Design, Implementation and Testing of a Tiny Multi-Threaded DNS64 Server

Gábor Lencse and András Gábor Soós

***Abstract*—DNS64 is going to be an important service (together with NAT64) in the upcoming years of the IPv6 transition enabling the clients having only IPv6 addresses to reach the servers having only IPv4 addresses (the majority of the servers on the Internet today). This paper describes the design, implementation and functional testing of MTD64, a flexible, easy to use, multi-threaded DNS64 proxy published as a free software under the GPLv2 license. All the theoretical background is introduced including the DNS message format, the operation of the DNS64 plus NAT64 solution and the construction of the IPv4-embedded IPv6 addresses. Our design decisions are fully disclosed from the high level ones to the details. Implementation is introduced at high level only as the details can be found in the developer documentation. The most important parts of a through functional testing are included as well as the results of some basic performance comparison with BIND.**

***Keywords*—DNS, DNS64, domain names, IPv4, IPv6, IPv6 transition.**

I. Introduction

Due to the depletion of the public IPv4 address pool [1] the ISPs (Internet Service Providers) will not be able to assign public IPv4 addresses to their new clients. Reference [2] classifies the possible IPv4 address sharing mechanisms and discloses their tradeoffs. From among them, many Hungarian ISPs have chosen to give private IPv4 addresses to the clients and use CGN (Carrier Grade NAT). However, this solution limits the reachability of the clients from the outside world, and does not support their transition to IPv6, which one must happen once (sooner or later). In our opinion, the deployment of IPv6 is the forward looking solution for the shortage of public IPv4 address. The new clients will get IPv6 addresses only and they can communicate with the native IPv6 servers directly, but the majority of the Internet servers still use IPv4 only. The combination of a DNS64 [3] service and a NAT64 [4] gateways is a suitable solution which enables the IPv6 only clients to communicate with IPv4 only servers [5]. We agree with the authors of [2] that: “The only actual address sharing mechanism that really pushes forward the transition to IPv6 is Stateful NAT64 (Class 4). All other (classes of) mechanisms are more tolerant to IPv4.” Therefore we expect that (because NAT64 needs it) DNS64 will become a widespread used service during the upcoming phase of the IPv6 transition. To use this solution, a DNS64 server has to be set as the DNS server in the IPv6 only computers. When a client program (e.g. web browser) requests a domain name resolution for the domain name of a server which it wants to connect to, then the DNS64 server acts like a proxy: it uses the normal DNS system to find out the IP address. If the DNS64 server gets an IPv6 address from the DNS system then it simply returns the IPv6 address to the client. However, if it gets no IPv6 address but only IPv4 address (recall that it happens in the vast majority of the cases today) then it synthesizes a so called IPv4-embedded IPv6 address [6] and it returns the synthesized IPv6 address to the client. In this case, the communication of the IPv6 only client and the IPv4 only server will happen with the help of a NAT64 gateway. See more details later in this paper.

There are a number of free software [7] (also called open source [8]) DNS64 implementations, e.g. BIND, Unbound, PowerDNS or TOTD but even the smallest of them, TOTD has about 10,000 lines of source code (excluding the source of SWILL, its built-in web server) [9]. In this paper, we propose *MTD64*, a tiny *M*ulti-*T*hreaded *D*NS*64* server, which one is very small in code size (less than 1300 lines of source code) but it is still flexible and convenient. The aim of our work is to provide a simple DNS64 implementation which has clear and disclosed design decisions and well documented source code to give a chance for others to improve it by adding further functionalities or changing some of the used solutions to more efficient ones. The software is planned to be developed mainly by university students under the supervision of the first author of this paper, but our free software license allows anyone to join by making an own fork of the source code. At its current stage, MTD64 is not meant to be used as a DNS64 server in real-life networks, but it is rather meant to be a base point for further developments and to serve also as a testbed for comparison of the efficiency of different possible solutions (e.g. different caching policies). Our long term goal is to develop a production quality DNS64 server step by step.

The design decisions of MTD64 were originally disclosed in our conference paper [10], which one is now extended with high level details of implementation and with the documentation of testing including a thorough functional testing and a basic performance comparison with BIND.

The remainder of this paper is organized as follows. First, the theoretical background is introduced to the reader: the DNS message format, the operation of the DNS64+NAT64 solution and the construction of the IPv4-embedded IPv6 addresses are described. Second, our design decisions are presented from the high level ones to the details. Third, the implementation is described at high level including the source files and their roles as well as the operation of the program in a nutshell. Fourth, a detailed functional testing of our DNS64 implementation is done including a short performance testing, too. Fifth, our future plans are summarized. Finally, our conclusions are given.

Manuscript received November 6, 2015, revised March 14, 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).

A. G. Soós was with Telenor Hungary, 1 Pannon út, H-2045 Törökbálint, Hungary (e-mail: soos.gabor.andras@gmail.com).

doi: 10.11601/ijates.v5i2.129

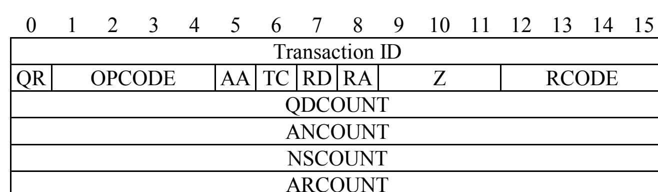

Fig. 1. DNS message *Header* section format.

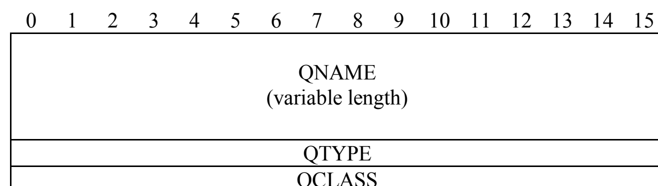

Fig. 2. DNS message *Question* section format.

II. THEORETICAL BACKGROUND

A. Format of DNS Messages

The DNS64 server has to work with various DNS messages: it must interpret, forward, prepare or synthesize them. Therefore we give a brief summary of the DNS message format [11].

DNS messages between a client and a server usually travel over UDP because both the requests and replies are usually short and sending them over UDP is much faster than establishing a TCP connection using the three-way handshake before the client-server communication and closing it at the end using the four-way handshake. If some of the messages happen to be lost then they can be resent.

1) Top level structure

A DNS message is built up by five sections: its *Header* section is always 12 bytes long and it is followed by four variable length sections (some of them may be empty): *Question*, *Answer*, *Authority*, *Additional*.

2) Header section format

The *Header* section can be further subdivided as shown in Fig. 1. The 16-bit *Transaction ID* field is used by the client to identify the answer of the server for different questions. It is generated by the requester (client) and it is copied by the server into the corresponding reply. The *QR* bit specifies whether this message is a query (0), or a response (1). The *OPCODE* field is used by the originator of the query to specify the kind of the query and it is copied by the server into the answer. Only the 0 value is of practical interest for us, it means standard query. The *AA* bit is valid only in responses and it signals if the answer is authoritative. The *TC* bit signals if the DNS message was truncated due to the limitations of the MTU of the transmission channel. The usage of the TC bit is clarified in section 9 of [12]. "The TC bit should not be set merely because some extra information could have been included, but there was insufficient room." It also states that: "When a DNS client receives a reply with TC set, it should ignore that response, and query again, using a mechanism, such as a TCP connection, that will permit larger replies."

