## Multi-Threaded DNS64 server Developer Documentation

**Introducion and presentation of header.h file:**

The software package consists of three .cpp files, a header.h file, a configuration file named settings.conf and a Makefile, each files are well documented. The header.h contains constants, function prototypes, class declaration and some macros which will prevent loading the necessary library files multiple times. The class declaration which belongs to the class named ConfigModule is the only class what MTD64 is using. The purpose of this class is to store all the settings loaded from the configuration file and based on those values it can perform some operations. The packet processing part of the program which is going to be run in separate threads will use this constant ConfigModule class to get all the necessary data and will use its function to perform a specific operation what will be described later. In the following three chapters I will introduce the config\_load.cpp, config\_module.cpp and dns64server.cpp files.

**Presentation of the config\_load.cpp file:**

This file is responsible for loading the data from the configuration file named settings.conf. Before it passes the prepared values to the ConfigModule class some checks are carried out in order to determine whether the actual setting is a valid one. In case of invalid setting the specific line will be ignored and a warning message will be written into the syslog. The config\_load.cpp file consists of one single function which will do all the previously mentioned tasks. This function is called in the main() function right after it initializes its variables and opens syslog. The program is going to read the configuration file line by line and will not interpret lines which starts with “#” or “//”. In the configuration file before and after the commands there could be arbitrary number of spaces and tabs if the line length is no longer than 254 characters. If the actual line is longer than 254 characters the program will try to interpret the first 254 characters and the rest of the data in the line will be ignored. You may find the available configuration file settings in the User Documentation. If the function finds a same valid setting which was loaded earlier the first setting will be used and the new ones will be ignored. In such cases a warning message will be written to syslog. The only exception from this rule is the nameserver command where in case of multiple “nameserver” settings the given DNS servers are going to be added together up to 50 which is the maximum number what program supports. If 50 nameservers is not enough for your purposes you can modify this number in the header.h file by rewrite the corresponding constant value. If any settings are missing from the configuration file the function will write a warning message into syslog and it will set the missing setting’s default value. The only case when error message could occur is when the settings.conf file cannot be opened for reading. In that case the MTD64 will be terminated.

**Presentation of the dns64server.cpp file:**

In the beginning of the dns64server.cpp file the implementation of the function prototypes can be found, furthermore the main() and the send\_response() functions. By entering the main() function the ConfigModule class is going to be instantiated first (after the definitions of the variables) which is followed by the opening of the syslog and the calling of the load\_config() function. After the loading of the configuration file ended successfully the socket configurations is going to be set. If any of the following system calls cannot be performed for some reason the program will be terminated. In such case an Error message which contains the errno value will be written into syslog. If we succeeded with the setting of the IPv6 listening port on UDP port 53 we will get into the infinite loop which will handle the incoming packets. If a new packet arrives the program will open a new thread in order to process the packet. This solution could provide better performance (especially if the system has multi-core support) and it also reduces the risk of having the buffer full which could cause packet loss. For processing and handling a packet the send\_response() function will be called.

In the send\_response() function, after a few necessary checks the DNS query message type will be inspected. Unless the query message has not the type of “AAAA” then the query will be sent to a DNS server, and the response message for that query will be sent back to the IPv6 client untouched. If the type of the query message is “AAAA” then this state will be stored in a boolean variable which indicated synthesizing could be necessary. This value will be used later.

After allocating the necessary containers (one is for the IPv4 response message and the other one is for the IPv6 response message) the sockets settings will occur and after that the sending process. If the original query type is not “AAAA” then only one query is going to be performed. Otherwise if the query type is “AAAA” and the response message contains at least one answer entry regarding the query, the response message will be returned to the client untouched and the earlier mentioned boolean variable will be unset. If the response message does not contain any “AAAA” answer entry then another query will be sent regarding the same domain name but this query will ask for “A” records. In this case the boolen variable remains set which indicates that server has to synthesize an “AAAA” response from the “A” response message. If the query sending part has ended successfully then the boolen variable will be checked whether synthesizing is necessary. If it is unset (which means false value) the actual response message will be returned untouched to the client, otherwise we will step into the synthesizing part.

In the first step of the synthesizing section, the program will calculate the size of the next block, and will check whether it can fit within the maximum length which was specified in the configuration file. If it cannot fit, then the actual block will not be inserted to the response message (which will be sent to the IPv6 client) and the program will set some parameters in the header of the DNS message accordingly in order to avoid inconsistency problems. If the actual block fits in we will proceed further. While reading the response message block by block if we find an “A” record then it will be converted to “AAAA”. In order to perform this, the type value will be changed to “AAAA” and the RDLEGTH value will be modified from 4 to 16 which is the size of an IPv6 address in bytes. After this we will use the function of the ConfigModule class named SetIpv4eIpv6Addr() to change the IPv4 address to an IPv4-Embedded IPv6 address in the RDATA field. Also if the program finds a pointer in the Name Field or in the RDATA field and it has to be changed because of the extra data then the pointer values will be modified accordingly. After we finished synthesizing a complete “AAAA” entry the program will copy the data from the next block of the original message to the modified response message location of the next block. If every “A” block has been converted or the maximum size has been reached we will send the response message back to the IPv6 client. In every cases if we will exit from the send\_response() function prior to that the allocated memories will be freed.