Performance Estimation of the Mtd64-ng DNS64 Implementation

Gábor Lencse

***Abstract*—DNS64 and NAT64 are IPv6 transition technologies enabling IPv6 only clients to communicate with IPv4 only servers. Mtd64-ng is a novel DNS64 implementation, being a successor of MTD64. In this paper, the performance of mtd64-ng is compared with that of MTD64 and BIND. The details of the measurements are fully disclosed. It is found that under heavy load conditions mtd64-ng can answer six times as many "AAAA" record requests per second than BIND. Mtd64-ng fixed two issues of MTD64 and also outperformed its predecessor by answering 46% more "AAAA" record requests per second under heavy load conditions.**

***Keywords*—BIND, DNS, DNS64, IPv6 transition, MTD64, mtd64-ng, performance comparison**

I. Introduction

The DNS64 [1] IPv6 transition technology (together with NAT64 [2]) enables clients having only IPv6 addresses to communicate with servers having only IPv4 addresses. Several free software [3] (also called open source [4]) DNS64 implementations exist. The stability and performance of BIND, TOTD, PowerDNS and Unbound were examined and compared in [5]. Two of them (BIND and PowerDNS) are multithreaded, thus they can benefit from the current multi-core CPUs, whereas the other two ones are single-threaded. A novel DNS64 implementation, namely MTD64 (Multi-Threaded DNS64) was developed at the Department of Networked Systems and Services, Budapest University of Technology and Economics [6]. The novelty of this implementation is that it starts a new thread for each request and therefore it can inherently utilize the computing power of all cores of a multi-core CPU. Its performance was compared to that of BIND and it was found that MTD64 seriously outperformed BIND concerning the number of answered "AAAA" record requests per second [7].

However, MTD64 is vulnerable to DoS (Denial of Service) attack by design: an attacker can force MTD64 to start a high number of threads, which may exhaust the memory of the computer. Therefore, MTD64 has been redesigned and re-implemented as mtd64-ng by Daniel Bakai [8]. The new design contains a thread pool of a fixed size (which is a configuration parameter), thus it spares the extra work of starting and terminating threads. In addition to that, mtd64-ng has a full object oriented design, whereas MTD64 was written mostly in C to achieve higher speed (C++ was used for convenient thread handling plus a class was used for storing the configuration parameters) [9]. MTD64 had another problem: memory leaking was experienced during its performance testing. This problem is fully eliminated in mtd64-ng by using the RAII idiom (Resource Acquisition Is Initialization) [10]. The most important design and implementation details of mtd64-ng can be found in the developer documentation of mtd64-ng [11].

The aim of this paper is to check whether mtd64-ng kept the high performance of MTD64 after its redesign and reimplementation. For this purpose, the performances of mtd64-ng, MTD64 and BIND are measured and compared using a similar test setup to that of [9].

Manuscript received July 28, 2016, revised October 1, 2016.

G. Lencse is with the Department of Networked Systems and Services, Budapest University of Technology and Economics, 2 Magyar tudósok körútja, H-1117 Budapest, Hungary (phone: +36-20-775-82-67; fax: +36-1-463-3263; e-mail: lencse@hit.bme.hu).

doi: 10.11601/ijates.v5i3.176

II. Method for Testing

A. Overview

The principles of the DNS64 testing method were laid down in [12]. In short, a high number of queries for "AAAA" records (IPv6 addresses) are sent to the DNS64 server. The requests contain different domain names which have only "A" records (IPv4 addresses) and no "AAAA" records. Therefore, the DNS64 server needs to synthesize them. It happens as follows. When the DNS64 server receives an "AAAA" record request for a particular domain name then first, it asks the normal DNS system for an "AAAA" record of the given domain name. Since it receives an empty answer, second, it sends an "A" record request to the DNS system for the same domain name. Now it receives a valid answer and it synthesizes a so-called *IPv4 embedded IPv6 address* using the prefix, which was set in its configuration file and embeds the 32 bits of the "A" record (IPv4 address). Finally it returns the synthesized IPv6 address.

