BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

NAT64 PERFORMANCE EVALUATION

TEST VÝKONNOSTI NAT64

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **JAN POKORNÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. MATĚJ GRÉGR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2019

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2018/2019

Master's Thesis Specification

21826

Student: **Pokorný Jan, Bc.**
Programme: Information Technology Field of study: Computer Networks and Communication
Title: **NAT64 Performance Evaluation**
Category: Networking
Assignment:

1. Study IPv6 transition mechanism - NAT64.
2. Get familiar with current NAT64 implementations (e.g., tayga, jool, cisco, etc.).
3. Compare NAT64 implementations with regards of supported features and networking performance.
4. Integrate selected NAT64 mechanism to NetX router platform used on BUT core network.
5. Analyse performance bottlenecks of the solution and propose an improvement to increase the performance.

Recommended literature:

- Bao, C., Huitema, C., Bagnulo, M., Boucadair, M., and X. Li, "IPv6 Addressing of IPv4/IPv6 Translators", RFC 6052, DOI 10.17487/RFC6052, October 2010
- Bagnulo, M., Matthews, P., and I. van Beijnum, "Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers", RFC 6146, DOI 10.17487/RFC6146, April 2011
- Chen, G., Cao, Z., Xie, C., and D. Binet, "NAT64 Deployment Options and Experience", RFC 7269, DOI 10.17487/RFC7269, June 2014

Requirements for the semestral defence:

- Items 1 to 3.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor: **Grégr Matěj, Ing., Ph.D.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2018
Submission deadline: May 22, 2019
Approval date: October 31, 2018

Abstract

This thesis is focused on the challenges of the transition between IP protocol version 4 and IP protocol version 6. The transition can be solved by many transition mechanisms and this thesis thoroughly describe Stateful NAT64 transition mechanism. The thesis aims to test various implementations of NAT64 and find a suitable implementation for NETX router. The goal is to find an implementation that would achieve traffic throughput of about 10 Gbps.

Several NAT64 implementations were evaluated in a testbed environment. Iperf and PF_Ring tools were used for throughput examination. Several different network traffic types have been measured to show the performance impact of each of the tested implementations.

The results showed that the most suitable implementation of NAT64 is Jool. Jool reached the required throughput, its development is still active and offers other advanced features, thus Jool was integrated into the NETX router. A command line extension for manipulating Jool instance was designed, implemented and integrated to NETX command line. Additionally, a package distribution process was developed through the RPM package system to fit the NETX build system.

The thesis outcome is full support of NAT64 transition mechanism in NETX platform achieving close to 10 Gbps.

Abstrakt

Tato práce se zabývá problematikou přechodu mezi IP protokolem verze 4 a IP protokolem verze 6. Přechod je možné řešit více mechanismy a tato práce je zaměřená na přechodový mechanismus Stateful NAT64. Cílem práce je otestovat různé implementace NAT64 a najít vhodnou implementaci pro router NETX. Za cíl bylo stanoveno najít implementaci, která bude dosahovat propustnosti 10 Gbps.

Několik NAT64 implementací bylo zkoumáno v testovacím prostředí. Měření probíhalo pomocí nástrojů Iperf a PF_Ring. Bylo změřeno několik různých druhů síťového provozu tak, aby bylo z výsledku patrné, jaký výkonnostní dopad má každá z testovaných implementací.

Z naměřených výsledků Jool vyšlo jako nejvhodnější NAT64 řešení. Jool splnil požadovanou propustnost a zároveň kromě stále aktivního vývoje nabízí i další pokročilé vlastnosti. Jool byl integrován do routeru NETX. Byla navrhnuta struktura příkazové řádky pro manipulaci s Jool instancí, která byla posléze implementována jako rozšíření NETX příkazové řádky. Dále byl vytvořen postup distribuce potřebných balíčku skrze balíčkovací systém RPM, tak aby zapadl do automatizovaného systému platformy NETX.

Výsledkem práce je plná podpora přechodového mechanismu NAT64 na platformě NETX dosahující propustnosti blízké 10 Gbps.

Keywords

IPv6, NAT64, transition mechanisms, Jool, Tayga, NETX, throughput evaluation, 10 Gbps

Klíčová slova

IPv6, NAT64, přechodové mechanismy, Jool, Tayga, NETX, měření propustnosti, 10 Gbps

Reference

POKORNÝ, Jan. *NAT64 Performance Evaluation.* Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Matěj Grégr, Ph.D.

NAT64 Performance Evaluation

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Mr. Ing. Matěj Grégr, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Jan Pokorný
May 18, 2019

Acknowledgements

I want to thank my supervisor Mr. Ing. Matěj Grégr, Ph.D. for the professional help and care he has provided me. Thanks also to Ing. Alberto Leiva Popper from Jool team for cooperation. Last but not least, I thank FIT VUTBR for the resources for my work.

Contents

Chapter 1

Introduction

This thesis deals with the Stateful NAT64 transition mechanism, which acts as a gateway between the old IP protocol version 4 (IPv4) and the new IP protocol version 6 (IPv6).

IPv6 solves some IPv4 scalability problems (insufficient address space), but at the cost of backward compatibility. This creates a new challenge to migrate the traditional widespread IPv4 network to the new protocol. The flag day[1] option turned out to be unrealistic, so it was necessary to figure out how to make the transition gradually. In principle, there are two solutions: Run both protocols simultaneously or run only one and offer the second protocol as a service on top of the first one. In my thesis, I deal with the second variant, when the network is running IPv6, and network hosts can still reach IPv4 services in a fashion that everything works transparently and the hosts have not recognized the difference. One mechanism that meets these requirements is called Stateful NAT64, and you can read more about it in chapter 2.

