Test Program for the Performance Analysis of DNS64 Servers

Gábor Lencse

***Abstract*—In our earlier research papers, bash shell scripts using the `host` Linux command were applied for testing the performance and stability of different DNS64 server implementations. Because of their inefficiency, a small multi-threaded C/C++ program (named dns64perf) was written which can directly send DNS AAAA record queries. After the introduction to the essential theoretical background about the structure of DNS messages and TCP/IP socket interface programming, the design decisions and implementation details of our DNS64 performance test program are disclosed. The efficiency of dns64perf is compared to that of the old method using bash shell scripts. The result is convincing: dns64perf can send at least 95 times more DNS AAAA record queries per second. The source code of dns64perf is published under the GNU GPLv3 license to support the work of other researchers in the field of testing the performance of DNS64 servers.**

***Keywords*—DNS64, DNS query, Linux, performance analysis, TCP/IP socket interface programming.**

I. Introduction

The combination of DNS64 [1] and NAT64 [2] is the best available solution which can enable IPv6 only clients to communicate with IPv4 only servers. There are several DNS64 and NAT64 implementations exist and the network operators need to choose the one that best fits for their purposes. Performance, stability and security are all very important aspects.

Several papers were published concerning the performance of DNS64 and NAT64 implementations, e.g. [3]-[5], however their authors measured the common performance of the combination of a given DNS64 implementation and a given NAT64 implementation. We have shown in [6] that the performance of the NAT64 and DNS64 implementations should be measured independently.

In our previous works [6]-[8], we measured the performance of several DNS64 implementations using single core and also multi-core test devices. We used simple bash shell scripts for load generation (see details in section II), but they proved to be inefficient and therefore we decided to prepare a small special purpose C/C++ test program. We do not know of any other similar test program for the performance analysis of DNS64 servers, except for the general DNS server performance test program called DNSPerf [9] which could be used, however it uses a text file as input [10] and thus the reading and processing of the text file may be a bottleneck. (The uncompressed size of its sample query input file [11] is 248,609,309 bytes.) As for the hardware platform of the load generator for our further DNS64 performance measurements, we plan to use a cluster of SBCs (Single Board Computer), similar to the one documented in [12] but with more powerful SBCs. (We have already benchmarked some of them [13] and we are currently designing a 16 element cluster.) The SBCs use micro SD cards as storage and they produce poor transfer rate (e.g. 10-20MB/s), thus the reading of hundreds of megabytes size files from a micro SD card may be a serious bottleneck. Therefore we considered that a small special purpose DNS64 performance test program was worth writing. In this paper, we discuss the design, implementation and testing of our DNS64 performance test program.

The remainder of this paper is organized as follows. The DNS64 server performance testing algorithm to be implemented by the test program is discussed in section II. The theoretical background including the structure of the DNS messages and the basics of TCP/IP socket interface programming is summarized in section III. The design and implementation questions are detailed in section IV. The method of the comparison of the performances of the old bash scripts and of the new test program is disclosed and the results are presented and discussed in section V. Our plans for future research are outlined in section VI.

II. Algorithm for the Test Program

We were satisfied with the testing method used in [6]-[8] therefore our goal was to design and implement a test program which performs very similar tests but at least one order of magnitude faster.

The original testing method was designed to eliminate the effect of caching therefore it requested name resolution for different domain names. Originally, we used the namespace of 10-{0..10}-{0..255}-{0..255}.zonat.tilb.sze.hu which was mapped to the 10.0.0.0 – 10.10.255.255 IPv4 address only and to no IPv6 address. (It was the task of the DNS64 server to synthesize the so called *IPv4 Embedded IPv6 Address*es [14] for these domain names.)

In order to make a well tunable load generator, we used 1-8 computers, which executed the same bash scripts. The *synchronized start* of the client scripts was done by using the "Send Input to All Sessions" function of the terminal program of KDE (called **`Konsole`**) in [6] and [7] and it was done by using multicast in [8].

To ensure non-overlapping namespace for each client computers, the number of the computers were included into the domain name generation: it was used as the number right

Manuscript received June 3, 2015, revised August 29, 2015.