The testing method has been improved over time. Originally, bash shell scripts were applied using the standard Linux host command. As its default behavior, it also requested an "MX" record [13]. Then, the request for the "MX" record was eliminated in order to focus on the "AAAA" record only [14]. Next, the shell script was partially rewritten in C to be able to provide high enough load for testing multi-core CPUs [5]. After that, the whole test program was implemented in C [15], which was named **`dns64perf`**. This program has added another factor of freedom: the user may set the number of threads to be able to tune the intensity of the load, however the number of sent queries was still

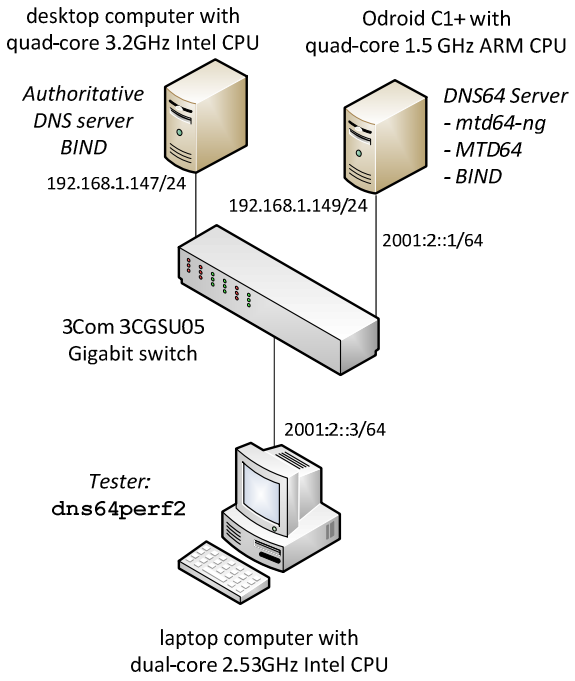

Fig. 1. Topology of the test network

fixed. Finally, this program was modified to be able to tune the number of sent queries and the second version of the program was used in our before mentioned paper for testing the performance of MTD64 [7]. The operation of the **dns64perf** program and its changes in the second version (**dns64perf2**) are well documented in [15] and in [7], respectively. Therefore, now we give only a short summary of the testing method.

B. Namespace

The **n1-n2-n3-n4.dns64perf.test** independent namespace is used, where n1, n1, n3 and n4 are integers in the [0, 255] interval. The elements of the namespace are mapped by a local authoritative DNS server to the n1.n2.n3.n3 IPv4 addresses.

C. Measurement Program Details

An *experiment* is composed of the queries for "AAAA" records of 256 different domain names. The 256 queries are sent by *n* threads, were *n* must be a power of 2 (e.g. 1, 2, 4, 8, 16, etc.) and each thread sends 256/*n* number of queries sequentially in a way that the next query can be sent after receiving the reply for the current one. Thus parameter *n* can be used to tune the intensity of the load. The execution time of an experiment is measured and printed to the standard output of the program (in milliseconds). Whereas the old version of the program (**dns64perf**) always performed exactly 256 experiments, **dns64perf2** has a further parameter: *b*, and it executes *b**256 number of experiments (to be able to perform longer tests continually).

D. Test Setup and Measurements

The topology of the test network is shown in Fig. 1. The three DNS64 implementations were executed by an Odroid C1+ single board computer (see top right) to be able to produce high enough load by the laptop computer (see at the bottom). The authoritative DNS server was a modern desktop computer to avoid being a bottleneck (see top left).

The measurements were performed using different parameters. First, the optimal value for the number of *working threads* of mtd64-ng was determined by executing a series of measurements using 1, 2, 4, 8, 16, 32, 64 or 128 working threads and generating the possible highest load by the **dns64perf2** program using 32 threads in it. Then the number of working threads of mtd64-ng was set fixed to the value that resulted in the best performance of mtd64-ng, and the performances of the three DNS64 implementations were compared under different load conditions produced by using different number of threads in the **dns64perf2** test program. Besides the execution time of the **dns64perf2** program, also the CPU utilization of the DNS64 server was measured to give more insight into the behavior of the three DNS64 implementations. Finally, MTD64 and mtd64-ng were tested against memory leaking by executing extremely long tests.

The measurements were carried out by several scripts disclosed in the next subsection.