Stateful NAT64 is a software that translates packets from IPv6 to IPv4 and vice versa. NAT64 can be part of a network device, or it is possible to run one of the open-source implementations. In my thesis, I deal with open-source NAT64 implementations, that are described in chapter 3. At the same time, I conducted a small survey of NAT64 support by well-known network router vendors.

Chapter 4 describes measurement methodology. The goal is to measure open-source implementations of NAT64, examine their throughput, and find their advantages and weaknesses. This work tries to find a solution, that can achieve 10 Gbps throughput.

In chapter 5, based on the measured results, a NAT64 implementation will be selected and integrated into the NETX router. The NETX is a special Linux distribution, that is being developed at BUT. This Linux distribution runs on specialized hardware and servers like a router in a network. The goal is to find appropriate NAT64 solutions for NETX router and make NETX able to serve as the NAT64 gateway in IPv6 only networks.

Chapter 6 deals with optimizing the chosen solution. There were several bottlenecks found in an analysis. There is also a description of how these bottlenecks have been optimized and how they impacted the overall performance of the solution.

Chapter 7 summaries and discuss future work.

[1] Flag day – A global shutdown of one protocol and turning on the other at a specific moment.

Chapter 2

Transition mechanisms and NAT64

If a network administrator wants to deploy IPv6 in its network while sustaining the availability of IPv4 only services, there are following options [24].

- **Dual-Stack** — Run both protocols independently of each other in parallel.
- **Translating** — Run only one protocol and **translate packets** from the other one.
- **Tunneling** — Run only one protocol and **encapsulate packets** from the other one.

Dual-Stack

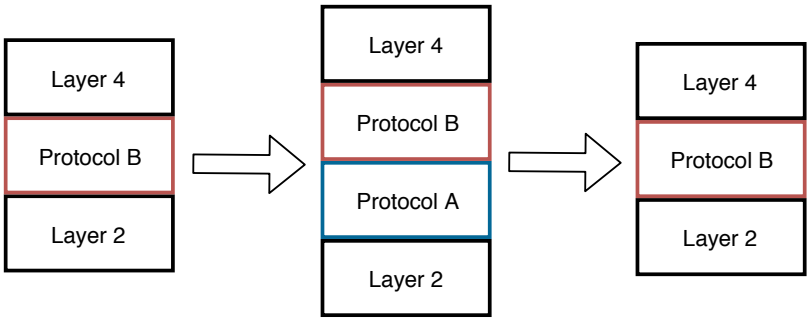
With Dual-Stack or Dual IP Layer Operation, we create two independent logical networks on the top of one physical topology. Each node in the network then can natively communicate on both protocols based on its preference. This duality has an advantage that, for example, if a problem with IPv6 network occurs, then the problem may not affect the IPv4 network. Thanks to this, IPv6 deployment can be done in full operation without impacting network performance. The major drawback of this solution is management, maintenance, and troubleshooting. Routing protocol instances, firewall rule bases, layer 2 security features, and others often require individual management. The maintenance and troubleshooting need to be done twice. For example, if there is an issue with the network, one of the first questions is whether the issue is IPv4, IPv6 or both related.

The Dual-Stack solution brings a full-fledged native experience from both protocols, without the need for compromise, but at the expense of difficult administration. Thanks to these features, the Dual-Stack is suitable for example in data centers, where high availability is needed, and it is the recommended approach by IETF RFC 6180 [11].

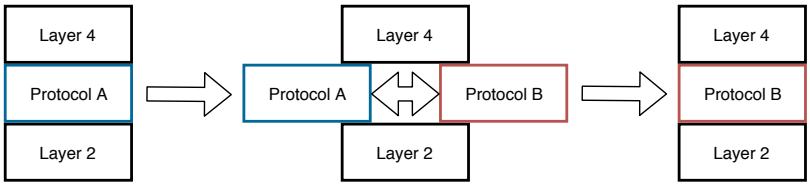
Address family transition

In this setup, we use just one address family in the core network, but we also offer a gateway (AFTR[1]), where packets from the other protocol are passed to their native network and vice versa. The other protocol's packet can be encapsulated into the core network protocol packets (Figure 2.1a) and then the AFTR de-encapsulates these packets and passes them to their native network, or the packets can be in the core network protocol format, and the AFTR translates them to the other protocol format (Figure 2.1b). The question is whether to run the core network on IPv4 or IPv6.

[1] AFTR – Address Family Transition Router. In the context of this work, the term AFTR is used for both tunneling and translating gateways.



(a) Protocol encapsulation



(b) Protocol translation

Figure 2.1: Difference between packet encapsulation and translation. Protocol A is the core network protocol.

If we already have an existing IPv4 network and we want to offer IPv6 features to network hosts, running the IPv4 protocol as the core protocol seems to be an option. In practice, it is easy to put it into operation, and there exist many mechanisms for this method like *6rd*, *6in4* or others, see RFC 7059 [26]. However, a problem with the lack of IPv4 addresses and the need for NAT in the network still persist. Also, if IPv6 adoption will increase in the future, the importance and burden of AFTR will increase. For these reasons, running the core IPv4 network and AFTR for IPv6 is not a long-lasting solution.

In contrast, IPv6 core network will solve the problem with the lack of IPv4 addresses. With the assumption that IPv4 traffic will dwindle in the future until it reaches the point where AFTR and NAT will not be needed anymore.

Compared to Dual-Stack, running only one protocol in the core network is easier to manage. On the other hand, it is always necessary to make several compromises with the protocol that is provided as a service. We can not afford these compromises, for example, in data center networks with an emphasis on high availability, but for internet service providers, affected by lack of IPv4 addresses, this could be a good choice for future growth.

2.1 Stateful NAT64

Stateful NAT64 belongs to the transition mechanisms that translates IPv6 packets to IPv4 and vice versa.

