**Developing a Software Defined CG-NAT**

Denis Plotnikov

<INTRO>

…

<WHAT IS NAT WHAT SHOULD IT DO (TYPES OF NAT and corresponding RFC overview)>

RFC

…

<WHAT IS CARRIER GRADE NAT, WHAT DOES IT DO, WHO CEARS>

RFC

…

<WHAT IS THE NOWADAYS APPROACH TO DO NAT>

Specific network devices

…

<WHY THIS APPROACH ISN’T GOOD: drawback of specialized NAT devices>

…

<WHAT ARE THEIR PRICES – the drawback #1>

My doc about prices with graph goes here

…

<HOW CAN WE DO BETTER>

Algorithmic(data structure) + technology(software/hardware architecture approaches (batching processing, NIC RSS queues, multithreading, “locklessness”, DDIO ) -> consequence DPDK already have all of those so let’s choose it)

…

<THE GOAL OF THIS WORK>

let’s try to make one reducing the price of nat as much as we can

…

<WHAT OUR NAT SHOULD BE ABLE TO DO>

The same as written in RFCs but light version of it just to prove that the desired packet processing speed can be achieved

…

<>

**NAT DEVELOPING**

<how we are going to do that and what do we need for that – BASICALLY THE INTRO OF THIS PART >

**NAT Performance Metrics**

Еру 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 [ref\_TT\_ROS\_TEL]. The second one is the performance specification claimed by one of the on-market available NAT device producers[ref\_RDP.RU] 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. (our\_approach)**

**Key Performance Characteristics**

* ***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.

**CG NAT Target Metric Values**

Based on the sources of information the performance requirements of the NAT device are set following:

* 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

In this document for evaluation of the performance another metric is used: ***Cycles per packet*** – the amount of processors’ cycles spent on processing of one packet. This metric 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 metric gives clearer impression of the performance because at least it doesn’t depend on CPU frequency. Thus, the main performance metric used in this work is Cycles Per Packet while packet processing rate, concurrent session support, connections setups rate and throughput play supplementary role and is mentioned where their role becomes important.

**NAT Design Approach**

The design process of the NAT system consists of two parts. In the first part the exploration of possible algorithms and data structures are made to choose the one which gives the best performance results. The second part is about choosing of technological approaches which give an answer to the question of how the program have to be structured and what technics have to be used to achieve the target performance.

**NAT testing application**

For choosing 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. Conditionally the program can be 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. The measuring part performs the number of tests set by the user and as an output calculates the average value of cycles per packet achieved and standard deviation of the result.

***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. 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. There are several mandatory actions which must be performed by any NAT to actually perform proper packets translations. They are: calculation of the check sum, setting time stamp and saving/acquiring translation information in a NAT lookup data structure. The last point is the main exploration area of this document.

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 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 ref\_rfc791] and TCP/UDP protocols [1.5 of ref\_rfc793 and rfc\_768].

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 [ref\_rfc4787].

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. These functions might be potential targets of performance optimization but are out of the scope of this document.

**“Ideal” NAT performance**

The most interesting part in the NAT system is the algorithm and data structure for storing the address translation information. In fact, two of the data structures are needed because of the necessity to store two pieces of data for a single address translation. The first one is the data about translation from LAN to WAN and second one is the data about the translation from WAN to LAN. From the first glance it isn’t clear how to organize them well. This issue is explored further in the document.

Before starting the exploration of the NAT translation algorithms and data structures it is essential to estimate the performance of the system which uses the ideal NAT translator. By word “ideal” the zero-time lookup data structure is implied. To get this estimation the bogus data structure has been used which returns deterministic result and requires computation time tends to be zero. Another words, it is a function which cyclically returns the same sequence of results.

In the figure 1 there are some results explaining the cost of one packet processing having the ideal lookup data structure. One packet processing takes around 120 cycles including calculation of checksums and timestamps settings processing. The packet processing routine takes around 40 cycles including processing of the test packet set which could be thought like simulation of packet acquiring from the network interface card queues. Thus, the overall overhead of packet data checksums calculation and timestamp/timeout processing is around 80 cycles.

