CIT. Journal of Computing and Information Technology, Vol. 25, No. 2, June 2017, 91–102
doi: 10.20532/cit.2017.1003419

Design, Implementation and Performance Estimation of mtd64-ng, a New Tiny DNS64 Proxy

Gábor Lencse and Dániel Bakai

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

In the current phase of the IPv6 transition, it is a typical situation that IPv6-only clients should be enabled to communicate with IPv4-only servers. The DNS64+NAT64 tool suite is an excellent solution to this problem. Although several free software DNS64 implementations exist, we point out that there is room for further high performance and computation efficient multithreaded DNS64 implementations. MTD64 was designed to be able to utilize several CPU cores. Whereas MTD64 outperformed BIND more than five times, two critical issues (memory leaking and potential vulnerability to DoS attacks) were identified. Therefore MTD64 was redesigned under a new name: mtd64-ng (not capitalized). This paper is about the design, implementation and initial performance estimation of mtd64-ng. The usage of object oriented decomposition and the RAII (Resource Acquisition Is Initialization) idiom ensures that raw, sensitive resources (e.g. memory, sockets) are always released and it greatly simplifies exception handling. Using the new features of the C++11 standard enabled us to write more efficient and better readable code. The performance of mtd64-ng is compared to that of BIND and MTD64 and it is found that mtd64-ng outperforms even its predecessor, MTD64.

ACM CCS (2012) Classification: Software and its engineering → Software creation and management → Designing software → Software design engineering

Networks → Network services → Naming and addressing

Networks → Network performance evaluation → Network performance analysis

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

1. Introduction

In the current phase of the IPv6 transition, it is a typical situation that IPv6-only clients, the number of which is increasing due to the depletion of the public IPv4 address pool, should be enabled to communicate with IPv4-only servers, which are in majority on the Internet today. The DNS64+NAT64 tool suite [1] is an excellent solution to this problem. There are several free software [2] (also called open source [3]) DNS64 [4] implementations and four of them were compared in [5]. It was found that two of them (TOTD and Unbound) are single threaded and the other two (BIND and PowerDNS) are multithreaded. Even though the single threaded ones may not benefit from the current trend of multi-core CPU design, and the performances of the tested implementations were compared up to four CPU cores, the single threaded Unbound showed the best performance among them in terms of served "AAAA" record requests per second. (As for the conditions of the measurements, worse case tests were performed: the requests were all different to eliminate the effect of caching and none of the requested domain names had "AAAA" records, thus the DNS64 implementations had to synthesize IPv4 embedded IPv6 addresses [6] using the "A" records. These conditions comply with the relevant Internet Draft [7]).

Therefore, we believe that there is room for high performance and computation efficient multithreaded DNS64 implementations.

MTD64 was designed to be able to utilize several CPU cores [8]. Whereas the experience was successful, as MTD64 outperformed BIND more than five times [9], two critical issues were identified:

- Memory leaking was experienced during the performance measurements.
- The design decision of starting a separate thread for each request made MTD64 potentially vulnerable of DoS attacks.

Hence, we decided to fundamentally redesign MTD64 under a new name: mtd64-ng (not capitalized). This paper is about the design, implementation and initial performance estimation of mtd64-ng.

The remainder of this paper is organized as follows. In Section 2, our design decisions are disclosed. In Section 3, the implementation questions are discussed. In Section 4, the testing method, the test environment and performance measurement results are presented and discussed. In Section 5, our plans for future research and development are disclosed. Section 6 concludes our work.

This topic was identified as being of importance to the development of DNS64 server implementations.

2. Design Decisions

2.1. Design Principles

The design principles of MTD64 [8] were basically kept, but we reinterpreted or further developed them as shown in Table 1.

The usage of object oriented decomposition and the RAII (Resource Acquisition Is Initialization) idiom ensures that raw, sensitive resources (e.g. memory, sockets) are always released and it greatly simplifies exception handling. Using the new features of the C++11 standard enabled us to write more efficient and better readable code. Move semantics help to avoid unnecessary copies, lambda expressions and initializer lists help achieving better code readability. The built-in thread, mutex, condition_variable and atomic classes and templates provide a standardized and cross-platform interface for multithreaded programming.

GPL was not a choice but a must, because we decided to reuse certain source code fragments of MTD64.

