Design 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 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+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.**

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

I. Introduction

DNS64 is expected by the authors to become a widespread used service during the upcoming phase of the IPv6 transition because the ISPs (Internet Service Providers) will not be able to assign public IPv4 addresses to their new clients due to the depletion of the public IPv4 address pool [1]. The clients will get IPv6 addresses instead but the vast majority of the Internet servers still use IPv4 only. The combination of the *DNS64* [2] service and the *NAT64* [3] gateways can be a suitable solution for this problem [4]. 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* [5] 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 [6] (also called open source [7]) 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) [8]. In this paper, we propose *MTD64*, a tiny *M*ulti-*T*hreaded *D*NS*64* server, which one is very small in code size but it is still flexible and convenient.

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, our future plans are summarized. Finally, our conclusions are given.

Manuscript received January 27, 2015.

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

A. Soós was with Department of Networked Systems and Services, Budapest University of Technology and Economics, 2 Magyar tudósok körútja, Budapest, Hungary (e-mail: soos.gabor.andras@gmail.com).

II. Theoretical Background

A. Format of the 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 [9].

DNS messages between a client and a server are 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.

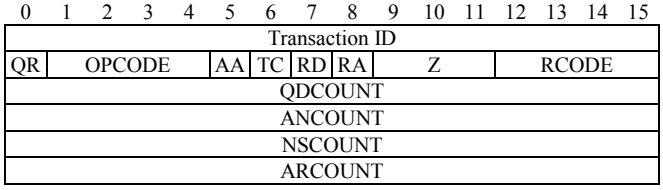

Fig. 1. DNS message *Header* section format.

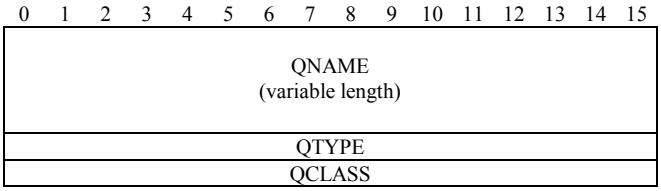

Fig. 2. DNS message *Question* section format.

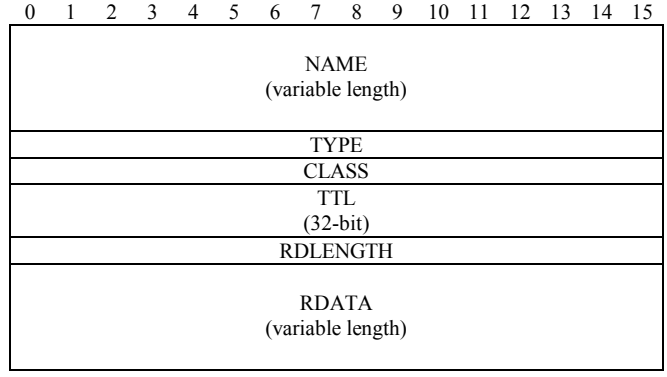

Fig. 3. DNS message *RR* format for *Answer*, *Authority*, *Additional* sections.

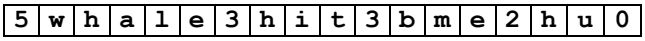

Fig. 4. DNS encoding of the **whale.hit.bme.hu** domain name.

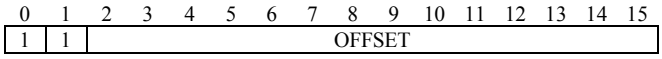

Fig. 5. The structure of a pointer.

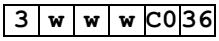

Fig. 6. Compressed encoding of the **www.hit.bme.hu** domain name using the fact that the domain name shown in Fig. 4 starts at offset 0x0030.

The usage of the TC bit is clarified in section 9 of [10]. "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 available. All the 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: Message 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 *RDLEGTH* 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 *labels* separated from each other by "." characters. The labels must be no longer than 63 characters. When domain names are encoded in DNS messages, the first character gives the length of the first label then the characters of the first label follow. After that, a character stands that specifies the length of the next label and the characters of the next label follow, etc. Finally, a zero character after the last label signals the end of the domain name. Fig. 4 illustrates the encoding of the **whale.hit.bme.hu** domain name. 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 addresses*. 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 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: [11]-[13].