G. Lencse is with the Department of Telecommunications, Széchenyi István University, Győr, Hungary (phone: +36-20-775-82-67, fax: +36-96-613-646, e-mail: lencse@sze.hu)

10.11601/ijates.v4i3.121

after the beginning "10-" string. The two other numbers from 0 to 255 made it possible for each clients to generate 65536 different domain names. They were grouped into 256 experiments (256 requests were sent in each of them) and the time of each experiments were measured. Two bash scripts were used. The first one was:

```
#!/bin/bash
i=`cat /etc/hostname|grep -o .$`
rm dns64-$i.txt
for b in {0..255}
do
    /usr/bin/time -f "%E" -o dns64-$i.txt \
       -a ./dns-st-c.sh $i $b
done
```

As it can be seen from the code above, the bash script performs 256 experiments and measures the execution time of each execution of the **dns-st-c.sh** script. The latter script was changed in [8] to request only the AAAA record (without the **-t AAAA** option it also requested the MX record in [6] and [7]). The change can be justified by the fact that though requesting the MX record is the default behavior of the host command, it is not need in the vast majority of the cases (e.g. when opening a web page). Thus we decided to implement the latest version of the script in our program. It is the following:

```
#!/bin/bash
for c in {0..252..4} # that is 64 iterations
do
    host -t AAAA 10-$1-$2-$c.zonat.tilb.sze.hu &
    host -t AAAA 10-$1-$2-$((c+1)).zonat.tilb.sze.hu &
    host -t AAAA 10-$1-$2-$((c+2)).zonat.tilb.sze.hu &
    host -t AAAA 10-$1-$2-$((c+3)).zonat.tilb.sze.hu
done
```

The script was designed to issue four **host** commands concurrently to utilize the computing power of all the four cores of our client computers.

Of course, the interpretation of the bash shell script and the starting of 65536 **host** commands by each clients are rather time and computing power consuming. We need a much more efficient test program to be able to measure the performance of multi-threaded DNS64 implementations executed by current multi-core computers with several CPU cores. Replacing the two bash shell scripts by a single (but multi-threaded) C/C++ program can be at least an order of magnitude faster.

III. Theoretical Background

To be able to implement the presented algorithm in C/C++ (or just to follow the design steps) one need to know the structure of the DNS messages and the basics of TCP/IP socket interface programming.

A. The Structure of DNS Messages

We have given an in-depth introduction to the structure of DNS messages in [15], where we disclosed the design principles of MTD64, our new multi-threaded DNS64 implementation. Now, we give a very short summary of the information relevant for our test program on the basis of [15]. We mention only those fields of the DNS message here, which were actually used in our program. (For more details see [15] or the specifying RFC [16].)

1) Top level structure

The DNS query and reply messages usually travel over UDP. A DNS message is built up by five sections, these are: *Header, Question*, *Answer*, *Authority*, *Additional* sections. Whereas the length of the *Header* section is always 12 bytes, the latter four sections have variable length and some of them may be empty.

2) Header section format

The *Header* section is built up by six 16-bit fields. The first one is *Transaction ID*, which is used by the client to identify the answer of the server for different questions. The second 16-bit field is decomposed into smaller fields and bits, out of which we had to set only the *RD* (Recursion Desired) bit to ask recursive query. (The *QR* bit should be set to 0 to specify query and the also 0 value of the *OPCODE* field means "Standard Query", what we need.) The third 16-bit field is the *QDCOUNT* field which specifies the number of entries in the *Question* section.

3) Question section format

From among the latter 4 sections we deal with the *Question* section only. It contains *QDCOUNT* number of entries (usually 1). An entry is build up by the following three fields: The variable length *QNAME* field contains the domain name using special encoding (see: Domain name encoding). The *QTYPE* filed specifies the *RR* (Resource Record) type by 16-bit long binary vales. An AAAA record (IPv6 address) is denoted by the value of 0x1C. The *QCLASS* field contains the 0x01 16-bit binary value for denoting the *IN* (Internet) class.