The IPv6 only network and the NAT64 transition mechanism are especially attractive for Internet service providers. It should be attractive for providers that are hitting their limits due to lack of IPv4 addresses. This solution is very similar to carrier-grade NATs. Network with carrier-grade NAT is a situation where providers are unable to allocate a single public IPv4 address to each customer, so the address translation is not performed at the customer's network boundary, but instead, a large central NAT is done at the provider's

network boundary. Central NAT becomes a critical point of the provider's network and a single point of failure. The end-to-end principle is lost for good.

In this case, NAT64 would replace the carrier-grade NAT with the expectation that, in the case of greater adoption of IPv6, the importance of NAT64 would be subdued. Compared to carrier-grade NAT, Stateful NAT64 should be less resource intensive by the traffic that could be already made over protocol IPv6.

The word NAT64 may be interpreted in several ways. In my work, unless otherwise stated, I am concerned with the Stateful NAT64 variant as defined in RFC 6146 [23].

2.1.1 Mapping IPv4 address in IPv6 address

Before we describe NAT64 in detail, the basics of mapping IPv4 to IPv6 address needs to be clarified.

One of the benefits of IPv6 protocol is its huge address space. The address space is so much larger than the IPv4 address space that we can take an IPv6 address prefix and make it encode every single IPv4 address.

