns64perf2` program sent a high number of AAAA record requests to the *DUT* (Device Under Test) for different domain names and received the replies. The *DUT* executed the MTD64 or the BIND[3] DNS64 server programs (the latter one served as a performance reference). As for authoritative DNS server, always BIND was used and it was executed by a desktop computer with significantly higher computing power to avoid being a bottleneck.

### B. Operation of `dns64perf2`

The details of the testing algorithm that was implemented in `dns64perf` can be found in [10]. Whereas the basics of the method were kept in `dns64perf2`, we had to do some technical modifications for having long enough test runs. This subsection contains the documentation of the operation of version 2.

*1) Testing Algorithm:* The core of the testing algorithm is very simple: AAAA record queries are sent for domain names, which ones do not have AAAA records but only A records; hence the DNS64 server needs to synthesize IPv4-embedded IPv6 addresses. An independent name space is used to be able to resolve the domain names without delay. This name space is the following: `n1-n2-n3-n4.dns64perf.test`, where n1, n2, n3 and n4 are integers from the [0, 255] interval. This name space can be easily mapped to IPv4: the corresponding IPv4 address is: n1.n2.n3.n4. However, the roles of the four numbers are very different. The first one, n1 is used as a fixed number during an execution of `dns64perf2`. Its task is to define an independent name space for each execution[4]. Each execution of `dns64perf2` contains several experiments (at least 256 and at most 65536), where an experiment contains the resolution of 256 domain names and their time is measured together. The next two numbers, n2 and n3 are used as counters (n2 is the high order one and n3 is the low order one, that is n2*256+n3 equals with the sequence number of the given experiment). Finally the last number, n4 is used within an experiment. An experiment is executed in *n* number of threads, where *n* must be a power of 2. Each thread is responsible for 256/*n* domain names, which ones are requested sequentially with no overlapping: the next one can be asked only after receiving the result of the current request. Nanosecond precision time stamps are taken at the beginning of each experiment (before starting the *n* threads) and at the end of the experiment (after all threads were joined). Their difference is calculated, converted to milliseconds, and printed by `dns64perf2`.

[2]The modification is only technical, essentially the original testing method was used, see details in subsection III-B.

[3]BIND was chosen because it is the most well-known and widespread used free software DNS server implementation. Its DNS64 performance was compared to that of three other DNS64 implementations in our former paper [8]. Using those results and the current performance comparison of MTD64 and BIND, one can also compare the performance of MTD64 to that of the DNS64 implementations included in [8].

[4]It can be useful for different purposes. In our earlier works [7], [8], we used multiple clients to generate high enough load, thus we needed independent name space for each client. In this paper, we use it in a different way.

*2) Parameters:* The program takes five or six command line arguments. In their presentation, we use the variable names from the source code [11] and the C language notation of the program arguments. The operation of the `dns64perf2` program is controlled by the following parameters:

**`a`** = `argv[1]` specifies an independent name space for each execution of `dns64perf2`, where `a` is an integer from the [0, 255] interval.

**`b`** = `argv[2]`. During an execution of the program, 256*`b` experiments are performed, where `b` is an integer from the [0, 255] interval.

**`n`** = `argv[3]`. In each experiment, `n` threads are used, where `n` must be a power of 2, e.g. 1, 2, 4, 8, etc. and each of the threads sends 256/`n` number of AAAA record requests.

**`timeout`** = `argv[4]`. The timeout value is given as a positive integer (interpreted as seconds) and it specifies the time while the program waits for a reply (typical values are 1 or 5).

**`server`** = `argv[5]` is the IPv6 address of the DNS64 server.

**`port`** = `argv[6]` is the port number on which the DNS64 server program listens (if not supplied then the default value of 53 is used).

In `dns64perf2`, parameter `b` was introduced because `dns64perf` performed only 256 experiments and their execution time proved to be too short for our measurements. The execution of the program could have been repeated multiple times, but we intended to use its continuous operation.