In practice, the word wide usage of the *NAT64 Well-Known Prefix* has several hindrances, see sections 3.1 and 3.2 of [5]. 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 [5]. 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.*

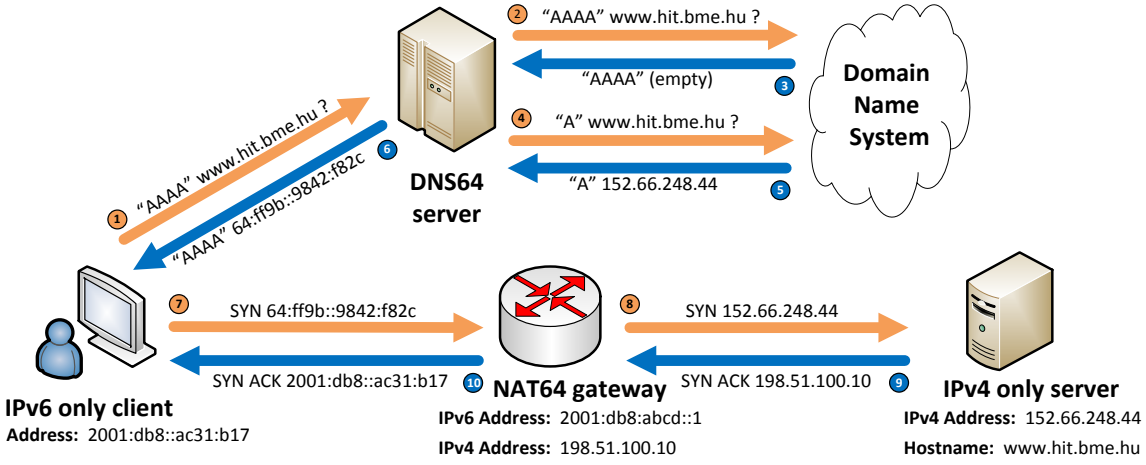

Fig. 7. The operation of the DNS64+NAT64 solution: an IPv6 only client communicates with and IPv4 only server

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 the 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.

We decided to omit caching. It may be added later if required.

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 [14] 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 a separate one contains the code for loading the settings. They both include the same single header file.

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 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 until 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 be using TCP, see [10].

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 [2] 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 two questions instead of one as well as taking care for which one has already arrived, therefore we decide 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 [15]. We also present a simple configuration file in the appendix, to give an impression of how flexibly MTD64 can be configured.

IV. Future plans

A. 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 [16]. We also plan to compare its performance to the before mentioned free DNS64 server programs, namely BIND, TOTD, Unbound and PowerDNS using the same method and test environment which was used for their performance analysis in [17] and [18].

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

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.

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 [15]. The program can be used, modified and redistributed under the GPLv2 license.

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

V. 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 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 conclude that our work may be useful as a workable DNS64 server and 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] 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)

[3] 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)

[4] 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

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

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

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

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

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

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

[11] 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

[12] 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 International Conference on Autonomous Infrastructure, Management, and Security (AIMS 2012)*, June 4-8, 2012, Luxembourg, Luxembourg

[13] S. Répás, T. Hajas and G. Lencse, "Application compatibility of the NAT64 IPv6 transition technology", in *Proc. 37th International Conference on Telecommunications and Signal Processing (TSP 2014)*, (Berlin, Germany, 2014. July, 1-3.) Brno University of Technology, pp. 49-55.

[14] G. Lencse, "Investigation of event-set algorithms", in *Proceedings of the 9th European Simulation Multiconference (ESM'95)* Prague, Czech Republic, 1995. June 5-7. SCS-Europe, pp. 821-825.

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

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

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

[18] G. Lencse, S. Répás, "Performance analysis and comparison of four DNS64 implementations under different free operating systems", *IEEE/ACM Transactions on Networking*, submitted for publication, [Online]. Available: http://www.hit.bme.hu/~lencse/publications/ToN-2014-DNS64-for-review.pdf

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
```