The IPv4 address is 32 bits long, and IPv6 address is 128 bits long. It is possible to concatenate a 96 bit IPv6 prefix with an IPv4 address and make an IPv6 address, that maps an IPv4 address see listing 2.1.

```
+--+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|PL| 0-------------32--40--48--56--64--72--80--88--96--104---------|
+--+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|          v6(96) prefix                              |    v4(32)     |
+--+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|          64:ff9b::                                  |   192.0.2.1   |
+--+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Listing 2.1: A diagram indicating IPv4 address mapping in IPv6

64:ff9b::/96 + 192.0.2.1 = 64:ff9b::192.0.2.1[2] = 64:ff9b::c000:201

This approach is not the only one how to map an IPv4 address in IPv6 address, but it is the one that is used with the NAT64. More different approaches and information about mapping IPv4 address in IPv6 address can be found in RFC 6052 [33]. The importance of these addresses will be explained in the following section 2.1.2.

The IPv6 prefix can be allocated from address space of an internal network – Network-Specific prefix or Well-Known Prefix 64:ff9b/96 can be chosen [33].

Choosing the Well-Known Prefix has some benefits. It is a simple solution that does not require concessions in an address plan. Configurations are also simplified as NAT64 components already have this prefix superseded. You can use public DNS64 servers that use Well-know prefix, for example, Google DNS64 server [7]. The role of DNS64 will be described in the following section 2.1.3. There are also some restrictions with the Well-Known Prefix as well described in RFC 6052 [33], but they are not fundamental to this work.

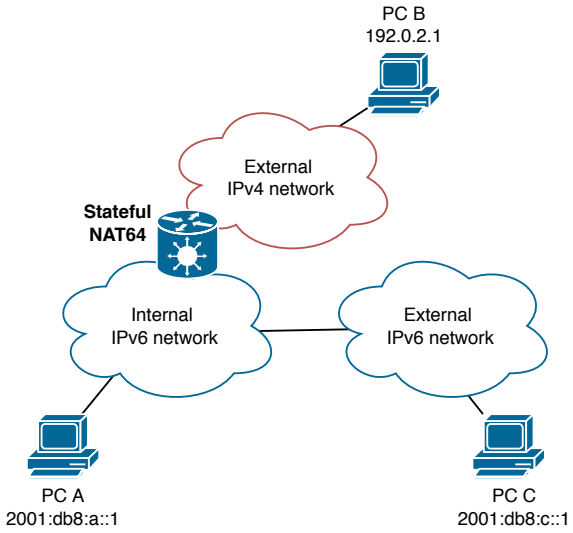


Figure 2.2: Network topology with Stateful NAT64. Communication between host A and C is native over protocol IPv6. Packets between host A and B are translated on NAT64 gateway.

2.1.2 Basic principles of Stateful NAT64

Figure 2.2 shows a simplified diagram of a network with NAT64 gateway. Host A is connected to an internal IPv6 only network. The internal network has connectivity to the external network, for example, the internet. The external network is logically divided into two parts. First part is a network with IPv6 connectivity and the second part is a network without IPv6 connectivity – IPv4 only network. In the external network there two hosts. Host B is connected to IPv4 only network, host C has IPv6 connectivity.

When host A wants to communicate with host C, the communication is native over protocol IPv6.

When host A wants to send a packet to host B, it misses a destination IPv6 address. Host B has no IPv6 connectivity, therefore no IPv6 address. In this case, host B IPv6 connectivity is simulated, and IPv6 address for host B is crafted. The host B IPv4 address 192.0.2.1 is taken and is appended to the Well-know NAT64 prefix 64:ff9b::/96. The result is 64:ff9b::192.0.2.1 The packet with this destination address is routed via an internal routing protocol to the NAT64 gateway. NAT64 gateway extracts the destination IPv4 address from the crafted destination IPv6 address. A destination port on a higher layer is preserved. The source address and port are chosen in the same way as a regular IPv4 NAT does. Finally, the translated IPv4 packet is forwarded to the external network. See figure 2.3.

[2]Please note the 64:ff9b::192.0.2.1 syntax. It is a simplified syntax, that makes the address more readable and easy to construct. The IPv4 part of the address will be expanded to the hexadecimal form.

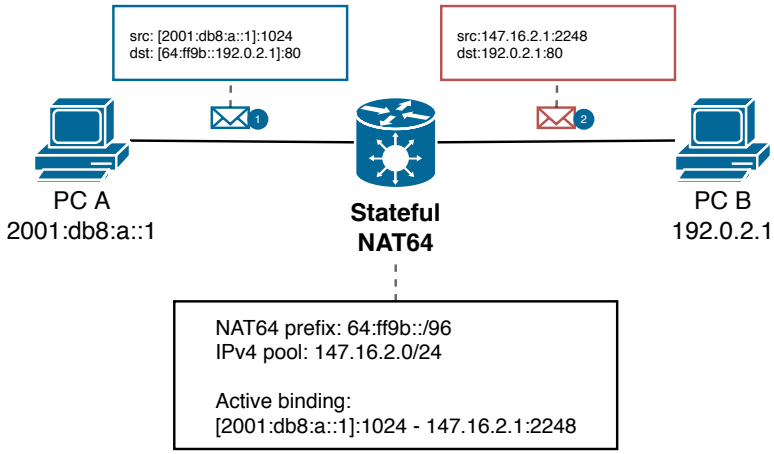


Figure 2.3: Packet passage through the NAT64 gateway. IPv6 packet from PC A is translated by NAT64 before is forwarded to PC B. A new connection session on NAT64 is created.

When host B responds to host A, an IPv4 packet is forwarded to the NAT64 gateway. The gateway searches for a session in the state table that matches incoming packets address and port. If the record is found, the destination IPv6 address and destination port are taken from the record. Source IPv6 address is crafted again from the NAT64 prefix and source IPv4 address. Translated IPv6 packet is forwarded to host B.

NAT64, as defined in RFC 6146 [23], can translate packets, that carries TCP, UDP and ICMP protocols. The above-described translation process has been simplified and did not neglect, for example, work with fragmentation and translation of ICMP messages. Further information about translating packets can be found in RFC 6145 [32].

2.1.3 Role of DNS64 or 464XLAT

In the NAT64 description above, we assumed that the computer automatically creates an IPv6 packet with a transformed IPv4 address when communicates with an IPv4 only network. However, this behavior is not self-acting and needs to be achieved. Two scenarios can occur. The end host network is aware of the existence of the NAT64 gateway and sends the IPv4 - IPv6 translated packets directly towards the NAT64 gateway, or the end host network is not aware of the existence of the NAT64 gateway, but the network pretends, that all services are reachable over IPv6.