2.2. High Level Design Decisions

Except for the two critical issues (memory leaking and potential vulnerability to DoS attacks) we were satisfied with the original design of MTD64 and kept the majority of the design decisions. Now we mention them only shortly, for the details please refer to subsection III.B of [8].

2.2.1 Decisions that Were Kept

Similarly to MTD64 (and TOTD), mtd64-ng acts as a forwarder. The capability of the recursion may be added later if necessary.

In the same way, we omitted caching. We consider caching useful and plan to add it in the next version, but now we focused on the essential functionalities of a DNS64 server.

Table 1. Design principles of MTD64 and mtd64-ng.

MTD64	**mtd64-ng**
be simple and therefore short (in source code)	kept
be fast (written in C, at most some parts in C++)	be fast and therefore written in C++11
be extensible (well structured and well documented)	improved: achieve better code quality by object oriented decomposition
be convenient and flexible in configuration	kept, only slight changes in configuration
be free software under GPL or BSD license	kept, and the previous GPL license was also kept
	be free of memory leaking (achieved by the RAII idiom)

The configuration file format and the configuration keywords were also kept, some slight changes were made (one new parameter was added and another setting was made more logical, see the details in subsection 3.3).

The logging by syslog was also kept and clarified.

2.2.2 Decisions that Were Changed

Though our performance measurements showed that our previous decision to start a new thread for each requests did not result in serious performance penalty [9], now we have chosen a different solution to avoid the potential vulnerability to DoS attacks. Instead, we used a thread pool, the size of which is a configuration parameter. In this way, the threads are reused, which may be beneficial concerning the performance, too.

MTD64 was written mostly in C for performance considerations. C++ was used for thread handling plus a single class was used for storing and retrieving configuration parameters. Now C++ was chosen mainly to achieve better code quality and to make the implementation extensible in several ways, but we contend that our new code written in C++11 is also very fast.

Finally, MTD64 is not a decent server program because when it is started it does not daemonize, but runs in the foreground. This weakness was also corrected in mtd64-ng.

2.2.3 FakeDNS

The most current version (1.1.0) of the mtd64-ng source code on GitHub [10] contains the experimental FakeDNS program. Its purpose is to replace the authoritative DNS server during the benchmarking of DNS64 implementations. Its operation, in a nutshell, is the following: it does not use a zone file but it calculates the IPv4 address from the information contained in the first label of the domain names. This operation is made possible by the method used for generating different domain names systematically [11]. FakeDNS reuses some of the code base of mtd64-ng. The easy implementation of FakeDNS is a justification of the object oriented design and implementation of mtd64-ng.

We note that FakeDNS is still very experimental and may be a subject of significant changes. In this paper, we focus on the design of version 1.0.0 of mtd64-ng, which has not included FakeDNS yet.

2.3. Important Design Details

We kept four out of five design decisions disclosed in section III.C of [8], that is:

1. Multiple authoritative DNS servers may be set and two selections modes are supported: random and round robin. (Random chooses a different authoritative DNS server from the list for every single request, whereas round robin chooses the next one from the list if the current one does not reply on time.)
2. Larger than 512 byte UDP message size may be enabled by a configuration option.
3. IPv6 transport protocol is used between the clients and mtd64-ng, and IPv4 is used between mtd64-ng and the authoritative DNS server.
4. The optional parallel request for "A" and "AAAA" resource records (mentioned in subsection 5.1.8 of [4]) is not supported.

Whereas MTD64 assembles its answer by moving as large as possible chunks of the reply message from the authoritative DNS server, mtd64-ng type casts the memory area of the reply from the authoritative DNS server to the appropriate class and assembles its own reply using the appropriate fields in the natural object oriented way. (Please refer to subsection 3.1.2 for more details.)

3. Implementation

3.1. Classes

Mtd64-ng is a genuine object oriented program, its architecture contains several classes. Now, we describe them. For the readers not familiar with the DNS message format, we recommend the parallel reading of Subsection II.A of [8] containing all the necessary information about the structure and field names of the DNS messages.

3.1.1 ThreadPool, WorkerThread

ThreadPool is a generic thread pool class written in C++11 using the built-in thread classes. Its constructor starts the given number of worker threads (using the WorkerThread functors to provide the main loop for the threads). An std::deque queues the tasks and an std::condition_variable is used to signal available tasks to sleeping worker threads.

Tasks are moved into the queue using an std::function template. This makes it possible to add functions, functors and lambda expressions to the queue.