E. Measurement Scripts

The "main" measurements were performed by the following bash script:

```
#!/bin/bash
#Paramaters:
server=2001:2::1  # IPv6 addr. of the DNS64 server
dns64=mtd64-ng    # DNS64 server (set manually)
b=4               # the length of the measurement

for (( i=0; i<6; i++ ))
do
  nth=$((2**i));  # number of threads
  ssh $server ./stats $dns64 $nth &
  sleep 1
  ./dns64perf2 $i $b $nth 1 $server > \
    odroid-${dns64}-$nth
  ssh $server killall dstat
  sleep 5
done
```

As it can be seen, variable **i** took the values from 0 to 5 that is the number of threads were: 1, 2, 4, 8, 16 and 32.

The above script started the **stats** script on the DNS64 server to log the CPU utilization using the **dstat** Linux command. The contents of the **stats** script was:

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

Before the execution of the "main" measurements, we needed to optimize the number of working threads to be set in the configuration file of mtd64-ng. For this purpose, the modified version of the first script was used:

```
#!/bin/bash
#Paramaters:
server=2001:2::1  # IPv6 addr. of the DNS64 server
dns64=mtd64-ng    # DNS64 server (set manually)
b=4               # the length of the measurement

for (( i=0; i<8; i++ ))
```

```
do
  nth=$((2**i));  # No.of mtd64-ng working threads
  ssh -l root $server ./set-mtd64-ng-wth $nth
  ssh $server ./stats $dns64 $nth &
  sleep 1
  ./dns64perf2 0 $b 32 1 $server > \
    odroid-${dns64}-$nth
  ssh $server killall dstat
  sleep 5
done
```

During the optimization process, the number of threads used in **dns64perf2** was always 32 (see its third parameter) to ensure the highest possible load. Variable **nth** (taking the values 1, 2, 4, 8, 16, 32, 64, 128) denoting the number of *working threads* was set in the configuration file of mtd64-ng by the **set-mtd64-ng-wth** script:

```
#!/bin/bash
killall mtd64-ng
cd /etc
cp mtd64-ng.conf.core mtd64-ng.conf
echo "num-threads $1" >> mtd64-ng.conf
mtd64-ng
```

The script for testing memory leaking was much simpler than the above measurement scripts. It executed only one but very long test, achieving it by using 255 for the value of **b**, and thus performing 255*256*256 "AAAA" queries. Its content was:

```
#!/bin/bash
#Paramaters:
server=2001:2::1  # IPv6 addr. of the DNS64 server
dns64=mtd64-ng    # DNS64 server (set manually)
b=255             # length of the measurement

ssh $server ./memstat $dns64 &
sleep 1
./dns64perf2 0 $b 32 1 $server > \
  odroid-${dns64}-mem
ssh $server killall pidstat
```

The above script started the **memstat** script on the DNS64 server to log the memory utilization using the **pidstat** Linux command. The contents of the **memstat** script was:

```
#!/bin/bash
nice -n 10 pidstat -h -r -p $(pidof $1) 1 | \
  grep $1 > dns64-mem-stats-$1
```

For the repeatability of our measurements, we provide configuration details in the following subsections.

F. Hardware and Software Parameters

1) 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 8.2 GNU/Linux operating system, 3.2.0-4-amd64 kernel, BIND 9.9.5-9+deb8u3-Debian

2) DNS64 server

Odroid C1+ single board computer with: 1.5GHz quad-core ARM Cortex A5 CPU (4 cores, 512kB cache), 1GB DDR3 SDRAM, 16GB Kingston micro SD card, 1000BaseTX Ethernet NIC; Ubuntu 14.04.4 LTS GNU/Linux operating system, 3.10.80-131 armv7l kernel, BIND 9.9.5-3ubuntu0.8-Ubuntu, MTD64 from [16], mtd64-ng from [8].

3) 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 8.2 GNU/Linux operating system, 3.2.0-4-amd64 kernel, **dns64perf2** from [17].

4) Switch

3CGSU05 5-port 3Com Gigabit Ethernet switch.