Let's start with the first variant. We need a component that accepts an IPv4 packet and translates it into an IPv6 packet with a synthesized IPv4 in IPv6 address. Then the packet can be sent via the IPv6 only internal network towards the NAT64 gateway. Back then a component that receives the translated packet from the NAT64 gateway and translates it back to the IPv4 packet. The component is called CLAT, and it is a part of a technique called 464XLAT [17]. 464XLAT combines Stateful NAT64 and a component that could be called NAT46. In 464XLAT terminology, it is a PLAT and a CLAT. The operating principle is shown in Figure 2.4. The CLAT component could be part of an operating system of a device or could be integrated into SOHO[3] router of end users network.

[3]SOHO – Small office/home office

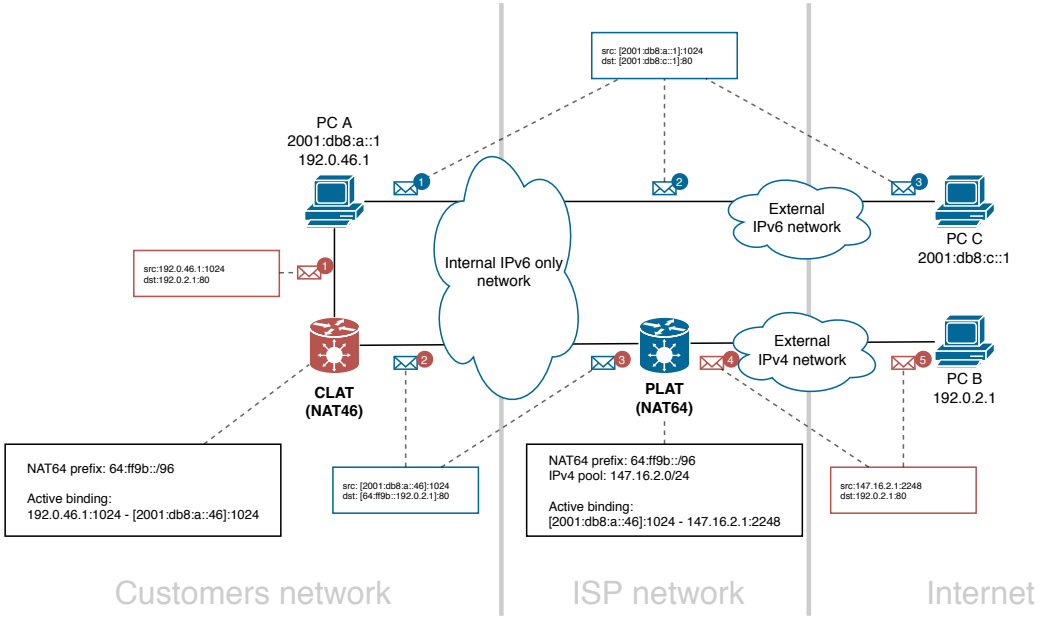


Figure 2.4: 464XLAT example network diagram. Host A and Host B communicate with each other natively over IPv6 protocol - blue packet numbers. Host B has no IPv6 connectivity. Host A crafts IPv4 packet for host B and forward it to CLAT. CLAT translates the IPv4 packet to IPv6 and forwards the packet to the PLAT through the **ISP (Internet Service Provider) IPv6 only** network. PLAT translates the packet back to IPv4 and forwards it to host C.

The second variant uses a DNS64 [22]. As the name implies, it is a recursive DNS[4] server with special functionality. If a domain name translation request arrives at the server, and the server finds the corresponding AAAA[5] record to the domain name, it sends the response to the querier in the same manner as the regular recursive DNS server does. If the domain name does not have a corresponding AAAA record, it is a service without IPv6 connectivity, and DNS64 synthesizes the answer. DNS64 resolves the corresponding A[6] record and creates AAAA from it by joining the pre-defined IPv6 prefix and the IPv4 address from the A record. It is the same method described in Section 2.1.1. For end host perspective, each domain name appears to have the corresponding AAAA record. If a domain query is preceded before each connection, the host works in the belief that each server is available via IPv6. This assumes the disadvantage that if the DNS query does not take place before work with IP address literals, the system fails. There may be problems with applications that have straight-encoded IPv4 literals instead of domain names. Another disadvantage is the manipulation with DNS records. Due to this manipulation, DNSSEC[7] is violated. In the way that a user delegates the DNSSEC check to the DNS64 server, the problem does not occur, but if the user wants to check the synthesized record itself, the corresponding DNSSEC signature will be missing, and the validation fails. On the other hand, trans-

[4]DNS – Domain Name System
[5]AAAA – IPv6 address record
[6]A – IPv4 address record
[7]DNSSEC –Domain Name System Security Extensions. Protection against malicious manipulation with DNS records based on cryptography and chains of trust [25].

parency is a big advantage. No extra configuration is required on the end host. The only condition is a proper DNS server (DNS64) address set up.

Both DNS64 and 464XLAT can be combined. In the Internet-Draft *Deployment Guidelines in Operator and Enterprise Networks* of J. Palet Martinez [16] the various combinations of these techniques and their advantages and disadvantages are well described.

Chapter 3

Available NAT64 implementations

In my work, I am looking for suitable NAT64 solution for integration into NETX router. Therefore, this chapter is primarily dedicated to existing NAT64 implementations. At the same time, I conducted a small survey of NAT64 support by well-known network equipment vendors.

3.1 Open-source implementations of Stateful NAT64

After research of existing open-source NAT64 solutions, I found four candidates – Tayga [15], Jool [12], WrapSix [31] and Ecdysis [29]. I eliminated the Ecdysis solution as it is a kernel-space implementation where kernel support is important, and this solution does not seem to be updated for several years, therefore without the support of the current kernels. WrapSix seemed to be a promising solution, but after several attempts, I did not get it into a working state. This left us with two solutions –Tayga and Jool both described in the following sections.

For each solution, there is a configuration example. The configuration is based on the topology that is shown in Figure 3.1. The same topology and configuration are used for testing in chapter 4.

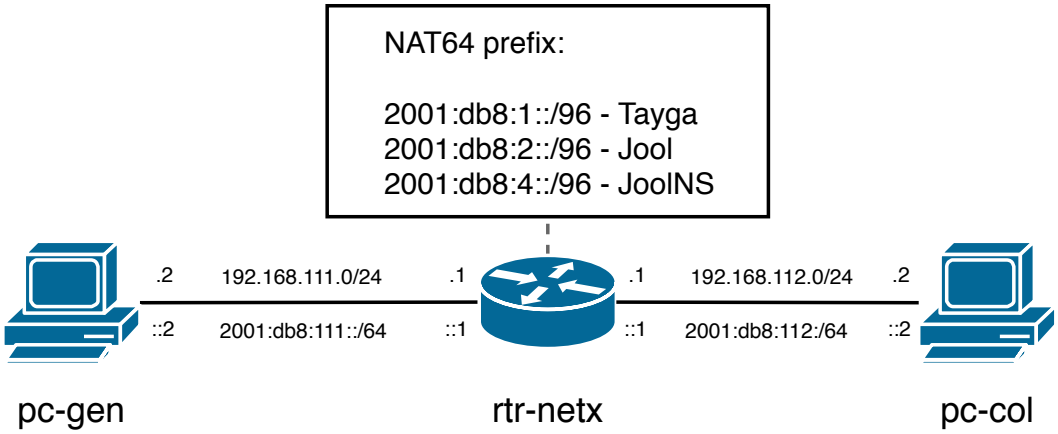


Figure 3.1: Used topology for NAT64 implementations testing.

3.1.1 Tayga

Tayga is an out-of-kernel stateless NAT64 implementation for Linux that uses TUN[1] driver to exchange IPv4 and IPv6 packets with the kernel [15].

The information from the official website shows that only one developer manages Tayga and the last release took place already a couple of years ago. Tayga claims to be fast, flexible, secure, and easy to install.

Most of these statements could be agreed because they stem from the nature of the implementation. Tayga runs in user-space and uses a TUN driver for the packet exchange. Thanks to this approach Tayga does not require its kernel module. Therefore the installation is easy, and further system updates will be without worry about kernel support. Tayga also does not demand root rights, which contributes to security. On the other hand, it can be assumed that implementation in the user space will not achieve the required performance.

Tayga is only a stateless NAT64 implementation, that means it does not overload IP addresses. For full stateful NAT64 behavior, Tayga needs to be used with common IPv4 NAT. Therefor Tayga is ordinarily used with iptables masquerade[2] – SNAT[3].

Installation

Source code can be downloaded from official website[4]. After unpacking the archive, source code needs to be compiled.