3.1.2 DNSHeader, DNSLabel, DNSQName, DNSQuestion, DNSResource, DNSPacket

These classes represent different parts of the DNS packet. DNSHeader represents the header of a DNS packet. It is not constructed, but casted on the raw data stream, making it more efficient. Polymorphic setter and getter functions perform the necessary bit masking and network byte order vs. host byte order conversions.

DNSQName (DNS Query Name) aggregates DNSLabels. DNSQuestions and DNSResources contain a DNSQName (besides other standard fields).

DNSPacket aggregates one DNSHeader, plus a given number of DNSQuestions and DNSResources. Setter and getter functions perform all the necessary conversions, including resizing the packet (and moving all the Questions, Resources and Labels) when the rdata field of a DNSQName or DNSResource is changed, which is an essential operation in synthesizing an *IPv4 embedded IPv6 address* ("AAAA" record) from IPv4 address ("A" record).

3.1.3 DNSSource, DNSClient

The DNSSource interface provides an interface to prepare and send DNS query packets to an authoritative DNS server and to receive answers. Implementing classes can use their own strategies to do that (forwarding or recursing, caching or not).

The DNSClient implements the DNSSource interface and acts as a non-caching forwarder to resolve DNS queries. This design makes it easy to implement recursion or caching later on.

3.1.4 Query

The Query class implements a DNS query. It is a functor which can be supplied to a ThreadPool as a task. It stores a pointer to the raw packet data and performs the business logic using the previous helper classes.

3.1.5 Server

The Server class implements the DNS64 server. It loads and stores the configuration, starts the thread pool, opens the sockets, then receives and converts DNS query packets into Query objects and stores them in a queue to be executed by the thread pool.

3.2. Flow of a Query

In this section we describe the generic flow of a query through the mtd64-ng server:

1. The Server receives a UDP packet on the configured port from a Client.
2. The Server creates a Query object from the packet and places it in the queue.
3. When there is an available worker thread, the Query starts to execute.
4. The Query objects determines whether the packet really is a DNS Query. If so, then it forwards the query to one of the configured DNS servers using the configured selection mode.
5. The Query object receives the answer of the DNS server. If it fails (timeout occurs at timeout-time), then Query object tries another DNS server, at most resend-attempts time.
6. The Query object determines whether a synthesis action is needed. Synthesis is required if: the question in the query is for an "AAAA" record AND the domain exists AND there is no "AAAA" record in the answer section.
7. If a synthesis is not needed, the answer is sent to the Client.

8. If a synthesis is needed, the Query class synthesizes the answer using the DNS packet manipulator classes and the Server sends the synthesized answer to the Client.

3.3. Configuration File Changes

Because of using a thread pool, its size was added as a new configuration parameter: num-threads. This is the number of the *working threads*, which process the requests from the clients. The size of the tread pool proves to be an important parameter, which significantly influences the performance of mtd64-ng. Please see further details in subsection 4.3.

The other change is a small clarification only. For setting the timeout time regarding the authoritative DNS servers, MTD64 uses two configuration parameters of *integer* type: timeout-time-sec and timeout-time-usec. For setting a timeout of 1.35s they should be set to 1 and 350000, respectively. This was found to be somewhat cumbersome, so they were replaced by a single parameter of *double* type, namely timeout-time.

All other parameters were kept unchanged.

4. Initial Performance Estimation

The full performance analysis of mtd64-ng should include its performance measurement in various hardware and software environments (including both different architecture CPUs and several operating systems). It could be an effort like the one documented in [5]. As the performance estimation of mtd64-ng is only one of the goals of this paper (besides the documentation of design and implementation of mtd64-ng), we do no perform such a detailed analysis. Rather, the novelty of our performance estimation method is its full conformance to the relevant Internet Draft [7].

4.1. Performance Estimation Method

The method for benchmarking DNS64 servers is described in Section 9 of [7]. Its test and traffic setup uses only two devices: the Tester and the DUT (Device Under Test), following the test setup of [12]. However, in DNS64 testing, the Tester plays two separate roles:

1. It sends "AAAA" record requests for different (systematically generated) domain names at a predefined rate to the DUT. It receives the replies from the DUT, and decides whether they are in time (arrived within timeout) as well as if they contain a valid answer. It counts the number of in time valid answers.
2. It provides authoritative DNS server functionality for the DUT as follows: it sends empty answers for all the "AAAA" record requests and valid answers for the "A" record requests.