4) Domain name encoding

The domain names stored in the *QNAME* field follow special encoding. A domain name is built up by so called *label*s separated from each other by "." characters. The labels must be no longer than 63 characters. When domain names are encoded in DNS messages, the first character gives the length of the first label then the characters of the first label follow. After that, a character stands that specifies the length of the next label and the characters of the next label follow, etc. Finally, a zero character after the last label signals the end of the domain name. We illustrated this encoding with a detailed example in [15].

B. Basics of TCP/IP Socket Interface Programming

Before a socket can be used for communication, it must be created (or opened) by the **socket()** function call. Whereas programmers must use the **bind()** function call for assigning IP address and port number information to server sockets, it can be omitted for clients sockets. Functions **sendto()** and **recvfrom()** can be used for sending and receiving UDP packets, respectively. The latter one is a so-called 'blocking' function call. To change its behavior, timeout can be set by using the **setsockopt()** function. The return value of the socket handling functions should be checked for error and the standard **perror()** function can be used for giving appropriate error messages to the user.

As for the most important data structures, the **sendto()** function takes the IP address and port number of the destination host (together with some other pieces of information) from a structure of type **struct sockaddr**. If

IPv6 is used then the actual structure is of type **`struct sockaddr_in6`** (and its pointer has to be type casted to **`struct sockaddr`**). The same type of data structures are used by **`recvfrom()`** to store the parameters of the sender of the received packet.

All the above mentioned functions and data structures are well documented in the man pages of the Linux operating system.

IV. Design and Implementation of the Test Program

A. High Level Requirements and Design Decisions

To achieve the highest possible speed, the whole functionality of the two bash shell scripts should be included into a single program. The program should be able to utilize the computing power of all the CPU cores of the computer used for its execution. Therefore, we decided that the program be multi-threaded. As for programming language, C was found to be appropriate and C++ was used only for thread handling.

The program should follow the testing method defined by the bash scripts but it should be suitable for other researchers using different test environments. Therefore an independent name space should be used for domain name resolution and the number of the threads should be a parameter.

The program should be as simple as possible for both minimizing the programming effort and making it easily understandable and modifiable by others. Therefore the program consists of a single file and takes only the essential input parameters.

Other researchers should be able to use, study its operation and modify the program as well as distribute their modified version. Therefore the test program is distributed under the GNU GPL v3 license [17].

B. Input Parameters

The program takes five input parameters:

1. The first one is the number of the executing client (stored in variable **`a`**; and is used in the domain name right after the beginning "10-" string).
2. The second one is the number of threads (stored in variable **`n`**). It MUST be a power of two (e.g. 1, 2, 4, 8, etc.), but it is not checked by the program.
3. The third one is the timeout value for the **`recvfrom()`** function measured in seconds (used when the test program waits for the reply of the DNS64 server).
4. The fourth one is the IPv6 address of the DNS64 server to be tested.
5. The fifth one is the port number at the server (where the DNS64 program listens). It was put to the last position because it is optional. If it is not specified then 53 is used as the default value.

C. Structure and Operation of the Test Program

The whole program contains three functions only:

- Function **`main()`** reads the input parameters, and executes a cycle for **`b`**=0..255. Each cycle contains an experiment. The execution time of each experiment is measured (and printed out). In each experiment, **`n`** number of threads are started (the program code of the threads is the **`dnstest()`** function), and they each are to perform **`N`**=256/**`n`** number of DNS queries for AAAA records.
- Function **`dnstest()`** is executed as a thread. It takes six parameters: **`a`**, **`b`**, **`c0`**, **`N`**, the timeout value (as a **`struct timeval *`** pointer), the IPv6 address of the DNS64 server to be tested (in the appropriate format in a **`struct addrinfo`**) and the port number at the DNS64 server. It issues **`N`** number of AAAA queries for domain names beginning with the **`10-a-b-{c0..c0+N-1}`** labels.
- Function **`dnsencode()`** encodes the domain name with the special encoding introduced in subsection III.A.4.