What is the relationship between the command line arguments of the program and the n1, n2, n3 and n4 numbers in the first label of the domain names in the AAAA record requests?

- In all requests, n1 takes the value of `a`.
- The two bytes long counter built up by n2 and n3 takes its values from the [0, 256*`b`-1] interval. (The whole measurement contains 256*`b` number of experiments, and the two bytes long counter identifies the experiments.)
- An experiment contains 256 AAAA record requests, which ones can be distinguished by the value of n4. Each one of the `n` threads sends 256/`n` number of requests.

The load generated by the program may be tuned by the value of `n`, the number of threads. It may be worth increasing `n` over the number of CPU cores of the computer used for the execution of `dns64perf2`, because the threads may be waiting for the replies.

We note that in our earlier works [7] and [8], we used different number of client computers (1, 2, 4 and 8) executing the test script to be able to exactly tune the measure of the load. In this way we were able to exactly double the measure of the offered load. This time we will increase the number of threads to tune the measure of the load. We do it for ease of testing now, and we admit that the offered load will not be exactly doubled when doubling the number of threads because all the threads are executed by the same computer with limited resources.

### C. Test Script Used for our Measurements

The tests were executed multiple times using different number of threads, to measure and compare the performances of MTD64 and BIND under different load conditions. The following test script was used:

```
#!/bin/bash
#Paramaters:
server=2001:2::1  # IPv6 address of the DNS64 server
dns64=mtd64       # type of the DNS64 server
b=10              # length of the measurement

for (( i=0; i<5; i++ ))
do
  nth=$((2**i));   # number of threads
  ssh $server ./stats $dns64 $nth &  # start dstat
  sleep 1
  ./dns64perf2 $i $b $nth 1 $server > \
    dns64perf2-results-${dns64}-$nth
  ssh $server killall dstat          # stop dstat
  sleep 5
done
```

As it can be seen from the script, the value of parameter `a` took the values 0, 1, 2, 3, 4. This way an independent name space was ensured for each execution of the `dns64perf2` program. The value of `b` was 10 that is 2560 experiments were performed in each run of `dns64perf2`. The number of threads took the values of 1, 2, 4, 8, 16 to increase the load offered by the Tester to the DUT. The timeout value was 1 second.

Though many times the DUT is considered as a black box, now we did not follow this approach, but measured the CPU utilization at the DNS64 server for a deeper understanding of the behavior of the tested DNS64 implementations. We started the `dstat` Linux command from a small `bash` script named `stats` executed by `ssh` (and stopped it by the `killall` command using `ssh`, too). The content of the `stats` script was:

```
#!/bin/bash
nice -n 10 dstat -c --output \
  dns64-stats-$1-$2.dstat > /dev/null
```

To calculate the CPU utilization, we used the idle time percentage from the output of `dstat`, and subtracted it from 100%. We did so because we considered less problem to include also the CPU utilization of some possible other processes, which were not taking part in DNS64 than leaving out the CPU utilization of some processes other than MTD64 or BIND, but doing some work for their interest (e.g. kernel processes sending and receiving packets, writing log files, etc.).

As we tested only two DNS64 implementations, they were started manually (and also their names were set manually in the test script).

### D. Hardware and Software Parameters of the Test Environment

For the repeatability of our measurements, we provide the most important hardware and software parameters of our test environment.

**Authoritative DNS Server** Desktop computer with 3.2GHz Intel Core i5-4570 CPU (4 cores, 6MB cache), 16GB 1600MHz DDR3 SDRAM, 250GB Samsumg 840 EVO SSD, Realtek RTL8111F PCI Express Gigabit Ethernet NIC; Debian GNU/Linux 8.2 operating system, 3.2.0-4-amd64 kernel,