```
[root@rt-netx-e tayga-0.9.2] ./configure
[root@rt-netx-e tayga-0.9.2] make
[root@rt-netx-e tayga-0.9.2] make install
```

A directory for persistent storage should be created. Tayga stores NAT64 bindings in the storage and therefore translations persist potential restarts.

```
[root@rt-netx-e tayga-0.9.2] mkdir -p /var/db/tayga
```

Now a configuration file should be created.

```
[root@rt-netx-e tayga-0.9.2] cat /usr/local/etc/tayga.conf
tun-device nat64
ipv4-addr 192.168.112.3
prefix 2001:db8:1::/96
dynamic-pool 10.64.0.0/16
data-dir /var/db/tayga
```

`Tun-device` is a name of a tun device, that will be created below. The `ipv4-address` is an IPv4 address dedicated for Tayga. It must not be assigned to any interface. `Prefix` is the NAT64 prefix.

`Dynamic-pool` is a pool of IPv4 addresses. Tayga maps IPv6 source addresses to these address 1:1. The pool should be big enough to handle all possible request from the IPv6 network. Dynamic map records are valid for 124 minutes after the last matching packet is seen. This dynamic mapping is stored in the persistent storage.

[1] TUN – virtual network kernel interface
[2] Iptables masquerade is an IPv4 NAT implementation in Linux iptables module.
[3] SNAT – Source Network Address Translation
[4] http://www.litech.org/tayga/

`Data-dir` is the directory with persistent storage, that was created above.
Now it is time to create the TUN interface, and set it up.

```
[root@rt-netx-e tayga-0.9.2] tayga --mktun # Creates device named NAT64
[root@rt-netx-e tayga-0.9.2] ip link set nat64 up
```

Next step is to create proper routes. The NAT64 prefix and dynamic-pool have to be forwarded to the TUN device.

```
[root@rt-netx-e tayga-0.9.2] ip route add 10.64.0.0/16 dev nat64
[root@rt-netx-e tayga-0.9.2] ip route add 2001:db8:1::/96 dev nat64
```

When it is done, we can start the Tayga daemon and verify its functionality. We can try ping PC-COL at address 2001:db8:1::192.168.112.2 from PC-GEN. As we can see from the Taygas output, a new binding has been created. IPv6 address 2001:db8:111::3 is mapped to 10.64.205.46 and PC-COL is from PC-GEN reachable.

```
[root@PC-GEN ~] ping6 2001:db8:1::192.168.112.2
PING 2001:db8:1::192.168.112.2(2001:db8:1::c0a8:7002) 56 data bytes
64 bytes from 2001:db8:1::c0a8:7002: icmp_seq=65 ttl=61 time=0.353 ms
```

```
[root@rt-netx-e tayga-0.9.2] tayga -d
starting TAYGA 0.9.2
Using tun device nat64 with MTU 1500
TAYGA's IPv4 address: 192.168.112.3
TAYGA's IPv6 address: 2001:db8:1::c0a8:7003
NAT64 prefix: 2001:db8:1::/96
Dynamic pool: 10.64.0.0/16
assigned new pool address 10.64.205.46 (2001:db8:111::3)
```

3.1.2 Jool

Jool is another NAT64 implementation. It is a kernel space solution, that is being developed under the auspices of NIC MEXICO. Jool is well documented on his official website and development is still active. Jool is not just a Stateful NAT64, but it also supports many other RFCs associated with IPv6 – IPv4 translations.

Because Jool is a kernel space application, working with it is a bit more difficult than with Tayga. Jool consists of two applications a kernel module and a user-space application, that manages the kernel module. This makes it more difficult to install and update with the need to check compatibility with the used Linux kernel.

Jool is a Netfilter module, like for example iptables, and hooks into the *prerouting chain*. Because Netfilter is not comfortable with packets changing layer-3 protocols, Jool has its own forwarding pipeline, which only translating packets traverse [12]. The architecture is shown in Figure 3.2.

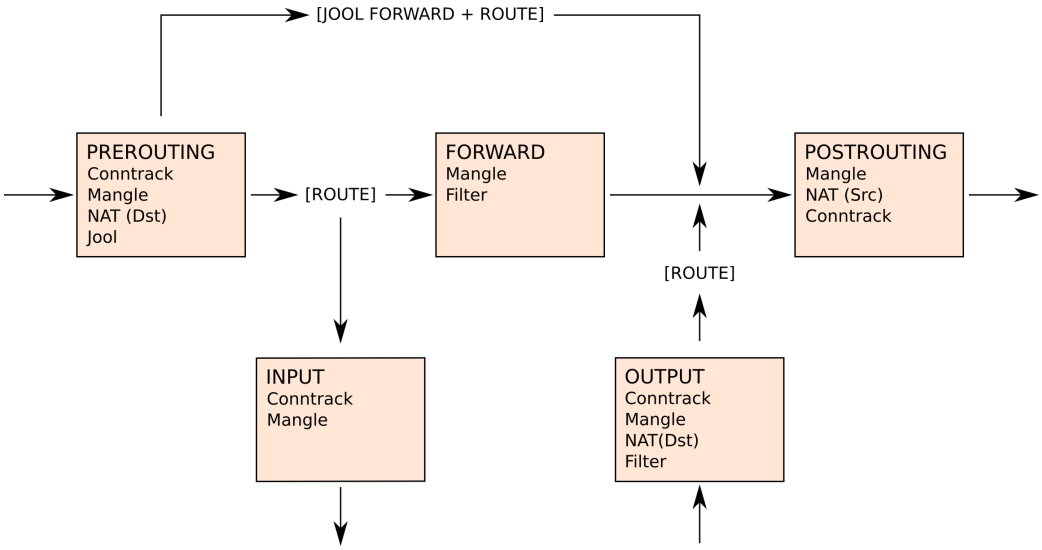


Figure 3.2: Jool Netfiler architecture. (taken from [12])

As you can see from Figure 3.2 with Jool architecture, Jool skips the forwarding chain. Skipping the forwarding chain violates the standard packet flow in Linux, resulting in some constraints. For example, if the router simultaneously filters traffic using rules in iptables, rules in the FORWARD chain will not apply to NAT64 traffic. In other words, Jool skips forward rules in iptables, and the NAT64 traffic cannot be filtered in the traditional manner. This unwanted feature can be circumvented by running Jool in its network namespace. The setup with a namespace will be described later.

Installation

Installation of Jool depends on a distribution of Linux and Jool version. The following installation is shown on Centos 7 (NetXOS release 7.5.1804 (Core)) and Jool v4.0.0.

After downloading the Jool, for this example from an official git repository, some dependencies need to be installed.