G. Authoritative DNS Server Configuration

1) BIND settings

The **/etc/bind/named.conf.local** file contained the following settings:

```
zone "dns64perf.test" {
        type master;
        file "/etc/bind/db.dns64perf.test";
};
```

2) Zone file

The **db.dns64perf.test** zone file was generated by the following bash script:

```
#!/bin/bash
cat > db.dns64perf.test << EOF

\$ORIGIN dns64perf.test.
\$TTL    86400
@  IN  SOA  localhost. root.localhost. (
        2016012901     ; Serial
            604800     ; Refresh
             86400     ; Retry
           2419200     ; Expire
             86400 )   ; Negative Cache TTL
;
@       IN      NS      localhost.

EOF

for a in {0..6} # to provide independent namespace
do
  for b in {0..10} # see parameter b of dns64perf2
  do
    for c in {0..255}
    do
      echo '$'GENERATE 0-255 $a-$b-$c-$ \
        IN A $a.$b.$c.$ >> db.dns64perf.test
    done
  done
done
echo "" >> db.dns64perf.test
```

The memory leaking tests required only single but much larger namespace. Therefore, the **for** cycle for **a** was omitted (the value of **a** was set to 0) and the value of **b** was running from 0 to 255, thus the size of the namespace was 256^3=16M.

H. DNS64 Server Configuration

1) BIND

The **/etc/bind/named.conf.options** file contained the following settings:

```
options {
        directory "/var/cache/bind";
        forwarders { 192.168.1.147; };
        forward only;
        dns64  2001:db8::/96 { };
```

TABLE I. DNS64 PERFORMANCE OF MTD64-NG AS A FUNCTION OF THE NUMBER OF WORKING THREADS, USING ALWAYS 32 THREADS IN DNS64PERF2

1	No. of working threads in **mtd64-ng**		1	2	4	8	16	32	64	128
2	Execution time of 256	average	82.53	53.05	37.18	29.37	27.40	36.89	36.79	36.93
3	queries (ms)	std. dev.	2.35	0.80	0.60	0.84	2.49	3.84	3.78	3.90
4	No. of served queries per sec. (q/s)		3102	4826	6885	8716	9342	6940	6959	6933

TABLE II. DNS64 PERFORMANCE OF BIND

1	Number of threads used in **dns64perf2**		1	2	4	8	16	32
2	Execution time of 256 queries (ms)	average	517.60	291.78	183.20	165.99	163.01	166.32
3		standard deviation	34.78	28.46	6.63	14.95	11.14	8.48
4	Number of served queries per second (query/sec)		495	877	1397	1542	1570	1539
5	DNS64 server CPU utilization (%)	average	34.55	52.45	82.42	90.45	92.32	93.03
6		standard deviation	8.20	2.58	1.90	3.59	2.73	2.28

TABLE III. DNS64 PERFORMANCE OF MTD64

1	Number of threads used in **dns64perf2**		1	2	4	8	16	32
2	Execution time of 256 queries (ms)	average	125.81	75.25	50.54	42.15	40.62	40.30
3		standard deviation	2.69	1.24	11.47	2.56	5.57	1.80
4	Number of served queries per second (query/sec)		2035	3402	5066	6073	6302	6353
5	DNS64 server CPU utilization (%)	average	23.43	39.28	61.44	76.58	87.22	88.45
6		standard deviation	2.76	1.12	3.03	1.19	1.95	1.39

TABLE IV. DNS64 PERFORMANCE OF MTD64-NG USING 16 WORKING THREADS

1	Number of threads used in **dns64perf2**		1	2	4	8	16	32
2	Execution time of 256 queries (ms)	average	117.20	67.65	44.80	34.90	30.74	27.56
3		standard deviation	0.65	0.97	1.21	1.07	2.29	2.62
4	Number of served queries per second (query/sec)		2184	3784	5714	7335	8328	9289
5	DNS64 server CPU utilization (%)	average	20.63	34.64	51.89	67.18	76.93	83.87
6		standard deviation	1.15	0.92	0.69	0.87	1.14	1.43

```
        dnssec-validation no;
        auth-nxdomain no
        listen-on-v6 { any; };
};
```

We note that dnssec validation was switched off for the fair comparison with the two other DNS64 implementations.

