Skoltech
Сколковский институт науки и технологий

Analysis of high-performance software NAT design approaches

Master's Educational Program: **Information Science and Technology**

Author: Denis Plotnikov

Certified by: ____________________
Areg Melik-Adamyan, PhD, GNU Toolchain Team Manager, Intel

____
Vadim Sukhomlinov, Strategic Business Development Manager, Intel

Accepted by: ____________________
Mats Hanson, Dean of Education, Skoltech

Moscow

June, 2015

Analysis of high-performance software NAT design approaches

By

Denis Plotnikov

Submitted to Skolkovo Institute of Science and Technology on April 24, 2015 in Partial Fulfillment of the Requirements for the Degree of Master of Science

ABSTRACT

These days each device tends to communicate with the rest of the world over the Internet. For message passing the stack of TCP/IP protocols is widely used. The protocol performing address resolving is called IP protocol. Today, the IPv4 is the internet communication standard.

There is a problem of address space exhaustion of IPv4. Although, IPv6 is designed to solve that problem and is available for using, transition to it is connected to a number of problems. The big Internet users, such as Telecom Carriers, Internet Service Providers and other enterprises with big data networks are not ready to attack these problems and prefer to solve these problems using Network Address Translators (NATs).

There are different NATs on the market that differs with functionality and performance. NAT, fulfilling the performance and functional requirements of the big Internet users is called CG-NAT.

The CG-NATs available on the market are specialized network devices requiring high level of investments but there are another ways to achieve high performance NAT functionality using a commodity computer.

This work is devoted to exploration of those ways aiming to fulfill the functionality and performance requirements for CG-NAT. The result of the work is experimentally tested approach description of high performance Network Address Translator developing which allows to reduce the CG-NAT price and to keep the performance at the level of specialized network devices.

Thesis advisor:

Areg Melik-Adamyan, PhD, GNU Toolchain Team Manager, Intel

Thesis co-advisor:

Vadim Sukhomlinov, Strategic Business Development Manager, Intel

TABLE OF CONTENTS

Part 1 NAT OVERVIEW

1.1 Introduction

These days each device tends to communicate with the rest of the world over the Internet. The internet is a gigantic data network used IP protocols for nodes identification and consists of big number of sub networks. These sub networks have a way of transparent communication between the nodes inside and outside the sub network. The transparent way is called NAT (*Network Address Translator*). The main its function is changing the source IP address and port number of the packet going from the inner network and changing the destination IP address and port number of coming to the inner network packets. This process is called *address translation.* There is a set of requirements for a NAT which stays the same but has some differences depending on the size of network and the use case.

The NAT solves an important problem of IPv4 address space exhaustion. Although, the next generation protocol, IPv6, is ready to use there are some reasons because of which the users of NAT are not ready to move to IPv6 from IPv4 right away [1, 2].

There are different types of NAT implementations: software and hardware. The software versions of NAT provide all the required functionality but have low performance level. The hardware versions provide full functionality and high performance but usually expensive. Nowadays, when the high performance needed the hardware NATs have no competitors.

This work is focused on finding the way of making a software version of NAT which would have the same performance as a hardware version. This is worth of making efforts because of the high prices of hardware NAT solutions.

The work organized as follows. Part 1 gives overview of NAT technology and the existing NAT solutions, formulates the requirements to the NAT in terms of functionality and the performance. Part 2 does the software design approach analysis and makes some assumptions about what means should be used in order to achieve desired results. Part 3 analyzes the results acquired from implementation of the assumptions described in part 2, makes a conclusion of what the best approach to use among described is and formulates the conclusion.

1.2. Background and motivation

1.2.1 NAT Purpose and Motivation of Using

NAT was invented as a way of using a single IP address for several network devises. Thus, the main reason of using it is to reduce the number of IP addresses used by a number of network devices. Besides of the main functionality the NAT gives some security benefits like internal network structure hiding while and ability to restrict the access to the outer network of an internal node. Some versions of NAT provide functionality of translating IPv4 address to IPv6 which is helpful when changing provider settings.

This time the number of network devices grows rapidly because personal devises with the Internet access become more popular and more people starts using it every day. Each of those devices needs a network address to communicate through the data network. Presently, the main protocol used in the Internet is IPv4. The problem with IPv4 is that at the time the number of address in the IPv4 has reached its limit and the organization affiliated to manage the addresses issues has stopped its free distribution at 2012 [3].

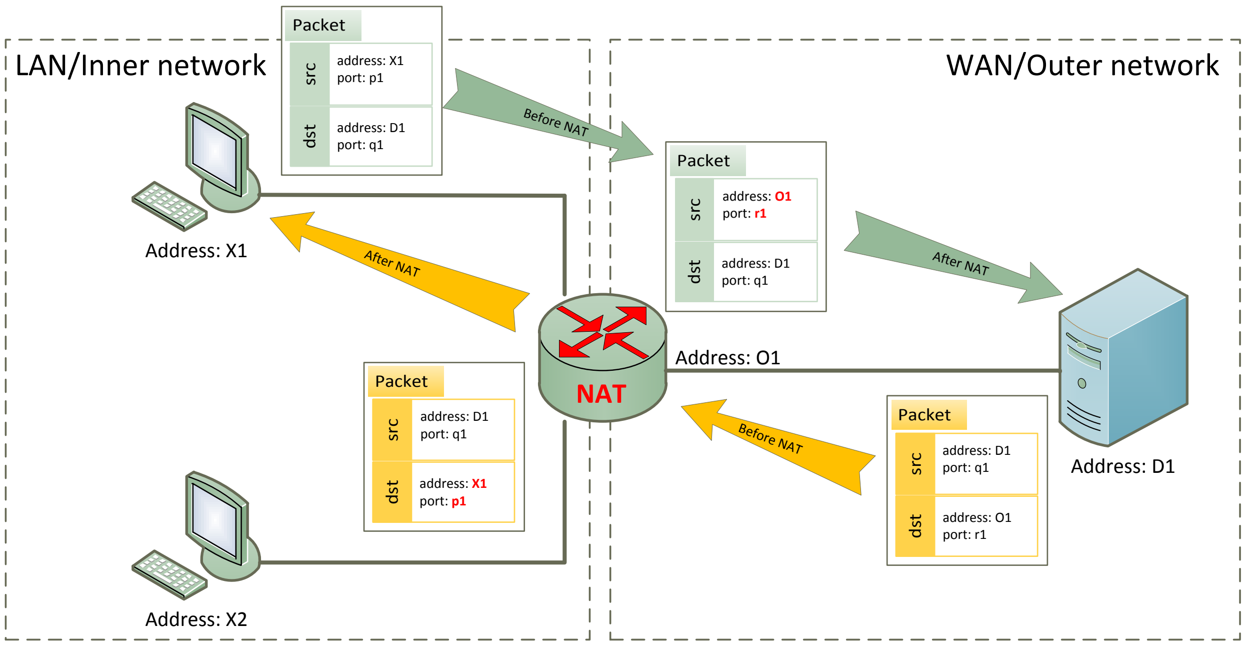
As the number of IPv4 addresses has reached its limit the addresses are turning into more and more valuable resource and the price of buying or renting it becomes higher and higher which means that more and more network devices will share the one IPv4 address for the Internet accessing [4].

Although, IPv6 seems to be a solution of the lack of IPv4 addresses the process of switching to IPv6 is quite slow. The most likely reason for that is the whole network setting changing necessity as well as a need to remove all legacy network equipment and software which are not compatible with IPv6. This will take a lot of efforts from the Internet providers and they are not ready to put many of them right away because of the investments needed. Instead, they keep working on IPv4 using NATs to reduce the number of “white” IPv4 addresses used.

1.2.2 NAT operation

This document is focused in exploring of traditional NAT setup as most frequently used. Although, other NAT setups also have its own application case but they are quite rare in the real world and omitted from the consideration.

The traditional NAT setup [5] looks as follows. There are two networks which considered by NAT as *inner* and *outer.* The inner network is the local area network served with NAT. The outer network is the wide area network which can communicate with inner network via the NAT only. There are two main NAT methods Basic NAT and NAPT (Network Address Port Translator). Basic NAT performs IP address translation only (i.e. changing a source IP address of a packet to an IP address allocated to NAT for translation)without changing of port number when NAPT does translation of a tuple {IP, port (TCP/UDP/ICMP)}. These days NAPT method is mostly used and furthers in this document it is implied when saying NAT.



Scheme 1. NAT operation routine

NAT operations as follows: a node from inner network sends a packet to a node in the outer network. The packet comes to the NAT. The NAT gets the packet, allocates an IP address and port number for translation. Then, the source IP address and port number stores in the NAT with respect to just allocated IP and port number to perform backwards translation. The packet source IP address and port number is replaced with the allocated tuple and checksums of IP and TCP/UDP/ICMP headers of the packet are recalculated. After that, the packet is sent to the destination node in the outer network. The outer network node receives the packet and sends a replying packet using the tuple of source IP address and porn number from the just received packet as the destination IP address and port number in its packet. The replying packet comes to the NAT. The NAT using the destination address and port number from the replying packet looks for the corresponding tuple of IP address and port number saved previously. Having found the tuple, the NAT performs destination IP address and port number replacement in the replying packet as well as changing of checksums in IP and TCP/UDP/ICMP packets. Then the replying packet is sent to the node in the inner network which was an originator of connection.

Although, NAT supplies transparency to inner and outer nodes communication (i.e. the nodes know nothing about NAT existing) it has a serious disadvantage. There are a set of application protocols (FTP, DNS, PPTP, H.323, etc.) working onto TCP/UDP that store the connection data (i.e. IP address and port number) inside their packet on the level upper than L4 of ISO model where TCP/UDP works. This fact leads to inconsistency of IP addresses and port numbers in translated by NAT packets where source IP and port number in TCP/IP headers doesn't match to the source IP and port number in the upper level protocol headers. This problem is solved with ALGs (Application Layer Gateway) working with NAT. There are different protocol specific ALG. Each ALG protocol are able to distinguish and process the packets belongs to the protocol accordingly. This document is not focused on using ALGs in NAT and only core traditional NAT functionality is taken into consideration.

1.2.3 NAT behavioral requirements

NAT behavioral requirements for TCP, UDP and ICMP are clearly stated in [6, 7] and [8] respectively. Here the generalized list of NAT behavioral requirements based on mentioned documents is shown.

1. A NAT must have an "Endpoint-Independent mapping" behavior. Endpoint-Independent mapping means that the NAT reuses port mapping for subsequent packets sent from the same internal IP address and port to any external IP address and port [7].
2. A NAT must support all valid sequences of TCP/UDP/ICMP packets for connections initiated both internally as well as externally, when the connection is permitted by the NAT. In addition to handling the TCP 3-way

handshake mode of connection initiation, A NAT must handle the TCP simultaneous-open mode of connection initiation

3. If application transparency is most important, it is RECOMMENDED that a NAT have an "Endpoint-Independent Filtering" behavior for TCP. If a more stringent filtering behavior is most important, it is RECOMMENDED that a NAT have an "Address-Dependent Filtering" behavior.

   Endpoint-Independent Filtering means that sending packets from the internal side of the NAT to any external IP address is sufficient to allow any packets back to the internal endpoint.

   Address-Dependent Filtering means that for receiving packets from a specific external endpoint, it is necessary for the internal endpoint to send packets **first** to that specific external endpoint's IP address

4. A NAT must not respond to an unsolicited inbound TCP SYN packet for at least 6 seconds after the packet is received. If during this interval the NAT receives and translates an outbound TCP SYN for the connection the NAT must silently drop the original unsolicited inbound TCP SYN packet. Otherwise, the NAT should send an ICMP Port Unreachable error (Type 3, Code 3) for the original TCP SYN. The NAT must silently drop the original TCP SYN packet if sending a response violates the security policy of the NAT

5. If a NAT cannot determine whether the endpoints of a TCP connection are active, it MAY abandon the session if it has been idle for some time. In such cases, the value of the "established connection idle-timeout" must not be less than 2 hours 4 minutes. The value of the "transitory connection idle-timeout" must not be less than 4 minutes. The value of the NAT idle-timeouts may be configurable.

6. A NAT UDP mapping timer must not expire in less than two minutes, unless the port number is from the well-known port range of 0-1023. In

that case NAT may have shorter UDP mapping timers. The value of the NAT UDP mapping timer MAY be configurable. A default value of five minutes or more for the NAT UDP mapping timer is recommended.

7. A NAT must not have a "Port assignment" behavior of "Port overloading". Another words, the NAT must not always use port preservation even in the case of port collisions
8. It is recommended that a NAT have an "IP address pooling" behavior of "Paired". It means that the NAT use the same external IP address mapping for all sessions associated with the same internal IP address. This requirement is not applicable to NATs that do not support IP address pooling
9. It is recommended that a NAT have a "Port parity preservation" behavior which means that after the NAT processing an even UDP port will be mapped to an even UDP port, and an odd UDP port will be mapped to an odd UDP port.
10. If a NAT includes ALGs it is RECOMMENDED that all of those ALGs (except for FTP) be disabled by default.
11. A NAT must support "hairpinning" for TCP/UDP/ICMP. A NAT's hairpinning behavior must be of type "External source IP address and port". This means that two nodes are behind the same NAT both use for communication to each other different tuples of external IP address and port allocated by NAT
12. Unless explicitly overridden by local policy, a NAT device must permit ICMP Queries and their associated responses, when the Query is initiated from a private host to the external hosts
13. An ICMP Query session timer must not expire in less than 60 seconds. It is recommended that the ICMP Query session timer be made configurable

14. When an ICMP Error packet is received, if the ICMP checksum fails to validate, the NAT should silently drop the ICMP Error packet. If the ICMP checksum is valid, do the following: If the IP checksum of the embedded packet fails to validate, the NAT should silently drop the Error packet; If the embedded packet includes IP options, the NAT device must traverse past the IP options to locate the start of the transport header for the embedded packet; the NAT device should not validate the transport checksum of the embedded packet within an ICMP Error message, even when it is possible to do so; if the ICMP Error payload contains ICMP extensions, the NAT device must exclude the optional zero-padding and the ICMP extensions when evaluating transport checksum for the embedded packet.
15. If a NAT device receives an ICMP Error packet from an external realm, and the NAT device does not have an active mapping for the embedded payload, the NAT should silently drop the ICMP Error packet. If the NAT has active mapping for the embedded payload, then the NAT must do the following prior to forwarding the packet, unless explicitly overridden by local policy: revert the IP and transport headers of the embedded IP packet to their original form, using the matching mapping; leave the ICMP Error type and code unchanged; modify the destination IP address of the outer IP header to be same as the source IP address of the embedded packet after translation.

16. If a NAT device receives an ICMP Error packet from the private realm, and the NAT does not have an active mapping for the embedded payload, the NAT should silently drop the ICMP Error packet. If the NAT has active mapping for the embedded payload, then the NAT must do the following prior to forwarding the packet, unless explicitly overridden by local policy:

revert the IP and transport headers of the embedded IP packet to their original form, using the matching mapping;

17. leave the ICMP Error type and code unchanged; if the NAT enforces Basic NAT function, and the NAT has active mapping for the IP address that sent the ICMP Error, translate the source IP address of the ICMP Error packet with the public IP address in the mapping. In all other cases, translate the source IP address of the ICMP Error packet with its own public IP address
18. While processing an ICMP Error packet pertaining to an ICMP Query or Query response message, a NAT device must not refresh or delete the NAT Session that pertains to the embedded payload within the ICMP Error packet.
19. When a NAT device is unable to establish a NAT Session for a new transport-layer (TCP, UDP, ICMP, etc.) flow due to resource constraints or administrative restrictions, the NAT device should send an ICMP destination unreachable message, with a code of 13 (Communication administratively prohibited) to the sender, and drop the original packet.
20. A NAT device MAY implement a policy control that prevents ICMP messages being generated toward certain interface(s).
21. Receipt of any sort of ICMP message MUST NOT terminate the NAT mapping or TCP connection for which the ICMP was generated.

The requirements listed above do not guarantee the compliance of NAT with all application protocols but fulfilling them significantly improves the likelihood of successful processing of any kinds of packets.

1.2.4 Carrier grade NAT (CG-NAT)

The NAT supplies the ability to share a single external IP address among several nodes in the external network. Some of internet service providers have started offering this service long before IPv4 address space shortage problem has arisen showing that there is another driven force of using NAT. Each subscriber at the ISP's network assigned a private address and the NAT, situated at the customer edge, translates traffic between public and private addresses [4].

Because of scales of ISP's networks NATs used there have some additional functional and determined performance requirements. The list of NAT functional requirements is added with following points (taken from [4]) extending the points shown previously in this chapter:

1. The CGN function should not have any limitations on the size or the contiguity of the external address pool. In particular, the CGN function must be configurable with contiguous or non-contiguous external IPv4 address ranges
2. A CGN MUST support limiting the number of external ports (or, equivalently, "identifiers" for ICMP) that has been assigned per subscriber. Per-subscriber limits must be configurable by the CGN administrator. Per-subscriber limits may be configurable independently per transport protocol. Additionally, it is recommended that the CGN include administrator-adjustable thresholds to prevent a single subscriber from consuming excessive CPU resources from the CGN (e.g., rate-limit the subscriber's creation of new mappings).
3. A CGN should support limiting the amount of state memory allocated per mapping and per subscriber. This may include limiting the number of sessions, the number of filters, etc., depending on the NAT implementation. Limits should be configurable by the CGN administrator.

Additionally, it should be possible to limit the rate at which memory-consuming state elements are allocated.

4. It must be possible to administratively turn off translation for specific destination addresses and/or ports.
5. Once an external port is deallocate, it should not be reallocated to a new mapping until at least 120 seconds have passed, with the exceptions being If the CGN tracks TCP sessions TCP ports MAY be reused immediately. If external ports are statically assigned to internal addresses, the assignment remains constant across state loss, than ports may be reused immediately. If the allocated external ports used address-dependent or address-and-port-dependent filtering before state loss, they may be reused immediately. The length of time and the maximum number of ports in this state must be configurable by the CGN administrator.
6. A CGN must implement a protocol giving subscribers explicit control over NAT mappings. That protocol SHOULD be the Port Control Protocol.
7. CGN implementers should make their equipment manageable. Standards-based management using standards such as "Definitions of Managed Objects for NAT" is recommended
8. When a CGN is unable to create a dynamic mapping due to resource constraints or administrative restrictions (i.e., quotas): it must drop the original packet; it should send an ICMP Destination Unreachable message with code 1 (Host Unreachable) to the sender; it should send a notification (e.g., SNMP trap) towards a management system (if configured to do so); it must not delete existing mappings in order to "make room" for the new one. (This only applies to normal CGN behavior, not to manual operator intervention.)

The requirements listed above do not guarantee the compliance of NAT with all application protocols and is based on the best practices. Fulfilling the requirements significantly improves the likelihood of successful processing of any kinds of packets while NAT working on the ISP edge.

Another important point is CGN performance. Although, there is nothing about it stated in the requirements, it worth a lot because of the nature of the network serving with CG-NAT. As CG-NAT is an ISP's appliance and ISP's network consist of tens of thousands of nodes the CG-NAT should provide sufficient performance level to satisfy the customer demands. In case of CG-NAT the performance seems to be a more critical characteristic than strict compliance with a standard. From the marketing point of view the performance could be more important than the fulfilling the standard functionality because no one would buy a system that fulfilled the functionality requirements and didn't perform them with proper pace and vice versa, if the performance are enough than one could possibly turn a blind eye to some lack of the features.