Based on this data it is possible to claim that than closer the performance of a NAT to “ideal” values than better the NAT setup is. In our case the ideal value is 120 cycles/pkt.

Figure 1. "Ideal" NAT performance

**NAT Bottleneck**

In this chapter the part of NAT system is to be investigated which has the most significant influence on overall NAT performance. Each NAT system should store information in some kind of data structure to be able to retrieve this information when it is necessary. 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 many cycles we could spend on a packet. Having a processor working on 2.4GHz frequency we could spend 436 cycles per packet.

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 [ref\_cormen]. 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 Figure 2.

Figure 2. NAT Performance: Linear search

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.

**Testing of NAT lookup data structures and algorithms**

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 could be reduced and it is shown in further chapters of this document but nevertheless it is still a big chunk of memory which cannot be placed in the fast CPU cache memory so the solution to be chosen cannot rely on that memory and should take into consideration software construction approaches that could help to eliminate cache misses which could be a key point in the race to the performance.

To find a suitable solution it is worth to look for a data structure which provides better lookup time than O(n) and doesn’t allow significant memory overheads because of the big given amount of data to be stored but leaving acceptable level of data locality[ref\_locality] for reducing of CPU overheads. In particular, this research is focused on the data structures and algorithms with search time equal or less than O(logN) because they significantly reduce the amount of reads from the memory. In our case, using an ideal algorithm with O(logN) search time on 65.5M gives around 26 memory reads. This number is quite theoretical because in the real word CPU needs a number of addition memory reads. The performance of this approach has been got and described in further chapters. Another group of algorithms and data structures is that one which employs hash-based technics. In theory, it could provide with O(1) searching speed. Number 1 here doesn’t mean exactly 1 but a constant number which represents the constant time of getting the requested value. This value can vary greatly depending on algorithm used and its parameters.

**NAT Translation Record Structure**

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.

**NAT lookup data structures and algorithm choosing**

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.

**Tree-based NAT table**

The simplest data algorithm and data structure with O(logN) search time is a simple binary tree. (ADD SOME EXPLANATION ABOUT WHAT THE BINARY TREE IS AND HOW IT WORKS – IF NEEDED)

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 first test was made using two tree-based NAT table. The test results are shown in Figure 3.

Figure 3. NAT performance. Binary tree based NAT table

This approach has reviled poor performance. With NAT table size of 100 000 entries it spends more than 3.7M cycles/pkt which is ridiculously big number having in mind a target of 436 cycles/pkt. Most likely reasons of the poor performance are unbalanced tree structure and CPU massive cache misses.

Simple binary tree is known with its drawback of data sequence storing dependency. In the worst case, when the inserting values set sorted, the binary tree turns into a linked list. In this case the search time becomes to be equal to O(n) the same as linear search has. Although, this scenario is highly unlikely to happen in the real world it shows the main problem of the simple binary tree: it could be highly unbalanced which means that the depth of one branch can differ of the depth of the other branch in the order of magnitude. Thus, the real search time is somewhere between O(n) and O(logN) depending on data sequence.

Another drawback of binary tree is low spatial locality[ref\_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 in different time frames are necessary. Because these allocated chunks of memory could be 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 CPU with a big cache, 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.

To solve the last issue there another test was performed. Despite, it is impossible to guess the network to be processed with NAT in prior, the NAT output data range (IPs and ports) is always known because it is the main resource that NAT uses for packet translation. Hence, as a data structure for backwards NAT translation a one-dimensional array of structures can be used. To access its data the hash-based approach can be used: the destination IP and port can be used to make a key and this key is used to access the data in the array (hash table with open addressing). IP could be thought as a segment number and port could be thought as an offset inside the segment. Having used this simple technic, we can, first, get rid of the second binary tree saving some memory, second, make backwards packet translation cheap because of O(1) searching time. The result of the experiment is shown in Figure 4.

Figure 4. Nat Performance. Binary tree-based NAT table for outgoing packets and 1D array for incoming packets. Comparison of basic and red-black binary trees.