TABLE I
DNS64 PERFORMANCE: MTD64

| 1 | Number of threads used in `dns64perf2` | | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|---|---|
| 2 | Exec. time of 256 queries (ms) | average | 61.34 | 33.23 | 28.53 | 17.90 | 15.12 |
| 3 | | std. dev. | 0.95 | 1.13 | 1.19 | 0.77 | 0.82 |
| 4 | Number of queries per second | | 4174 | 7705 | 8972 | 14300 | 16929 |
| 5 | DNS64 server CPU utilization (%) | average | 20.65 | 37.42 | 38.96 | 67.25 | 84.47 |
| 6 | | std. dev. | 1.41 | 1.77 | 0.84 | 0.67 | 0.49 |

TABLE II
DNS64 PERFORMANCE: BIND

| 1 | Number of threads used in `dns64perf2` | | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|---|---|
| 2 | Exec. time of 256 queries (ms) | average | 166.99 | 101.51 | 93.09 | 84.85 | 88.13 |
| 3 | | std. dev. | 3.57 | 8.29 | 10.49 | 9.12 | 17.79 |
| 4 | Number of queries per second | | 1533 | 2522 | 2750 | 3017 | 2905 |
| 5 | DNS64 server CPU utilization (%) | average | 50.51 | 72.37 | 68.87 | 83.18 | 86.63 |
| 6 | | std. dev. | 1.06 | 2.26 | 6.75 | 4.68 | 3.93 |

TABLE III
DNS64 PERFORMANCE: MTD64, DYNAMIC CPU FREQUENCY SCALING ENABLED

| 1 | Number of threads used in `dns64perf2` | | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|---|---|
| 2 | Exec. time of 256 queries (ms) | average | 93.27 | 51.23 | 38.56 | 23.80 | 16.87 |
| 3 | | std. dev. | 0.93 | 0.70 | 1.29 | 0.66 | 0.58 |
| 4 | Number of queries per second | | 2745 | 4997 | 6640 | 10757 | 15175 |
| 5 | DNS64 server CPU utilization (%) | average | 11.71 | 22.41 | 28.88 | 50.56 | 76.60 |
| 6 | | std. dev. | 2.93 | 3.51 | 1.18 | 1.63 | 2.02 |

TABLE IV
DNS64 PERFORMANCE: BIND, DYNAMIC CPU FREQUENCY SCALING ENABLED

| 1 | Number of threads used in `dns64perf2` | | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|---|---|
| 2 | Exec. time of 256 queries (ms) | average | 205.71 | 113.95 | 95.86 | 84.30 | 87.52 |
| 3 | | std. dev. | 8.72 | 7.68 | 8.97 | 8.22 | 17.05 |
| 4 | Number of queries per second | | 1244 | 2247 | 2670 | 3037 | 2925 |
| 5 | DNS64 server CPU utilization (%) | average | 40.38 | 70.08 | 65.94 | 83.99 | 87.40 |
| 6 | | std. dev. | 1.81 | 1.97 | 3.45 | 4.16 | 3.70 |

BIND 9.9.5-9+deb8u3-Debian

**DNS64 server** Desktop computer with 2.2GHz AMD Athlon 64 X2 Dual Core CPU 4200+ (2 cores, 512kB cache), 2GB 667MHz DDR2 SDRAM, 320GB Samsung HD321KJ HDD, nVidia CK804 Gigabit Ethernet NIC; Debian GNU/Linux 8.2 operating system, 3.2.0-4-amd64 kernel, BIND 9.9.5-9+deb8u4-Debian, MTD64 from [12] (Latest commit: January 4, 2015)

**Tester** Dell Latitude E6400 series laptop with 2.53GHz Intel Core2 Duo T9400 CPU (2 cores, 6MB cache), 4GB 800MHz DDR2 SDRAM, 250GB Samsumg 840 EVO SSD, Intel 82567LM Gigabit Ethernet NIC; Debian GNU/Linux 8.2 operating system, 3.2.0-4-amd64 kernel, `dns64perf2` from [11]

**Switch** 3CGSU05 5-port 3Com Gigabit Ethernet switch

All three computers are able to use dynamic CPU frequency scaling. First, this feature was disabled on all three computers during our main measurements to eliminate its potential effect to the results. However, as this feature is quite common, we have also examined the case when it was enabled on all three computers.