1.2.5 NAT Performance metrics

The goal of this work is to develop a working prototype of software defined carrier-grade network address translator (SD CG-NAT). To make sure that the SD CG-NAT is close to reality in terms of performance it is necessary to define the performance metrics and set their values. In order to get those metrics, a couple of sources are used. The first one is Rostelecom technical requirements for CG-NAT [9]. The second one is the performance specification claimed by one of the on-market available NAT device producers [10] which employ the same approach as this research does: **to use not task specific computer (a commodity server) to make a network specific solution using a mix of algorithmic and technological approaches.**

NAT performance metrics:

- ***Packets processing rate*** – (packets per second [PPS]) – the router's maximum rate of packet processing. This is the main metric describing the packet processing abilities of a NAT device.
- ***Concurrent session support*** – (number) – the maximum number of sessions produced by served network. It describes the maximum network size which can be served by the NAT device. As described later in this document than bigger the network than harder to maintain translations to its nodes.
- ***Connections setups rate*** – (connection setups per second [csps]) – the number of new NAT records to be created in a second. This metric shows the NAT ability to create new NAT records and could be a drawback of the NAT device in a certain modes of network work like when the networks nodes start creating of new connections actively, for example in the beginning working hours
- ***Throughput*** – (bit per second [bps]) **–** it isn't very clear metric of the NAT device because it is mostly defined with NIC (network interface card) performance used by NAT device. If the NAT device won't have enough of packet processing rate its throughput can't achieve the maximum throughput provided with NIC and vice versa. The main sense of having it in the metric list is to make sure that NAT device is able to transfer needed amount of information.
- ***Latency*** – seconds [sec] – time needed for one packet processing. This metric is important when evaluating the minimal time frame of one packet processing to know what part of runtime is needed for changing the packet data. This can be helpful when comparing performance growth and scalability.

This set of characteristics is usually used by equipment vendors while describing their competitive advantages. Thus, using it will allow one to be on the same page with all the professionals working in the field of computer network devices.

In this document for evaluation of the performance another characteristic is used: ***Cycles per packet*** – [cps] – the amount of processors' cycles spent on processing of one packet. This characteristic seems to be more descriptive than others while describing the NAT performance because there are a lot different processors which differ to each other with CPU frequency and technologies used which makes it harder to compare the performance of the NAT on different processors using the set of metrics described earlier in this chapter. Cycles per packet characteristic gives clearer impression of the performance because at least it doesn't strongly depend on CPU frequency however there are other limiting factors influencing on the characteristic value such as system bus frequency and memory frequency. Another drawback is that this characteristic becomes quite confusing when trying to describe the performance on multiple cores. Thus, the main performance characteristic used in this work for assessing the performance is Cycles Per Packet and is used mainly for choosing the best working approach. The target metrics values are set in the following paragraph and are used as the requirements to the NAT settings and abilities.

1.3 NAT implementations comparison and analysis

There are two types of NAT: software and hardware.

1.3.1 Software NAT

Software NAT is a program that implements NAT functionality and works on the top of operating system. There are two kinds of software NAT implementation. One is a user program working in the user mode totally like *NAT32 IP Router* [11] or *WinGate* for windows or *IPFilter* for Linux and using the OS resources to access the network. It can be installed or uninstalled by user's demands. There are commercial versions of this kind of NAT as well as free once.

Another kind of software NAT is a program working in the kernel space like Linux module or is a part of operation system like *ICS* in Windows or *iptables* in Linux. As this kind of NAT distributed along the operation systems it is usually free. Some of UNIX family software NATs have opened source code and are free for changing.

1.3.2 Hardware NAT

Hardware NAT is a specialized network device or a feature of specialized network device like firewall or router. These devices usually have their own specific operation system and interfaces for management. The hardware core of hardware NAT device consist of a specific processor designed for fast packet processing as well as associative memory and another chips that increase the performance in specific operations. Hardware NATs are produced by telecommunication producers like Cisco, Juniper and others. These devices have high performance and are expensive.

1.3.3 Software and hardware NAT comparison

Functionality: As hardware NAT is usually a part of industrial firewall or a router which is used at Internet Services Provider's or Data Center's facilities it is equipped with vast functionality where NAT is one of many. The functionality includes firewall, VPN support and crypto security features allowing a customer to have many of needed abilities in one box. Software NAT is often is an integral part of operation system and its functionality is not that advanced because it is usually used for small offices or tiny private networks. Talking about NAT core functionality, both of them provide full support of NAT requirements.

Performance: Hardware NAT is specially designed for high performance used the cutting edge hardware for achieving it. In particular, for the performance improvement special memory units are used called CAM (Content-addressable memory a.k.a. associative memory). CAM(TCAM) is extremely fast in tasks of comparing input data against stored data and returning matched data as a result. Although CAM is fast it has low memory capacity and very high price. One unit with capacity of 80Kbytes costs around $180 while hundreds of these units needed in a router working at an ISP’s rack.

Software NAT is installed in commodity computers and has a serious performance limiting factor. As software NAT use OS system as the source of network resources it uses OS system calls to get them and, hence, is limited by the performance of that system calls which means that the NAT is not able to outperform the OS it uses. Thus, the main limiting factor is the network stack used by the operation system. The experiment revealed the packet performance rate for Linux 3.16 kernel around 260 Kpps.

Upgradability: The hardware NAT is a set of hardware mounted in some kind of chassis with a piece of specialized software pre-installed which is hardly be compatible with different set of hardware because of technical and vendor limitations. Another thing is that because of the software, controlling the hardware NAT is proprietary and upgrading might be an issue. This makes the only way of upgrading this equipment possible: buying a new software update, support plan, new hardware NAT device from the vendor. Unlike hardware NAT, software NAT can be installed at any system supporting the OS the software NAT specialized. Thus, increasing of performance is possible by updating the hardware where the software NAT installed. As software NAT is just a program than the upgrading/updating could be done by using the same approach as other pieces of software used with paying no attention to the hardware used. This makes the software NAT more flexible in terms of modification and customization.

Price: The price of the system implementing NAT functions based on software NAT consists of several parts: price of the computer, price of the OS and the price of software NAT itself. The price of software NAT program was around $2200 (Wingate [12]) for maximum functional version at the time of writing. There are free versions of software NAT working on free OS like iptables on Linux. In that case the price depends on the price of computer used only.

As for hardware NAT the prices are much higher. This is mainly due to pieces of hardware used in producing of such devices but the vendor interests also play a big role. The marked research has been done to evaluate the prices and the results are shown in Figure 1.

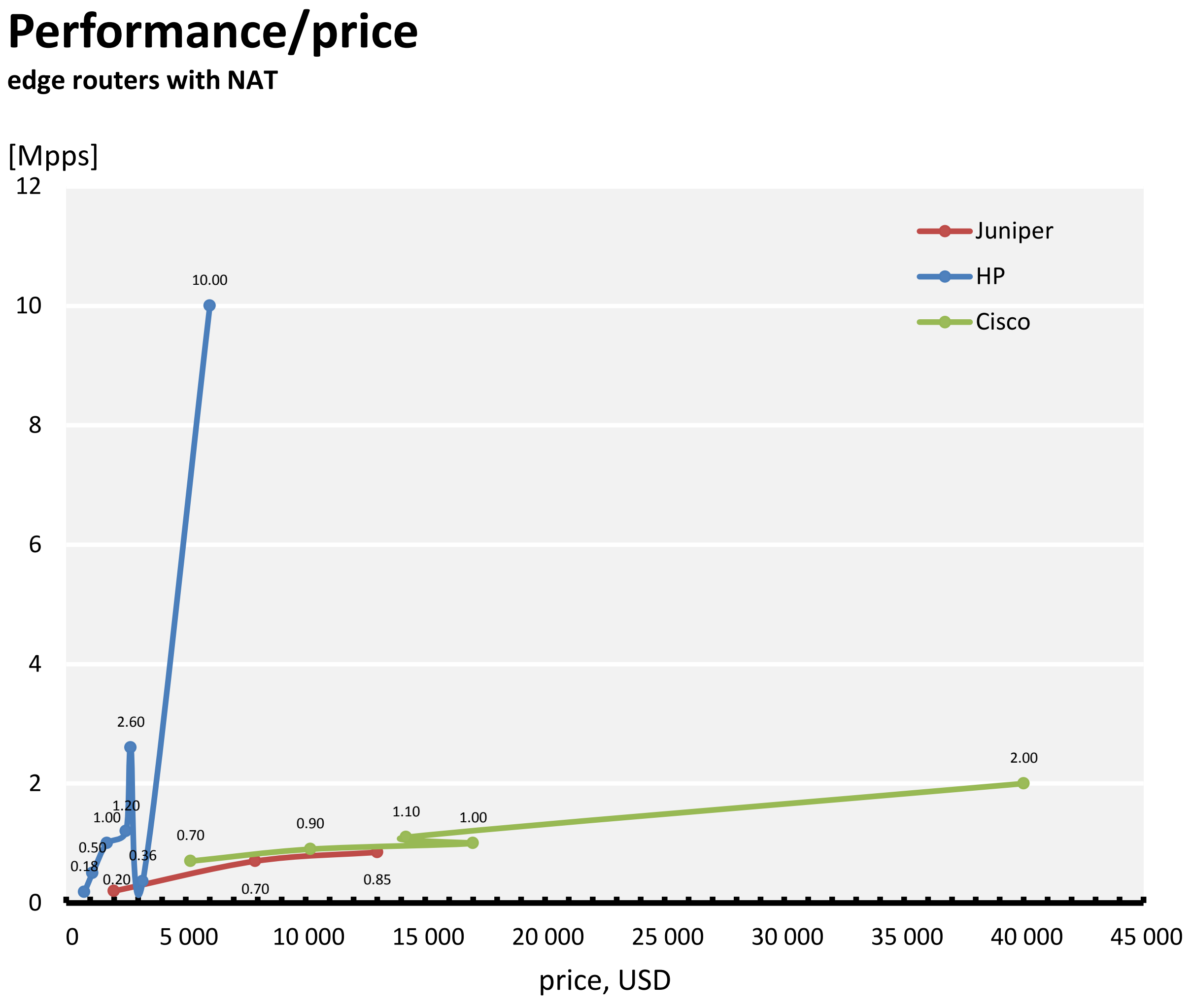


Figure 1. Performance/price relation plot of the modern edge routers supporting NAT

For evaluation a set of edge routers was taken produced by three well known telecommunication equipment vendors: HP, Cisco and Juniper. The performance of the NAT mode has not been found and firewall on mode was used as an approximation. The packet processing rate and related price for the set of HP devices seems to be not consistent with the same data for Cisco and Juniper devices. This is could be a consequence of difference in the packet processing rate definition and measuring methodology used by different vendors. When claiming performance, Juniper and Cisco use packet processing rate with firewall turned on. HP doesn't give any clarification about the mode they used for packet processing rate measuring. The data found for Juniper and Cisco routers shows pretty much the same trend and, thus, it can be considered as trustworthy and showing the current situation on the market.

It is seen from the chart that high performance costs a lot: roughly 15 000 USD for each 1 Mpps which makes buying of the router the matter of capital investments for ISPs.

The data for the chart was collected from the public resources on the Internet. The prices are relevant for Russian market, for other countries prices may vary. The list of models used is in Appendix A.

1.3.4 The comparison and analysis results

The results of comparison are shown in Table 1. The analysis revealed the situation as following. Hardware NAT provides vast functionality and high performance wherein the price is much higher than software NAT provides. The reason for the lower software NAT performance is limited packet processing ability of underlying OS. Also, Hardware NAT is hard to upgrade because of the high cost whereas Software NAT is easily updatable because it is just a program working on top of operation system.

	Hardware NAT	Software NAT
Additional functionality	High	Low
Performance	High	Low
Upgradeability	Low	High
Price	High	Low

Table 1. Advantages and disadvantages comparison of hardware and software NATs. The advantages marked with red color

1.4 NAT improvements. The software CG-NAT. The CG-NAT target characteristics.

How to improve NAT: Based on the analysis, the reasonable improvement would be increasing of performance of the Software NAT. In this case it could significantly increase the number of cases where Software NAT could be used instead of Hardware NAT. It is implied that software NAT would be cheaper than hardware NAT but this is a matter of exploration. If the software NAT was cheaper than the hardware one it would reduce the investments on building the network. Having these properties, the software NAT could be applied at ISP's facilities playing the role of a CG-NAT. In that case the software NAT should have a certain values of performance metrics.

This work is focused on exploring feasibility and reasonability of software NAT development that could be compared with hardware NAT in performance and could be used as a CG-NAT. Further in this work the name "the software CG-NAT" is used to denote the goal of exploration.

The performance metrics target values: As it was mentioned earlier the software CG-NAT should have the target metrics values. Based on the sources of information [9, 10] it is reasonable to set the performance requirements of the software CG-NAT device to the following values:

- Packet processing rate: ≥ 5.5 Mpps
- Concurrent session support: ≥ 65.5M (a B-class network with up to 1000 active ports for each node)
- Connection setups rate: ≥ 3 Mcsps
- Throughput: 10 Gbps
- Latency: ≤ 180 ns

These values are relevant to performance demands of the ISP's which are the users of CG-NAT.

Part 2. ANALYSIS OF NAT DESIGN APPROACHES

2.1 Overview of high performance software design principles

The essence of NAT is packet processing. To increase the ability of NAT to process the packets certain kinds of software design patterns should be employed. There are a number of works and books related to software design for fast packet processing] and underlying in chapter 5 of describing the technics capable to increase the packet processing performance [13, 14, 15, 16] and to boost overall system performance (chapter 5, 6, 7 of [17]). The list of these principles as following:

Zero-copy policy: Each data copying spends hundreds of processor cycles to perform the memory allocation for new copy of coping itself. To eliminate this unnecessary operation the zero-copy policy should be employed. Zero-coping means that while processing a packet the software doesn't make any copies of the data and for data manipulation the pointers are used. Using this technic allows making only one copy of data for the whole packet processing round when copying the data to transmission queue which significantly reduce the amount of possible memory allocations.

Cache optimized data access: Modern processors use multilevel cache memory based on fast but expensive SRAM. The cache represents a table consist of cache lines where the data stored. The cache line stores the chunk of continuous memory. To get the access to some memory address the processor, firstly, check if this address is currently in the cache and if not, than load this chunk of memory from the main memory. This situation is called a cache miss. If the processor finds the chunk of memory in its cache it is called a cache hit. A cache miss costs a processor around a hundred of cycles which increase overheads of runtime. To decrease the cache miss rate using principle of locality should be considered. There are two types of localities. Due to the cache line nature of memory loading for reducing the reads from the main memory the data in memory have to be stored in continuous manner. This allows storing the data, to be used, in the cache in advance and then use it without additional readings. This is called spatial locality. Some data is needed to be used frequently during the runtime. This is called temporal locality. This kind of data has to stay in the cache but can be discarded from the cache because of a program's memory access pattern. Due to that while the program development this specialty should be taken into the account and the memory pattern access should be adopted accordingly.

Batching: It is a method that allows eliminating possible delays for reading packets from a network interface card as well as employs cache optimized data access by processing the packets in blocks. Processing the block of data gives increasing of data and instruction locality which reduces the overheads in comparison with sequential packet processing approach.

Parallel processing: Processing the data in parallel manner makes use of multiple cores available on modern processors. Theoretically, splitting the works into several threads gives maximum boost of performance directly proportional to the number of threads used. In practice, this boost is not achievable as stated in Amdahl's law [18]. Therefore, to benefit from the parallel technics the program should use designed so, that the sequential portion would be as small as possible. Another pitfall of parallel programming hides in the processor's hardware design. If some threads read from and write into the same shared data structure and the data structure is not adopted to multithreading use it is highly possible that the performance of the program will suffer from the consequences related to the cache coherence. The problem is that a thread can write to some value which is stored in some memory address which, in turn, is stored in cache of several other cores because of cache line properties. This write induces the other cores, having this address in their caches, to update the address's value. This operation takes some time and is an unnecessary overhead when the value is not relevant to the algorithm essence. This property of the multi-core processors cache should be taken into consideration when using multi-thread programming technics.

Avoiding OS's networking facility: … don't use Linux/Windows TCP/IP stack

Operation Systems implement a large number of network protocols providing vast functionality related to the networking. These network facilities are OS specific and optimized to be used by the operation system demands. The disadvantage is that this functionality is controlled by OS, accessed via system calls and is not designed and, thus, destined for using in high performance network applications. Instead of using the OS-supplied network functionality, alternative ways have to be considered like using a high performance network stack or a Data Plane framework.

Interruption avoiding: Both types of interruptions: software and hardware incur OS context switching and switches the processor from user mode to kernel mode where the interruption handler works which is fraught with doing additional work by processor increasing overheads and, therefore, program runtime. To avoid these overheads from the software interruptions point of view the system functions should be used with great carefulness or even be discarded if possible. From the hardware interruptions point of view the special network card drivers should be used which does not induce hardware interruptions.

2.2 Software NAT design exploration

2.2.1 NAT design overview

The most interesting part in the NAT system is the algorithm and data structure for storing the address translation information. There was a number of works related to this issue [19, 20, 21] Although, some technics described in these works can be employed but most of them are related to special network processors and equipment. In this work the implementation of NAT using a commodity computer is the matter of exploration. The commodity computer uses processors with the architecture different from the network ones. In the most cases it is x86 architecture which requires certain ways of high performance achieving.

The Network Address Translation process consists of several parts: packet receiving, packet translating (making a decision about packet changing, changing the packet headers) and sending the packet out.

Packet receiving and sending are related to the OS and hardware environment while the packet translating is related to internal program organization. To choose the design which would fulfill performance requirements it is necessary to determine the part of the system where performance reduction can occur.