The improvements made have given a remarkable performance increasing about 5700% on 100 000 nodes data set. From the chart it is clearly seen that although this result is quite good but has two serious flaws: The first one is that the performance is still below the desired level of 436 cycles and to fulfill that requirement it is necessary to speed up the lookup in 3.5 times. The second one is that the tree isn’t balanced which means that the search time for different nodes takes different time. To fix that flaw any balanced-tree data structure can be used, for example, a well-known and widely spread red-black tree[ref\_cormen]. The figure 4 shows the results of using red-black tree instead of regular binary tree. The performance improvement that gives red-black tree data structure is around 13% and still 3 times below the desired value.

The results of using tree-based data structures revealed the fact that these 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.

**Hash-based NAT table**

The hash table is a data structure which provides the constant time O(1) of getting the value using a key. The key is produced with by a hash function. The hash function has a deterministic value for a given input.

Figure 5. Nat Performance. Hash-based NAT table for outgoing packets and 1D array for incoming packets.

The biggest issue while using a hash table is collisions 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. It provides with good performance allowing to use simple (i.e. computationally cheap) hash functions.

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. As in case of the binary tree exploration, the hash table is used for outgoing packets. For incoming packets the same 1-dimensional array is used. 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 2, 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 value the initial size of the hash table varies: than bigger the devisor that bigger its initial size. How to adjust these values properly is the question out of the scope of this document. The short estimations gave the value of 2^23 for the devisor which means the initial size of the hash table is around 67 Mb and the table maximum size is around 2.5 GB. In these settings the maximum list size is 512 elements and the translation data can be found in around 500 cycles (see Figure 2.). In the Figure 5 the value of 567 cycles is shown at 8M connections, which is very close to the numbers estimated.

This performance value is very close to target one, 37% more that needed, but still doesn’t achieve the desired value.

Further speed up can be gotten using parallelization of the translation process.

**Hash-based parallel NAT table**

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.

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.

The approach described is simulated in the test system using ***pthread*** library. Each 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 NAT with fully independent data structures (i.e NAT tables) which eliminates all the drawbacks described in the previous paragraph but leads to memory overheads. As performance is the priority of this research the memory overhead issues were postponed to the future work.

Figure 6. Nat Performance. Parallel hash-based NAT table for outgoing packets and 1D array for incoming packets. Results for different number of simultaneously working cores.

In figure 6 the results is shown for different number of cores. The experiments were set for 4 threads maximum because the testing system has 4 cores processor 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 performance boost.

The result shows that in Figure 6 shows that the parallel approach gives the desired results: than more cores are available than grater performance improvement is. Adding of an additional core gives around 30% of performance boost.

The experiment wasn’t very clear because of using the hyper-threaded processor which in fact has only 2 physical cores while OS sees 4. A processor with fully-functional 4 physical cores can give better performance improvement.

**SUMMARY**

Using of parallel hash-tables gives the result which is greater than the target one. 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, for example cuckoo hashing, but this is the matter of further research.

**REFERENCES**

[ref\_TT\_ROS\_TEL] file:TT CGNAT 2014\_26\_06v1.doc

[ref\_RDP.RU] <http://rdp.ru/>

[ref\_rfc4787] Network Address Translation (NAT) Behavioral Requirements for Unicast UDP <https://tools.ietf.org/html/rfc4787#page-5>

[ref\_rfc791] Internet protocol <https://www.ietf.org/rfc/rfc791.txt>

[ref\_rfc793] Transmission Control Protocol <https://www.ietf.org/rfc/rfc793.txt>

[ref\_rfc768] User Datagram Protocol <https://www.ietf.org/rfc/rfc768.txt>

[ref\_cormen] Introduction to algorithms, By [Thomas H. Cormen](http://mitpress.mit.edu/authors/thomas-h-cormen), [Charles E. Leiserson](http://mitpress.mit.edu/authors/charles-e-leiserson), [Ronald L. Rivest](http://mitpress.mit.edu/authors/ronald-l-rivest) and [Clifford Stein](http://mitpress.mit.edu/authors/clifford-stein) ISBN: 978026203384

[ref\_locality] http://en.wikipedia.org/wiki/Locality\_of\_reference**e**