```
[root@rt-netx-e Jool] yum install automake kernel-devel kernel-headers
libnl3-devel iptables-devel dkms
```

The installation is separated for the kernel module and the user-space application. The userspace application is in `usr` directory and is build and installed via common build tools.

```
[root@rt-netx-e Jool] cd usr
[root@rt-netx-e usr] ./autogen.sh
[root@rt-netx-e usr] ./configure
[root@rt-netx-e usr] make
[root@rt-netx-e urs] make install
```

The kernel module could be installed via Kbuild or DKMS[5]. The DKMS is more user-friendly and straightforward and is officially recommended.

```
[root@rt-netx-e Jool] dkms install .
```

[5]DKMS – Dynamic Kernel Module Support. Linux framework/subsystem that manages kernel modules.

After successful installation, the kernel module can be inserted into the Linux kernel.

```
[root@rt-netx-e Jool] modprobe jool
```

When the kernel module is mode probed, a new Jool instance can be created with the userspace CLI[6] tool. A NAT64 prefix it is the only required parameter, that needs to be assigned to the instance.

```
[root@rt-netx-e usr] jool instance add --netfilter --pool6 2001:db8:2::/96
```

This is a minimal setup, and the connectivity from PC-GEN to PC-COL should be achieved.

```
[root@rt-netx-f ~] ping6 2001:db8:2::192.168.112.2
PING 2001:db8:2::192.168.112.2(2001:db8:2::c0a8:7002) 56 data bytes
64 bytes from 2001:db8:2::c0a8:7002: icmp_seq=1 ttl=63 time=0.192 ms
```

3.1.3 Translating in a virtual network namespace

A network namespace is a Linux feature that allows creating network virtualization. A network namespace is a set of network interfaces, routing table, Netfilter chains, and other related components. The default network set up of a Linux machine is also a namespace and is called a regular namespace. On a single Linux machine, many other network namespaces can be created. Network interfaces in newly created namespaces can be taken from the regular namespace, in other words, the physical interfaces from the machine can be assigned to a namespace. If we want to connect namespaces together, a special virtual pair interface is used. The virtual pair interface is a pair of two interfaces. One interface from the pair is assigned to one namespace and the second to another namespace, and these two interfaces creates a bridge.

To understand the following text, imagine network namespace as a connected virtual router as shown in Figure 3.3.

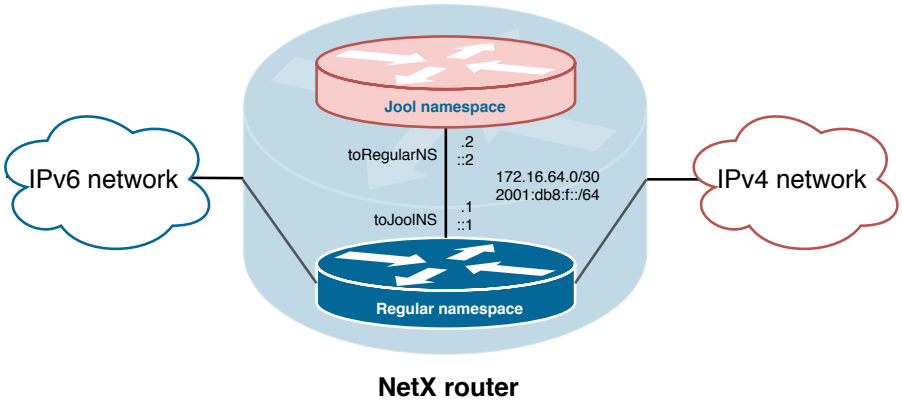


Figure 3.3: Visualization of an intra-router topology with virtual network namespace. Netx router is connected with its physical interfaces to the IPv4 and IPv6 network – regular namespace. Besides the regular namespace, a virtual namespace with a Jool instance exists – Jool namespace. The two namespaces are connected with a virtual pair interface. Subnet 172.16.64.0/30 and 2001:db8:f::/64 is used for subnetting as noted in the figure.

[6] CLI – Command line interface

This kind of virtualization can be used while working with Jool. Jool instance can be placed into a namespace. This concept will separate the address translation logic from the router's core, making it easier to manage. The solution is very similar to Tayga. In both solutions, there is a virtual interface that accepts a packet for translation a sends it translated back. The problem with NAT64 traffic filtering is solved because now we can apply filter rules when the packet is forwarded to the namespace and when it translated returns.

The configuration is shown below. First, we create a network namespace `joolNS` and pair interface `toJoolNS` and `toRegularNS`. The `toRegularNS` interface is added to the newly created namespace.

```
[root@rt-netx-e ~] ip netns add joolNS
[root@rt-netx-e ~] ip link add name toJoolNS type veth peer name toRegularNS
[root@rt-netx-e ~] ip link set dev toRegularNS netns joolNS
```

We set up the `toJoolNS` interface and add IP addresses.

```
[root@rt-netx-e ~] ip link set toJoolNS up
[root@rt-netx-e ~] ip addr add 172.16.64.1/30 dev toJoolNS
[root@rt-netx-e ~] ip addr add 2001:db8:f::1/64 dev toJoolNS
```

Then we switch to the `joolNS` and similarly set up the `toRegularNS` interface.

```
[root@rt-netx-e ~] ip netns exec joolNS bash
[root@rt-netx-e ~] ip link set toRegularNS up
[root@rt-netx-e ~] ip addr add 172.16.64.2/30 dev toRegularNS
[root@rt-netx-e ~] ip addr add 2001:db8:f::2/64 dev toRegularNS
```

Proper routes need to be created; we forward all traffic from the namespace via the `toRegularNS` interface.

```
[root@rt-netx-e ~] ip route add 0.0.0.0/0 via 172.16.64.1
[root@rt-netx-e ~] ip route add ::/0 via 2001:db8:f::1
```

When addressing and routing is done a Jool instance can be started and we can exit the `joolNS` namespace.