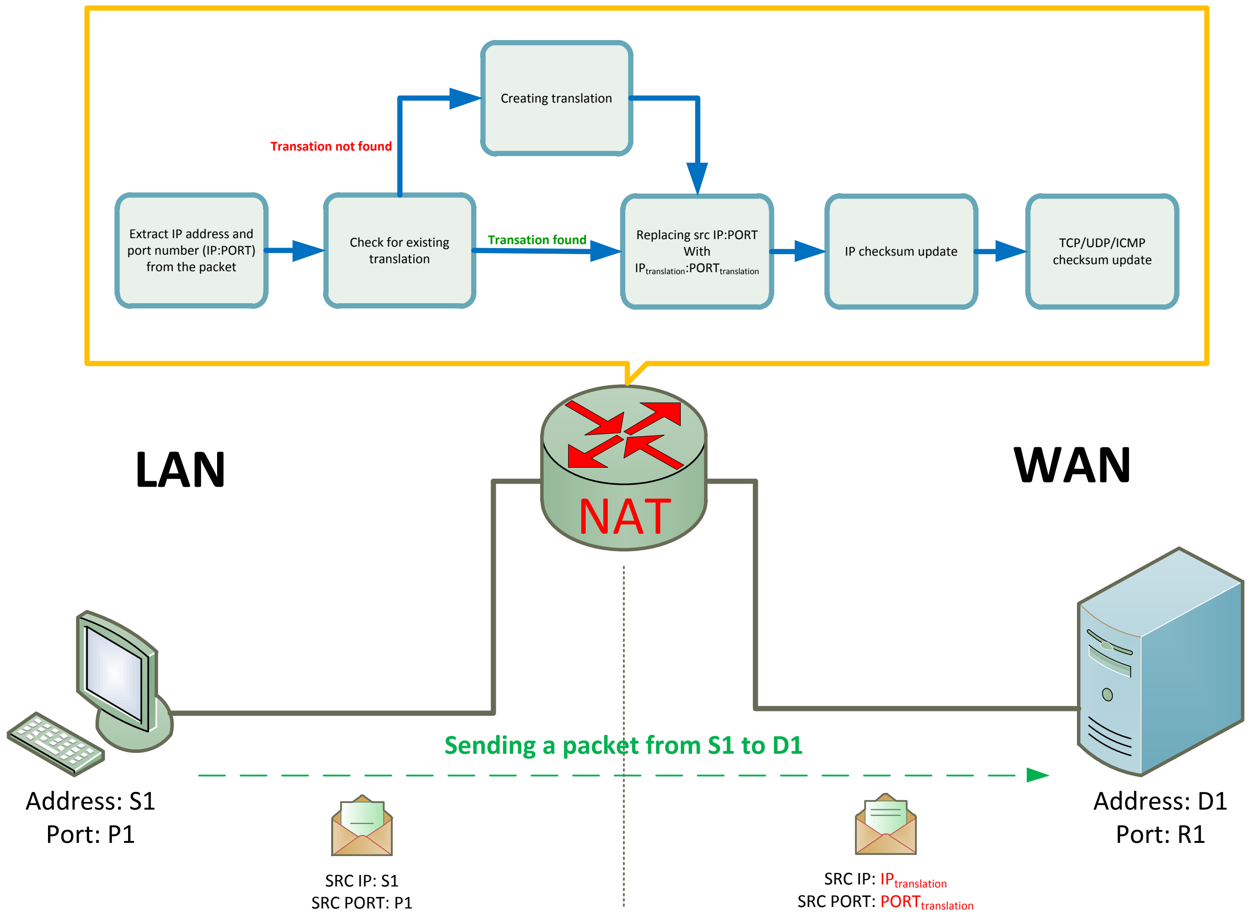
2.2.2 NAT bottleneck

For considering the bottlenecks the NAT system is conditionally split into 2 parts: transmitting part (packet receiving and sending) and processing part (translation).

The transmitting part is closely related to hardware and OS. Modern hardware (CPUs and NICs) provide exceptional level of performance. The previous works [22] show that the maximum raw packet retransmitting rate is 23.06 Mpps per 10 Gb port for 64-byte packet size. On bigger packet sizes the full throughput can be achieved. This limitation exists due to using PCIe Gen2 system bus. The processor is able to forward up to 92.22 Mpps. Having these numbers in mind, it is possible to claim that the hardware and OS (excluding its TCP/IP facilities) are able to exceed the required level of performance and, thus, are not the performance bottlenecks. This performance rate could be achieved by using Intel's DPDK framework [23].

The processing part consists of packet changing and translation managing parts. The packet changing part performs the packet headers modifications and has a constant runtime because it doesn't depend on input data. The tests, more closely described in part 3, shows that performing packet header changes only the packet processing rate can reach 20.1Mpps. This number also exceeds the required level of performance and is not a bottleneck.

The translation managing part is the most interesting one. It consists of several operations: storing the translation data, looking for the stored translation data, resources allocating and others. Storing the translation data and looking for the translation data are the core operations. They use a data structure and algorithm that perform storing and retrieving the data about translation to be done.


Creating translation
Transation not found
Extract IP address and port number (IP:PORT) from the packet
Check for existing translation
Transation found
Replacing src IP:PORT With IP$_{translation}$:PORT$_{translation}$
IP checksum update
TCP/UDP/ICMP checksum update
LAN
NAT
WAN
Sending a packet from S1 to D1
Address: S1
Port: P1
Address: D1
Port: R1
SRC IP: S1
SRC PORT: P1
SRC IP: IP$_{translation}$
SRC PORT: PORT$_{translation}$

Scheme 2. Processing of packet going from LAN to WAN

The NAT translation is a critical part because of the required amount of data to be stored. Than bigger the amount of data stored than bigger the time of searching in this data chunk is. Thus, the NAT translation managing part is the bottleneck. To reduce the amount of runtime to be spent on packet translation the translation storing data structure and searching algorithm should be found that provides higher or equal level of performance than required.

Having in mind that the NAT should be able to support 65.5M unique translations, it is easy to conclude that its lookup data structure have to be able to store 65.5M records and the search process will take a majority of packet processing time because of the size of this data structure. To achieve the target packet processing rate (5.5M pps) it is easy to calculate how much time available for a packet processing. The time frame for a packet processing is around 180 ns. Considering a processor working on frequency 2.4GHz we could spend no more than 436 cycles per packet. Hence, our target performance characteristic upper bound is 180 ns or 436 cycles per packet. This upper bound is used when evaluating the design approaches suggested further in this chapter in chapter 3.

2.2.3 The bottleneck overcoming. Exploration methodology

The bottleneck of the NAT is translation managing data structure and algorithm. It’s not obvious from the first glance how to choose them properly. Furthermore, there are a lot of data structures and algorithms to test and trying each of them seems to be a too time consuming task. The scope of this work is to find a solution for the NAT that satisfies the certain requirements, so the goal is not to find the optimal solution and it is not required to cover all possible data structures and algorithms. Instead of this, the following methodology is used. The data structures and algorithms searching started from the simplest variants. If the solution doesn’t satisfy the requirements, another, more complex, solution is suggested and tested. This process lasts while the suitable solution is found. The decision whether a solution is satisfying or not is made by testing it in the specially developed performance measurement program. The result, given by the tested algorithm, should be equal or lower the upper bound determined in the previous section.

2.2.4 Data structures and algorithms exploration

Linear search: So the first question to investigate is how fast the searching process is and does it really necessary to choose the algorithm and data structure. To answer that question the test has been performed which uses as a storing data structure a simple linear array with linear search algorithm [24]. This algorithm is known as having O(n) search time and can be a good starting point of performance exploration. With high probability this approach is not suitable because of the linear nature of performance degradation. Having in mind the necessity to store the data for 65.5M instances the linear search is not able to provide with high performance. It can give some benefits when searching in small data sets consisting of tens of records.

Binary tree: To increase the performance another group of algorithms is taken into consideration. The group of algorithms based on tree-like data structures provide O(logN) theoretical search time. The simplest algorithm in that group is a simple binary search.

Its performance looks potentially promising but it consumes additional memory on tree node linking, in particular, each node uses 3 additional pointers to keep link with its parent node and 2 child nodes (left and right). Each of these links consumes at least 4 bytes of memory (12 in total) which leads to increasing of a NAT table entry size at least to 60%. So the overall memory overhead is more or equal than 60% depending on the CPU architecture and OS used.

The disadvantage of this algorithm is dependence from the input data storing sequence. In the worst case the binary tree can turn into a linked list. Searching in linked list has O(n) searching time and has no benefits in comparison to the linear search. In fact, this situation is highly unlikely in the real world but has an implication of the binary tree being unbalanced. “Unbalanced” term means that the left and right branches of the binary tree have divers depth differing to each other more than 1. The consequence of this is that in the real world binary tree search cannot achieve O(logN) searching speed and the search time differs depending on the branch having the data to be searched.

RB-tree: There is an improved data structure and searching algorithm that fix the unbalancing property of the binary tree leaving the search time at the same level of O(logN). The well-known representative of that group is red-black tree data structure. Searching the data saved in an rb-tree has the same workflow as searching in a binary tree. The distinction is in the storing (removing) logic. The main difference is that after each storing a piece of data in the rb-tree, the rb-tree is modified in a way to keep its structure balanced. This slightly increases leaving at the same order of O(logN) the time of storing the data but prevents the rb-tree from turning into a linked list.

Hash lists: Another group of the data structures and algorithms to be explored is one that uses a hash table based technics. The advantage of this group is a constant time (O(1)) of searching. For searching the data stored a key is used. The key is produced with a hash function from some data. The hash function has a deterministic value for a given input. It provides with good performance allowing to use simple (i.e. computationally cheap) hash functions.

The main issue while using a hash table is collision arising which occur when the hash function produce the same result for any two or more inputs. There are several schemes of this problem resolving. The most frequently used is separate chaining with linked lists. This means that each cell of the hash table is the first element of a linked list. If the hash function produces the same output for two or more different inputs all these values are stored in the same linked list. For searching the data scanning of the linked list should be done which might be a problem if the linked-list is long because of the O(n) linked-list scanning time. Hence, the search time for the algorithm in case of using the modulo hash function is O(n/modulo_value). Other option of collisions resolving can be using a tree-like data structure instead of linked-list. Because of the O(logN) tree-based data structures searching time the search time for tree-like collision resolving in case of using the same modulo function as hash function is O[log(n/modulo_value)]. Theoretically the last collision resolving scheme gives better performance but in the real world the performance boost might be not so good because of the CPU memory using schemes. To fix the memory issues it is possible to employ “cache-friendly” technics. It means that the memory for storing the collision resolving chains is allocated in chunks which are less or equal to the processors cache line size. This gives the elimination of cache misses improving the overall performance. The results of using this technic is discussed in Part 3.

Parallel processing: Almost all modern CPUs offer several cores on a single chip. Thus, using several cores for NAT routine seems to be a promising idea.

When employing multicores technics for software developing there are several issues to aware of which make significant influence on performance speed up.

The first issue is the cache coherence which arises when using shared data structures. Because of a copy of current processing data is stored in a core’s cache, changing the data by one core leads to updating the data in all cores currently use it. This means that the data have to overwritten in some common place of memory and then once again reloaded by other cores. This process is expensive and could cost hundreds of cycles which leads to considerable performance degradation.

The second issue is keeping the data in consistent state which is closely related to using special data structures known as locks. A lock also could be a problem because it makes the cores get access to the data in a sequence manner which can lead to core idling, decreasing the degree of parallelization. In the worst case it can lead to the result when the multicore code works with the same (or even less) performance it single core version.

Trying to avoid these issues the following approach was used. The biggest degree of parallelization can be achieved in case when a process running on core is fully independent from other processes (i.e. isn’t used the shared data). In case of our system this is possible because modern network interface cards provides multiple queues which can be used by different cores in associated manner when a given queue is associated with one and only one core. In this case it can be seen as if a separate NAT process uses a core and a single network card and the amount of network cards installed in the system is more or equal to number of processors.

2.3 Implementation

For exploring the approach of building the NAT the testing application has been made. To simulate the NAT workflow several solutions have been implemented which use different data structures and software organization options described in paragraphs above. Conditionally the program is split into 3 parts: measuring part, generation part and simulating part.

The measuring part consists of the environment that performs testing routine and calculates the performance results. The metric produced by this part is cycles per packet. This metric is acquired by using **rdtsc** instruction which reads the internal processor tick counters. Then it recalculates the result from cycles to other metrics taking into account the CPU speed. The measuring part performs the number of tests, set by the user, and, produces the performance values.

The generation part generates a packet set to be processed by the simulation part. It imitates uniformly distributed network activity and stores generated packets in a one-dimensional array of structures which is the input to the simulation part. The packet set is fully stored in RAM of the computer. Time of packet set generation isn’t taken into account when calculating the performance of the algorithm.

The simulation part is a core of the testing application and consists of NAT routine actions. There are several mandatory actions which must be performed by any NAT to perform proper packets translations. They are: calculation of the checksums, setting time stamp and saving/acquiring translation information in a NAT lookup data structure.

There are several necessary action to be performed by the NAT in order to perform address translation properly besides changing of packet's IP address and number of TCP/UDP port in the corresponding headers. They are: calculation of the checksum for IP and TCP/UDP/ICMP headers and storing the timestamp of the particular translation.

The checksum calculation is related to the packet processing. This action should be performed each time when the packet translation occurs and a packet IP and port number changed in order to be consistent with the requirements of the IP [1.4 of 25] and TCP/UDP/ICMP protocols [1.5 of 26, 27, 8].

The storing of the timestamp translation in the NAT translation data structure is necessary and cannot be eliminated because of the "Mapping Refresh" requirement for NATs [7].

For the testing purposes in the NAT testing program the following function implementations are used. For checksum calculation *ip_fast_csum()* from Linux kernel is used. For getting/setting the timestamp *gettimeofday()* Linux system call is used. However, some different, faster, source of timestamp data can be used, for example CPU ticks counter which is faster but trickier when converting it to the physical time. These functions might be potential targets of performance optimization but are out of the scope of this document.

To store data about each address translation it is necessary to make the following record:

- source packet IP address [4 bytes] – IPv4
- source packet port number [2 bytes]
- IP address assigned by translation [4 bytes] – IPv4
- port number assigned by translation [2 bytes]
- translation timestamp – to calculate timeout [4 bytes]
- some additional service info (L4 protocol, flags) [4 bytes]

The total record size is 20 bytes. As it is needed to perform 2 translations for each connection it is necessary to save 2 records associated with that connection. So the total amount of data to save is 40 bytes per connection. This amount of data can be reduced. The memory reducing technic is described in the following chapters.

The target capacity of the NAT translation information data structure is 65.5M records which makes the NAT data structure space consuming. Having stored IP address and port number for each unique translation, minimum size of the data structure size is 65.5M * [4 (IP) + 2 (port) + 2(timestamp)] = 524 Mb. The NAT must be able to perform two translations for a single connection: from its inner network to its outer network and vice versa. So, it should have 2 similar data structures to store corresponding translation information. Thus, the amount of memory to be allocated is 2 * [data structure size] which is 1048 Mb in our case. This amount of memory is reduced by using a hash table with direct addressing for incoming packets. All translation specific data is stored it that hash table. The address for the hash table access is produced from incoming packet destination IP address and port using IP address as offset and port number as an index. The outgoing NAT table refers to that hash table and doesn't store any translation specific data.

There are implemented several version of the code: one for each approach suggested in 2.2.4 to get the data about the performance and make a decision about approach applicability.

The testing application was developed using C programming language with GCC 4.8.2 compiler(compiler flags were: -std=c99 -O3 -mavx2) . The host operation system was native 64-bit Ubuntu 14.04 LTS (kernel version 3.13.0).

For multi-threading ***pthread*** library was used. The approach of parallelizing the code is following. A process shares the same generated packet set but the set is split into parts and this parts associated to each thread so that no process packets reads or writes interfere to other process packets reads or writes. Each thread is a separate network address translator with fully independent data structures (i.e NAT tables) which eliminates nearly all the pitfalls described 2.1 in the previous paragraph but leads to memory overheads.

Part 3. RESULTS

3.1 Evaluation

3.1.1 Measurement setup

All tests were done using a Lenovo U430p laptop with the following characteristics:

- CPU: Intel i5-4210U (Haswell-ULT)
- RAM 8Gb Single-channel DDR3 @800MHz
- Motherboard: LENOVO Cherry 4A (U3E1)
- OS: Ubuntu 14.10

3.1.2 Experimental methodology

In getting metrics values the test setup plays a key role. The values of the metrics are highly dependent on the test methodology.

To get the values of interest the following setup was used. There were packets with mostly unique (more than 99%) tuples of IP and Port number in the packet set. The test routine gave this packet set as an input to the NAT system. The NAT system processed each packet and changed the values of IP, port number and checksums in the packet saving this data at the same packet set. Once all packets processed the routine performed backward translation simulating the response of the node from the outer network to the just translated and transmitted packet. After all packet processing has been done the check for translation correctness was performed.

This testing routine was used in order to simulate the most intensive regime of network working: the nodes of the network are constantly trying to communicate with nodes in the outer network but the NAT device is offline, then the NAT device is switched on and right away starts serving the nodes connection, creating and performing new translations. This routine is more computationally intense than packet translation because the creating of a new connection costs more than a packet translating as it includes search for the connection translation data and if it was not found creating of a new translation entry is done.

This is the worst case scenario of network operating. The kind of testing used, allows getting the fair level of the NAT device performance.

3.2 Results discussion

The following paragraphs show sequential improvement attempts and results achieved.

From now and further in this document the NAT translation lookup data structure is called a NAT table and the translation record is called NAT table entry. The NAT table and the NAT table entry structures and sizes can differ in further described experiments depending on underlying data structure used in each particular experiment.

3.2.1 Results achieved

As it is seen from the Figure 1 the processing time of one packet excluding searching for translation data is a constant. This process is quite fast and can be compared with processors L3 cache miss penalty which is around 100 cycles.

So the first question to investigate is how fast the searching process is and does it really necessary to choose the algorithm and data structure. To answer that question the test has been performed which uses as a lookup data structure a simple linear array with linear search algorithm [24]. This algorithm is known as having O(n) search time and can be a good starting point of performance exploration.

The results of testing the algorithm are shown in Figures 1 and 2. Linear search revealed the high linear performance degradation with increasing of the NAT records capacity: at size of 2000 entries the time of a packet processing is 3 times higher than at size of 500 entries and 3 times higher than the target performance.

These results show that the translation data search is a real and quite serious bottleneck of the NAT performance and to solve this problem some effective algorithms and data structures are needed.