→ *The DNS64 server program should not set the TC bit for leaving out some of the Additional RRs at the end of the message.*

The *RD* bit is used by the requester to ask recursive query. The *RA* bit is used by the server to signal if recursion is

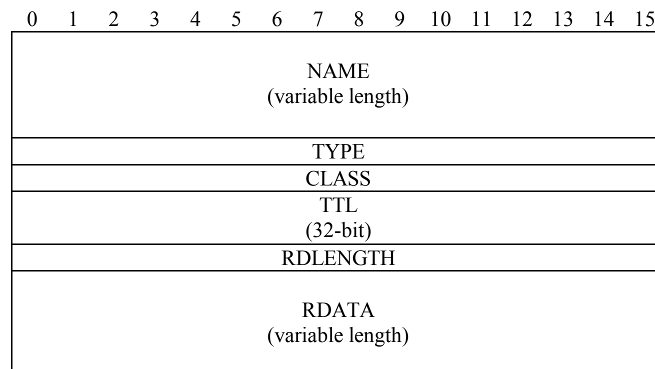

Fig. 3. DNS message *RR* format for *Answer*, *Authority*, *Additional* sections.

available. All four bits of the *Z* field must be set to 0 in all queries and responses (it is reserved for future use). The *RCODE* field of the responses specifies the error code: 0 value means no error. The *QDCOUNT* field specifies the number of entries in the *Question* section. In practice, clients send only one question in a DNS message. The *ANCOUNT*, *NSCOUNT* and *ARCOUNT* fields specify the number of resource records in the *Answer*, *Authority* and *Additional* sections, respectively.

3) Question section format

The *Question* section contains *QDCOUNT* number of entries (usually 1). An entry follows the format shown in Fig 2. The variable length *QNAME* field contains the domain name using special encoding (see: Domain name encoding and message compression). The *QTYPE* filed specifies the *RR* (Resource Record) type by 16-bit long binary vales. Some examples are:

- *A* (0x01) – IPv4 ***A***ddress
- *AAAA* (0x1C) – IPv6 Address (***4*** times size of ***A***)
- *CNAME* (0x05) – ***C***anonical ***NAME*** (alias)
- *MX* (0x0F) – ***M***ail e***X***changer
- *NS* (0x02) – ***N***ame ***S***erver
- *PTR* (0x0C) – used for reverse mapping (***P***oin***T***e***R***).

The *QCLASS* field contains the 0x01 16-bit binary value for denoting the *IN* (Internet) class. The other theoretically possible values for *CH* (Chaos) or *HS* (Hesiod) are not used.

4) Resource record format

The *RR* (Resource Record) format – used in the *Answer*, *Authority* and *Additional* sections – is shown in Fig. 3. The first three fields correspond to that of the Question section. The 32-bit unsigned integer in the *TTL* (Time to Live) field specifies the time interval in seconds while the *RR* may be cached. The 16-bit unsigned integer in the *RDLENGTH* field gives (in octets) the length of the *RDATA* field, which contains the octets of the given resource (e.g. the 4 octets of the IPv4 address or the 16 octets of the IPv6 address).

5) Domain name encoding and message compression

The domain names stored in the *QNAME* or *NAME* fields 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 and 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. Fig. 4 illustrates the encoding of the domain name

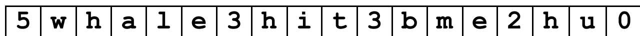

Fig. 4. DNS encoding of the **whale.hit.bme.hu** domain name.

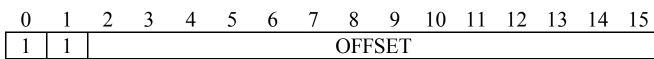

Fig. 5. The structure of a pointer.

Fig. 6. Compressed encoding of the **www.hit.bme.hu** domain name using the fact that the domain name shown in Fig. 4 starts at offset 0x0030.

whale.hit.bme.hu.

The addition of *pointer*s to this encoding scheme makes possible an efficient compression if there are repetitions of entire domain names or label sequences at the end of the domain names in DNS messages. A pointer is a two octet sequence where the first two bits of the first octet are ones, see Fig. 5. Note that the length of a label is at most 63 octets, therefore the first two bits of the octet expressing its length are always zeros, thus a pointer can be easily distinguished from a label. The *OFFSET* field of the pointer specifies the offset of the pointed label sequence from the beginning of the DNS message. Let us demonstrate it with an example. If the domain name in Fig. 4 starts at offset 0x0030 in a DNS message then we can compress the **www.hit.bme.hu** domain name in the same DNS message as it is shown in Fig. 6. The beginning three **w** characters are encoded in the usual way and then follows the 0xC0 value. The "11" values of its first two bits show that this is a pointer and the octet is to be interpreted together with the next one. The value of the offset field is 0x0036, which points to the second label of the domain name in Fig. 4.

→ *The DNS64 server program must be able to handle correctly this encoding and compression scheme.* (See later its consequences: the server program must be able to decode the domain name for logging purposes and it must also be able to modify the pointer if the pointed *RR* is moved within the DNS message.)

B. Operation of the DNS64 + NAT64 Solution

The operation of the DNS64 + NAT64 solution is demonstrated in Fig. 7. It shows a scenario where an IPv6 only client communicates with an IPv4 only web server. The DNS64 server uses the 64:ff9b::/96 *NAT64 Well-Known Prefix* for generating *IPv4-embedded IPv6 address*es. A prerequisite for the proper operation is that packets towards the 64:ff9b::/96 network are routed to the NAT64 gateway (routing must be configured that way). Let us follow the steps:

1. The client asks its DNS server (which one is actually a DNS64 server) about the IPv6 address of the **www.hit.bme.hu** web server.
2. The DNS64 server asks the DNS system about the IPv6 address of **www.hit.bme.hu**.
3. No IPv6 address is returned.
4. The DNS64 server then asks the DNS system for the IPv4 address of **www.hit.bme.hu**.
5. The 152.66.148.44 IPv4 address is returned.
6. The DNS64 server synthesizes an *IPv4-embedded IPv6 address* by placing the 32 bits of the received 152.66.148.44 IPv4 address after the 64:ff9b::/96 prefix and sends the result back to the client.
7. The IPv6 only client sends a TCP SYN segment using the received 64:ff9b::9842:f82c IPv6 address and it arrives to the IPv6 interface of the NAT64 gateway (since the route towards the 64ff9b::/96 network is set so in all the routers along the path).
8. The NAT64 gateway constructs an IPv4 packet using the last 32 bits (0x9842f82c) of the destination IPv6 address as the destination IPv4 address (this is exactly 152.66.248.44), its own public IPv4 address (198.51.100.10) as the source IPv4 address and some other fields from the IPv6 packet plus the payload of the IPv6 packet. It also registers the connection into its connection tracking table (and replaces the source port number by a unique one if necessary). Finally it sends out the IPv4 packet to the IPv4 only server.
9. The server receives the TCP SYN segment and sends a SYN ACK reply back to the public IPv4 address of the NAT64 gateway.
10. The NAT64 gateway receives the IPv4 reply packet. It constructs an appropriate IPv6 packet

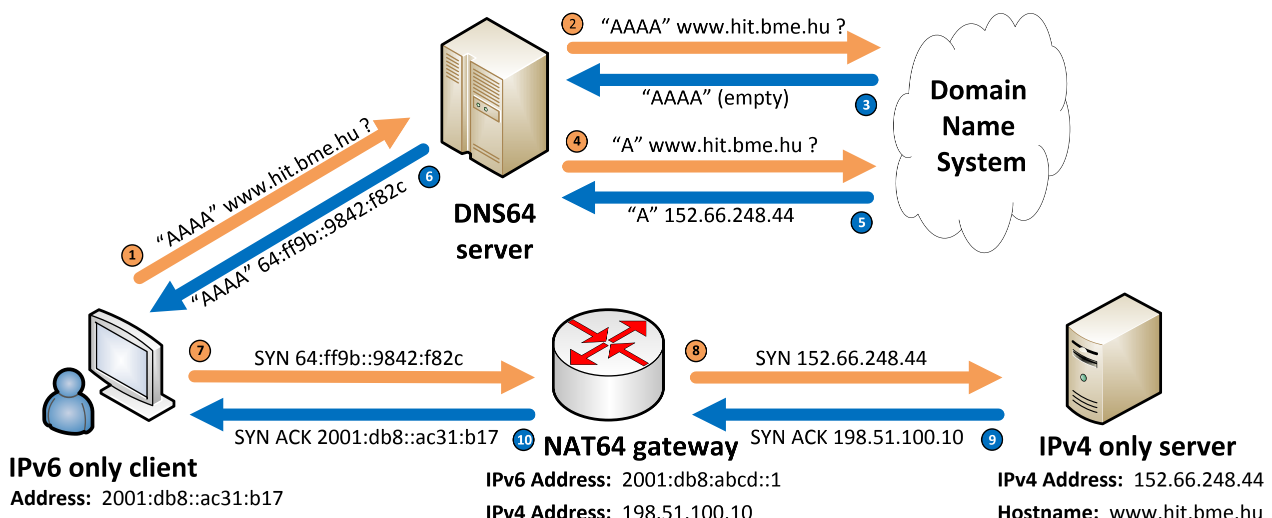

Fig. 7. The operation of the DNS64+NAT64 solution: an IPv6 only client communicates with and IPv4 only server.