D. Implementation Details

1) Name space for the experiments

An independent name space was chosen so that the program could be used anywhere by anyone. The program requests the AAAA records of the **`10-a-b-c.dns64perf.test`** domain names, where the values of the **`a`**, **`b`** and **`c`** variables are printed out in decimal format.

2) Time measurement

The standard **`clock_gettime()`** function is used for time measurements. It uses nanosecond resolution, however its accuracy is not necessarily 1 nanosecond. The result is printed out is milliseconds as the usual values of the execution time of one experiments is from a couple times ten to a couple time hundred milliseconds in a typical setup. The source code can be easily modified if a different (e.g. microsecond) resolution is preferred.

3) Preparation of the DNS queries

When a new DNS query is prepared, first, its whole memory area is initialized to 0x00 by the **`memset()`** function. One may decide to omit it as the majority of the area is overwritten with new values. However, then care must be taken to set some bytes to zero, as the current code assumes that the memory area has just been initialized to zero.

The unique value for the 16-bit *Transaction ID* is composed of the values of variables **`a`** and **`b`**. The next 16-bit field was set to ask standard recursive query. The number of questions was set to 1. The *QNAME* field of the *Question* section was encoded by the before mentioned **`dnsencode()`** function. Finally, the values of the *QTYPE* and *QCLASS* fields were set as we specified them in subsection III.A.3.

4) Compilation and linking

The test program was compiled and linked using the following command line:

```
g++ -std=c++11 -O4 dns64perf.cc -lrt -lpthread \
  -o dns64perf
```

V. Testing and Results

The performance of our new test program named *dns64perf* was compared to that of the original bash scripts using identical hardware and software environment.

A. Test Network

A very simple test network was used, see Fig. 1. Both the

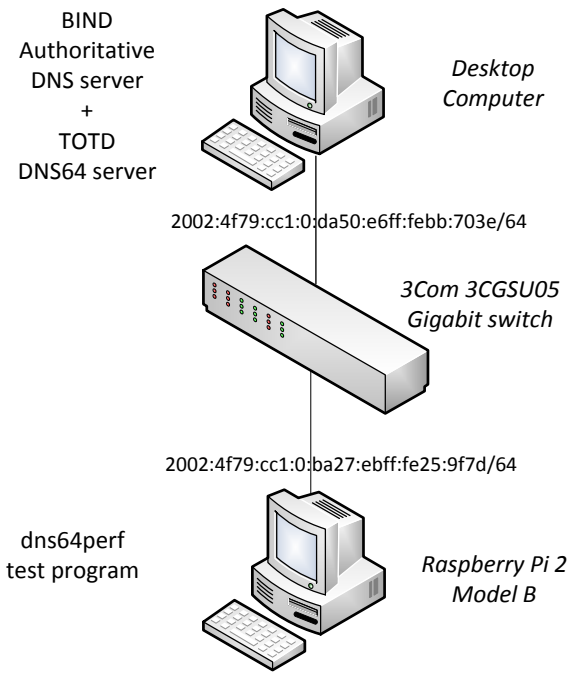

Fig. 1. DNS64 test network.

DNS64 server and authoritative DNS server were running on the same Intel desktop computer shown at the top of the figure. The test program was executed by the Raspberry Pi 2 single board computer (see at the bottom of the figure). The two computers were interconnected by a 3Com switch.
The devices had the following hardware and software configuration:
Desktop computer: 3200GHz Intel Core i5-4570 CPU (4 cores, 6MB L3 cache), 16GB 1600MHz DDR3 SDRAM, 250GB SSD; Debian GNU/Linux 7.8 operating system.
Raspberry Pi 2 Model B: 900MHz Broadcom BCM2836 CPU (4 cores, ARM Cortex-A7), 1GB RAM, 16GB microSD card (Class 10); Debian GNU/Linux 8.0 operating system.
switch: 3CGSU05 5-port 3Com Gigabit Ethernet switch

B. Testing Method

1) Authoritative DNS server

BIND 9.8.4 was used as authoritative DNS server. The zone file for the **dns64perf.test** zone was generated by the following script:

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

\$ORIGIN dns64perf.test.
\$TTL    86400
@       IN      SOA     localhost. root.localhost. (
           2015033101     ; Serial
               604800     ; Refresh
                86400     ; Retry
              2419200     ; Expire
                86400 )   ; Negative Cache TTL
;
@       IN      NS      localhost.