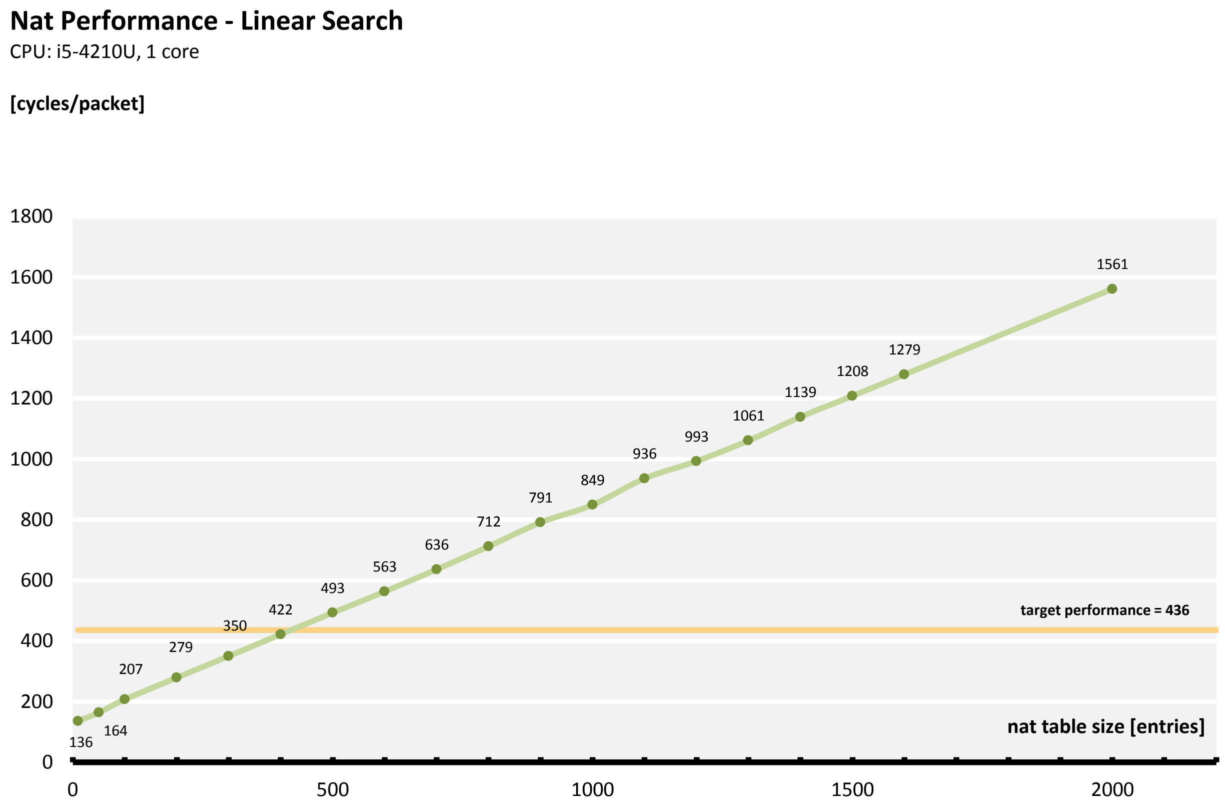


Figure 1. NAT Performance: Linear search in cycles per packet

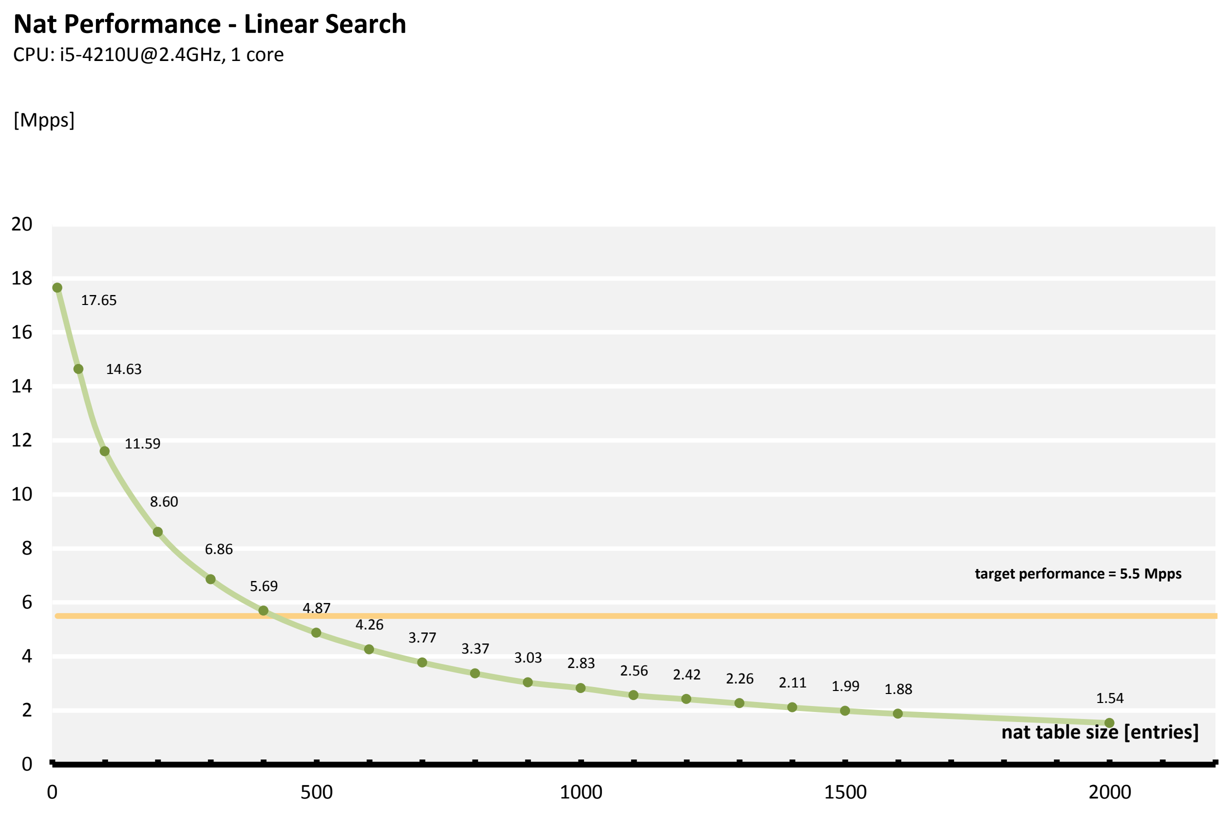


Figure 2. NAT Performance: Linear search in packets per second

The performance of tree-based data structures look potentially promising but it consumes additional memory on tree node linking, in particular, each node uses 3 additional pointers to keep the link with its parent node and 2 child nodes (left and right). Each of these links consumes at least 4 bytes of memory (12 in total) which leads to increasing of a NAT table entry size at least to 60%. So the overall memory overhead is more or equal than 60% depending on the CPU architecture and OS used.

The drawback of binary from the hardware point of view is low level of spatial locality. This happens because of the binary tree node's creation routine. The binary tree allows storing as many values as needed but, because of that, many memory allocations happen in different time frames are. Because these allocated chunks of memory are in the different parts of physical memory the CPU has to load each node in the memory separately instead of loading several of them at a time. As the NAT table size is much larger than a cache, even on a highly advanced

CPUs with a big cache size, only some of the most frequently used nodes stays in the cache. All other nodes are constantly removed and stored again which means that the CPU spends a majority of time on data transferring from the main memory to the cache and vice versa.

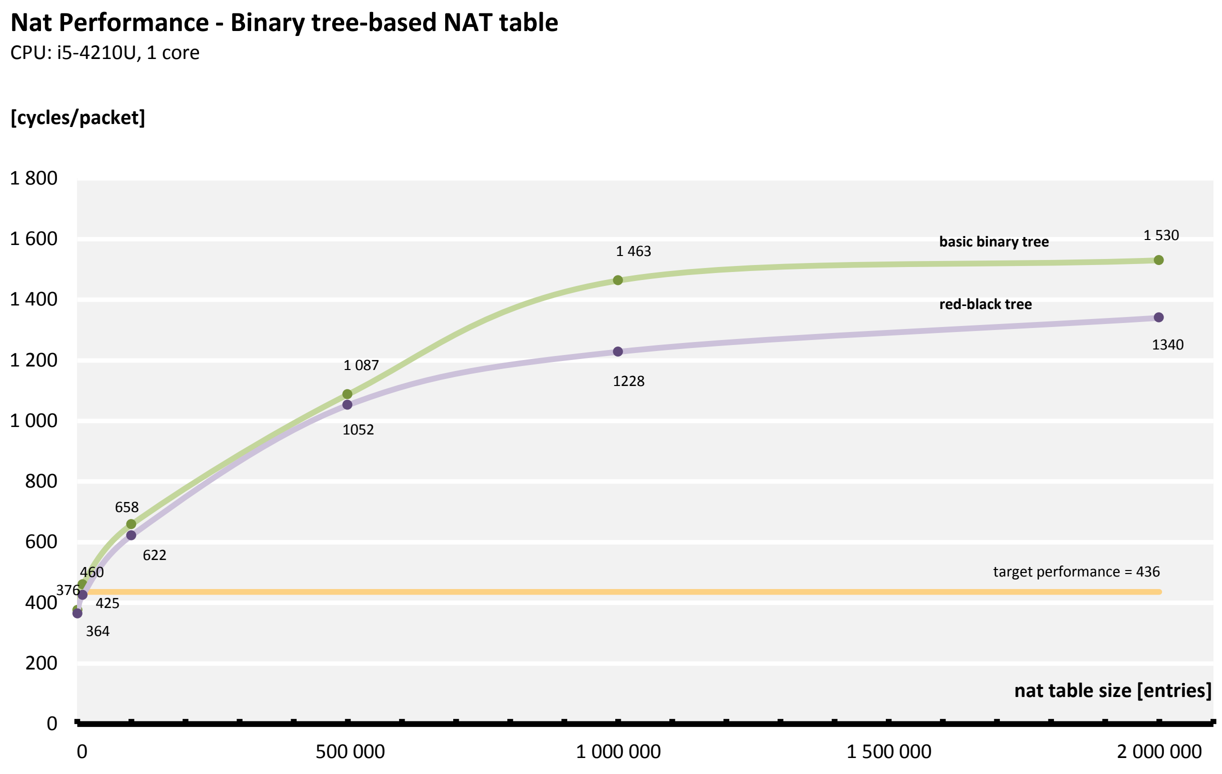


Figure 3. Nat Performance in cycles per packet. Binary tree-based NAT tables. Comparison between simple unbalanced and red-black binary trees.

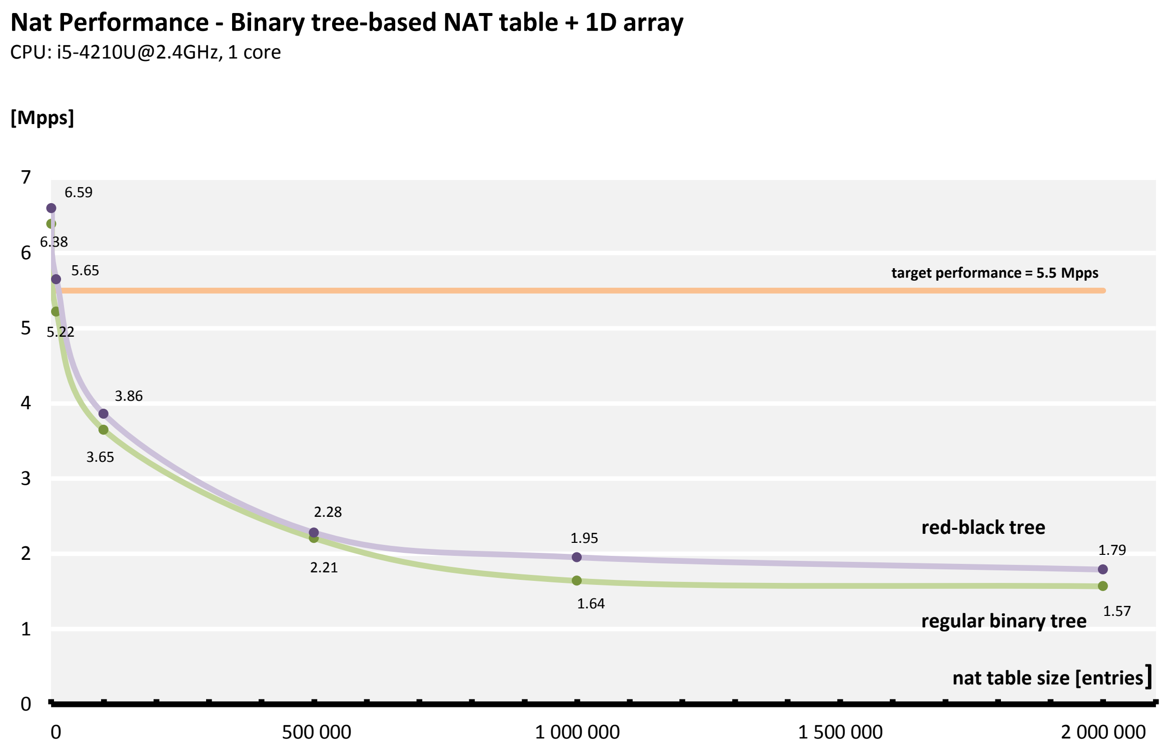


Figure 4. Nat Performance in packets per second. Binary tree-based NAT tables. Comparison between simple unbalanced and red-black binary trees.

The figure 3 and 4 show the results of using simple unbalanced and red-black binary trees. The performance improvement that gives red-black tree data structure is around 14% and still 3 times below the desired value.

The results of using tree-based data structures revealed the fact that the binary-trees group of data structures cannot be used for achieving of the target level of performance. The maximum result they could provide is 3 times slower than needed. For further investigation of the performance more fast data structures should be taken into consideration.

The next group of algorithms to be tested is hash-table based algorithms.

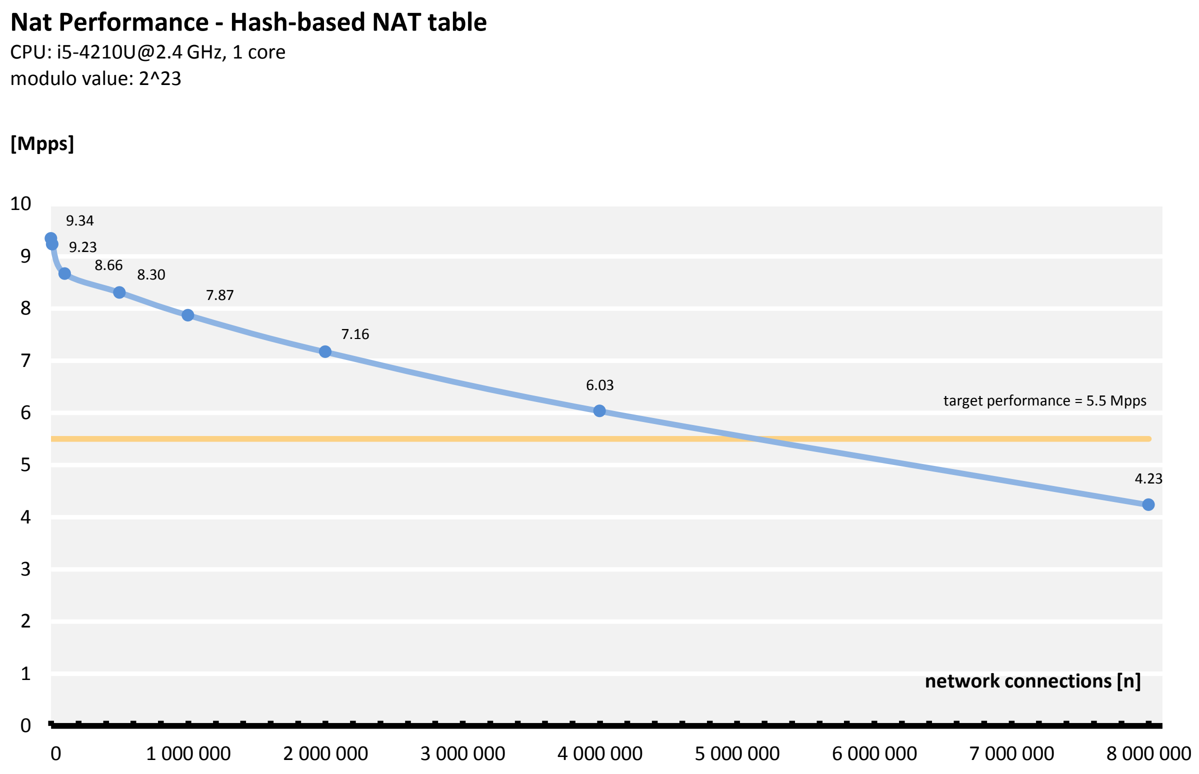


Figure 5. Nat Performance in packets per second. Hash-based NAT table.

For implementing the NAT table modulo division operation was used as the hash function. The NAT table capacity is set to store 1000 translation for each of 65536 nodes of the supported network. The hash table cell includes supplementary data which includes a link to a corresponding cell in the incoming array for accelerating of new translation creating time. The results of using the hash table are shown in Figure 5. There is a significant performance increasing in comparison with the tree-based NAT tables: it is 4 times faster. Because of the test system limitations it is not seen from the chart what the NAT performance is at the target level of connections number. Although, it is not shown in Figure 5, it is possible to estimate the full load performance, using the essence of the separate chaining linked list hash table and the results of linear search. The worst case scenario search time for this data structure is determined by the maximum length of the linked list associated with the cell calculated by hash function. Thus, changing the devisor value we could adjust the search time. But with changing the initial value the size of the hash table varies: than bigger the devisor that bigger its initial size.

This leads to significant memory overheads. Using the big hash table significantly exceeding the number of possible connections significantly reduce the probability of collision (the same key appearing) which, in turn, reduces the length of the chain. This leads to big memory overheads. The overheads could be eliminated by using alternative schemes of hash table collision resolving. The 4 of them were tested. The best result were shown by hash table with cache optimized list chaining storing 3 list items in a processor's cache line. The increasing of list items amount could further speed up the performance but is limited with a cache line size of 64 bytes. The results of comparison are shown in Figure 6 and 7.

Taking into account the fact that the 3-item cache line optimized hash table supplies the target level of performance for number of connection 4 times bigger than its initial capacity, the estimated initial size (and modulo value) of the hash table capable to support the target value of connections (65.5M) is $2^{26}/4 = 2^{24}$ (around 300M). Relatively small memory consumption of the hash table becomes important when one trying to take advantage from parallelization.

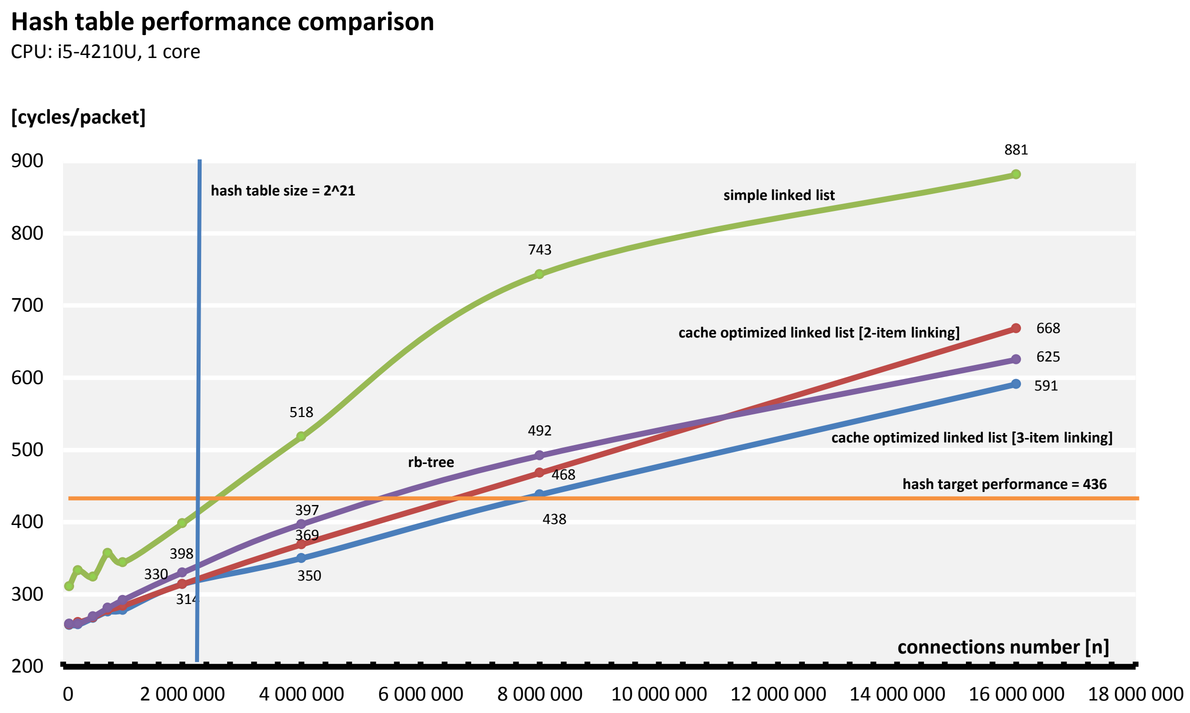


Figure 6. The comparison of hash tables with different types of collisions resolving in cycles per packet.