The DUT executes the examined DNS64 implementation, which is illustrated in Figure 1. The messages from 1 to 6 can be followed in the figure. It receives "AAAA" record requests (1) from the Tester, asks first for "AAAA" records (2), and after the empty reply (3), for "A" records (4) for the same domain name, then synthesizes the *IPv4 embedded IPv6 address* using the received valid "A" record (5) and finally, returns the synthesized "AAAA" record (6).

The Internet Draft requires that the Tester must send the "AAAA" record requests at a predefined rate and it must decide if it received valid answers for all of them from the DUT on time. In practice, binary search is used to determine the highest rate at which the DUT can reply to all the queries with valid answers in time.

Up to now, there is only one Internet Draft compliant DNS64 performance measurements tool available: dns64perf++, documented in [11] and downloadable from Github [13]. It performs a single measurement (at a required rate and during the required duration) and is to be called from a program (e.g. a bash shell script) which performs the binary search.

The Internet Draft is required to run 60 seconds long measurements in each step of the binary search and to perform the binary search 20 times, thus producing 20 results, and to summarize the results by calculating the median, as well as the 1st and 99th percentiles which correspond to the minimum and maximum values of the 20 results.

The Internet Draft is also required to perform a so called self-test to ensure that the tester itself is not a bottleneck, see subsection 9.2.1 of [7].

4.2. Test Environment

4.2.1. Test Setup

The topology of the DNS64 test network is shown in Figure 1. Though the Internet Draft uses only two devices (Tester and DUT), we have used two different physical devices for the two functionalities of the Tester. Tester/Measurer was a laptop for the execution of the dns64perf++ program, whereas Tester/AuthDNS was a desktop computer for the execution of the authoritative DNS server. The DUT was an Odroid C1+ SBC (Single Board Computer). We justify our hardware selections after their detailed specification in subsection 4.2.3.

4.2.2. Measurements and Parameters

To determine the performance of mtd64-ng, we needed to know what size thread pool should be used. For this reason, a series of measurements was performed increasing the size of the thread pool from 1 to 12. Later on, another series of measurements was performed using larger thread pools.

Finally we also tested BIND and MTD64 for comparison purposes. We note that BIND used 4 listeners and 4 working threads. MTD64 used one listener and as many working threads as there were requests under processing.

The value of the timeout parameter of dns64perf++ was always set to 1 second. This choice is justified in [14]. Thus the timeout value for the self-test was set to 0.25 second conforming to subsection 9.2.1 of [7].

4.2.3. Hardware and Software

For the repeatability of our measurements, we present the hardware and software parameters of the devices of our test environment.

Tester/Measurer: Dell Latitude E6400 series laptop with 2.53GHz Intel Core2 Duo T9400 CPU, 4GB 800MHz DDR2 SDRAM, 250GB Samsung 840 EVO SSD, Intel 82567LM Gigabit Ethernet NIC; Debian 8.4 GNU/Linux operating system, 3.2.0-4-amd64 kernel, dns64perf++ from [13].