using the necessary information from its state table. It sends the IPv6 packet back to the IPv6 only client.

The communication may continue on. It seems to the clients that it communicates to an IPv6 server. Similarly, the server "can see" an IPv4 client. If it logs the IP addresses of the clients than it will log the public IPv4 address of the NAT64 gateway.

Most client-server applications can work well with the DNS64+NAT64 solution. See more information about the application compatibility in: [13]–[15].

In practice, the world wide usage of the *NAT64 Well-Known Prefix* has several hindrances, see sections 3.1 and 3.2 of [6]. Therefore the network operators allocate a subnet from their own network for this purpose. It is called *Network Specific Prefix* (NSP).

→ The DNS64 server must enable the user to set the appropriate prefix for synthesizing the IPv4-embedded IPv6 address.

C. Construction of the IPv4-Embedded IPv6 Addresses

The construction of the IPv4-embedded IPv6 addresses is defined in [6]. When using *Network-Specific Prefix*, the network administrator has to decide the size of the prefix. There are some constraints:

- The prefix size must be exactly one of 32, 40, 48, 56, 64 or 96.
- The 64-71 bits of the IPv6 address must be 0.
- The 32 bits of the IPv4 address are stored right after the prefix but the above mentioned 0 bits have to be left out (or jumped over).
- If there are unused bits at the end of the IPv6 address then they must be filled with 0-s.

→ The DNS64 server should be able to check the prefix size and accept only the permitted ones.

D. Operation Requirements for the DNS64 server

The DNS64 server is set as the normal DNS server of the client.

→ Therefore the DNS64 server must be able to act as a proxy for any other requests than the AAAA records (e.g. MX).

Even though DNS64 is intended as an IPv6 transition solution for the IPv6 only clients, the clients might use dual stack.

→ Therefore A record requests and their replies must also be forwarded untouched.

III. Design Decisions

A. Design Principles

Our intention was to create a DNS64 server program that can be a viable alternative to the existing free software DNS64 implementations. Its attributes must include ease of use, high performance and ease of modification. In our position, a program like this should be:

- simple and therefore short (in source code)
- fast (written in C, at most some parts in C++)
- extensible (well structured and well documented)
- convenient and flexible in configuration
- free software under GPL or BSD license

B. High Level Design Decisions

1) Forwarder or recursor

A DNS server may operate in two modes. If it works as a *recursor* then it performs the recursion itself: starting from a top level DNS server it performs a series of iterative queries until it receives an authoritative answer. If it works as a *forwarder* then it acts like a proxy: forwards the queries to another DNS server and simply returns its answer to the client. (It may also cache the information.) As for the before mentioned four free DNS64 implementations, BIND and PowerDNS can act as both recursor and forwarder. TOTD can act as a forwarder only. Unbound can be either of them if it is used as a DNS server only, but it may perform the DNS64 functionality only in the case if it is set as a recursor.

We decided that MTD64 will operate as a forwarder only. It complies with the principle of simplicity.

2) Caching

On the one hand caching may significantly improve the performance of a DNS server, but on the other hand it seriously increases complexity. In addition to that the most common desktop operating systems, i.e. the different versions of Windows and Linux use DNS caching, thus they do not send the subsequent requests of the clients concerning the same domain name to the DNS server. However, if the DNS64 server is used by several clients then many of them may send requests for the same set of domain names thus caching is very likely to be beneficial.

We decided to omit caching from the first version of MTD64 because implementing it would have required more time than it was available during the final project (MSc thesis) of the second author of this paper. It is planned to be added later as a separate project for another student.

3) Storing the requests or not

When the DNS64 server receives a request from the client and forwards it to the DNS system, the DNS64 server should preserve the information about the client while waiting for the reply to be able to send back the reply (or the synthesized IPv6 address) to the client. The requests from the different clients may arrive in high number therefore an expandable data structure should be chosen e.g. linked list, balanced or unbalanced trees. Their operations (insert, find, delete) involve programming complexity and the operations may involve significant time complexity if the data structure has high number of elements. Unfortunately there is a trade-off between the programming complexity and the speed. E.g. the operations of the linked list are simple but their time complexity is $O(n)$, where n denotes the number of elements in the data structure. The time complexity of the operations of the balanced trees is $O(\log n)$, but their operations require more programming work. For more information see [16] and its references.

We decided not to store explicitly the client information but start a new thread for each request. It means the information is stored on the stack in the local variables (and on the heap in dynamically allocated data structures held by pointers). We hope that this solution will not fight back through high memory consumption but it will turn out during performance testing. As a positive consequence of our decision, MTD64 will be able to utilize all the CPU cores of the server.

4) Programming language and program structure

The C++ programming language was chosen mainly for

its thread handling. Only one class is used: its tasks are to store the parameters set by the user and to make them available by member function calls. The majority of the source code is written in the C language to be as fast as possible. One main source code file contains the most important operation of the server program and two separate ones contains the code for loading and storing the settings. They all include the same single header file. See more details later on.

5) Configuration file format

Simple text format was chosen. The configuration file is line oriented: a keyword is followed by the values for the given setting. Both "#" and "//" can be used for denoting comments.

6) Logging

The MTD64 program uses the standard syslog facility for logging. The program uses multiple log levels and the amount of the logged information can also be set by the user in the configuration file of MTD64.

7) License

The GPL v2 license was chosen. It ensures that the derivatives of MTD64 will remain also free software.

C. Important Design Details

1) DNS servers and selection between them

Multiple name servers may be set. They can be added by using multiple lines. Also the configured name servers from the (Linux) operating system can be loaded. Two DNS server selection modes are supported. *Round Robin* uses the first one from the list while it replies on time. If time-out occurs, than it takes the next one from the list. *Random* chooses one randomly for every request. Note that this solution makes it possible to use the DNS servers balanced or unbalanced: e.g. one of them is specified 10 times and the other one is specified 20 times.

The random DNS server selection mode will also be useful when testing the performance of the MTD64 software: multiple DNS servers can be used so that their performance will not limit the performance of the MTD64 software.

2) DNS message length

DNS messages carried over the UDP transport protocol are limited to 512 octets. A DNS server may return multiple RR entries in its answer, thus its size may be close to 512 octets. When IPv4-embedded IPv6 addresses are synthesized from IPv4 addresses it results in a 16-4=12 octets growth for each IP address. Therefore care must be taken to the 512 octet limit. As certain programs may handle larger datagrams and others may not, we decided to entrust the decision to the user. Therefore the maximum length of the response of the MTD64 server can be set in the configuration file. If a resource record does not fit in the specified size of DNS reply message, the program leaves out the resource record and also logs the event. It does not set the TC bit, because by doing so it would force the client to repeat the query by using TCP, see [12].