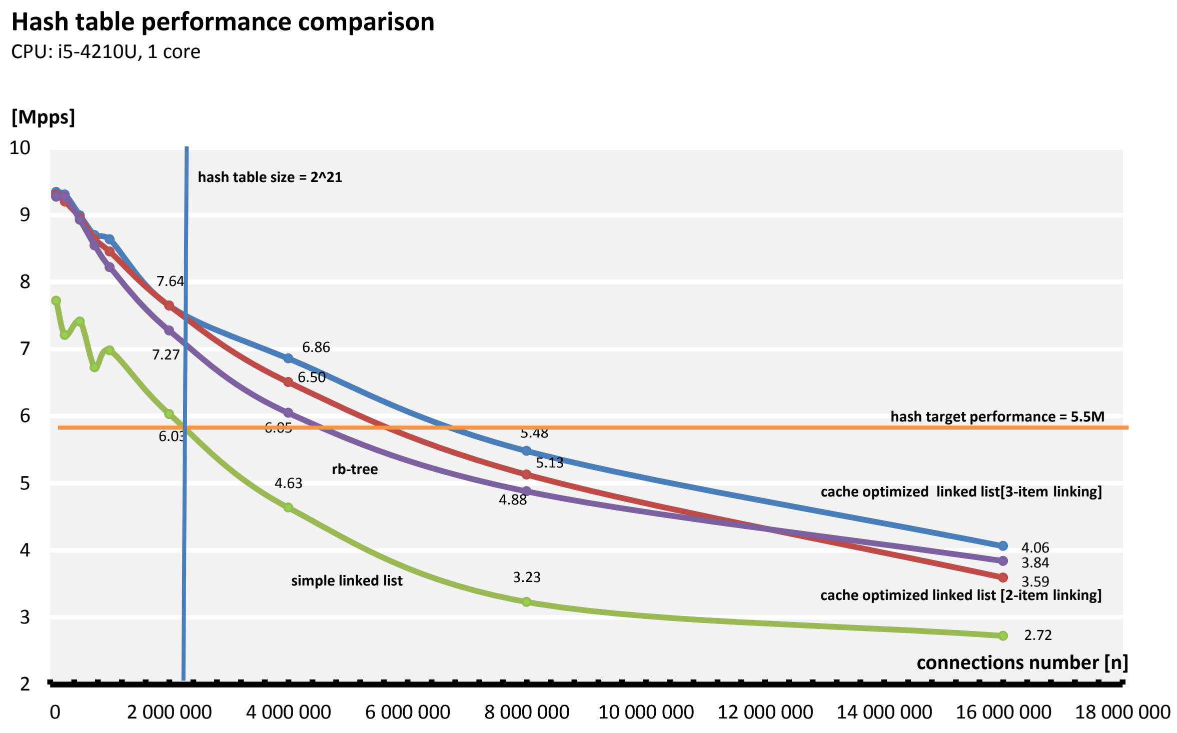


Figure 7. The comparison of hash tables with different types of collisions resolving in packets per second.

Further speed up can be gotten using parallelization of the translation process. All the results described previously in this document were gotten with 1 core at a multicore processor. Almost all modern CPUs offer several cores on a single chip. Thus, using several cores for NAT routine seems to be a promising idea.

In figure 8 the results are shown for different number of cores. The experiments were set for 4 threads maximum because the testing system has 4 hardware threads and setting experiment with more threads are meaningless because of the maximum number of simultaneously running cores is equal to number of cores available on the CPU chip and the OS scheduling issues, i.e. an operating system spends additional time on context switching and doesn’t give any performance boost. The results shown reveal that the parallel approach gives the desired results: than more cores are available than grater performance improvement would be. Furthermore, the solution has a good scalability: adding an additional thread gives the same increasing of performance. This happens because of the design used: each thread is an NAT with fully independent data structures which is used by a certain CPU core number only.

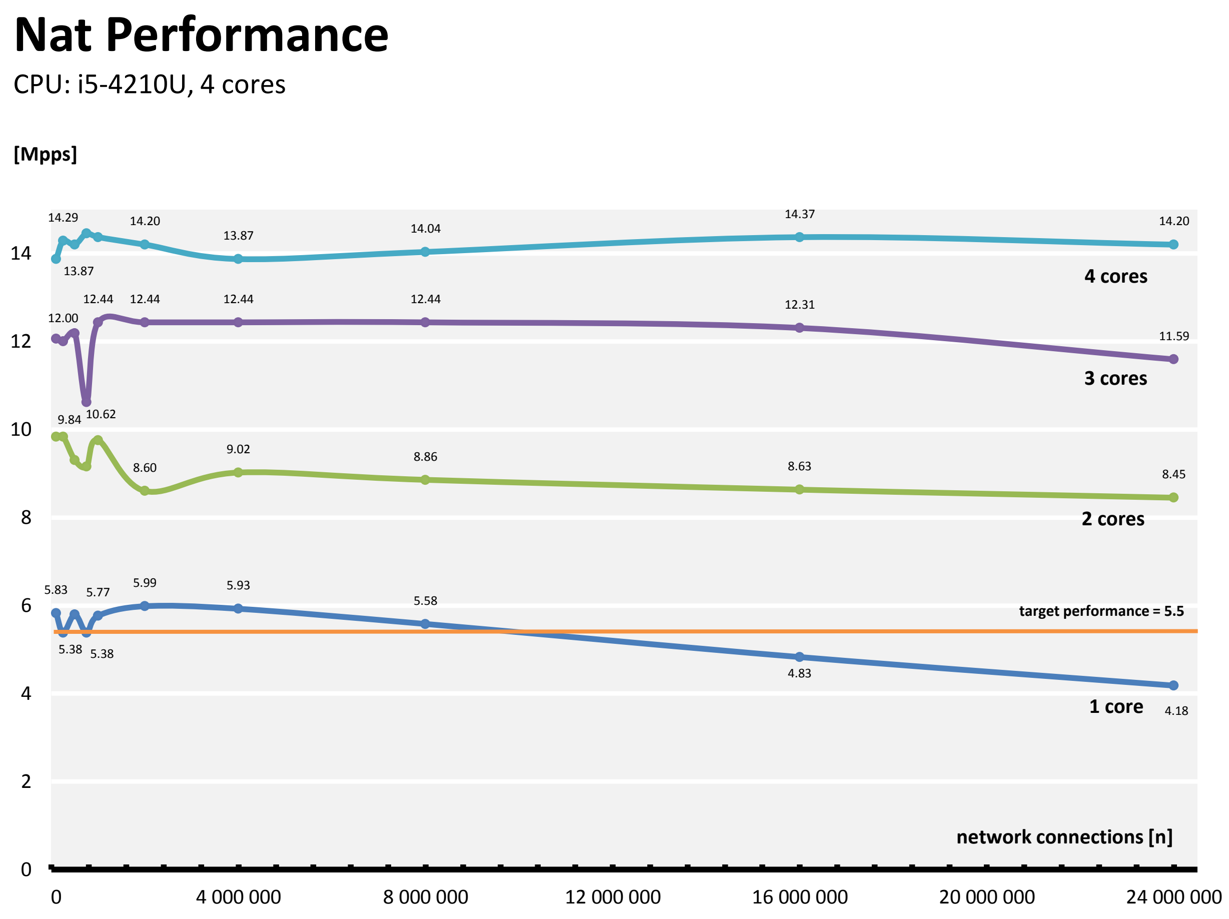


Figure 8. Nat Performance when using parallel 2-item optimized hash-based NAT table. Results for different number of simultaneously working cores.

This design is applicable in this case because the resources can be split evenly between the threads and modern Network Interface Cards provide the ability to split the incoming packets in several independent queues used by the threads independently.

3.2.2 Final design solution

Summing up the results, it is reasonable to conclude that using cache-friendly (3-item optimized) hash tables as a data structure for storing the translations information which works in parallel manner is a reasonable choice for the high-performance NAT. It gives the performance rate 260% more than target level and has a significant reserve for further NAT algorithms complications. This data structure is used to store outgoing translations. For incoming translation a one-dimensional array is used. All the translation data is stored in this array and the hash-table for outgoing translation stores references to the translation data stored in the array. All other translation routine is quite straightforward and standard approaches for doing that are used.

The code for the application is written in C, except function for the fast checksum calculation written using Assembler language and taken from Linux kernel. This approach promises to give more performance while using more advanced processors and compilers in the future.

Further modifications and optimizations of the NAT table structure based on hash-table could be done to improve the currently achieved performance values by using more sophisticated data structures, as well as other hardware-friendly optimization technics for example cuckoo hashing [22], but this is the matter of further research.

3.2.3 The software CG-NAT price estimation

Having the achieved results and assuming that this level is feasible not only in the simulation but in the real world, it is reasonable to estimate the price of NAT system using the results of this research and compare it with competing systems.

As a base for the system the average prices on the Russian market is used. To estimate the price, the computer setup is used with similar characteristics that the computer has which was used for testing in this work. The setup is the following:

Component name	Model	Price, $
Processor	Intel Core i5-4690 [28]	220
RAM Modules	SiliconPower **SP016GXLYU16ANDA** x2 [29]	320
Motherboard	SuperMicro X10SLL-S (Intel C222) [30]	200
Hard Disk	Intel DC S3610 [31]	275
Network Interface Card	Lenovo 10Gb X540-T2 [32]	600
System Unit	SuperMicro CSE-732D2-500B [33]	200
	Total:	**$1815**

Table 2. Software NAT system cost price estimation

As it is seen from the estimation, one spending a little bit more than 1800 USD can build a NAT system which could work as a NAT having the same performance rate as specialized devices which cost at least 3 times higher than tested one (see Figure 1).

3.3 Summary

In this work feasibility of developing a high-performance Network Address Translator which uses a regular cheap commodity server was explored. The results revealed that it can be developed by using special technics of translation data managing based on the hash tables with meeting all characteristics that a specially designed hardware NAT has. The software NAT possible on market price would be a couple of times cheaper than available on market devices. The performance evaluation was done using a laptop processor which makes it possible to assume that using of more advanced processor will give more performance.

Although, the results achieved exceed the target values, the evaluated NAT was developed using a simulated environment that performs only the core NAT functionality related to packet processing. The policy of IP address and port allocations was not implemented. The implementation of these functions can cause the performance reduction but is out of the scope of this work.

Another point is that the functionality of working with a network interface card was not implemented too and it was implied that there is a framework, Intel$^{©}$ DPDK, which gives the performance rate exceeding the target values and, because of that, couldn’t be considered as a performance bottleneck. This framework can be used for implementing the functionality of communicating with a network card without the performance reduction.

The developed NAT system falls within the concept of NFV (Network Function Virtualization) which implies separating the functionality of a network device from its hardware implementation. The NFV approach is a next generation of network building principles aimed to increase flexibility and reduce the price of deploying new or renovating existing data networks in the age of cloud computing.

REFERENCES

[1] IPv4 to IPv6: Challenges, solutions, and lessons, Stanford L. Levin a, Stephen Schmidt. Telecommunications Policy 38 (2014) 1059–1068

[2] Deploying IPv6 in the Google Enterprise Network. Lessons learned. Haythum Babiker, Irena Nikolova, Kiran Kumar Chittimaneni, Google, USENIX LISA, 2011

[3] https://www.ripe.net/publications/ipv6-info-centre/about-ipv6/ipv4-exhaustion

[4] Common Requirements for Carrier-Grade NATs (CGNs) http://www.rfc-base.org/txt/rfc-6888.txt

[5] Traditional IP address translator. https://www.ietf.org/rfc/rfc3022.txt

[6] NAT Behavioral Requirements for TCP https://tools.ietf.org/html/rfc5382

[7] Network Address Translation (NAT) Behavioral Requirements for Unicast UDP https://tools.ietf.org/html/rfc4787#page-5

[8] NAT Behavioral Requirements for ICMP https://tools.ietf.org/html/rfc5508

[9] Технические требования к оборудованию CG-NAT для сети IP/MPLS ОАО «Ростелеком» (Редакция 1)

[10] http://rdp.ru/

[11] NAT 32 IP Router http://v2.nat32.com/index.html

[12] http://www.wingate.com/purchase/wingate/purchase.php

[13] A Protected Dataplane Operating System for High Throughput and Low Latency. Adam Belay, Stanford University; George Prekas, École Polytechnique Fédérale de Lausanne (EPFL); Ana Klimovic, Samuel Grossman, and Christos Kozyrakis, Stanford University; Edouard Bugnion, École Polytechnique Fédérale de Lausanne (EPFL) https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[14] Understanding the Packet Processing Capability of Multi-Core Servers. Norbert Egi‡, Mihai Dobrescu†, Jianqing Du†, Katerina Argyraki†, Byung-Gon Chun§, Kevin Fall§, Gianluca Iannaccone§, Allan Knies§, Maziar Manesh§, Laurent Mathy‡, Sylvia Ratnasamy§ § Intel Research, † EPFL, ‡ Lancaster University

[15] M. Dobrescu, N. Egi, K. J. Argyraki, B.-G. Chun, K. R. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP ’09), pages 15–28, 2009

[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. ACM Trans. Comput. Syst., 18(3):263–297, 2000.

[17] Computer Systems: A Programmer's Perspective (2nd Edition) by Randal E. Bryant, David R. O'Hallaron, ISBN-13: 978-0136108047

[18] Validity of the single processor approach to achieving large scale computing capabilities, Gene M Amdahl, IBM SunnyvaleCalifornia

[19] NAT implementation for the NetFPGA platform, Omar Choudary, David J. Miller, University of Cambridge, Cambridge, UK

[20] An Efficient Hardware-based Multi-hash Scheme for High Speed IP Lookup, Socrates Demetriades, Michel Hanna, Sangyeun Cho and Rami Melhem, University of Pittsburgh, 16th IEEE Symposium on High Performance Interconnects

[21] Exploiting a Computation Reuse Cache to Reduce Energy in Network Processors, Bengu Li, Ganesh Venkatesh, Brad Calder, and Rajiv Gupta, University of Arizona, University of California

[22] Scalable, High Performance Ethernet Forwarding with CUCKOOSWITCH, Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen Carnegie Mellon University; Michael Kaminsky (Intel Labs), 10th USENIX Symposium on Networked Systems Design and Implementation, 2013

[23] Intel Data Plane Development Kit (Intel DPDK) Overview. http://www.intel.com/content/dam/www/public/us/en/documents/presentation/dpdk-packet-processing-ia-overview-presentation.pdf

[24] Introduction to algorithms, By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein ISBN: 978026203384

[25] Internet protocol https://www.ietf.org/rfc/rfc791.txt

[26] Transmission Control Protocol https://www.ietf.org/rfc/rfc793.txt

[27] User Datagram Protocol https://www.ietf.org/rfc/rfc768.txt

[28] http://ark.intel.com/ru/products/80810/Intel-Core-i5-4690-Processor-6M-Cache-up-to-3_90-GHz

[29]http://www.nix.ru/autocatalog/memory_modules_SiliconPower/Silicon_Power_SP016GXLYU16ANDA_DDRIII_16Gb_8Gb_PC312800_CL9_204263.html?set_id=be30bbd1ed9211e4a20b002590c35102&vs_id=be1945beed9211e4a20b002590c35102&sort=0&from=sch&sch_id=34&sch_good=190779

[30] http://www.srv-trade.ru/catalog/1824837288/CSE-732D2-500B/
http://www.nix.ru/autocatalog/motherboards_supermicro/SuperMicro_X10SLLS_LGA1150_C222_PCIE_2GbLAN_SATARAID_microATX_2DDRIII_159355.html

[31]http://www.nix.ru/price/price_list.html?section=ssd_all&set_id=9e029764ed9711e4a20b002590c35102#cid=900&fn=900&set_id=a574a849ed9711e4a20b002590c35102&sort=+p1710&thumbnail_view=1

[32] http://shop.lenovo.com/us/en/itemdetails/0C19497/460/BF8C0B7DC9BB4C13B5EF4FE54E3ABB39

[33] http://www.srv-trade.ru/catalog/1824837288/CSE-732D2-500B/

Appendix A

List of router models used for market research

Vendor	Router Model	Packet processing rate, Mpps	Price, USD	URL
HP	MSR2021	0.18	760	http://www8.hp.com/ru/ru/products/networking-routers/product-detail.html?oid=5054094#!tab=specs
HP	MSR1002-4	0.50	1 100	http://www8.hp.com/us/en/products/networking-routers/product-detail.html?oid=6288749#!tab=specs
HP	MSR2003	1.00	1 700	http://www8.hp.com/ru/ru/products/networking-routers/product-detail.html?oid=5408900#!tab=specs
HP	MSR50-40	1.20	2 500	http://www8.hp.com/us/en/products/networking-routers/product-detail.html?oid=4199527#!tab=specs
HP	MSR3012	2.60	2 700	http://www8.hp.com/us/en/products/networking-routers/product-detail.html?oid=6288370&jumpid=reg_r1002_usen_c-001_title_r0002#!tab=specs
HP	MSR3040	0.36	3 200	http://www8.hp.com/ru/ru/products/networking-routers/product-detail.html?oid=4199541#!tab=specs
HP	MSR4060	10.00	6 000	http://www8.hp.com/us/en/products/networking-routers/product-detail.html?oid=5408896
CISCO	7301	1.00	10 000	http://www.cisco.com/web/RU/products/hw/routers/ps352/ps4972/index.html
CISCO	ASA5515-IPS-K9	0.50	4 800	http://ciscosales.ru/katalog_produkcii/cisco/mezhsetevye_ekrany_i_fil_try/cisco_asa_5500_series_accessories/asa5515-ips-k9/ http://www.cisco.com/c/en/us/products/security/asa-5500-series-next-generation-firewalls/models-comparison.html#~tab-a
CISCO	ASA 5525-X	0.70	5 200	http://www.cisco.com/c/en/us/products/security/asa-5500-series-next-generation-firewalls/models-comparison.html#~tab-b
CISCO	ASA 5545-X	0.90	10 200	http://www.cisco.com/c/en/us/products/security/asa-5500-series-next-generation-firewalls/models-comparison.html#~tab-b
CISCO	ASA 5555-X	1.00	17 000	http://www.cisco.com/c/en/us/products/security/asa-5500-series-next-generation-firewalls/models-comparison.html#~tab-b

CISCO	5585-X SSP10	1.10	14 200	http://www.cisco.com/c/en/us/products/security/asa-5500-series-next-generation-firewalls/models-comparison.html#~tab-c http://ciscosales.ru/katalog_produkcii/cisco/mezhsetevye_ekrany_i_fil_try/cisco_asa_5500_series_firewall_edition_bundles/asa5585-s10-k9/
CISCO	5585-X SSP20	2.00	40 000	http://www.cisco.com/c/en/us/products/security/asa-5500-series-next-generation-firewalls/models-comparison.html#~tab-c http://ciscosales.ru/katalog_produkcii/cisco/mezhsetevye_ekrany_i_fil_try/cisco_asa_5500_series_firewall_edition_bundles/asa5585-s20-k8/http://ciscosales.ru/katalog_produkcii/cisco/mezhsetevye_ekrany_i_fil_try/cisco_asa_5500_series_firewall_edition_bundles/asa5585-s20-k8/
CISCO	5585-X SSP40	4.00	80 000	http://www.cisco.com/c/en/us/products/security/asa-5500-series-next-generation-firewalls/models-comparison.html#~tab-c http://ciscosales.ru/katalog_produkcii/cisco/mezhsetevye_ekrany_i_fil_try/cisco_asa_5500_series_firewall_edition_bundles/asa5585-s40-k8/
CISCO	5585-X SSP60	10.00	128 300	http://www.cisco.com/c/en/us/products/security/asa-5500-series-next-generation-firewalls/models-comparison.html#~tab-c http://ciscosales.ru/katalog_produkcii/cisco/mezhsetevye_ekrany_i_fil_try/cisco_asa_5500_series_firewall_edition_bundles/asa5585-s60-2a-k9/
JUNIPER	SRX240	0.20	2 000	http://www.juniper.net/us/en/local/pdf/datasheets/1000281-en.pdf http://www.srv-trade.ru/catalog/976735833/SRX240H/?gclid=CKXzk-_4ysMCFYPUcgodBmUALQ
JUNIPER	SRX550	0.70	7 900	http://www.juniper.net/us/en/local/pdf/datasheets/1000281-en.pdf
JUNIPER	SRX650	0.85	13 000	http://www.juniper.net/us/en/local/pdf/datasheets/1000281-en.pdf