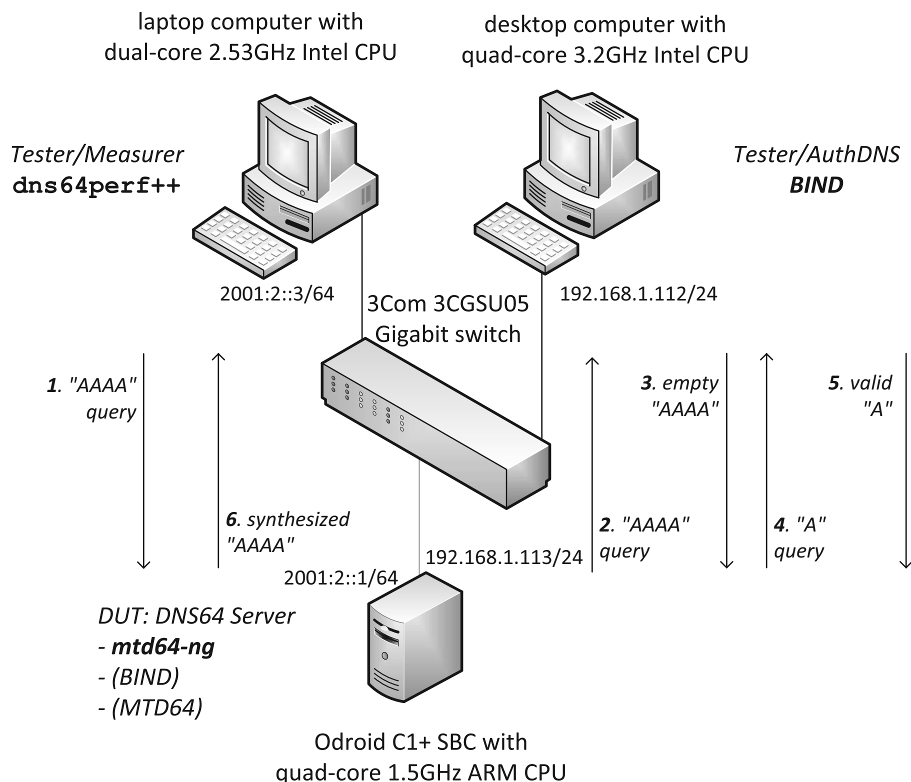

Figure 1. Test and traffic setup for DNS64 performance measurements.

Tester/AuthDNS: Desktop computer with 3.2GHz Intel Core i5-4570 CPU, 16GB 1600MHz DDR3 SDRAM, 250GB Samsung 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.

DUT: Odroid C1+ with 1.5GHz quad-core ARM Cortex A5 CPU (Amlogic S805), 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, mtd64-ng from [10] (Latest commit: Mar 21, 2016), BIND 9.9.5-3ubuntu0.8-Ubuntu, MTD64 from [15] (Latest commit: January 4, 2015),

Switch: 3CGSU05 5-port 3Com Gigabit Ethernet switch.

We note that it is not a typical choice to use a single board computer with an ARM CPU as a DNS64 server. We have chosen this device for two reasons:

1. It has low performance compared to the two computers, thus the DUT is ensured to be the performance bottleneck in the test setup.
2. It has four cores, thus we expect that it better models the current multi-core servers than the old desktop computer with two cores used in [9].

To avoid being a bottleneck, we have chosen a modern quad-core desktop computer to be the authoritative DNS server. As dns64perf++ can utilize only two CPU cores, the dual-core laptop was good enough for its execution.

4.2.4. Configuration Settings

The most important configuration files of the BIND authoritative DNS server were the following ones.

The /etc/bind/named.conf.local file contained:

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

The db.dns64perf.test zone file was generated by the bash shell script shown in Figure 2.

As for the configuration file of mtd64-ng, it had to be modified to set different number of work-

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

\$ORIGIN dns64perf.test.
\$TTL    86400
@       IN      SOA     localhost. root.localhost. (
                     2015112001         ; Serial
                         604800         ; Refresh
                          86400         ; Retry
                        2419200         ; Expire
                          86400 )       ; Negative Cache TTL
;
@       IN      NS      localhost.

EOF

for a in {0..63}
do
  for b in {0..255}
  do
    for c in {0..255}
    do
      printf "010-%.3i-%.3i-%.3i IN A 10.%i.%i.%i\n" $a $b $c $a $b $c >> \
        db.dns64perf.test
    done
  done
done
echo "" >> db.dns64perf.test
```

Figure 2. Generator script for the zone file.

ing threads. Therefore its configuration file was prepared by a script using the mtd64-ng.conf.core file containing the non-modified settings. They were:

```
nameserver 192.168.1.112
dns64-prefix 2001:db8:dead:beef::/96
debugging no
timeout-time  1.0
resend-attempts   1
response-maxlength  512
port 53
```

The modifications were performed by the set-mtd64-ng-wth script, which received the number of working threads as a parameter:

```
#!/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 stopped mtd64-ng, prepared the new configuration file and started mtd64-ng.)

The settings.conf configuration file of MTD64 contained logically the same information as the mtd64-ng.conf.core file (only the syntax of the timeout setting was different), therefore we do not include the file.

The named.conf.options configuration file of BIND had the following settings:

```
options {
  directory "/var/cache/bind";
  forwarders { 192.168.1.112; };
  forward only;
  dns64  2001:db8:dead:beef::/96 { };
  dnssec-validation no;
  auth-nxdomain no;
  listen-on-v6 { any; };
};
```

The measurements were performed by scripts. The binary search script belonging to [14] was used with the permission of its authors. It was modified according to our needs (e.g. adding an extra for cycle to test mtd64-ng with different number of working threads). The script contained several lines for logging or for displaying information. We present the script without those lines, keeping only the important parts, see Figure 3.