3) Client and DNS server IP version

The IP version for the client side is obvious: the IPv6 only clients use IPv6. What IP version should be used to reach the DNS system? Theoretically the request for the "AAAA" record might also be sent over IPv6, but we found a safe and simple choice to use always IPv4. (It simplifies both the setting of the DNS servers in the configuration file and the communication with them.)

4) Order of questions and answers

Section 5.1.8 of [3] states that: "The DNS64 MAY perform the query for the AAAA RR and for the A RR in parallel, in order to minimize the delay." However, this possible speed up has its price in assembling and sending always two questions instead of one[1] as well as taking care for which one the answer has already arrived, therefore we decided not to do this, but rather follow the order shown in Fig 7.

5) Preparation of the answers to the clients

If the question of the client was different than an "AAAA" record (e.g. "A" record, "MX" record, etc.) or the client asked for an "AAAA" record and the DNS system responded with an "AAAA" record than it is enough to forward its reply to the client. (It can be done without any changes, because even the Transaction ID is matching since MTD64 forwarded the request of the client untouched to the DNS system which one also kept the Transaction ID.) When an "AAAA" record must be synthetized from an "A" record, we saw two possible ways for completing this task:

1. The complete reply can be assembled step-by-step "from scratch" using the information piece-by-piece from the reply of the DNS system. (It requires a lot of steps, see the fields of the DNS messages.)
2. The reply can be built in larger chunks by copying as long as possible memory areas from the reply of the DNS system.

The second one was chosen to achieve higher speed. The size of the chunks is limited by the occurrences of the "A" records: the 4 octet long IPv4 addresses have to be replaced by the synthesized IPv6 addresses which requires 16 octets space. Special care must be taken for the domain names containing pointers whether they have to be adjusted. (Recall that the RRs in the DNS answers also contain the questions with specially encoded and possibly compressed the domain names.)

D. Further Design Details

The presentation of all the design details would exceed the limitations of this paper. They are included in the "Programmer Documentation". For those who would like only to use MTD64, we recommend the "User Documentation". They can be found together with the commented source code on GitHub [17]. We also present a simple configuration file in the appendix, to give an impression of how flexibly MTD64 can be configured.

IV. Implementation

As both programmer documentation and commented source code are available on GitHub [17], we give only a high level overview here.

A. Source Files and their Responsibilities

There is a single header file **`header.h`** containing all the necessary system header file includes, the definition of the **`ConfigModule`** class and some function prototypes. It is included by all program files.

[1] We note that more and more Internet hosts will have IPv6 addresses in the future and, therefore, the "A" record will have to be asked less frequently.

```
yoso@Yoso-UBU:~$ host www.yandex.ru
www.yandex.ru has address 213.180.193.3
www.yandex.ru has address 93.158.134.3
www.yandex.ru has address 213.180.204.3
www.yandex.ru has IPv6 address 2001:db8:63a9:2ef5:dead:beef:d5b4:c10f
www.yandex.ru has IPv6 address 2001:db8:63a9:2ef5:dead:beef:d5b4:cc1b
www.yandex.ru has IPv6 address 2001:db8:63a9:2ef5:dead:beef:5d9e:8603
www.yandex.ru mail is handled by 0 mx-corp.yandex.ru.
```

Fig. 8. Output of the functional testing command.

The **`config_load.cpp`** file contains a single function **`load_config()`**, which is responsible for loading all parameters from the **`settings.conf`** configuration file and storing them in an instance of the **`ConfigModule`** class.

The member functions of the **`ConfigModule`** class are defined in **`config_module.cpp`** and their task is to set and retrieve all data members of the class containing configuration information as well as to synthesize *IPv4-embedded IPv6 address*es.

The main source file is called **`dns64server.cpp`** and it contains some string and error handling functions, the **`main()`** function and the **`send_response()`** function which one is responsible for the lion's share of the tasks of MTD64.

A workable sample configuration file **`settings.conf`** and a **`Makefile`** are also provided.

B. Operation of MTD64 in a Nutshell

The **`main()`** function calls the **`load_config()`** function to read the parameters, opens an UDP socket for receiving DNS queries and then starts an infinite loop, in which it receives DNS queries and starts a separate thread for handling each of them. The executed code of the thread is the **`send_response()`** function. It checks and records if an "AAAA" record was requested by the client. Then it forwards the query to the appropriate DNS server (determined by the settings) keeping everything (including the *Transaction ID*) untouched in it. Next, it sets the appropriate timeout (defined by the user in the configuration file) using the **`setsockopt()`** socket handling function and calls the **`recfvrom()`** function for receiving the reply from the DNS server. (If timeout occurs then it resends the query by at most as many times as the number set by the user in the configuration file, but now we do not go into deeper details.) If finally a response is received then it checks in its own records, if an "AAAA" record was requested by the client. If yes, and no "AAAA" record was received in the answer of the DNS server then it prepares a DNS query for an "A" record by modifying the query type in the preserved DNS query message from the client. Then it sends the query to the appropriate DNS server and receives its reply in the above mentioned manner (using timeout and resending the query if necessary). If it receives at least one "A" record then it modifies the message from the DNS server by replacing all "A" records with synthesized "AAAA" records (note that there may be more than one of them) taking care also for modifying pointers if necessary. It also considers the maximum DNS message length allowed by the standard (512 octets) or set for some other value by the user. (It may be necessary to omit some records at the end of the DNS message.) Finally, it returns either the untouched reply of the DNS server (if no change was necessary) or the modified one – containing one or more *IPv4-embedded IPv6 address*(es) – to the client. Note that the *Transaction ID* was always left unchanged thus the reply will meet the expectation of the client.

V. Testing

Unless stated otherwise, the settings of the sample configuration file were kept and the standard Linux **`host`** command was used for testing. The IPv6 address of the client was fe80::221:ccff:fe69:9f0a and the IPv6 address of the MTD64 server was: fe80::8e89:a5ff:fec5:5bef during all the functional and message length tests.

A. Functional Testing

1) Command line testing and observation

The **`host www.yandex.ru`** command was used for testing. As no other parameters were specified, the **`host`** command sent three queries and they asked "A", "AAAA" and "MX" records. The result of the command is shown in Fig. 8. It can be observed that the domain has three IPv4 addresses and no IPv6 address, therefore three IPv6 addresses were synthesized by MTD64 using the IPv4 addresses and the 2001:db8:63a9:2ef5:dead:beef::/96 prefix. Please note that the IPv4 and the corresponding *IPv4-embedded IPv6 addresses* are in a different order; we shall show its reason soon.

2) Wireshark capture analysis

The network traffic of the MTD64 server was captured by Wireshark during the execution of the **`host`** command for a more thorough checking of the operation of our DNS64 implementation. The captured packets are shown in Fig. 9. By observing the first four of them, we can see that MTD64

Time	Source	Destination	Len	Info
0.470918	fe80::221:ccff:fe69:9f0a	fe80::8e89:a5ff:fec5:5bef	93	Standard query 0x52d7 A www.yandex.ru
0.471167	192.168.0.100	8.8.4.4	73	Standard query 0x52d7 A www.yandex.ru
0.473407	8.8.4.4	192.168.0.100	121	Standard query response 0x52d7 A 213.180.193.3 A 93.158.134.3
0.473535	fe80::8e89:a5ff:fec5:5bef	fe80::221:ccff:fe69:9f0a	141	Standard query response 0x52d7 A 213.180.193.3 A 93.158.134.3
0.474556	fe80::221:ccff:fe69:9f0a	fe80::8e89:a5ff:fec5:5bef	93	Standard query 0x1032 AAAA www.yandex.ru
0.474782	192.168.0.100	8.26.56.26	73	Standard query 0x1032 AAAA www.yandex.ru
0.506820	8.26.56.26	192.168.0.100	134	Standard query response 0x1032
0.506947	192.168.0.100	8.26.56.26	73	Standard query 0x1032 A www.yandex.ru
0.538833	8.26.56.26	192.168.0.100	245	Standard query response 0x1032 A 213.180.193.3 A 213.180.204.3
0.539047	fe80::8e89:a5ff:fec5:5bef	fe80::221:ccff:fe69:9f0a	325	Standard query response 0x1032 AAAA 2001:db8:63a9:2ef5:dead:be
0.539961	fe80::221:ccff:fe69:9f0a	fe80::8e89:a5ff:fec5:5bef	93	Standard query 0x75e0 MX www.yandex.ru
0.540199	192.168.0.100	8.8.4.4	73	Standard query 0x75e0 MX www.yandex.ru
0.542343	8.8.4.4	192.168.0.100	97	Standard query response 0x75e0 MX 0 mx-corp.yandex.ru
0.542457	fe80::8e89:a5ff:fec5:5bef	fe80::221:ccff:fe69:9f0a	117	Standard query response 0x75e0 MX 0 mx-corp.yandex.ru