Appendix B

Source code

Measuring Part (File: cycles.c):

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>

#define LOOPS 1
#define EXP_NUM 16
#define MEANINGFUL_LIMIT 5000000
#define CPU_SPEED 2400000000

struct Packet {
        int ip;
        int port;
};

static __inline__ unsigned long long rdtsc(void){
    unsigned int lo, hi;
    __asm__ volatile ("rdtsc\n" : "=a" (lo), "=d" (hi));
    return ((unsigned long long) hi << 32) | lo;
}


/*
 *      Test function interface
 */
extern void test_create(int *params, int num_params);
extern void test_show_info();
extern void test_init();
extern void test_run(int loops);
extern int test_make_checks();
extern void test_clean();

int
main(int argc, char *argv[]){
        static struct Packet pkt;
        unsigned long long start, end;
        int i, j, good_res_amnt;
        double sum = 0;
        double result = 0;
        int num_packets = 1000;
        double res_avg = 0;
        double std_dev = 0;
        double result_arr[EXP_NUM];
        int ri = 0;
        good_res_amnt = 0;

        // command line params processing
        if (argc == 2) {
                num_packets = atoi(argv[1]);
        }

        // Performing test
        test_create(&num_packets, 1);
        test_show_info();

        for (i = 0; i < EXP_NUM; i++)
        {
                test_init();
                start = rdtsc();

                //test_function_1(&pkt);
                test_run(LOOPS);

                end = rdtsc();
                 // * 2 - because of back and forth packet processing
                result = (end-start)/LOOPS/num_packets/2;
                printf("Try %d: [ticks: %llu] %lf\n", i, (end-start), result);
                if (result < MEANINGFUL_LIMIT)
                {
                        sum += result;
                        good_res_amnt++;
                        result_arr[ri++] = result;
                }

                if (test_make_checks() < 0) {
                        printf("TEST STOPPED\n");
                        return 0;
                }

                test_clean();
        }

        if (good_res_amnt == 0) {
                printf("There were no successful attempts.\n");
                printf("Check MEANINGFUL_LIMIT value.\n");
                printf("It might be too low\n");
                return 0;
        }

        res_avg = sum/(double) good_res_amnt;

        // calculating standard deviation
        for(i = 0; i < ri; i++)
                std_dev += pow(result_arr[i]-res_avg, 2);

        std_dev = sqrt(std_dev/ri);


        printf("Cycles per packet: %.1lf [~%.2lf]\n", res_avg, std_dev);
        printf(
                "Processing speed on %.1lf GHz: %.2lf Kpps\n",
                (double) CPU_SPEED/1000000000,
                CPU_SPEED/res_avg/1000
        );
        printf("Successful attempts: %d out of %d\n\n", good_res_amnt, EXP_NUM);
        return 0;
}
```

Generation and simulation parts (File: nat.c):

RB-tree version:

```
#include <stdbool.h>
#include "stdio.h"
#include <sys/time.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include "tree.h"


#define PORTS 5500000
#define WANIP 1 // 83 for 5.5 Mpackets
#define MAX_NAT_ENTRIES PORTS
#define SEC 1000000 // usec in a sec
#define PORTS_RESERVED 1024
#define NET_NUM_NODES 255
#define NUM_PORTS_TO_GENERATE 21500

#define NET_ADDR_0 192
#define NET_ADDR_1 168
#define NET_ADDR_2 1
#define NET_ADDR_WAN "10.10.10.10"
#define NET_MASK 0xFF // /24
#define TIMEOUT 120 // in seconds

#define SUCCESS 0
#define FAULT 1
#define NOT_FOUND 2
```

```
#define TIMEOUT_EXCEEDED 3
#define EMPTY 4
#define NOT_VALID_ENTRY 5
#define LACK_OF_FREE_PORTS 6;

#define VERBOSE 0

extern double randn (double mu, double sigma);

static struct nat_entry nat_table[MAX_NAT_ENTRIES];
static unsigned long long timeout = TIMEOUT * SEC;
static struct ippkt *pkt_set = NULL;
static struct ippkt *pkt_test_set = NULL;
static struct port_item *port_free_list = NULL;
static int pkt_set_size;
static unsigned int wan_ip;

static struct TreeNode *indexTree;

/*
 *      fast checksum calculation
 *      taken from
 *      /usr/src/linux-headers-3.13.0-44/arch/x86/include/asm/checksum_64.h
 */
static inline unsigned int
ip_fast_csum(const void *iph, unsigned int ihl)
{
        unsigned int sum;

        __asm__ ("  movl (%1), %0\n"
            "  subl $4, %2\n"
            "  jbe 2f\n"
            "  addl 4(%1), %0\n"
            "  adcl 8(%1), %0\n"
            "  adcl 12(%1), %0\n"
            "1: adcl 16(%1), %0\n"
            "  lea 4(%1), %1\n"
            "  decl %2\n"
            "  jne      1b\n"
            "  adcl $0, %0\n"
            "  movl %0, %2\n"
            "  shrl $16, %0\n"
            "  addw %w2, %w0\n"
            "  adcl $0, %0\n"
            "  notl %0\n"
            "2:"
        /* Since the input registers which are loaded with iph and ihl
           are modified, we must also specify them as outputs, or gcc
           will assume they contain their original values. */
            : "=r" (sum), "=r" (iph), "=r" (ihl)
            : "1" (iph), "2" (ihl)
            : "memory");

        return sum;
}

/*
 *      timestamp in usec
 */
static inline unsigned long long
get_seconds() {
       struct timeval t;
       gettimeofday(&t, NULL);
       return t.tv_sec * SEC + t.tv_usec;
       //return 1;
}


int inline
get_port_index(unsigned int port) {
       return port-PORTS_RESERVED;
}

int inline
get_index_port(unsigned int index) {
       return index+PORTS_RESERVED;
}

/*
 *      test for get_seconds() (timestamp) function
 */
void
test_get_seconds(int loops){
        unsigned long long t;
        int i;
        for (i = 0; i < loops; i++) {
                t = get_seconds();
                printf("Time stamp: %llu\n", t);
        }
}

/*
 *      convert a string IPv4 to int IPv4 in big-endian order
 */
unsigned int
str_to_ip(char *ipstr) {
        unsigned int ip;
        unsigned int a[4];

        sscanf(ipstr, "%u.%u.%u.%u", &a[0], &a[1], &a[2], &a[3]);
        ip = (a[0] << 24) | (a[1] << 16) | (a[2] << 8) | a[3];
        return ip;
}
/*
 *      converts to ip address in big-endian notation out of 4 int numbers
 */
unsigned int
make_ip(int a0, int a1, int a2, int a3) {
        unsigned int ip;
        ip = (a0 << 24) | (a1 << 16) | (a2 << 8) | a3;
        return ip;
}

/*
 *      print an ip-address
 */
void
print_ip_port(struct ippkt *pkt)
{
         printf("%d.%d.%d.%d:%d\n",
                pkt->ip >> 24 & 0xFF,
                pkt->ip >> 16 & 0xFF,
                pkt->ip >> 8 & 0xFF,
                pkt->ip & 0xFF,
                pkt->port
        );
}

/*
 *      packet generation functions
 */
int
get_rand_val(int max_val) {
         int i;
         double r;

         r = -4;
         while(abs(r) > 1.5)
                 r = randn(0, 2);

         r = r + 2;
         r = round(r/4 * max_val + 1);

         return (int) r;
}

void
generate_ippkt_set(struct ippkt *pkt_set, int num) {
        int i;
        unsigned int host_ip, host_port;

        srand(time(NULL));

        for(i = 0; i < num; i++) {
                host_ip = get_rand_val(253);

                pkt_set[i].ip = make_ip(
                        NET_ADDR_0, NET_ADDR_1, NET_ADDR_2, host_ip
                );
                host_port = get_rand_val(NUM_PORTS_TO_GENERATE)+PORTS_RESERVED;

                pkt_set[i].port = host_port;
        }


        // save a copy of pkt_set
        memcpy(pkt_test_set, pkt_set, sizeof(struct ippkt) * num);

        #if VERBOSE
                printf("Packet (%d) generation...\tDONE\n", num);
        #endif
}


/*
 *      free port list managing functions
 */
static inline int
get_port(struct port_item **free_ports) {
        int port;
        struct port_item *used_item;

        if (*free_ports == NULL)
                return -EMPTY;

        #if VERBOSE
                printf("get_port is not empty\n");
        #endif

        used_item = *free_ports;
        port = used_item->port;

        (*free_ports) = used_item->next;
        free(used_item);

        #if VERBOSE
                printf("get_port: %d\n", port);
        #endif

        return port;
}

static inline void
add_port(struct port_item **free_ports, int port) {
        struct port_item *prev_port;
        prev_port = *free_ports;
        *free_ports = (struct port_item*) malloc(sizeof(struct port_item));

        (*free_ports)->port = port;
        (*free_ports)->next = prev_port;
}


static void
print_free_ports(struct port_item *free_ports) {
        if (free_ports == NULL) {
                printf("\n");
                return;
        }

        printf("%d ", free_ports->port);
        print_free_ports(free_ports->next);
}

static void
test_free_ports() {
        int i;

        for(i = 0; i < 3; i++) {
                add_port(&port_free_list, i);
        }

        printf("Free ports...");

        for(i = 0; i < 3; i++)
                if (get_port(&port_free_list) != 2-i) {
                        printf("\tFAIL\n");
                        return;
                }

        printf("\tOK\n");

}

static void
clean_free_ports(struct port_item **free_ports) {
        struct port_item *next_port, *cur_port;
        cur_port = *free_ports;

        while(cur_port != NULL) {
                next_port = cur_port->next;
                free(cur_port);
                cur_port = next_port;
        }

        *free_ports = NULL;
        #if VERBOSE
                printf("Free ports list cleaning...\tDONE\n");
        #endif
}

void
create_free_ports(struct port_item **free_ports) {
        int i;
        for (i = 0; i < PORTS; i++) {
                add_port(free_ports, PORTS_RESERVED + i);
        }
        #if VERBOSE
                printf("Free port list generation...\tDONE\n");
        #endif
}



/*
 *      Makes a key for nat tree out of ip and port
 */
static inline _key_
makeKey(unsigned int addr, unsigned int port) {
         _key_ key;

         key = (addr & NET_MASK) << 24 | port;

         return key;
}

/*
 *      Makes a key for nat tree out of ip and port
 */
static inline int
find_nat_entry(int saddr, int sport) {
         int  index;
        struct TreeNode* n;
        n =  find(makeKey(saddr, sport), indexTree);

        if (n == NULL)
                return -NOT_FOUND;

        index = n->data;

        if (get_seconds()-nat_table[index].sec > timeout){
                nat_table[index].valid = false;
                  return -TIMEOUT_EXCEEDED;
         }

        return index;
}

static inline int
make_nat_entry(unsigned int ip, unsigned int port){
         int index, p;
        struct TreeNode *n;
        _data_ data;

         if ((p = get_port(&port_free_list)) < 0)
                  return -FAULT;

         index = get_port_index(p);
         nat_table[index].lan_ipaddr= ip;
         nat_table[index].lan_port = port;
         nat_table[index].valid = true;
         nat_table[index].sec = get_seconds();

        data = index;
        n = insertData(makeKey(ip, port), data, &indexTree);

        if (n == NULL)
                return -FAULT;

         #if VERBOSE
                  printf("New NAT entry port: %d\n", p);
         #endif
         return index;

}

static inline int
translate_lan_to_wan(struct ippkt *pkt) {
         int index, port;

         #if VERBOSE
                  printf("Searching for nat entry %u:%d -> ", pkt->ip, pkt->port);
         #endif

         if((index = find_nat_entry(pkt->ip, pkt->port)) < 0)
         {
                  #if VERBOSE
                          printf("Proper index hasn't been found\n");
                  #endif
                  if (index == -NOT_FOUND) {
                          if ((index = make_nat_entry(pkt->ip, pkt->port)) < 0)
                                  return -FAULT;
                  }
                  else if(index == -TIMEOUT_EXCEEDED) {
                          printf("Exiting with %d\n", index);
                          return -FAULT;
                  }
                  else {
                          printf("Exiting with %d\n", index);
                          return -FAULT;
                  }
         }

         #if VERBOSE
                  printf("Index found: %d\n", index);
         #endif

         pkt->ip = wan_ip;
         pkt->port = get_index_port(index);
         pkt->chksum = ip_fast_csum((void*) pkt, sizeof(struct ippkt));
         nat_table[index].sec = get_seconds();
         return SUCCESS;
}

static inline int
translate_wan_to_lan(struct ippkt *pkt) {
         int index;
         index = get_port_index(pkt->port);
         #if VERBOSE
                  printf("Port to index from wan: %d -> %d\n", pkt->port, index);
         #endif

         if (true) {
                  pkt->ip = nat_table[index].lan_ipaddr;
                  pkt->port = nat_table[index].lan_port;
                pkt->chksum = ip_fast_csum((void*) pkt, sizeof(struct ippkt));
                  nat_table[index].sec = get_seconds();
                  return SUCCESS;
         }
         else
                  return -NOT_VALID_ENTRY;
}

/*
 *      Checks
 *
 */
```

```
int
pkt_set_check() {
        int i;
        for (i = 0; i < pkt_set_size; i++)
                if (
                        pkt_set->ip != pkt_test_set->ip ||
                        pkt_set->port != pkt_test_set->port
                )
                        return -FAULT;

        return SUCCESS;
}

int
test_nat_capacity() {
        if (NET_NUM_NODES * NUM_PORTS_TO_GENERATE > PORTS) {
                  printf("Nat capacity is too low!!!\n");
                printf("needed: %d, avalable: %d\n",
                        NET_NUM_NODES * NUM_PORTS_TO_GENERATE,
                        PORTS);

                return -FAULT;
        }
        else
                return SUCCESS;
}

/*
 *      testing interface functions
 */
void
test_create(int *params, int num_params) {
        if (test_nat_capacity() < 0){
                printf("Exiting...\n");
                exit(0);
        }

        wan_ip = str_to_ip(NET_ADDR_WAN);
        pkt_set_size = params[0];
        pkt_set = (struct ippkt*) malloc (sizeof(struct ippkt) * pkt_set_size);
        pkt_test_set =
                (struct ippkt*) malloc (sizeof(struct ippkt) * pkt_set_size);
}

void
test_show_info() {
        printf("===== TEST INFO =====\n");
        printf("Max of ip-port combinations:\t%d\n",
                NET_NUM_NODES * NUM_PORTS_TO_GENERATE);
        printf("NAT capacity:\t%d\n", MAX_NAT_ENTRIES);
        printf("NAT entry size:\t%lu\n", sizeof(struct nat_entry));
        printf("NAT table size:\t%lu\n", sizeof(nat_table));
        printf("Tree node size:\t%lu\n", sizeof(struct TreeNode));
        printf("Packet size:\t%lu\n", sizeof(struct ippkt));
        printf("Packet amount:\t%d\n", pkt_set_size);
        printf("Pkt set size:\t%'lu\n", sizeof(struct ippkt)*pkt_set_size);
        printf("=====================\n\n");
}


void
test_init() {
        create_free_ports(&port_free_list);
        generate_ippkt_set(pkt_set, pkt_set_size);
}


void
test_clean() {
        unsigned int tree_size;
        memset(nat_table, 0, sizeof(nat_table));

        memset(pkt_set, 0, sizeof(struct ippkt) * pkt_set_size);

        memset(pkt_test_set, 0, sizeof(struct ippkt) * pkt_set_size);

        clean_free_ports(&port_free_list);

        tree_size = getTreeSize(indexTree);
        printf("Tree size was: %d\n", tree_size);
        freeTree(&indexTree);
}

void
print_err(int packet_id) {
        printf("Translation of packet [%d] FAILED\n", packet_id);
}


void
test_run(int loops) {
        int i, j;
        int ip = 0;
        for(i = 0; i < loops; i++) {
                // translation from lan to wan
                for (j = 0; j < pkt_set_size; j++)
                        if(translate_lan_to_wan(&pkt_set[j]) < 0)
                                print_err(j);
                // translation from wan to lan
                for (j = 0; j < pkt_set_size; j++)
                        if(translate_wan_to_lan(&pkt_set[j]) < 0)
                                print_err(j);
        }
}

int
test_make_checks() {
        if (pkt_set_check() < 0) {
                printf("Packet inconsistancy found!\n");
                return -FAULT;
        }

        return SUCCESS;
}
```

Hash table based version (3-item cache optimized):

```
#include <stdbool.h>
#include <stdio.h>
#include <sys/time.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>


#define PORTS 65000000
#define WANIP 1 // 83 for 5.5 Mpackets
#define MAX_NAT_ENTRIES PORTS*WANIP
#define SEC 1000000 // usec in a sec
#define PORTS_RESERVED 1024
#define GEN_ADDR_MAX_VAL 10
#define TIMEOUT 180 // timeout in seconds

#define NET_ADDR_0 192
#define NET_ADDR_1 168
#define NET_ADDR_2_RANGE 200
#define NET_ADDR_3_RANGE 250
#define NET_ADDR_WAN "10.10.10.10"
#define NET_MASK 0xFFFF
#define NUM_PORTS_TO_GENERATE 10000

#define HASHING_VAL 1<<24

#define SUCCESS 0
#define FAULT 1
#define NOT_FOUND 2
#define TIMEOUT_EXCEEDED 3
#define EMPTY 4
#define NOT_VALID_ENTRY 5

#define VERBOSE 0

struct port_item {
        int port;
        struct port_item* next;
};
```

```
struct l4pkt {
        unsigned int chksum;
        unsigned int opt;
};

struct ippkt {
        unsigned int ip;
        unsigned int port;
        unsigned int chksum;
        struct l4pkt tcp_pkt; // a packet of the level 4 protocol
};

struct nat_entry {
         int  lan_ipaddr;
         int lan_port;
         unsigned long long sec;      /*timestamp in micro-seconds*/
         bool valid;
};

struct HashListItem {
        int nat_table_index;
        unsigned int key;
        struct HashListItem *next;
        bool valid; // [0] - valid (bool)  {1|0}
        char num;
//};
} __attribute__((__packed__));

extern double randn (double mu, double sigma);

static struct nat_entry nat_table[MAX_NAT_ENTRIES];
static struct HashListItem* nat_hash_table[HASHING_VAL];
static unsigned long long timeout = TIMEOUT * SEC;
static struct ippkt *pkt_set;
static struct ippkt *pkt_test_set;
static struct port_item *port_free_list;
static int pkt_set_size;
static unsigned int wan_ip;

// each item of the nat_hash_table is a pointer to the first
// element of the linked list of nat_entries

/*
 *      fast checksum calculation
 *      taken from
 *      /usr/src/linux-headers-3.13.0-44/arch/x86/include/asm/checksum_64.h
 */
static inline unsigned int
ip_fast_csum(const void *iph, unsigned int ihl)
{
         unsigned int sum;