## IV. RESULTS AND DISCUSSION

### A. Presentation and Interpretation of the Results

Our DNS64 performance measurement results are presented in Table I and Table II. Both tables follow the same structure. The number of threads used in `dns64perf2` is given in the first row. (We note that the number of threads is a parameter of the Tester and not of the tested DNS64 servers. It is used to tune the intensity of the load provided by the Tester so that the DNS64 server can be tested under different load conditions.) The average and the standard deviation of the execution time of an experiment (256 queries) are shown in row 2 and row 3, respectively. In row 4, we also displayed the number of answered queries per second calculated by using the average execution time from row 2. The average and the standard deviation of DNS64 server CPU utilization can be found in row 5 and row 6, respectively.

As for the performances of the two DNS64 implementations, MTD64 seriously outperformed BIND even when a single thread was used: MTD64 processed 4174 queries per second whereas BIND could do only 1533. This difference was growing further when the number of threads was increased. When 16 threads were used, MTD64 outperformed BIND more than five times by processing 16929 queries per second

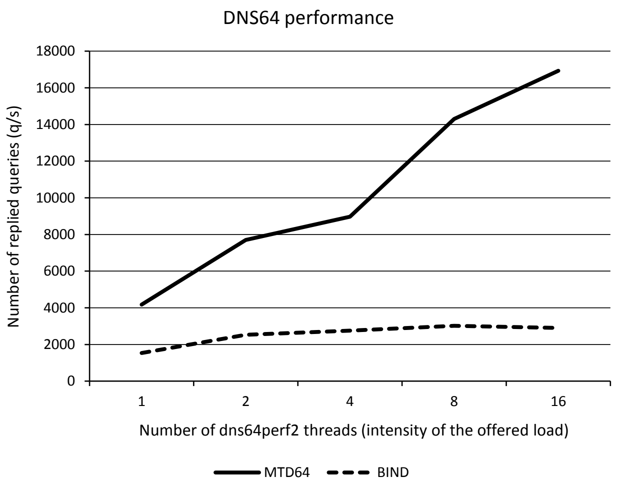

Fig. 2. DNS64 performance of MTD64 and BIND

whereas BIND could do only 2905. The explanation is clear: MTD64 used much less computing power than BIND, e.g. 20.65% vs. 50.51% at one thread or 37.42% vs. 72.37% at two threads, therefore MTD64 had still significant amount of spare CPU capacity at two threads and thus it could cope with the higher loads produced by 4, 8 and 16 threads execution of `dns64perf2`. Whereas the performance of BIND (2905qps) was somewhat less at 16 threads than at 8 threads (3017qps), MDT64 could still significantly increase its performance from 14300qps to 16929qps when the number of threads was increased from 8 to 16.

Fig. 2 provides a graphical comparison of the DNS64 performances of MTD64 and BIND as a function of the number of `dns64perf2` threads, that is, the intensity of the load.

## B. Discussion

We can lay down that the results justify the design principles of MTD64. However, we can consider this excellent performance result only as a "proof of concept". We do not recommend MTD64 to be used as a real life DNS64 server for several reasons, including:

- MTD64 was not written to be a productive MTD64 server. It is neither supported, nor maintained.
- Though our tests have shown that it can perform DNS64 server functionality properly [6], it has not undergone extensive testing and may contain bugs.
- In its current state, it is not a real server program: it is not daemonized but runs in the foreground.
- MTD64 was not tested against DoS attacks for which it is vulnerable by design.

Thus our final evaluation is that the experiment of creating MTD64, a tiny multi-threaded DNS64 server was successful, and the our principles seem to be viable for the design and implementation of a production class DNS64 server.

We note that MTD64 has been released as a free software under GPL v2 license, thus anyone can make a fork of its source code available on Github [12] and may further develop it.

We also note that MTD64 is a light-weighted software omitting many real world scenarios (e.g. controls, exceptions, etc.) thus its performance comparison with BIND is not completely fair.