Domain Name Service (dns), 263 bytes

Fig. 9. DNS messages during the basic functional testing – captured and displayed by Wireshark.

```
▼Queries
 ▶www.yandex.ru: type A, class IN
▼Answers
 ▶www.yandex.ru: type A, class IN, addr 213.180.193.3
 ▶www.yandex.ru: type A, class IN, addr 93.158.134.3
 ▶www.yandex.ru: type A, class IN, addr 213.180.204.3
▼Authoritative nameservers
 ▶yandex.ru: type NS, class IN, ns ns2.yandex.ru
 ▶yandex.ru: type NS, class IN, ns ns1.yandex.ru
▼Additional records
 ▶ns1.yandex.ru: type A, class IN, addr 213.180.193.1
 ▶ns1.yandex.ru: type AAAA, class IN, addr 2a02:6b8::1
 ▶ns2.yandex.ru: type A, class IN, addr 93.158.134.1
 ▶ns2.yandex.ru: type AAAA, class IN, addr 2a02:6b8:0:1::1
▼Queries
 ▶www.yandex.ru: type AAAA, class IN
▼Answers
 ▶www.yandex.ru: type AAAA, class IN, addr 2001:db8:63a9:2ef5:dead:beef:d5b4:cc0f
 ▶www.yandex.ru: type AAAA, class IN, addr 2001:db8:63a9:2ef5:dead:beef:d5b4:c11b
 ▶www.yandex.ru: type AAAA, class IN, addr 2001:db8:63a9:2ef5:dead:beef:5d9e:8603
▼Authoritative nameservers
 ▶yandex.ru: type NS, class IN, ns ns2.yandex.ru
 ▶yandex.ru: type NS, class IN, ns ns1.yandex.ru
▼Additional records
 ▶ns1.yandex.ru: type AAAA, class IN, addr 2001:db8:63a9:2ef5:dead:beef:d5b4:c131
 ▶ns1.yandex.ru: type AAAA, class IN, addr 2a02:6b8::1
 ▶ns2.yandex.ru: type AAAA, class IN, addr 2001:db8:63a9:2ef5:dead:beef:5d9e:8601
 ▶ns2.yandex.ru: type AAAA, class IN, addr 2a02:6b8:0:1::1
```

Fig. 10. How MTD64 synthesizes IPv4-embedded IPv6 messages?

acted like a proxy when an "A" record was requested: the first message was sent from the client to the MTD64 server over IPv6. The second message contained the same query and is was sent from the MTD64 server to a public DNS server at 8.8.4.4 over IPv4. Similarly the response of the public DNS server was simply forwarded to the client by MTD64. The next six lines show the resolution of the query for an "AAAA" record. Similarly to the case of the query for the "A" record, MTD64 forwarded the request of the client to the public DNS server. Let us observe that the 0x1032 *Transaction ID* was also kept. However, MTD64 received an "empty" answer. Therefore, it sent a query for an "A" record using the same *Transaction ID*. We can observe that the public DNS server sent the same three IPv4 addresses as before but in a different order (now, the first two begin with the 213.180/16 prefix). This is the explanation of our earlier observation of the order change (based on the output of the host command shown in Fig. 8.) And let us also observe that whereas the length of the response of the public DNS server is 245 bytes, the length of the response of the MTD64 server is 325 bytes. The difference is 80 bytes. We shall explain its reason soon by a deeper analysis of the two messages. The last four lines show the resolution of the query for an "AAAA" record. Here, MTD64 acted as a proxy again, thus we do not go into details.

```
yoso@Yoso-UBU:~$ host -v -t AAAA cc00033.h.cnc.ccgslb.com.cn
Trying "cc00033.h.cnc.ccgslb.com.cn"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 5673
;; flags: qr rd ra; QUERY: 1, ANSWER: 10, AUTHORITY: 6, ADDITIONAL: 6

;; QUESTION SECTION:
;cc00033.h.cnc.ccgslb.com.cn.   IN      AAAA

;; ANSWER SECTION:
cc00033.h.cnc.ccgslb.com.cn. 89 IN      AAAA    2001:db8:63a9:2ef5:dead:beef:3df0:8794
cc00033.h.cnc.ccgslb.com.cn. 89 IN      AAAA    2001:db8:63a9:2ef5:dead:beef:3df0:8710
cc00033.h.cnc.ccgslb.com.cn. 89 IN      AAAA    2001:db8:63a9:2ef5:dead:beef:3df0:8718
cc00033.h.cnc.ccgslb.com.cn. 89 IN      AAAA    2001:db8:63a9:2ef5:dead:beef:77bc:8b4c
cc00033.h.cnc.ccgslb.com.cn. 89 IN      AAAA    2001:db8:63a9:2ef5:dead:beef:77bc:8ba1
cc00033.h.cnc.ccgslb.com.cn. 89 IN      AAAA    2001:db8:63a9:2ef5:dead:beef:77bc:8b8c
cc00033.h.cnc.ccgslb.com.cn. 89 IN      AAAA    2001:db8:63a9:2ef5:dead:beef:3df0:8711
cc00033.h.cnc.ccgslb.com.cn. 89 IN      AAAA    2001:db8:63a9:2ef5:dead:beef:b676:4d3b
cc00033.h.cnc.ccgslb.com.cn. 89 IN      AAAA    2001:db8:63a9:2ef5:dead:beef:b676:4d2b
cc00033.h.cnc.ccgslb.com.cn. 89 IN      AAAA    2001:db8:63a9:2ef5:dead:beef:da3a:d18d

;; AUTHORITY SECTION:
cnc.ccgslb.com.cn.      34145   IN      NS      ns7.cnc.ccgslb.com.cn.
cnc.ccgslb.com.cn.      34145   IN      NS      ns9.cnc.ccgslb.com.cn.
cnc.ccgslb.com.cn.      34145   IN      NS      ns12.cnc.ccgslb.com.cn.
cnc.ccgslb.com.cn.      34145   IN      NS      ns13.cnc.ccgslb.com.cn.
cnc.ccgslb.com.cn.      34145   IN      NS      ns14.cnc.ccgslb.com.cn.
cnc.ccgslb.com.cn.      34145   IN      NS      ns15.cnc.ccgslb.com.cn.

;; ADDITIONAL SECTION:
ns7.cnc.ccgslb.com.cn.  37321   IN      AAAA    2001:db8:63a9:2ef5:dead:beef:7b7d:1367
ns9.cnc.ccgslb.com.cn.  1954    IN      AAAA    2001:db8:63a9:2ef5:dead:beef:3a44:8d04
ns12.cnc.ccgslb.com.cn. 37385   IN      AAAA    2001:db8:63a9:2ef5:dead:beef:2a3e:b92
ns13.cnc.ccgslb.com.cn. 52277   IN      AAAA    2001:db8:63a9:2ef5:dead:beef:77bc:8c46
ns14.cnc.ccgslb.com.cn. 63710   IN      AAAA    2001:db8:63a9:2ef5:dead:beef:3df0:8a86
ns15.cnc.ccgslb.com.cn. 11019   IN      AAAA    2001:db8:63a9:2ef5:dead:beef:1bc3:9224

Received 605 bytes from fe80::8e89:a5ff:fec5:5bef%2#53 in 116 ms
```

Fig. 11. A DNS message which is longer than 512 octets.