2) MTD64

The **settings.conf** file contained the following settings:

```
nameserver 192.168.1.147
dns64-prefix 2001:db8::/96
debugging no
timeout-time-sec  1
timeout-time-usec 0
resend-attempts   1
response-maxlength  512
```

3) Mtd64-ng

The **/etc/mtd64-ng.conf** file contained the following settings:

```
nameserver 192.168.1.147
dns64-prefix 2001:db8::/96
debugging no
timeout-time  1.0
resend-attempts   1
response-maxlength  512
port 53
num-threads 16 # for the "main" measurements
```

III. RESULTS

A. Number of Working Threads for Mtd64-ng

The measurement results of the experiment series for determining the optimal number of working threads of mtd64-ng are presented in Table I. The first row specifies the number of working threads used in mtd64-ng. The average and the standard deviation of the execution time of an experiment (256 queries) are shown in rows 2 and 3, respectively. The number of the replied "AAAA" record queries per second (N) is shown in row 4, which was calculated according to (1), where T denotes the average execution time of one experiment (resolution of 256 "AAAA" record queries) specified in milliseconds.

$$N = \frac{256 \frac{query}{exp} * 1000 \frac{ms}{s}}{T \frac{ms}{exp}} \quad (1)$$

As it was expected, first, the number of server requests per second increased with the number of working threads. It reached its maximum value at 16 working threads and then it showed degradation up to 128 working threads. Therefore, the number of working threads was set to 16 for the following experiments and all three DNS64 implementations were tested under the same conditions.

B. Performance Comparison

The performance measurement results are presented in identical tables for all three DNS64 implementations: Table II, Table III and Table IV contain results of BIND, MTD64 and mtd64-ng, respectively. In each table, the first row specifies the number of threads used in **`dns64perf2`**. The number of threads is used for setting higher and higher loads, but we note that doubling the number of threads does not result in exactly double intensity of the load. The average and the standard deviation of the execution time of an experiment (256 queries) are shown in rows 2 and 3, respectively. The number of served queries per second was calculated according to (1) and it is given in row 4. The average and the standard deviation of the CPU utilization of the DNS64 server computer are displayed in rows 5 and 6, respectively. (100% denotes the aggregated CPU capacity of the four cores.) We note that the CPU utilization was calculated as subtracting the idle time percentage from 100%. (See [5] for the justification of this method.)

Our most important result is that mtd64-ng has seriously outperformed BIND at any load conditions by answering 4-6 times higher number of queries than BIND (it was 2184 vs. 495 at 1 thread and 9289 vs. 1539 at 32 threads.) The performance of mtd64-ng was similar to that of MTD64 under low load (1 thread) and the difference increased with the increase of the load: mtd64-ng significantly outperformed MTD64 under high load (it was 9289 q/s vs. 6353 q/s at 32 threads).

The observation of the CPU utilization values gives a deeper understanding of the behavior of the three DNS64 implementations. BIND used visibly more computing power (34.55%) at 1 thread than MTD64 (23.43%) or mtd64-ng (20.63%). BIND could increase its performance until its CPU utilization approached 90% at 8 threads, and then neither the number of served queries nor the CPU utilization could significantly grow. MTD64 needed significantly less CPU power and its performance showed similar saturation at 16 – 32 threads (6302q/s – 6353q/s). As the CPU utilization of mtd64-ng was even lower it could significantly increase its performance even during the 16 – 32 threads change (8328q/s – 9289q/s).

C. Discussion

As for the question why both MTD64 and mtd64-ng could seriously outperform BIND, we can mention multiple reasons. First of all, our measurement method eliminates the possible performance gain of caching. Whereas this aspect of our testing method complies with the requirements of the relevant Internet Draft [18], the measurement method impairs the measured performance of DNS64 implementations that use caching as they waste a significant amount of CPU cycles with maintaining their caches (without any possible performance gain). Another factor can be that both MTD64 and mtd64-ng are simple and tiny thus their working sets [19] better fit into the L1 or L2 cache of the CPU of the DNS64 server than that of BIND, thus they can be executed faster than BIND.