EOF

for a in {0..255}
do
    for b in {0..255}
    do
        echo '$'GENERATE 0-255 10-$a-$b-$ IN A \
            10.$a.$b.$ >> db.dns64perf.test
    done
done
echo "" >> db.dns64perf.test
```

Note that the **$GENERATE** directive is a shorthand. One line of this kind is equivalent to 256 traditional A record lines.
To tell BIND about the new zone, the **named.conf.local** configuration file of BIND was appended with the following lines:

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

BIND was made to listen on port 1053 using the following line in the **named.conf.options** file:

```
listen-on port 1053 { 127.0.0.1; };
```

2) DNS64 server

TOTD 1.5.2 was used as DNS64 server. Its setup was very simple. TOTD was made known that the authoritative DNS server was listening on port 1053 of **localhost** and the prefix for the generation of IPv4 embedded IPv6 addresses was set to dead:beef::/96 using the following two lines in the **totd.conf** file:

```
forwarder 127.0.0.1 port 1053
prefix dead:beef::
```

3) Measurement with the shell scripts

First, the **dns-st-c.sh** script was modified to use the new independent namespace:

```
#!/bin/bash
for c in {0..252..4} # that is 64 iterations
do
    host -t AAAA 10-$1-$2-$c.dns64perf.test &
    host -t AAAA 10-$1-$2-$((c+1)).dns64perf.test &
    host -t AAAA 10-$1-$2-$((c+2)).dns64perf.test &
    host -t AAAA 10-$1-$2-$((c+3)).dns64perf.test
done
```

Then our TOTD DNS64 server was set as the primary DNS server of the Linux system. The following line was inserted as the first line of the **/etc/resolv.conf** file at the Raspberry Pi 2 computer:

```
nameserver 2002:4f79:cc1:0:da50:e6ff:febb:703e
```

After that, the DNS64 name resolution was working correctly. It was tested by the following command:

```
host -t AAAA 10-1-1-1.dns64perf.test
10-1-1-1.dns64perf.test has IPv6 address
dead:beef::a01:101
```

Finally, the bash shell scripts were executed. The results can be found in Table 1.

4) Measurement with dns64perf

The test program was executed as the "first" client (a=1), using all the 4 cores of the CPU and the timeout was set to 1 second using the following command line:

```
./dns64perf 1 4 1 2002:4f79:cc1:0:da50:e6ff:febb:703e
```

TABLE I
DNS64 PERFORMANCE TEST RESULTS

		bash shell scripts	dns64perf 4 threads	dns64perf 1 thread
Execution time of one experiment (s)	average	4.663	0.049	0.167
	std. dev.	0.018	0.007	0.011
	maximum	4.710	0.079	0.193
Performance (requests/s)		**55**	**5234**	1535

The program was also executed using only a single thread for the purpose of comparison.

C. Results and Evaluation

Table 1 shows the results of the tests. For all three tests, average and standard deviation of the execution times of the 256 experiments were calculated and also the maximum values were selected.

The performance (the number of AAAA record requests per second) was calculated according to (1).

$$N_{AAAA} = \frac{256}{t_E} \quad (1)$$

Where N_{AAAA} and t_E denote the number of AAAA record requests per second and the average execution time of an experiment (that is sending of 256 AAAA record requests), respectively.

The results show that dns64perf (using 4 threads) could perform an average of 5234 AAAA record queries per second, whereas the shell scripts could do only 55. This means that our programming efforts resulted in a 95 times speed-up.

It can also be observed that the standard deviation of the execution time was as low as 0.018s (which is less than 0.4% of the 4.663s average value) using the bash scripts. But it became relatively much higher using dns64perf: its 0.007s value is more than 14% of the 0.049s average. And also the maximum value of the execution time (0.079s) is now significantly higher than the average. We consider that it was caused by the way the threads are used: they always have to wait for the latest finishing one before the experiment can be completed. We checked our hypothesis with the results of the single thread execution. In this case, the 0.011s standard deviation value is only 6.6% of the 0.167s average. And also the maximum value of the execution time (0.193s) is much closer to the average.

D. Discussion of the Results

We have also checked the CPU utilization[1] of both the desktop computer and the Raspberry Pi 2 using the **`top`**[2] Linux command. These values were observed by human eyes only, and they were even fluctuating, therefore the following numbers are to be interpreted as order of magnitude estimations. They will be used for qualitative analysis only, to find out the limits of the testing environment, that is: which component of the system limited the results?

When the bash shell scripts were used for testing, both BIND and TOTD used only about 1-2% of the computing power of a CPU core of the desktop computer, while the CPU idle time was only 5-6% (of the computing power of all 4 cores) at the Raspberry Pi 2 (that is, it was nearly fully utilized). Therefore it is clear that *the performance of the client limited the performance of the system*.

When dns64perf used 4 threads, TOTD used 90-95% of a CPU core of the desktop computer (BIND was under 40%, so it was not the bottleneck) while the Raspberry Pi 2 had more than 80% idle time. (The network could not be the bottleneck: DNS AAAA queries and answers were about 100-110 bytes and 210-220 bytes long, respectively. Let us consider the longer ones: about 5300 answers being each 220 bytes long would result in less than 10 Mbit per second whereas Raspberry Pi 2 has Fast Ethernet NIC.) Therefore, the *bottleneck was the single threaded TOTD DNS64 implementation*. We have also checked the CPU utilization value of TOTD when dns64perf used only one thread: TOTD used 45-47% of the computing power of a CPU core. (And it was 77-82% with dns64perf using 2 threads.) Thus we can clearly state that *dns64perf can perform significantly better*!

On the one hand it would be interesting to measure how many times dns64perf is faster than the method using bash shell scripts, but on the other hand the result would be useful for being proud of it only and nothing else. The performance ratio which has a real significance in DNS64 testing, is the following: *dns64perf executed by a single board computer can produce enough load to test the performance of a DNS64 server implementation executed by the CPU of a modern PC*. This result opens up the possibility of testing modern servers (e.g. with 16, 32 or even more cores) using dns64perf executed by a cluster of SBCs. This can be a very much cost effective solution.

VI. DIRECTIONS OF OUR FUTURE RESEARCH

A. Testing with an SBC cluster

Our next step is to build a 16 element cluster of ODROID C1+ [18] SBCs for load generation. We plan to use dns64perf to test the same four DNS64 implementations which we tested in [8], but now we will be able to test their performance up to 16 cores (instead of up to 4 cores).

B. Plans for Benchmarking RFC

Benchmarking methodology for network interconnect devices was described in RFC 2544. It was expanded to address some IPv6 specificities in RFC 5180. However, it is stated there, that IPv6 transition mechanisms are outside the scope of the RFC.