3) Deeper analysis of the messages

Now, we compare two messages: the one with the "A" records from the public DNS server to the MTD64 server and the one with the synthesized "AAAA" records from the MTD64 server to the client. They are shown in Fig. 10. The first message contains the query, the three answers as "A" records, the names of two authoritative name servers and four additional records. From the four additional records, two ones are of type "A" and the other two ones are of type "AAAA". MTD64 transformed this message into the second one. How many differences can be observed? There are five of them: the three "A" records as answers and the two "A" records from among the additional records were transformed into "AAAA" records. Each transformation is responsible for a length growth by 12 octets. The IP headers are not displayed here but it can be checked in Fig. 9 that the first message travelled over IPv4 and the second one was sent over IPv6. The length of the standard IPv4 and IPv6 headers are 20 octets and 40 octets, respectively. Now, we have shown that the difference between the lengths of the two messages is exactly 5*12+20=80 octets.

B. Testing DNS Message Length Issues

As we have just seen, the DNS64 functionality increases the length of the DNS messages. What happens if we reach

Time	Source	Destination	Len	Info
0.94060	fe80::221:ccff:fe69:9f0a	fe80::8e89:a5ff:fec5:5bef	107	Standard query 0x1629 AAAA cc00033.h.cnc.ccgslb.com.cn
0.94078	192.168.0.100	8.26.56.26	87	Standard query 0x1629 AAAA cc00033.h.cnc.ccgslb.com.cn
0.97346	8.26.56.26	192.168.0.100	132	Standard query response 0x1629
0.97360	192.168.0.100	8.26.56.26	87	Standard query 0x1629 A cc00033.h.cnc.ccgslb.com.cn
1.05627	8.26.56.26	192.168.0.100	455	Standard query response 0x1629 A 61.240.135.148 A 61.240.135.16 A 61.240.13
1.05660	fe80::8e89:a5ff:fec5:5bef	fe80::221:ccff:fe69:9f0a	667	Standard query response 0x1629 AAAA 2001:db8:63a9:2ef5:dead:beef:3df0:8794

Domain Name Service (dns), 605 bytes

Fig. 12. DNS messages with 700 octets maximum length limit set – captured and displayed by Wireshark.

Time	Source	Destination	Length	Info
1.050888000	fe80::221:ccff:fe69:9f0a	fe80::8e89:a5ff:fec5:5bef	107	Standard query 0x6c15 AAAA cc00033.h.cnc.ccgslb.com.cn
1.051162000	192.168.0.100	8.26.56.26	87	Standard query 0x6c15 AAAA cc00033.h.cnc.ccgslb.com.cn
1.083156000	8.26.56.26	192.168.0.100	132	Standard query response 0x6c15
1.083274000	192.168.0.100	8.26.56.26	87	Standard query 0x6c15 A cc00033.h.cnc.ccgslb.com.cn
1.669715000	8.26.56.26	192.168.0.100	455	Standard query response 0x6c15 A 61.179.105.13 A 61.240.135.24 A
1.670063000	fe80::8e89:a5ff:fec5:5bef	fe80::221:ccff:fe69:9f0a	555	Standard query response 0x6c15 AAAA 2001:db8:63a9::3db3:690d AAAA

Domain Name Service (dns), 493 bytes

Fig. 13. DNS messages with standard 512 octets maximum length limit set – captured and displayed by Wireshark.

```
▼Queries
 ▶cc00033.h.cnc.ccgslb.com.cn: type A, class IN
▼Answers
 ▶cc00033.h.cnc.ccgslb.com.cn: type A, class IN, addr 61.179.105.13
 ▶cc00033.h.cnc.ccgslb.com.cn: type A, class IN, addr 61.240.135.24
 ▶cc00033.h.cnc.ccgslb.com.cn: type A, class IN, addr 112.84.133.26
 ▶cc00033.h.cnc.ccgslb.com.cn: type A, class IN, addr 210.76.58.31
 ▶cc00033.h.cnc.ccgslb.com.cn: type A, class IN, addr 36.250.90.25
 ▶cc00033.h.cnc.ccgslb.com.cn: type A, class IN, addr 119.188.140.141
 ▶cc00033.h.cnc.ccgslb.com.cn: type A, class IN, addr 211.90.28.25
 ▶cc00033.h.cnc.ccgslb.com.cn: type A, class IN, addr 210.76.58.10
 ▶cc00033.h.cnc.ccgslb.com.cn: type A, class IN, addr 210.76.58.32
 ▶cc00033.h.cnc.ccgslb.com.cn: type A, class IN, addr 61.240.135.50
▼Authoritative nameservers
 ▶cnc.ccgslb.com.cn: type NS, class IN, ns ns7.cnc.ccgslb.com.cn
 ▶cnc.ccgslb.com.cn: type NS, class IN, ns ns9.cnc.ccgslb.com.cn
 ▶cnc.ccgslb.com.cn: type NS, class IN, ns ns12.cnc.ccgslb.com.cn
 ▶cnc.ccgslb.com.cn: type NS, class IN, ns ns13.cnc.ccgslb.com.cn
 ▶cnc.ccgslb.com.cn: type NS, class IN, ns ns14.cnc.ccgslb.com.cn
 ▶cnc.ccgslb.com.cn: type NS, class IN, ns ns15.cnc.ccgslb.com.cn
▼Additional records
 ▶ns7.cnc.ccgslb.com.cn: type A, class IN, addr 123.125.19.103
 ▶ns9.cnc.ccgslb.com.cn: type A, class IN, addr 58.68.141.4
 ▶ns12.cnc.ccgslb.com.cn: type A, class IN, addr 42.62.11.146
 ▶ns13.cnc.ccgslb.com.cn: type A, class IN, addr 119.188.140.70
 ▶ns14.cnc.ccgslb.com.cn: type A, class IN, addr 61.240.138.134
 ▶ns15.cnc.ccgslb.com.cn: type A, class IN, addr 27.195.146.36
```

```
▼Queries
 ▶cc00033.h.cnc.ccgslb.com.cn: type AAAA, class IN
▼Answers
 ▶cc00033.h.cnc.ccgslb.com.cn: type AAAA, class IN, addr 2001:db8:63a9::3db3:690d
 ▶cc00033.h.cnc.ccgslb.com.cn: type AAAA, class IN, addr 2001:db8:63a9::3df0:8718
 ▶cc00033.h.cnc.ccgslb.com.cn: type AAAA, class IN, addr 2001:db8:63a9::7054:851a
 ▶cc00033.h.cnc.ccgslb.com.cn: type AAAA, class IN, addr 2001:db8:63a9::d24c:3a1f
 ▶cc00033.h.cnc.ccgslb.com.cn: type AAAA, class IN, addr 2001:db8:63a9::24fa:5a19
 ▶cc00033.h.cnc.ccgslb.com.cn: type AAAA, class IN, addr 2001:db8:63a9::77bc:8c8d
 ▶cc00033.h.cnc.ccgslb.com.cn: type AAAA, class IN, addr 2001:db8:63a9::d35a:1c19
 ▶cc00033.h.cnc.ccgslb.com.cn: type AAAA, class IN, addr 2001:db8:63a9::d24c:3a0a
 ▶cc00033.h.cnc.ccgslb.com.cn: type AAAA, class IN, addr 2001:db8:63a9::d24c:3a20
 ▶cc00033.h.cnc.ccgslb.com.cn: type AAAA, class IN, addr 2001:db8:63a9::3df0:8732
▼Authoritative nameservers
 ▶cnc.ccgslb.com.cn: type NS, class IN, ns ns7.cnc.ccgslb.com.cn
 ▶cnc.ccgslb.com.cn: type NS, class IN, ns ns9.cnc.ccgslb.com.cn
 ▶cnc.ccgslb.com.cn: type NS, class IN, ns ns12.cnc.ccgslb.com.cn
 ▶cnc.ccgslb.com.cn: type NS, class IN, ns ns13.cnc.ccgslb.com.cn
 ▶cnc.ccgslb.com.cn: type NS, class IN, ns ns14.cnc.ccgslb.com.cn
 ▶cnc.ccgslb.com.cn: type NS, class IN, ns ns15.cnc.ccgslb.com.cn
▼Additional records
 ▶ns7.cnc.ccgslb.com.cn: type AAAA, class IN, addr 2001:db8:63a9:2ef5::7b7d:1367
 ▶ns9.cnc.ccgslb.com.cn: type AAAA, class IN, addr 2001:db8:63a9:2ef5::3a44:8d04
```