         __asm__ ("  movl (%1), %0\n"
             "  subl $4, %2\n"
             "  jbe 2f\n"
             "  addl 4(%1), %0\n"
             "  adcl 8(%1), %0\n"
             "  adcl 12(%1), %0\n"
             "1: adcl 16(%1), %0\n"
             "  lea 4(%1), %1\n"
             "  decl %2\n"
             "  jne      1b\n"
             "  adcl $0, %0\n"
             "  movl %0, %2\n"
             "  shrl $16, %0\n"
             "  addw %w2, %w0\n"
             "  adcl $0, %0\n"
             "  notl %0\n"
             "2:"
         /* Since the input registers which are loaded with iph and ihl
            are modified, we must also specify them as outputs, or gcc
            will assume they contain their original values. */
             : "=r" (sum), "=r" (iph), "=r" (ihl)
             : "1" (iph), "2" (ihl)
             : "memory");

         return sum;
}

/*
```

```
 *      timestamp in usec
 */
static inline unsigned long long
get_seconds() {
        struct timeval t;
        gettimeofday(&t, NULL);
        return t.tv_sec * SEC + t.tv_usec;
        //return 1;
}


int inline
get_port_index(unsigned int port) {
        return port-PORTS_RESERVED;
}

int inline
get_index_port(unsigned int index) {
        return index+PORTS_RESERVED;
}

/*
 *      test for get_seconds() (timestamp) function
 */
void
test_get_seconds(int loops){
        unsigned long long t;
        int i;
        for (i = 0; i < loops; i++) {
                t = get_seconds();
                printf("Time stamp: %llu\n", t);
        }
}

/*
 *      convert a string IPv4 to int IPv4 in big-endian order
 */
unsigned int
str_to_ip(char *ipstr) {
        unsigned int ip;
        unsigned int a[4];

        sscanf(ipstr, "%u.%u.%u.%u", &a[0], &a[1], &a[2], &a[3]);
        ip = (a[0] << 24) | (a[1] << 16) | (a[2] << 8) | a[3];
        return ip;
}
/*
 *      converts to ip address in big-endian notation out of 4 int numbers
 */
unsigned int
make_ip(int a0, int a1, int a2, int a3) {
        unsigned int ip;
        ip = (a0 << 24) | (a1 << 16) | (a2 << 8) | a3;
        return ip;
}

/*
 *      print an ip-address
 */
void
print_ip_port(struct ippkt *pkt)
{
         printf("%d.%d.%d.%d:%d\n",
                pkt->ip >> 24 & 0xFF,
                pkt->ip >> 16 & 0xFF,
                pkt->ip >> 8 & 0xFF,
                pkt->ip & 0xFF,
                pkt->port
        );
}

/*
 *      packet generation functions
 */
int
get_rand_val(int max_val) {
         int i;
         double r;

         r = -4;
        while(abs(r) > 1.5)
                r = randn(0, 2);

        r = r + 2;
        r = round(r/4 * max_val + 1);

        return (int) r;
}

void
generate_ippkt_set(struct ippkt *pkt_set, int num) {
        int i;
        unsigned int host_ip2, host_ip3, host_port;

        srand(time(NULL));

        for(i = 0; i < num; i++) {
                host_ip2 = get_rand_val(NET_ADDR_2_RANGE);
                host_ip3 = get_rand_val(NET_ADDR_3_RANGE);

                pkt_set[i].ip = make_ip(
                        NET_ADDR_0, NET_ADDR_1, host_ip2, host_ip3
                );


                host_port = get_rand_val(NUM_PORTS_TO_GENERATE)+PORTS_RESERVED;

                pkt_set[i].port = host_port;

        }

        // save a copy of pkt_set
        memcpy(pkt_test_set, pkt_set, sizeof(struct ippkt) * num);

        #if VERBOSE
                printf("Packet (%d) generation...\tDONE\n", num);
        #endif
}

/*
 *      free port list managing functions
 */
static inline int
get_port(struct port_item **free_ports) {
        int port;
        struct port_item *used_item;

        if (free_ports == NULL)
                return -EMPTY;

        used_item = *free_ports;
        port = used_item->port;

        (*free_ports) = used_item->next;
        free(used_item);
        #if VERBOSE
                printf("get_port: %d\n", port);
        #endif
        return port;
}

static inline void
add_port(struct port_item **free_ports, int port) {
        struct port_item *prev_port;
        prev_port = *free_ports;
        *free_ports = (struct port_item*) malloc(sizeof(struct port_item));

        (*free_ports)->port = port;
        (*free_ports)->next = prev_port;
}


static void
print_free_ports(struct port_item *free_ports) {
        if (free_ports == NULL) {
                printf("\n");
                return;
        }

        printf("%d ", free_ports->port);
        print_free_ports(free_ports->next);
}

static void
test_free_ports() {
        int i;

        for(i = 0; i < 3; i++) {
                add_port(&port_free_list, i);
        }

        printf("Free ports...");

        for(i = 0; i < 3; i++)
                if (get_port(&port_free_list) != 2-i) {
                        printf("\tFAIL\n");
                        return;
                }

        printf("\tOK\n");

}

static void
clean_free_ports(struct port_item **free_ports) {
        struct port_item *next_port, *cur_port;
        cur_port = *free_ports;

        while(cur_port != NULL) {
                next_port = cur_port->next;
                free(cur_port);
                cur_port = next_port;
        }

        *free_ports = NULL;
        #if VERBOSE
                printf("Free ports list cleaning...\tDONE\n");
        #endif
}

void
create_free_ports(struct port_item **free_ports) {
        int i;
        for (i = 0; i < PORTS; i++) {
                add_port(free_ports, PORTS_RESERVED + i);
        }
        #if VERBOSE
                printf("Free port list generation...\tDONE\n");
        #endif
}

/*
 *       Makes a key for nat out of ip and port
 */
static inline unsigned int
makeKey(unsigned int addr, unsigned int port) {
         unsigned int key;

         key = port << 16 | (addr & NET_MASK);

         return key;
}

//
//      returns NAT hash table index
//
static inline unsigned int
getHashTableIdx(unsigned int addr, unsigned int port) {
        unsigned int key;
        unsigned int hash_mod = HASHING_VAL;
        key = makeKey(addr, port);
        key = key % hash_mod;
        return key;
}

//
//      returns index in the nat table
//
static inline int
getNatTableIndex(unsigned int ip, unsigned int port) {
        struct HashListItem *item;
        int hash_idx;
        int idx;
        unsigned int key;

        hash_idx = getHashTableIdx(ip, port);
        key = makeKey(ip, port);
        #if VERBOSE
                printf("Hash index is %u\n", hash_idx);
        #endif
        if (nat_hash_table[hash_idx] == NULL)
                return -NOT_FOUND;

        #if VERBOSE
                printf("Searching for the data in the list");
        #endif

        item = nat_hash_table[hash_idx];

        while(item != NULL) {
                if (item->key == key) {
                        if (item->valid)
                                return item->nat_table_index;
                }

                item = item->next;
        }

        return -NOT_FOUND;

}

// put a new record into hat_hash_table
// the record consists of ip, port, validity and index to the nat_table
static inline void
putDataIntoHashTable(unsigned int src_ip, unsigned int src_port, int nat_tbl_idx){
        struct HashListItem *item, *new_item;
        unsigned int hash_idx;


        hash_idx = getHashTableIdx(src_ip, src_port);

        if (nat_hash_table[hash_idx] == NULL) {
                new_item = (struct HashListItem*)
                        malloc(sizeof(struct HashListItem)*3);
                new_item->num = 0;
                new_item->key = makeKey(src_ip, src_port);
                new_item->valid = true;
                new_item->nat_table_index = nat_tbl_idx;
                new_item->next = NULL;
                nat_hash_table[hash_idx] = new_item;
                return;
        }

        item = nat_hash_table[hash_idx];
// ====================================================
        while(true) {
                if (item->next == NULL) {
                        if (item->num == 2){
                                new_item = (struct HashListItem*)
                                        malloc(sizeof(struct HashListItem)*3);
                                new_item->num = 0;
                        }
                        else {
                                new_item = (item+1);
                                   new_item->num = item->num+1;
                        }

                        new_item->key = makeKey(src_ip, src_port);
                        new_item->valid = true;
                        new_item->nat_table_index = nat_tbl_idx;
                        new_item->next = NULL;
                        item->next = new_item;
                        return;
                }
                else
                        item = item->next;
        }
}
```

```
//      clears the nat_hash_table and prints its size
static inline void
clearHashTable() {
        int i;
        unsigned int itemCnt;
        struct HashListItem *item, *next;

        itemCnt = 0;

        for(i = 0; i < HASHING_VAL; i++) {
                item = nat_hash_table[i];

                while(item != NULL) {
                        itemCnt++;
                        next = item->next;

                        if (item->num == 0) {
                                free(item);
                        }

                        item = next;
                }

                nat_hash_table[i] = NULL;
        }
        printf("Hash table size was [%u] %lu\n",
                itemCnt,
                sizeof(struct HashListItem)*itemCnt + sizeof(nat_hash_table));
}

/*
 *      Finds a valid slot (ip:address) for address translation
 */
static inline int
find_nat_entry(int saddr, int sport) {

        int idx;
        idx = getNatTableIndex(saddr, sport);

        if (idx < 0)
                return idx; // <- NOT_FOUND

        if (get_seconds()-nat_table[idx].sec > timeout) {
                nat_table[idx].valid = false;
                return -TIMEOUT_EXCEEDED;
        }

        return idx;
}

static inline int
make_nat_entry(unsigned int ip, unsigned int port){
        int index, p;

        if ((p = get_port(&port_free_list)) < 0)
                return -FAULT;

        index = get_port_index(p);

        nat_table[index].lan_ipaddr= ip;
        nat_table[index].lan_port = port;
        nat_table[index].valid = (char) true;
        nat_table[index].sec = get_seconds();
        putDataIntoHashTable(ip, port, index);

        #if VERBOSE
                printf("New NAT entry port: %d\n", p);
        #endif
        return index;

}

static inline int
translate_lan_to_wan(struct ippkt *pkt) {
        int index, port;
        #if VERBOSE
                printf("Searching for nat entry %u:%d -> ", pkt->ip, pkt->port);
        #endif

        if((index = find_nat_entry(pkt->ip, pkt->port)) < 0)
        {
                #if VERBOSE
                        printf("Proper index hasn't been found\n");
                #endif
                if (index == -NOT_FOUND) {
                        if ((index = make_nat_entry(pkt->ip, pkt->port)) < 0)
                                return -FAULT;
                }
                else if(index == -TIMEOUT_EXCEEDED) {
                        printf("Exiting with %d\n", index);
                        return -FAULT;
                }
                else {
                        printf("Exiting with %d\n", index);
                        return -FAULT;
                }
        }

        #if VERBOSE
                printf("Index found: %d\n", index);
        #endif

        pkt->ip = wan_ip;
        pkt->port = get_index_port(index);
        pkt->chksum = ip_fast_csum((void*) pkt, sizeof(struct ippkt));
        nat_table[index].sec = get_seconds();
        return SUCCESS;
}

static inline int
translate_wan_to_lan(struct ippkt *pkt) {
        int index;
        index = get_port_index(pkt->port);
        #if VERBOSE
                printf("Port to index from wan: %d -> %d\n", pkt->port, index);
        #endif

        if (true) {
                pkt->ip = nat_table[index].lan_ipaddr;
                pkt->port = nat_table[index].lan_port;
                pkt->chksum = ip_fast_csum((void*) pkt, sizeof(struct ippkt));
                nat_table[index].sec = get_seconds();
                return SUCCESS;
        }
        else
                return -NOT_VALID_ENTRY;
}

/*
 *      Checks
 *
 */
int
pkt_set_check() {
        int i;
        for (i = 0; i < pkt_set_size; i++)
                if (
                        pkt_set->ip != pkt_test_set->ip ||
                        pkt_set->port != pkt_test_set->port
                )
                        return -FAULT;

        return SUCCESS;
}

int
test_nat_capacity() {
        return SUCCESS;
}

/*
 *      testing interface functions
 */
void
test_create(int *params, int num_params) {
         if (test_nat_capacity() < 0){
                 printf("Exiting...\n");
                 exit(0);
         }
        wan_ip = str_to_ip(NET_ADDR_WAN);
        pkt_set_size = params[0];
        pkt_set = (struct ippkt*) malloc (sizeof(struct ippkt) * pkt_set_size);
        pkt_test_set =
                (struct ippkt*) malloc (sizeof(struct ippkt) * pkt_set_size);
}

void
test_show_info() {
         printf("===== TEST INFO =====\n");
         printf("Max of ip-port combinations:\t%d\n",
                NET_ADDR_2_RANGE * NET_ADDR_3_RANGE * NUM_PORTS_TO_GENERATE);
         printf("NAT capacity:\t%d\n", MAX_NAT_ENTRIES);
         printf("NAT entry size:\t%lu\n", sizeof(struct nat_entry));
         printf("NAT table size:\t%lu\n", sizeof(nat_table));
        printf("NAT hash table entry size:\t%lu\n",
                sizeof(struct HashListItem));
        printf("NAT hash table size:\t%lu\n", sizeof(nat_hash_table));
         printf("Packet size:\t%lu\n", sizeof(struct ippkt));
         printf("Packet amount:\t%d\n", pkt_set_size);
         printf("Pkt set size:\t%'lu\n", sizeof(struct ippkt)*pkt_set_size);
         printf("=====================\n\n");
}


void
test_init() {
        create_free_ports(&port_free_list);
        generate_ippkt_set(pkt_set, pkt_set_size);
}


void
test_clean() {
        memset(pkt_set, 0, sizeof(pkt_set));
        memset(nat_table, 0, sizeof(nat_table));
        memset(pkt_set, 0, sizeof(struct ippkt) * pkt_set_size);
        memset(pkt_test_set, 0, sizeof(struct ippkt) * pkt_set_size);
        clearHashTable();
        clean_free_ports(&port_free_list);
}

void inline
print_error(int packet_id) {
        printf("Translation of packet [%d] FAILED\n", packet_id);
}


void
test_run(int loops) {
        int i, j;
        int ip = 0;
        #if VERBOSE
                printf("Test has been started\n");
        #endif
        for(i = 0; i < loops; i++) {
                // translation from lan to wan
                for (j = 0; j < pkt_set_size; j++) {
                        if(translate_lan_to_wan(&pkt_set[j]) < 0)
                                print_error(j);
                }
                // translation from wan to lan
                for (j = 0; j < pkt_set_size; j++)
                        if(translate_wan_to_lan(&pkt_set[j]) < 0)
                                print_error(j);
        }
}

int
test_make_checks() {
        if (pkt_set_check() < 0) {
                printf("Packet inconsistancy found!\n");
                return -FAULT;
        }

        return SUCCESS;
}
```

Parallel hash table based version:

```
#include <stdbool.h>
#include "stdio.h"
#include <sys/time.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <pthread.h>

#define PORTS 64000000
#define WANIP 1 // 83 for 5.5 Mpackets
#define MAX_NAT_ENTRIES PORTS*WANIP
#define SEC 1000000 // usec in a sec
#define PORTS_RESERVED 1024
#define GEN_ADDR_MAX_VAL 10
#define TIMEOUT 180 // timeout in seconds

#define NET_ADDR_0 192
#define NET_ADDR_1 168
#define NET_ADDR_2_RANGE 254
#define NET_ADDR_3_RANGE 254
#define NET_ADDR_WAN "10.10.10.10"
#define NET_MASK 0xFFFF
#define NUM_PORTS_TO_GENERATE 15000
#define NET_NUM_NODES NET_ADDR_2_RANGE*NET_ADDR_3_RANGE

#define HASHING_VAL 1<<24

#define SUCCESS 0
#define FAULT 1
#define NOT_FOUND 2
#define TIMEOUT_EXCEEDED 3
#define EMPTY 4
#define NOT_VALID_ENTRY 5

#define VERBOSE 0

struct PortItem {
        int port;
        struct PortItem* next;
};

struct L4Pkt {
        unsigned int chksum;
        unsigned int opt;
};

struct Pkt {
        unsigned int ip;
        unsigned int port;
        unsigned int chksum;
        struct L4Pkt tcp_pkt; // a packet of the level 4 protocol
};

struct NatEntry {
         int  lan_ipaddr;
         int lan_port;
         unsigned long long sec;      /*timestamp in micro-seconds*/
         bool valid;
};

struct HashEntry {
        unsigned int nat_table_index;
        unsigned int key;
        struct HashEntry *next;
        bool valid;
        char num;
} __attribute__((__packed__));


extern double RandN (double mu, double sigma);

static struct NatEntry** nat_table_pool;
static struct HashEntry*** nat_hash_table_pool;
static struct PortItem** port_free_list_pool;
static int* nat_table_indices;
static int* nat_table_sizes;

static unsigned long long timeout = TIMEOUT * SEC;
static struct Pkt *pkt_set;
static struct Pkt *pkt_test_set;
static int pkt_set_size;
static unsigned int g_wan_ip;
static int g_threads_num;

static __inline__ unsigned long long rdtsc(void) {
    unsigned int lo, hi;
    __asm__ volatile ("rdtsc\n" : "=a" (lo), "=d" (hi));
    return ((unsigned long long) hi << 32) | lo;
}

// each item of the nat_hash_table is a pointer to the first
// element of the linked list of nat_entries

/*
 *      fast checksum calculation
 *      taken from
 *      /usr/src/linux-headers-3.13.0-44/arch/x86/include/asm/checksum_64.h
 */
static inline unsigned int
IpFastCsum(const void *iph, unsigned int ihl)
{
        unsigned int sum;

        __asm__ ("  movl (%1), %0\n"
            "  subl $4, %2\n"
            "  jbe 2f\n"
            "  addl 4(%1), %0\n"
            "  adcl 8(%1), %0\n"
            "  adcl 12(%1), %0\n"
            "1: adcl 16(%1), %0\n"
            "  lea 4(%1), %1\n"
            "  decl %2\n"
            "  jne      1b\n"
            "  adcl $0, %0\n"
            "  movl %0, %2\n"
            "  shrl $16, %0\n"
            "  addw %w2, %w0\n"
            "  adcl $0, %0\n"
            "  notl %0\n"
            "2:"
        /* Since the input registers which are loaded with iph and ihl
           are modified, we must also specify them as outputs, or gcc
           will assume they contain their original values. */
            : "=r" (sum), "=r" (iph), "=r" (ihl)
            : "1" (iph), "2" (ihl)
            : "memory");

        return sum;
}

/*
 *      timestamp in usec
 */
static inline unsigned long long
GetSeconds() {
       struct timeval t;
       gettimeofday(&t, NULL);
       return t.tv_sec * SEC + t.tv_usec;
       //return 1;
}


int inline
GetPortIndex(unsigned int port, unsigned int ports_shift) {
       return port - ports_shift;
}

int inline
GetIndexPort(unsigned int index, unsigned int ports_shift) {
       return index + ports_shift;
}

/*
 *      test for GetSeconds(() (timestamp) function
 */
```

```
void
TestGetSeconds(int loops){
        unsigned long long t;
        int i;
        for (i = 0; i < loops; i++) {
                t = GetSeconds();
                printf("Time stamp: %llu\n", t);
        }
}

/*
 *      convert a string IPv4 to int IPv4 in big-endian order
 */
unsigned int
StrToIp(char *ipstr) {
        unsigned int ip;
        unsigned int a[4];

        sscanf(ipstr, "%u.%u.%u.%u", &a[0], &a[1], &a[2], &a[3]);
        ip = (a[0] << 24) | (a[1] << 16) | (a[2] << 8) | a[3];
        return ip;
}
/*
 *      converts to ip address in big-endian notation out of 4 int numbers
 */
unsigned int
MakeIp(int a0, int a1, int a2, int a3) {
        unsigned int ip;
        ip = (a0 << 24) | (a1 << 16) | (a2 << 8) | a3;
        return ip;
}

/*
 *      print an ip-address
 */
void
PrintIpPort(struct Pkt *pkt)
{
         printf("%d.%d.%d.%d:%d\n",
                pkt->ip >> 24 & 0xFF,
                pkt->ip >> 16 & 0xFF,
                pkt->ip >> 8 & 0xFF,
                pkt->ip & 0xFF,
                pkt->port
        );
}