Our results definitely show that mtd64-ng not only kept the high performance of MTD64 but even significantly outperformed it. We identify one of the reasons as the usage of a thread pool: thus no thread creation is necessary for the processing of every single new requests. Since the processing of "AAAA" record requests does not require much computation but only constructing and sending two requests (first, for an "AAAA" record and, after an empty answer, for an "A" record) to the authoritative DNS server and synthesizing the reply and sending it back to the client, the thread creation overhead may be significant. We also consider that mtd64-ng has a better quality source code than MTD64, which may also result in higher performance.

D. Memory Leaking and Vulnerability to DoS Attacks

As for the memory leaking tests, on the one hand, MTD64 showed so high memory leaking that the test could not be fully performed, because MTD64 was unable to respond (in time) and the **`dns64perf2`** ran out of available sockets. On the other hand, mtd64-ng showed no memory leaking at all: both VSS (virtual set size) and RSS (resident set size) were constant during the measurements. Thus mtd64-ng proved to be totally free of memory leaking.

MTD64 starts a separate thread for every single request and thus it is susceptible to the kind of DoS attacks where the memory of the DNS64 server is exhausted by sending too many "AAAA" record requests per second. Using a fixed sized thread pool, mtd64-ng is no more susceptible to this kind of DoS attack.

IV. Conclusion

We conclude that mtd64-ng, the successor of MTD64 fixed the memory leaking and vulnerability to DoS attacks issues of MTD64 and also significantly outperformed it. We plan to test and develop this promising DNS64 implementation further.

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.

[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.

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

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

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

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

[7] G. Lencse, "Performance analysis of MTD64, our tiny multi-threaded DNS64 server implementation: Proof of concept", *Internat. J. of Adv. in Telecommun., Electrotechn., Signals and Systems*, vol. 5, no 2, pp. 116–121, DOI: 10.11601/ijates.v5i2.166

[8] D. Bakai, "Mtd64-ng: A lightweight C++11 DNS64 server", source code, [Online]. Available: https://github.com/bakaid/mtd64-ng

[9] G. Lencse, A. G. Soós, "Design, implementation and testing of a tiny multi-threaded DNS64 server", *Internat. J. of Adv. in Telecommun.,*

Electrotechn., Signals and Systems, vol. 5. no. 2, pp. 68–78, DOI: 10.11601/ijates.v5i2.129

[10] B. Stroustrup, The C++ Programming Language, 3rd ed. Addison-Wesley Longman. Reading Mass. USA.

[11] D. Bakai, "mtd64-ng: A lightweight C++11 DNS64 server" developer documentation, [Online]. Available: https://github.com/bakaid/mtd64-ng/tree/master/doc

[12] G. Lencse and G. Takács, "Performance analysis of DNS64 and NAT64 solutions", *Infocommunications Journal*, vol. 4, no. 2, pp. 29–36, June 2012.

[13] G. Lencse and 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, Spain, 2013, pp. 877-884. DOI: 10.1109/AINA.2013.80

[14] G. Lencse and S. Répás, "Improving the performance and security of the TOTD DNS64 implementation", *Journal of Computer Science and Technology*, ISSN: 1666-6038, vol. 14, no. 1, pp. 9–15. Apr. 2014.

[15] G. Lencse, "Test program for the performance analysis of DNS64 servers", *Internat. J. of Adv. in Telecommun., Electrotechn., Signals and Systems*, vol. 4, no. 3, pp. 60–65. Sep. 2015. DOI: 10.11601/ijates.v4i3.121

[16] A. G. Soós, "MTD64: Multi-Threaded DNS64 server" source code, [Online]. Available: https://github.com/Yoso89/MTD64

[17] G. Lencse, "dns64perf2" source code, [Online]. Available: http://www.hit.bme.hu/~lencse/dns64perf2

[18] M. Georgescu and G. Lencse, "Benchmarking methodology for IPv6 transition technologies", Internet Draft, IETF BMWG, July 7, 2016, [Online]. Available: https://tools.ietf.org/html/draft-ietf-bmwg-ipv6-tran-tech-benchmarking-02

[19] P. J. Denning, "The working set model for program behavior", *Communications of the ACM*, vol. 11, no. 5, pp. 323–333, May 1968. DOI: 10.1145/363095.363141

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 and also 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 IEICE (Institute of Electronics, Information and Communication Engineers, Japan).