Fig. 14. Comparison of the DNS reply with the A records (on the left) and the MTD64 reply with synthesized AAAA records (on the right). Some of the additional RRs were omitted due to the standard 512 octets DNS message size limit.

the 512 octets limit? To be able to test this situation, we had to find an appropriate domain name. An appropriate one can be found as follows: the **host -t AAAA www.gmw.cn** command results in several CNAME-s and one of them, namely **cc00033.h.cnc.ccgslb.com.cn** is suitable for this purpose.

1) Testing with 700 octets limit

The configuration file was modified as follows:

```
response-maxlength 700
```

Fig. 11 shows the results of the execution of the **host** command querying the "AAAA" record of the above mentioned domain name. The command also displayed the length of the DNS message: 605 bytes (marked by a red rectangle). Please note that the host command did not give an error message because of the DNS message was longer than 512 octets – even in the verbose mode (set by the **-v** option). Wireshark also displayed this message with no error, see Fig. 12. (The DNS payload size is also reported as 605 bytes in the bottom left corner of the figure.)

2) Testing with standard 512 octets limit

The configuration file was modified as follows:

```
response-maxlength 512
dns64-prefix 2001:0db8:63a9::/96
```

The same **host** command was issued again. Fig. 13 shows the Wireshark results. Now the payload length is only 493 bytes. Fig. 14 shows the reply of the public DNS server (with the "A" records) and the reply of the MTD64 server (with synthesized "AAAA" records). It can be seen that some of the additional RRs were omitted by MTD64 due to the standard 512 octets DNS message size limit.

C. Basic Performance Testing

During the review process of this journal paper, we have compared the performance of MTD64 to that of BIND, and it was found that MTD64 significantly outperformed BIND concerning the number of answered AAAA record requests per second [18]. However, as that paper is still under review (and therefore it is not citable yet), we have performed more measurements using different DNS64 server hardware to avoid copyright issues.

1) Test setup

The topology of our performance test network is shown in Fig. 15. A Raspberry Pi 2 Model B+ single-board computer was used as DUT (Device Under Test) to execute the DNS64 server programs to be compared. The **dns64perf** [19] test program was executed by a laptop computer for performance measurement. The authoritative DNS server was executed by a high performance desktop computer. The elements were interconnected by a Gigabit Ethernet switch. This setup was prepared so that the DUT be the bottleneck, thus its performance determined the overall performance of the test system.

2) Hardware and software parameters

For the repeatability of our measurements, we provide hardware and software details.

The authoritative DNS server was a desktop computer with 3.2GHz Intel Core i5-4570 CPU (4 cores, 6MB cache),

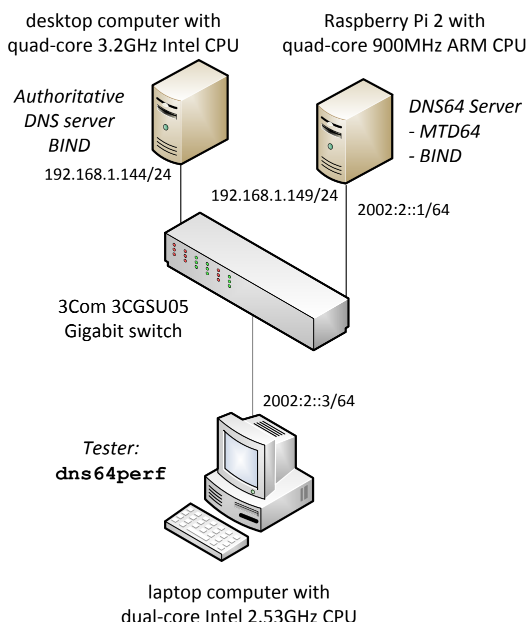

Fig. 15. Topology of the performance test network

TABLE I
BASIC PERFORMANCE COMPARISON OF MTD64 AND BIND

DNS64 Implementation		MTD64	BIND
Execution time of one experiment (ms)	average	65.87	122.27
	std. dev.	5.08	4.67
Replied AAAA queries per second		3886	2094

16GB 1600MHz DDR3 SDRAM, 250GB Samsung 840 EVO SSD, Realtek RTL8111F PCI Express Gigabit Ethernet NIC; Debian GNU/Linux 8.2 operating system, 3.2.0-4-amd64 kernel, BIND 9.9.5-9+deb8u3-Debian

The DUT was a Raspberry Pi 2 Model B single-board computer with 900MHz quad-core ARM Cortex A7 CPU, 1GB 400MHz LPDDR2 SDRAM, 16GB Kingston micro SD card, 100BaseTX Ethernet NIC; Debian GNU/Linux 8.0 operating system, 3.18.0-trunk-rpi2 kernel, BIND 9.9.5-9+deb8u2-Debian, MTD64 from [17] (Latest commit: January 4, 2015).

The tester device was a Dell Latitude E6400 series laptop with 2.53GHz Intel Core2 Duo T9400 CPU (2 cores, 6MB cache), 4GB 800MHz DDR2 SDRAM, 250GB Samsung 840 EVO SSD, Intel 82567LM Gigabit Ethernet NIC; Debian GNU/Linux 8.2 operating system, 3.2.0-4-amd64 kernel, **`dns64perf`** test program from [20].

The devices were interconnected by a 3CGSU05 5-port 3Com Gigabit Ethernet switch.

3) Testing method

The **`dns64perf`** program sent AAAA record queries for the domain names **`10-0-`*`b`*`-`*`c`*`.dns64perf.test`**, where variables *b* and *c* took their values from the [0..255] interval, thus altogether 65536 queries were sent. The domain names were resolved to the 10.0.*b*.*c* IPv4 addresses by the authoritative DNS server. As it is documented in [19], the **`dns64perf`** program organizes the 65536 name resolutions into 256 "experiments". During an experiment, variable *b* has a fixed value and the program can use several threads specified by the user. The program measures the execution time of the experiments and prints out the 256 results (in milliseconds). For further details, see [19]. For this time, 16 threads were used to send 16 queries concurrently in order to ensure high enough load.

4) Results

The result are presented in Table I. The *N* number or the replied AAAA record queries per second is calculated according to (1), where *T* denotes the execution time of one experiment (resolution of 256 AAAA record queries) specified in milliseconds.

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

The results are convincing: MTD64 could reply 3886 AAAA record queries per second whereas BIND could do only 2094. For further results on performance comparison, please see [18].

VI. FUTURE PLANS

A. Detailed Performance Analysis

We plan to test MTD64 under heavy load conditions to investigate its stability, CPU and memory requirements and also to check if it complies with the graceful degradation principle [21]. We also plan to compare its performance to the before mentioned free DNS64 server programs, namely BIND, TOTD, Unbound and PowerDNS with a similar test method which was used for their performance analysis in [22], [23], and [24].