## C. The Effect of Dynamic CPU Frequency Scaling

The DNS64 measurements were also performed with dynamic CPU frequency scaling enabled on all three computers. Table III and Table IV show the results.

The results produced under moderate load (using a single thread only) are significantly different from those produced when the dynamic CPU frequency scaling was disabled: MTD64 processed only 2745qps instead of 4174qps and BIND served 1244qps instead of 1533qps. Thus this result shows that disabling such mechanisms is a must when performance measurements are taken.

On the other side of the coin, the heavier the load was, the better the results of measurements with dynamic CPU frequency scaling approximated the results of the measurements without dynamic CPU frequency scaling. (As for BIND, the results for 4-16 threads are very similar; as for MTD64, the result for 16 threads are getting similar.) This observation justifies the application of dynamic CPU frequency scaling in production systems: the computers can still provide their full performance under high load conditions, and energy may be saved under lower load.

## V. Conclusion

We conclude that the design principles of MTD64, our tiny multi-threaded DNS64 server can be useful in the design of a high performance production class DNS64 server.

## 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.

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

[3] Free Software Foundation, "The free software definition", Available: http://www.gnu.org/philosophy/free-sw.en.html

[4] Open Source Initiative, "The open source definition", Available: http://opensource.org/docs/osd

[5] G. Lencse, and A. G. Soós, "Design of a tiny multi-threaded DNS64 server", in *Proc. 38th Internat. Conf. Telecommunications and Signal Processing*, Prague, 2015, pp. 27–32. DOI: 10.1109/TSP.2015.7296218

[6] G. Lencse, and A. G. Soós, "Design, implementation and testing of a tiny multi-threaded DNS64 server", *International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems*, vol. 5. no. 2, pp. 68–78, Mar. 2016, DOI: 10.11601/ijates.v5i2.129

[7] G. Lencse, and S. Répás, "Performance analysis and comparison of different DNS64 implementations for Linux, OpenBSD and FreeBSD", in *Proc. IEEE 27th Internat. Conf. Advanced Information Networking and Applications*, Barcelona, 2013, pp. 877–884. DOI: 10.1109/AINA.2013.80

[8] G. Lencse and S. Répás, "Performance analysis and comparison of four DNS64 implementations under different free operating systems", *Telecommun. Syst.*, in press, DOI: 10.1007/s11235-016-0142-x

[9] G. Lencse, "Investigation of event-set algorithms", in *Proc. 9th European Simulation Multiconference (ESM'95)*, Prague, 1995, pp. 821–825.

[10] G. Lencse, “Test program for the performance analysis of DNS64 servers”, *International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems*, vol. 4. no. 3. pp. 60–65, Sept. 2015, DOI: 10.11601/ijates.v4i3.121

[11] G. Lencse, “The `dns64perf2` DNS64 performance tester”, Available: http://www.hit.bme.hu/~lencse/dns64perf2

[12] A. G. Soós, “Multi-threaded DNS64 server” source code and documentation, Available: https://github.com/Yoso89/MTD64

**Gábor Lencse** received his MSc in electrical engineering and computer systems at the Technical University of Budapest in 1994, and his PhD in 2001.

He has been working for the Department of Telecommunications, Széchenyi István University in Győr since 1997. He teaches Computer networks and the Linux operating system. Now, he is an Associate Professor. He is responsible for the specialization of the information and communication technology of the BSc level electrical engineering education. He is a founding member and also a core member of the Multidisciplinary Doctoral School of Engineering Sciences, Széchenyi István University. The area of his research includes discrete-event simulation methodology, performance analysis of computer networks and IPv6 transition technologies. He has been working part time for the Department of Networked Systems and Services, Budapest University of Technology and Economics (the former Technical University of Budapest) since 2005. There he teaches Computer architectures and Computer networks.

Dr. Lencse is a member of IEEE, IEEE Communications Society, and the Institute of Electronics, Information and Communication Engineers (IEICE).