We note the Internet Draft has to perform the tests eliminating the effect of caching. Therefore the test script for the BIND DNS64 implementation contained some extra lines to restart it after each step of the binary search. (The other two implementations do not support caching.)

4.3. Performance Measurement Results

The results of the self-test were always above 24000 queries per second, thus even if we choose the value of delta to be 0.2, the Tester may be used for testing up to 10000 queries per second (see subsection 9.2.1 of [7]).

The results of our first series of measurements are presented in Table 2. The results are shown as a function of the number of working threads. Besides median, 1st and 99th percentiles whose values are required by the Internet Draft, we also presented the average and the standard deviations of the 20 results, because we contend

Table 2. DNS64 performance of mt64-ng as a function of the number of working threads (1-12).

Number of working threads		1	2	3	4	5	6	7	8	9	10	11	12
DNS64 performance (queries per second)	median	3132	4747	5885	6849	7490	7940	8249	8542	8446	8470	8496	8433
	avg.	3050	4664	5796	6907	7570	7980	8356	8522	8459	8469	8438	8288
	s. dev.	175	167	160	240	171	354	276	216	220	266	276	436
	min.	2431	4095	5563	6141	7391	7167	8017	8191	8189	7871	7663	6975
	max.	3141	4773	5963	7273	7937	8453	8833	9089	9217	8833	8737	8711

```
#!/bin/bash
server=2001:2::1

# iteration for the number of working threads (nth)
for (( nth=1; nth<13; nth+=1 ))
do
  ssh -l root $server ./set-mtd64-ng-wth $nth # set the No. of working threads
  sleep 5 # Wait until mtd64-ng is surely ready

  # the original binary search program
  #Parameters
  xpts=60 # duration (in seconds) of an experiment instance
  xpt=$((1000000000*xpts)) # xpts given in nanoseconds (dns64perf++ uses ns)
  xptlim=$((xpt*101/100)) # at least 1% (cumulated) timer accuracy is required
  max=16384 # maximum query rate
  to=1 # timeout in seconds
  sleept=10 # sleeping time between the experiments
  e=1 # error: difference between higher and lower bounds of binary search
  srv_IP=$server # IPv6 address of the tested DNS64 server
  range=10.0.0.0/10 # name space used for testing (used by dns64perf++)
  no_exp=20 # number of experiments

  for (( n=1; n <= $no_exp; n++ )) # Execute $no_exp number of experiments
  do # Execute a binary search in the [l, h] interval
    l=0 #Initialize lower bound to 0
    h=$max #Initialize higher bound with maximum query rate
    for (( i=1; $((h-l)) > $e; i++ ))
    do
      f=$(((h+l)/2)) # initialize the test frequency with (h+l)/2
      ./dns64perf++ $srv_IP 53 $range $((xpts*f)) 1 $((1000000000/f)) $to \
        > temp.out 2>&1  # Execute the test program
      # Collect and evaluate the results
      sent=$(grep 'Sent' temp.out | awk '{print $3}')   # No. of sent queries
      valid=$(grep 'Valid' temp.out | awk '{print $3}') # No. of valid replies
      err1=$(grep 'Full' temp.out | awk '{print $5}')   # real exec. time
      # If the dns64perf++ program could not keep up, we MUST exit testing
      if [ $((err1+0)) -gt $xptlim ]; then
        exit;
      fi
      # If all the replies are valid, we choose the upper half interval
      if [ $valid == $sent ]; then
        l=$f
      # Otherwise we choose the lower half interval
      else
        h=$f
      fi
      date +'%Y-%m-%d %H:%M:%S.%N, '"$i"', '"$f"  > temprate.csv
      sleep $sleept # Sleep to give DUT a chance to relax
    done # (end of the binary search)
    summary=$(cat temprate.csv) # Collect results
    echo "$n, $xpts, $(($max/2)), $to, $summary" >> rate.csv
    rm temp*
  done # (end of the $no_exp number of experiments)
done # (end of the iteration for the number of working threads)
```

Figure 3. Extract from the measurement script (used with mtd64-ng).

that they give further insight:

- If there is a significant difference between the average and the median, it indicates that the distribution of the results is skewed.
- If the standard deviation is high (e.g. higher than 10% of the average) it indicates that the results are too scattered.