/*
 *      packet generation functions
 */
int
GetRandVal(int max_val) {
         int i;
         double r;

         r = -4;
         while(abs(r) > 1.5)
                 r = RandN(0, 2);

         r = r + 2;
         r = round(r/4 * max_val + 1);

         return (int) r;
}

void
GeneratePktSet(struct Pkt *pkt_set, int num) {
        int i;
        unsigned int host_ip2, host_ip3, host_port;

        srand(time(NULL));

        for(i = 0; i < num; i++) {
                host_ip2 = GetRandVal(NET_ADDR_2_RANGE);
                host_ip3 = GetRandVal(NET_ADDR_3_RANGE);

                pkt_set[i].ip = MakeIp(
                        NET_ADDR_0, NET_ADDR_1, host_ip2, host_ip3
                );
                host_port = GetRandVal(NUM_PORTS_TO_GENERATE)+PORTS_RESERVED;

                pkt_set[i].port = host_port;
        }

        // save a copy of pkt_set
        memcpy(pkt_test_set, pkt_set, sizeof(struct Pkt) * num);

        #if VERBOSE
                printf("Packet (%d) generation...\tDONE\n", num);
        #endif
}

/*
 *      free port list managing functions
 */
static inline int
GetPort(struct PortItem **free_ports) {
        int port;
        struct PortItem *used_item;

        if (free_ports == NULL)
                return -EMPTY;

        used_item = *free_ports;
        port = used_item->port;

        (*free_ports) = used_item->next;
        free(used_item);
        #if VERBOSE
                printf("GetPort: %d\n", port);
        #endif
        return port;
}

static inline void
AddPort(struct PortItem **free_ports, int port) {
        struct PortItem *prev_port;
        prev_port = *free_ports;

        *free_ports = (struct PortItem*) malloc(sizeof(struct PortItem));

        (*free_ports)->port = port;
        (*free_ports)->next = prev_port;
}

static void
CreateFreePorts(struct PortItem **free_ports, int start, int num) {
        int i;

        for (i = start; i < start + num; i++) {
                AddPort(free_ports, i);
        }

        #if VERBOSE
        printf("Free port list generation at %p from %d to %d...\tDONE\n",
                free_ports, start, start+num);
        #endif
}

static void
CreateFreePortsPool() {
        int i, reminder, pool_size;;

        for (i = 0; i < g_threads_num; i++)
                CreateFreePorts(&port_free_list_pool[i],
                        nat_table_indices[i], nat_table_sizes[i]);
}

static void
ClearFreePorts(struct PortItem **free_ports) {
        struct PortItem *next_port, *cur_port;
        cur_port = *free_ports;

        while(cur_port != NULL) {
                next_port = cur_port->next;
                free(cur_port);
                cur_port = next_port;
        }
        *free_ports = NULL;
        #if VERBOSE
                printf("Free ports list cleaning at %p...\tDONE\n", free_ports);
        #endif
}

static void
ClearFreePortsPool() {
        int i;
        for (i = 0; i < g_threads_num; i++)
                ClearFreePorts(&port_free_list_pool[i]);
}

//
//      Makes a key for nat out of ip and port
//
static inline unsigned int
MakeKey(unsigned int addr, unsigned int port) {
         unsigned int key;

         key = (addr & NET_MASK) << 16 | port;

         return key;
}

//
//      returns NAT hash table index
//
static inline unsigned int
GetHashTableIdx(unsigned int addr, unsigned int port) {
        unsigned int key;
        unsigned int hash_mod = HASHING_VAL;
        key = MakeKey(addr, port);
        key = key % hash_mod;
        return key;
}


//
//      returns index in the nat table
//
static inline int
GetNatTableIdx(struct HashEntry** hash_table, signed int ip, unsigned int port) {
        struct HashEntry *item;
        int hash_idx;
        int idx;
        unsigned int key;

        hash_idx = GetHashTableIdx(ip, port);
        key = MakeKey(ip, port);
        #if VERBOSE
                printf("Hash index is %u\n", hash_idx);
        #endif
        if (hash_table[hash_idx] == NULL)
                return -NOT_FOUND;

        #if VERBOSE
                printf("Searching for the data in the list");
        #endif

        item = hash_table[hash_idx];

        while(item != NULL) {
                if (item->key == key) {
                        if (item->valid)
                                return item->nat_table_index;
                }

                item = item->next;
        }

        return -NOT_FOUND;
}

// put a new record into hat_hash_table
// the record consists of ip, port, validity and index to the nat_table
static inline void
PutDataIntoHashTable(
        struct HashEntry** hash_table,
        unsigned int src_ip, unsigned int src_port,
        int nat_tbl_idx) {

        struct HashEntry *new_item, *item;
        unsigned int hash_idx;

        hash_idx = GetHashTableIdx(src_ip, src_port);

        if (hash_table[hash_idx] == NULL) {
                new_item = (struct HashEntry*)
                        malloc(sizeof(struct HashEntry)*3);
                new_item->num = 0;
                new_item->key = MakeKey(src_ip, src_port);
                new_item->valid = true;
                new_item->nat_table_index = nat_tbl_idx;
                new_item->next = NULL;
                hash_table[hash_idx] = new_item;
                return;
        }

        item = hash_table[hash_idx];

        while(true) {
                if (item->next == NULL) {
                        if (item->num == 2){
                                new_item = (struct HashEntry*)
                                        malloc(sizeof(struct HashEntry)*3);
                                new_item->num = 0;
                        }
                        else {
                                new_item = (item+1);
                                new_item->num = item->num+1;
                        }

                        new_item->key = MakeKey(src_ip, src_port);
                        new_item->valid = true;
                        new_item->nat_table_index = nat_tbl_idx;
                        new_item->next = NULL;
                        item->next = new_item;
                        return;
                }
                else
                        item = item->next;
        }

}

//      clears the nat_hash_table and prints its size
static inline void
ClearHashTable(struct HashEntry** hash_table) {
        unsigned long i, hashing_val;
        unsigned int itemCnt, rootItemCnt;
        struct HashEntry *item, *next;

        itemCnt = 0;
        rootItemCnt = 0;
        hashing_val = (unsigned long long) HASHING_VAL;

        for(i = 0; i < hashing_val; i++) {
                item = hash_table[i];

                if (item != NULL)
                        rootItemCnt++;


                while(item != NULL) {
                        itemCnt++;
                        next = item->next;
                        if (item->num == 0)
                                free(item);
                        item = next;
                }

                hash_table[i] = NULL;
        }

        printf("Hash table size was [%d] %llu\n",
                itemCnt,
                sizeof(struct HashEntry)*(unsigned long long)(itemCnt-rootItemCnt)
                + sizeof(hash_table)*hashing_val
        );
```

```
}

//
//      Finds a valid slot (ip:address) for address translation
//
static inline int
FindNatEntry(
        struct HashEntry** hash_table, struct NatEntry* nat_table,
        int saddr, int sport) {

        int idx;
        idx = GetNatTableIdx(hash_table, saddr, sport);

        if (idx < 0)
                return idx; // <- NOT_FOUND

        if (GetSeconds()-nat_table[idx].sec > timeout) {
                nat_table[idx].valid = false;
                return -TIMEOUT_EXCEEDED;
        }

        return idx;
}

static inline int
MakeNatEntry(
        struct HashEntry** hash_table, struct NatEntry* nat_table,
        struct PortItem** port_free_list, unsigned int ports_shift,
        unsigned int ip, unsigned int port) {

        int index, p;

        if ((p = GetPort(port_free_list)) < 0)
                return -FAULT;

        index = GetPortIndex(p, ports_shift);

        nat_table[index].lan_ipaddr= ip;
        nat_table[index].lan_port = port;
        nat_table[index].valid = true;
        nat_table[index].sec = GetSeconds();
        PutDataIntoHashTable(hash_table, ip, port, index);

        #if VERBOSE
                printf("New NAT entry port: %d\n", p);
        #endif
        return index;

}

static inline int
TranslateLanToWan(
         struct HashEntry** hash_table, struct NatEntry* nat_table,
        struct PortItem** free_ports, unsigned int ports_shift, struct Pkt* pkt) {

        int index, port;
        #if VERBOSE
                printf("Searching for nat entry %u:%d -> ", pkt->ip, pkt->port);
        #endif

        if((index = FindNatEntry(hash_table, nat_table, pkt->ip, pkt->port)) < 0)
        {
                #if VERBOSE
                        printf("Proper index hasn't been found\n");
                #endif
                if (index == -NOT_FOUND) {
                        if (
                                (index = MakeNatEntry(hash_table, nat_table,
                                                      free_ports, ports_shift,
                                                      pkt->ip, pkt->port)
                        ) < 0)
                                return -FAULT;
                }
                else if(index == -TIMEOUT_EXCEEDED) {
                        printf("Exiting with %d\n", index);
                        return -FAULT;
                }
                else {
                        printf("Exiting with %d\n", index);
                        return -FAULT;
                }
        }

        #if VERBOSE
                printf("Index found: %d\n", index);
        #endif

        pkt->ip = g_wan_ip;
        pkt->port = GetIndexPort(index, ports_shift);
        pkt->chksum = IpFastCsum((void*) pkt, sizeof(struct Pkt));
        nat_table[index].sec = GetSeconds();
        return SUCCESS;
}

static inline int
TranslateWanToLan(
        struct NatEntry* nat_table, unsigned int ports_shift,
        struct Pkt *pkt) {

        int index;
        index = GetPortIndex(pkt->port, ports_shift);
        #if VERBOSE
                printf("Port to index from wan: %d -> %d\n", pkt->port, index);
        #endif

        if (true) {
                pkt->ip = nat_table[index].lan_ipaddr;
                pkt->port = nat_table[index].lan_port;
                pkt->chksum = IpFastCsum((void*) pkt, sizeof(struct Pkt));
                nat_table[index].sec = GetSeconds();
                return SUCCESS;
        }
        else
                return -NOT_VALID_ENTRY;
}

//
//      Checks
//

int
PktSetCheck() {
        int i;
        for (i = 0; i < pkt_set_size; i++)
                if (
                        pkt_set->ip != pkt_test_set->ip ||
                        pkt_set->port != pkt_test_set->port
                )
                        return -FAULT;

        return SUCCESS;
}

int
TestNatCapacity() {
                 return SUCCESS;
}

//
//      testing interface functions
//
void
TestCreate(int *params, int num_params) {
         int i, reminder, pool_size;
        size_t size;

         if (TestNatCapacity() < 0) {
                 printf("Exiting...\n");
                 exit(0);
         }

        g_wan_ip = StrToIp(NET_ADDR_WAN);
        pkt_set_size = params[0];
        g_threads_num = params[1];

    pool_size = PORTS / g_threads_num;
    reminder = PORTS % g_threads_num;

        nat_table_indices = (int*) malloc(sizeof(int) * g_threads_num);
        nat_table_sizes = (int*) malloc(sizeof(int) * g_threads_num);
        // splitting ports into parts saving each part index and size
        for(i = 0; i < g_threads_num; i++) {
                nat_table_indices[i] = pool_size * i;
                nat_table_sizes[i] = pool_size;
        }

        if (reminder > 0)
                nat_table_sizes[i-1] += reminder;

        // get memory for packet sets
        pkt_set = (struct Pkt*) malloc(sizeof(struct Pkt) * pkt_set_size);
        pkt_test_set = (struct Pkt*) malloc(sizeof(struct Pkt) * pkt_set_size);

        // get memory for nat_tables, hash_tables and free_port_lists
        // they are arrays to be passed to the threads
        port_free_list_pool =
             (struct PortItem**)
                malloc(sizeof(struct PortItem*) * g_threads_num);
        nat_table_pool =
              (struct NatEntry**)
                malloc(sizeof(struct NatEntry*) * g_threads_num);
        nat_hash_table_pool =
             (struct HashEntry***)
                malloc(sizeof(struct HashEntry**) * g_threads_num);

        // clear memeory for just created nat structures
        for(i = 0; i < g_threads_num; i++) {
                size = sizeof(struct NatEntry)*nat_table_sizes[i];
                nat_table_pool[i] = (struct NatEntry*)malloc(size);
                memset(nat_table_pool[i], 0, size);

                size = sizeof(struct HashEntry*)*HASHING_VAL;
                nat_hash_table_pool[i] = (struct HashEntry**) malloc(size);
                memset(nat_hash_table_pool[i], 0, size);
        }
}

void
TestShowInfo() {
    printf("===== TEST INFO =====\n");
    printf("Max of ip-port combinations:\t%d\n",
             (unsigned) NET_NUM_NODES *  NUM_PORTS_TO_GENERATE);
    printf("NAT capacity:\t%d\n", MAX_NAT_ENTRIES);
    printf("NAT entry size:\t%lu\n", sizeof(struct NatEntry));
        printf("NAT table size:\t%lu\n", sizeof(struct NatEntry)*PORTS);
        printf("NAT hash table entry:\t%lu\n", sizeof(struct HashEntry));
        printf("NAT hash table size:\t%lu\n",
                sizeof(struct HashEntry*) *
                (unsigned)g_threads_num *
                (unsigned) HASHING_VAL
        );
    printf("Packet size:\t%lu\n", sizeof(struct Pkt));
    printf("Packet amount:\t%d\n", pkt_set_size);
    printf("Pkt set size:\t%'lu\n", sizeof(struct Pkt) * pkt_set_size);
    printf("====================\n\n");
}


void
TestInit() {
        int i;
        size_t size;
        CreateFreePortsPool();
        GeneratePktSet(pkt_set, pkt_set_size);
}

void
TestClean() {
        int i;
        memset(pkt_set, 0, sizeof(pkt_set));
        memset(pkt_set, 0, sizeof(struct Pkt) * pkt_set_size);
        memset(pkt_test_set, 0, sizeof(struct Pkt) * pkt_set_size);

        ClearFreePortsPool();

        for (int i = 0; i < g_threads_num; i++) {
                #if VERBOSE
                        printf("Cleaning nat_table_pool[%d]<%p>:%d\n",
                                i, nat_table_pool[i], nat_table_sizes[i]);
                #endif

                memset(nat_table_pool[i], 0,
                        sizeof(struct NatEntry)*nat_table_sizes[i]);

                #if VERBOSE
                        printf("Cleaning nat_hash_table_pool[%d]\n", i);
                #endif

                ClearHashTable(nat_hash_table_pool[i]);
        }
}

void inline
PrintError(int packet_id) {
        printf("Translation of packet [%d] FAILED\n", packet_id);
}

struct WorkerParams{
        unsigned int in_id;
        struct NatEntry* in_nat_table;
        struct HashEntry** in_hash_table;
        struct PortItem* in_port_free_list;
        struct Pkt *in_packet_set;
        unsigned int in_packets_amount;
        unsigned int in_start_index;
        unsigned long long out_ticks_elapsed;
};

void*
Worker(void *thread_params) {
        unsigned int id;
        unsigned int packets_amount, start_index;
        unsigned int i, j, k;
         unsigned long long start, end, ticks_elapsed;
        struct Pkt *packet_set;
        struct WorkerParams *params;
        struct NatEntry* nat_table;
        struct HashEntry** hash_table;
        struct PortItem* port_free_list;

        params = (struct WorkerParams*) thread_params;

        nat_table = params->in_nat_table;
        hash_table = params->in_hash_table;
        port_free_list = params->in_port_free_list;

        id = params->in_id;
        packets_amount = params->in_packets_amount;
        packet_set  = params->in_packet_set;
        start_index = params->in_start_index;

         #if VERBOSE
                 printf("[%d]Hi! Thread data: nat %p, hash %p, ports %p\
                        packets amount %u, packet_set %p, start idx %d\n",
                        id, nat_table, hash_table, port_free_list,
                        packets_amount, packet_set, start_index);

         #endif

        j = 0;

         start = rdtsc();

        for (i = 0; i < packets_amount; i++) {
                j++;nat_table_indices;
                k = packet_set[i].ip;
        }

        #if VERBOSE
                printf("Worker ID[%i]: j = %u k = %u\n", id, j, k);
        #endif
        // translation from lan to wan
        for (i = 0; i < packets_amount; i++)
                 if(TranslateLanToWan(
                                hash_table, nat_table, &port_free_list,
                                start_index, &packet_set[i]
                        ) < 0
                )
                        PrintError(i);
        // translation from wan to lan
        for (i = 0; i < packets_amount; i++)
                if(TranslateWanToLan(
                                nat_table, start_index, &packet_set[i]
                        ) < 0
                )
                        PrintError(i);

        end = rdtsc();
        ticks_elapsed = end - start;

        #if VERBOSE
        printf("Ticks elapsed: %llu\n", ticks_elapsed);
        #endif

        params->out_ticks_elapsed = ticks_elapsed;

        #if VERBOSE
                printf("JOB DONE. My id: %u\n", id);
        #endif
}

unsigned long long
TestRun() {
        int i, set_chunk_size;
        pthread_t threads[g_threads_num];
        struct WorkerParams worker_params[g_threads_num];
        int set_chunk_reminder;
        unsigned long long ticks_sum = 0;

        // splitting the pkt_set
        set_chunk_size = pkt_set_size/g_threads_num;
        set_chunk_reminder = pkt_set_size % g_threads_num;

        for(i = 0; i < g_threads_num; i++) {
                worker_params[i].in_id = i;
                worker_params[i].in_nat_table = nat_table_pool[i];
                worker_params[i].in_hash_table = nat_hash_table_pool[i];
                worker_params[i].in_port_free_list = port_free_list_pool[i];
                worker_params[i].in_packet_set = &pkt_set[set_chunk_size*i];
                worker_params[i].in_packets_amount = set_chunk_size;
                worker_params[i].in_start_index = nat_table_indices[i];
                worker_params[i].out_ticks_elapsed = 0;
        }

        // if packet set isn't roundly dividable to g_threads_num than
        // make the last thread to do the rest of the work
        if (set_chunk_reminder != 0) {
                worker_params[i-1].in_packets_amount += set_chunk_reminder;
        }

        #if VERBOSE
                for (i = 0; i <g_threads_num; i++) {
                        printf("Chunk #%d ID: %u amount: %u adr %p\n",
                                i,
                                worker_params[i].in_id,
                                worker_params[i].in_packets_amount,
                                worker_params[i].in_packet_set
                        );
                }
        #endif

        // start all threads
        for (i = 0; i < g_threads_num; i++) {
                if (pthread_create(
                        &threads[i], NULL, Worker, (void*)&worker_params[i]
                        )
                ) {
                        printf("Can't join thread id:%u\n",worker_params[i].in_id);
                        return 0;
                }
        }


        // join all the threads
        for (i = 0; i < g_threads_num; i++) {
                if (pthread_join(threads[i], NULL)) {
                        printf("Can't join thread id:%u\n",worker_params[i].in_id);
                        return 0;
                }

        }

        // collecting the results

        for (i = 0; i < g_threads_num; i++) {
                #if VERBOSE
                        printf("Ticks: Thread #%d : %llu\n",
                                i, worker_params[i].out_ticks_elapsed);
                #endif
                ticks_sum += worker_params[i].out_ticks_elapsed;
        }

        #if VERBOSE
                printf("Ticks attempt sum: %llu\n",  ticks_sum);
        #endif

        return ticks_sum/g_threads_num/2;

}

int
TestMakeChecks() {
        if (PktSetCheck() < 0) {
                printf("Packet inconsistancy found!\n");
                return -FAULT;
        }

        return SUCCESS;
}
```

e