Efforts were made to define benchmarking methodology for IPv6 transition technologies [19]. For more details of the measurements, see [20]. Marius Georgescu (the first author of [20]) has invited the author of this paper to take part in the further development of the draft RFC [19], to cover the testing methodology of DNS64 servers, too.

[1] It was done not during the above measurements because it could have influenced the execution speed of the programs, but during repeated ones.

[2] This command gives both a summary of the CPU states (e.g. user, system, idle, wait, etc.) and a list of the most resource consuming processes displaying their percentage of CPU and memory consumption. However, the interpretation of 100 percent is different in the two cases. In the CPU summary, 100% means all the computing power of the existing CPU cores. In the process list, 100% means the computing power of a single core. Therefore we will always specify, how "100%" is to be interpreted.

VII. Conclusion

We conclude that our efforts to make an efficient test program for the performance analysis of DNS64 servers were successful. Our new test program, dns64perf can perform at least 95 times more AAAA record queries per second than the old bash shell scripts could, but we have shown that its performance is even higher than that. We hope that dns64perf may be a useful testing tool for many researchers interested in the performance analysis of different DNS64 server implementations. The source code of the program is available under the GNU General Public License v3 from [21].

Acknowledgment

The author thanks Gábor Horváth, Dept. of Networked Systems and Services, Budapest University of Technology for lending the Raspberry Pi 2 single board computer.

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, April 2011. ISSN: 2070-1721 (RFC 6147)

[2] M. Bagnulo, P. Matthews and I. Beijnum, "Stateful NAT64: Network address and protocol translation from IPv6 clients to IPv4 servers", IETF, April 2011. ISSN: 2070-1721 (RFC 6146)

[3] K. J. O. Llanto and W. E. S. Yu, "Performance of NAT64 versus NAT44 in the context of IPv6 migration", in *Proc. International MultiConference of Engineers and Compuer Scientists 2012 (IMECS 2012*), Hong Kong, March 14-16, 2012, vol. I, pp. 638-645.

[4] C. P. Monte et al, "Implementation and evaluation of protocols translating methods for IPv4 to IPv6 transition", *Journal of Computer Science & Technology*, ISSN: 1666-6038, vol. 12, no. 2, (August, 2012). pp. 64-70.

[5] S. Yu, B. E. Carpenter, "Measuring IPv4 – IPv6 translation techniques", Technical Report 2012-001, Department of Computer Science, The University of Auckland, January 2012.

[6] 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

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

[8] G. Lencse, S. Répás, "Performance analysis and comparison of four DNS64 implementations under different free operating systems", unpublished.

[9] Nominum, "Measurement tools: DNSPerf and ResPerf downloads", https://nominum.com/measurement-tools/

[10] die.net, "dnsperf(1) – Linux man page", http://linux.die.net/man/1/dnsperf

[11] Nominum, "Sample query input file", ftp://ftp.nominum.com/pub/nominum/dnsperf/data/queryfile-example-10million-201202.gz

[12] S. J. Cox, J. T. Cox, R. P. Boardman, S. J. Johnston, M. Scott, N. S. O'Brien, "Iridis-pi: a low-cost, compact demonstration cluster", *Cluster Computing*, vol. 17, no. 2, June 2014, pp. 349-358. DOI: 10.1007/s10586-013-0282-7

[13] G. Lencse and S. Répás, "Benchmarking Single Board Computers for Building a Mini Supercomputer for Simulation", *38th Internat. Conf. Telecomm. and Signal Proc. (TSP 2015)*, Prague, Czech Republic, July 9-11, 2015, Brno University of Technology, pp. 246-251.

[14] C. Bao, C. Huitema, M. Bagnulo, M Boucadair and X. Li, "IPv6 addressing of IPv4/IPv6 translators", IETF RFC 6052, 2010.

[15] G. Lencse, A. G. Soós, "Design of a tiny multi-threaded DNS64 server", in *Proc. 38th Internat. Conf. Telecomm. and Signal Proc. (TSP 2015)*, Prague, Czech Republic, 2015, Brno University of Technology, pp. 27-32.

[16] P. Mockapetris, "Domain names – implementation and specification", IETF, November 1987. (RFC 1035)

[17] Free Software Foundation, *GNU General Public License*, Version 3, June 29, 2007, http://www.gnu.org/licenses/gpl-3.0.en.html

[18] Hardkernel, *Odroid C1+*, (product description of the manufacturer), http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143703355573

[19] M. Georgescu, "Benchmarking Methodology for IPv6 Transition Technologies", IETF 93, Prague, Czech Republic, slides available: https://www.ietf.org/proceedings/93/slides/slides-93-bmwg-6.pdf

[20] M. Georgescu, H. Hazeyama, Y. Kobayashi, S. Yamaguchi, "Empirical analysis of IPv6 transition technologies using the IPv6 Network Evaluation Testbed", *EAI Endorsed Transactions on Industrial Networks and Intelligent Systems*, vol 2, no 2, e1, (2015), DOI: 10.4108/inis.2.2.e1

[21] G. Lencse, "dns64perf source code", http://ipv6.tilb.sze.hu/dns64perf/

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, Computer architectures, IP-based telecommunication systems 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 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 the Institute of Electronics, Information and Communication Engineers (IEICE).