We have also presented the median values as a graph in Figure 4. It is clearly visible that the increase of the number of working threads resulted in the increase of the DNS64 performance up to 8 working threads (from 3132 queries/s to 8542 queries/s). After that, no more increase can be seen, rather the performance shows some degradation (down to 8433 queries/s), but the degradation is less than the standard deviation

of the measurement results, therefore we considered that further measurements were necessary before stating anything about the tendency. During the second series of measurements the size of the thread pool was increased from 10 to 100. The results are presented in Table 3 and the median values are also displayed as a graph in Figure 5. Now, the tendency is clearly visible: the DNS64 performance of mtd64-ng decreased to about 6000 queries per second at 30 working threads but it stabilized at that value.

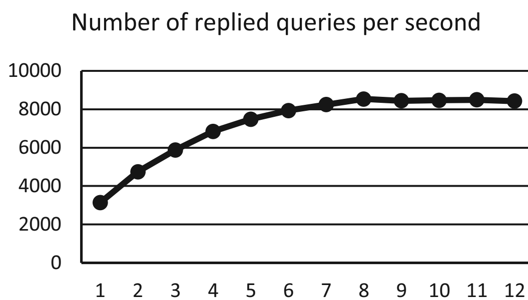

Figure 4. DNS64 performance of mtd64-ng as a function of the number of working threads (1-12).

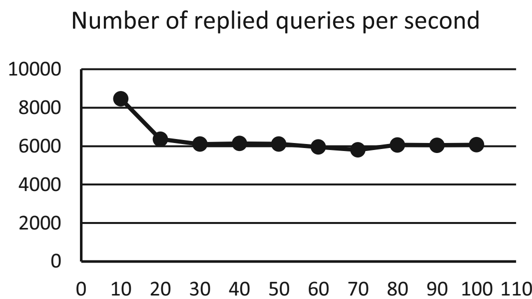

Figure 5. DNS64 performance of mtd64-ng as a function of the number of working threads (10-100).

As for the interpretation of the results, the reason of the initial increase of the performance is very simple: the increase of the number of working threads from 1 to 4 made it possible to utilize all four CPU cores. The further increase of the performance from 4 to 8 working threads can be explained by the fact that the DNS64 server had to send two queries to the authoritative DNS server for each request and while waiting for replies, the CPU cores could be utilized by the processing of other requests. The DNS64 performance did not show significant decrease from 8 to 12 working threads, but it was about 30% less for 30 or higher numbers of threads. We consider a possible explanation of this phenomenon the less effective use of the CPU cache (less cache hits), due to the higher memory usage of the unnecessarily high number of working threads.

Table 3 also contains the DNS64 performance results of BIND and MTD64. As we expected, mtd64-ng outperformed both BIND and MTD64. Considering the best performance of mtd64-ng at 8 working threads (8542 queries/s), it outperformed BIND (1374 queries/s) and MTD64 (5602 queries/s) 6.2 times and 1.5 times, respectively. Even if one prefers to compare the performance of mtd64-ng using 4 working threads with the performance of BIND, which uses also 4 working threads (but 4 listeners whereas mtd64-ng uses only 1 listener), the proportion is still 4.98, which is to be rounded to 5.0. And even if far too many threads are used, mtd64-ng is still somewhat faster than MTD64.

Table 3. DNS64 performance of mtd64-ng as a function of the number of working threads (10-100) + comparison with BIND (4 working threads) and MTD64.

Number of working threads		**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**	**BIND**	**MTD64**
DNS64 performance (queries per second)	median	8470	6368	6108	6146	6116	5956	5804	6071	6054	6074	1374	5602
	avg.	8469	6346	6027	6041	6038	5947	5789	6043	6027	6043	1354	5523
	s. dev.	266	473	253	252	175	191	258	153	201	183	86	374
	min.	7871	5119	5119	5247	5755	5599	5247	5797	5599	5611	1023	4607
	max.	8833	7169	6241	6305	6337	6209	6153	6273	6273	6401	1473	6145

5. Plans for Future Research and Development

Our long term goal is to develop a high performance DNS64 server, which can be used in production systems. This goal is planned to be achieved step by step, adding different functions and optimizing mtd64-ng gradually.

One of the short term development tasks is to use multiple listeners instead of the current single one to avoid being a bottleneck and then test the performance of mtd64-ng up to 8 or 12 CPU cores.

Our next goals are implementation of caching and testing its efficiency. We believe that our current object oriented redesign has been a great help in adding functionalities like caching more easily.

In the long run, we also plan some improvements like the ones listed in subsection VI.B of [8].