We are especially interested in how the extensive use of threading influences the memory consumption of the program.

We consider that our current performance results and the result of [18] partially justify our design decisions but we need to perform further tests, especially concerning the effects of a possible DoS (Denial of Service) attack, when the attacker sends needless AAAA record requests to exhaust the resources of the server.

B. Implementing Further Functions

We plan to implement recursion, caching and concurrent look-up of "AAAA" and "A" records, too. We plan to add these functions one by one and compare the performance of the new software to the original one to check whether the additional complexity required by these functions results in speed-up or slow-down of the software.

Our long term plans include the support of TCP as transport protocol for DNS messages and after its inclusion, it will be possible to add also DNSSEC [25].

EDNS(0) makes it possible to use larger than 512 bytes message size over UDP, see section 4.3 of [26]. We will consider implementing this feature.

The tiny size of the source code makes it possible to oversee the program as a whole and thus to change its behavior and add functions as we find the best.

C. Expecting Feedback from the Users

MTD64 was released as free software, sharing the source code and documentation on GitHub [17]. The program can be used, modified and redistributed under the GPLv2 license. We would like to warn our potential users that the software is not yet ready to be used in production systems, but it can be tested and/or further developed.

Any questions, comments, suggestions, experiences, test reports are welcome by the authors of this paper.

D. The Development of MTD64 is Kept Going

Dániel Bakai has taken over the further development of our DNS64 implementation in 2015. He made a fork and named the new version mtd64-ng. We plan to report his results soon.

VII. CONCLUSION

We have introduced all the necessary details about the DNS message format, the operation of the DNS64+NAT64 solution and the construction of IPv4-embedded IPv6 addresses.

We have disclosed our design principles for a high performance, easy to use and modify DNS64 server.

We have fully described our design decisions from the top level ones to the details.

We have summarized the most important implementation details in this paper and also published the source code and documentation of our multi-threaded DNS64 server (called MTD64) on GitHub as a free software under the GPLv2 License.

We have conducted a thorough functional testing and also

checked the DNS message size issues.

During the basic performance testing, we have found that MTD64 significantly outperformed BIND when they were executed by a Raspberry Pi 2 Model B+ single-board computer.

Stability testing under heavy load conditions and a detailed performance analysis including comparison with several other free DNS64 implementations are planned future tasks.

We conclude that MTD64 may be useful also as a starting point for later development for anyone interested in.

ACKNOWLEDGEMENT

The development of the MTD64 server was the MSc thesis (final project) work of the second author at the Department of the Networked Systems and Services, Budapest University of Technology and Economics under the supervision of the first author.

REFERENCES

[1] The Number Resource Organization, "Free pool of IPv4 address space depleted" [Online]. Available: http://www.nro.net/news/ipv4-free-pool-depleted

[2] N. Skoberne, O. Maennel, I. Phillips, R. Bush, J. Zorz, and M. Ciglaric, "IPv4 address sharing mechanism classification and tradeoff analysis", *IEEE/ACM Transactions on Networking*, vol. 22, no 2, April 2014, pp. 391–404.DOI: 10.1109/TNET.2013.2256147

[3] 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)

[4] 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)

[5] M. Bagnulo, A. Garcia-Martinez and I. Van Beijnum, "The NAT64/DNS64 tool suite for IPv6 transition", *IEEE Communications Magazine*, vol. 50, no. 7, July 2012, pp. 177–183. DOI: 10.1109/MCOM.2012.6231295

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

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

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

[9] F. W. Dillema, TOTD 1.5.3 source code, [Online]. Available: https://github.com/fwdillema/totd

[10] 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

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

[12] R. Elz and R. Bush, "Clarifications to the DNS Specification", IETF, July 1997. (RFC 2181)

[13] N. Škoberne and M. Ciglarič, "Practical evaluation of stateful NAT64/DNS64 translation" *Advances in Electrical and Computer Engineering*, vol. 11, no. 3, August 2011, pp. 49–54. DOI: 10.4316/AECE.2011.03008

[14] V. Bajpai, N. Melnikov, A. Sehgal and J. Schönwälder, "Flow-based identification of failures caused by IPv6 transition mechanisms" in *Proc. 6th IFIP WG 6.6 Internat. Conf. on Autonomous Infrastructure, Management, and Security (AIMS 2012)*, Luxembourg, 2012, pp 139–150. DOI: 10.1007/978-3-642-30633-4_19

[15] S. Répás, T. Hajas and G. Lencse, "Application compatibility of the NAT64 IPv6 transition technology", in *Proc. 37th Internat. Conf. on Telecommunications and Signal Processing (TSP 2014)*, Berlin, 2014, pp. 49–55. DOI: 10.1109/TSP.2015.7296383

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

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

[18] G. Lencse, "Performance analysis of MTD64, our tiny multi-threaded DNS64 server implementation: Proof of concept", review version available: http://www.hit.bme.hu/~lencse/publications/

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

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

[21] NTIA ITS, "Definition of 'graceful degradation' " [Online]. Available: http://www.its.bldrdoc.gov/fs-1037/dir-017/_2479.htm

[22] 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, 2013, pp. 877–884. DOI: 10.1109/AINA.2013.80

[23] G. Lencse and S. Répás, "Improving the Performance and Security of the TOTD DNS64 Implementation", *Journal of Computer Science and Technology (JCS&T)*, ISSN: 1666-6038, vol. 14, no. 1, pp. 9–15. Apr. 2014.

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

[25] R. Arends, R. Austein, M. Larson, D. Massey, S. Rose, "DNS Security Introduction and Requirements", IETF, March 2005. (RFC 4033)

[26] J. Damas, M. Graff, P. Vixie, "Extension Mechanisms for DNS (EDNS(0))", IETF, April 2013. (RFC 6891)

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).

András Gábor Soós received his Bsc and MSc in computer engineering from the Budapest University of Tecnology and Economics in 2013 and 2015, respectively. For his BSc thesis, he added DHCPv6 functionality to an open source GGSN software. For his MSc thesis, he has implemented a fully functional DNS64 server program called MTD64.

He worked for Telenor Global services as a Voice Core Network Operator Engineer providing second and third line support, being responsible for solving voice related issues in 2014 and 2015. Currently he is working at Avaya Global Support Services as a Backbone Engineer porividing third line enterprise support on Avaya Voice products.

APPENDIX: CONFIGURATION POSSIBILITIES OF MTD64

```
# Sample configuration file for MTD64, a tiny Multi-Threaded DNS64 server

// Uncomment the following line for name servers to be read from /etc/resolv.conf
#nameserver defaults

// Or you can add name servers manually
nameserver 8.8.8.8
nameserver 195.46.39.39

// Set DNS server selection mode
# selection-mode random // The given DNS servers will be used in random order
selection-mode round-robin // If a DNS server does not respond until timeout, the next one will be used

// Accepted IPv6 prefix length values are: 32, 40, 48, 56, 64, 96
dns64-prefix 2001:0db8:63a9:2ef5:dead:beef::/96

debugging yes  // Results in more verbose logging

# Sample settings for the timeout value of 1.35 sec
timeout-time-sec 1 // Maximum value is 32767
timeout-time-usec 350000 // Maximum value is 999999

# How many times will the DNS64 server try to resend a DNS query message if there is no answer
resend-attempts 2 // Maximum value is 32767

# This will set the maximum length of the IPv6 response message (UDP payload).
# Blocks which fall outside this value will be cut off.
# It is highly recommended not to change from 512 since it is the RFC standard.
# Some programs can accept UDP DNS response messages longer than 512 bytes.
# Note that only Answer, Authority, Additional blocks can be cut off.
# Queries block is going to be sent even if the message length is longer therewith
response-maxlength 512 // Accepted range for this setting is 0-32767
```