Before recommending mtd64-ng to be used in production systems, we plan to do its extensive functional and security testing.

6. Conclusion

We conclude that the object oriented redesign and reimplementation of MTD64, resulting in mtd64-ng, was a significant step in the life cycle of this new DNS64 server program, not only by the elimination of two significant flaws (memory leaking and its potential vulnerability to DoS attacks), but also by resulting in a better quality source code and paving the path for adding further functionalities.

As for the high performance of the DNS64 server, we have shown that it is not only kept but even further improved.

We conclude that the development of mtd64-ng is worth continuing until a production class DNS64 server implementation is achieved.

References

[1] M. Bagnulo *et al.*, "The NAT64/DNS64 Tool Suite for IPv6 Transition", *IEEE Communication Magazine*, vol. 50, no 7, pp. 177–183, 2012. http://dx.doi.org/10.1109/MCOM.2012.6231295

[2] Free Software Foundation, "The Free Software Definition" http://www.gnu.org/philosophy/free-sw.en.html

[3] Open Source Initiative, "The Open Source Definition" http://opensource.org/docs/osd

[4] M. Bagnulo *et al.*, "DNS64: DNS Extensions for Network Address Translation from IPv6 Clients to IPv4 Servers", *IETF RFC 6147*. http://dx.doi.org/10.17487/rfc6147

[5] G. Lencse and S. Répás, "Performance Analysis and Comparison of Four DNS64 Implementations under Different Free Operating Systems", *Telecommunication Systems*, vol. 63, no. 4, pp. 557–577. http://dx.doi.org/10.1007/s11235-016-0142-x

[6] C. Bao *et al.*, "IPv6 Addressing of IPv4/IPv6 Translators", *IETF RFC 6052*. http://dx.doi.org/10.17487/rfc6052

[7] M. Georgescu and G. Lencse, "Benchmarking Methodology for IPv6 Transition Technologies", *IETF BMWG*, Internet Draft https://tools.ietf.org/html/draft-ietf-bmwg-ipv6-tran-tech-benchmarking-08

[8] 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. 2016. http://dx.doi.org/10.11601/ijates.v5i2.129

[9] G. Lencse, "Performance Analysis of MTD64, Our Tiny Multi-Threaded DNS64 Server Implementation: Proof of Concept", *International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems*, vol. 5, no 2, pp. 116–121, 2016. http://dx.doi.org/10.11601/ijates.v5i2.166

[10] D. Bakai, "A Lightweight Multithreaded C++11 DNS64 Server", mtd64-ng source code. https://github.com/bakaid/mtd64-ng

[11] G. Lencse and D. Bakai, "Design and Implementation of a Test Program for Benchmarking DNS64 Servers", *IEICE Transactions on Communications*, vol. E100-B, no. 6, pp. 947–954, 2017. http://dx.doi.org/10.1587/transcom.2016EBN0007

[12] S. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices", *IETF RFC 2544*. http://dx.doi.org/10.17487/rfc2544

[13] D. Bakai, "A C++11 DNS64 Performance Tester, dns64perf++", source code. https://github.com/bakaid/dns64perfpp

[14] G. Lencse *et al.*, "Benchmarking Methodology for DNS64 Servers", *Computer Communications*, published online http://dx.doi.org/10.1016/j.comcom.2017.06.004

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

Received: September 2016
Revised: April 2017
Accepted: May 2017

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

Dániel Bakai
Department of Networked Systems and Services
Budapest University of Technology and Economics
2 Magyar tudósok körútja
H-1117 Budapest
Hungary
e-mail: bakaid@kszk.bme.hu

Gábor Lencse received his MSc and PhD in computer science from the Budapest University of Technology and Economics, Budapest, Hungary in 1994 and 2001, respectively. He works for the Department of Telecommunications, Széchenyi István University, Győr, Hungary since 1997. Now, he is an Associate Professor. He is also a part time Senior Research Fellow at the Department of Networked Systems and Services, Budapest University of Technology and Economics since 2005. His research interests include performance analysis of communication systems, parallel discrete event simulation methodology and IPv6 transition methods.

Dániel Bakai is a BSc student studying computer science at the Budapest University of Technology and Economics, Budapest, Hungary. He does project work for the Department of Networked Systems and Services, Budapest University of Technology and Economics since February 2015. He is also the author of the dns64perf++ DNS64 server benchmarking program.