```
[root@rt-netx-e ~] jool instance add --netfilter --pool6 2001:db8:4::/96
[root@rt-netx-e ~] exit
```

Now, in the regular namespace, we add a route, that forward NAT64 traffic to the namespace. Then the connectivity should be secured as shows the ping result.

```
[root@rt-netx-e ~] ip route add 2001:db8:4::/96 via 2001:db8:f::2

[root@rt-netx-f ~] ping6 2001:db8:4::192.168.112.2
PING 2001:db8:4::192.168.112.2(2001:db8:4::c0a8:7002) 56 data bytes
64 bytes from 2001:db8:4::c0a8:7002: icmp_seq=1 ttl=61 time=0.313 ms
```

The packet flow is shown in Figure 3.4. If we compare traceroute between native Jool and namespaced Jool, we can see two extra hops, which corresponds to the packet transfer between the namespaces.

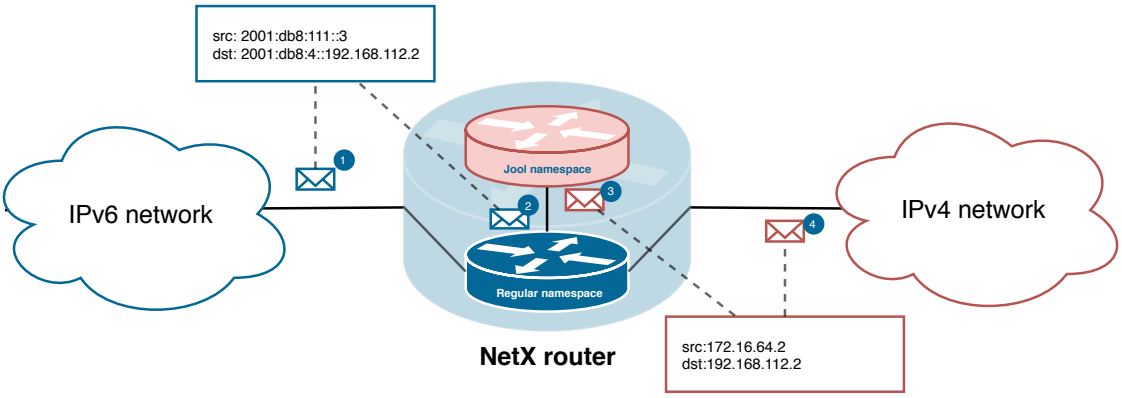


Figure 3.4: Packet passing in a topology with NAT64 translation inside a virtual network namespace. The blue IPv6 packet arrives at the NETX router. Regular network namespace forwards the packet to the Jool network namespace, where the translation is done. The translated red IPv4 packet is returned from the Jool network namespace and passed to the IPv4 network.

```
[root@rt-netx-f ~] traceroute6 2001:db8:2::192.168.112.2
traceroute to 2001:db8:2::192.168.112.2 (2001:db8:2::c0a8:7002),
 1 2001:db8:111::1 (2001:db8:111::1) 0.247 ms 0.230 ms 0.240 ms
 2 2001:db8:2::c0a8:7002 (2001:db8:2::c0a8:7002) 0.258 ms 0.260 ms 0.259 ms

[root@rt-netx-f ~] traceroute6 2001:db8:4::192.168.112.2
traceroute to 2001:db8:4::192.168.112.2 (2001:db8:4::c0a8:7002),
 1 2001:db8:111::1 (2001:db8:111::1) 0.254 ms 0.198 ms 0.208 ms
 2 2001:db8:f::2 (2001:db8:f::2) 0.412 ms 0.415 ms 0.407 ms
 3 2001:db8:4::c0a8:7002 (2001:db8:4::c0a8:7002) 0.451 ms 0.428 ms 0.427 ms
 4 2001:db8:4::c0a8:7002 (2001:db8:4::c0a8:7002) 0.453 ms 0.461 ms 0.418 ms
```

3.2 Stateful NAT64 support by well-known network router vendors

This section describes a list of well-known network router vendors and their NAT64 support. The list of manufacturers was created based on personal experience and references.

Direct Stateful NAT64 support was investigated. If NAT64 was not available, at least NAT-PT support was sought. NAT-PT is a deprecated predecessor of Stateful NAT64 [27]. Stateful NAT64 is a reduced NAT-PT algorithm, and therefore the core of Stateful NAT64 is included in NAT-PT.

For a device that I could personally try out, an example of configuration is given.

The information found is from commonly available manufacturers websites and is valid as of March 19, 2019.

Cisco

According to data from the Cisco feature navigator, Stateful NAT64 is supported on higher IRS series and ASR routers [4].

Cisco also supports NAT-PT. The NAT-PT support is through several router series with IOS 12.3 and higher. During my work, I had the opportunity to try a Cisco router 2911 with IOS 15.3. An example of NAT64 (NAT-PT) configuration can be seen in 3.1.

```
no ipv6 cef
ipv6 unicast-routing
!
interface GigabitEthernet0/0.111
 encapsulation dot1Q 111
 ipv6 address 2001:DB8:111::4/64
 ipv6 enable
 ipv6 nat
!
interface GigabitEthernet0/0.112
 encapsulation dot1Q 112
 ip address 192.168.112.4 255.255.255.0
 ipv6 enable
 ipv6 nat
!
!
ipv6 nat v6v4 source list NAT64_SRC_IPV6_ACL interface g0/0.112 overload
ipv6 nat prefix 64:ff9b::/96 v4-mapped NAT64_DST_IPV6_ACL
!
ipv6 access-list NAT64_SRC_IPV6_ACL
 permit ipv6 2001:DB8:111::/64 any
!
ipv6 access-list NAT64_DST_IPV6_ACL
 permit ipv6 any 64:ff9b::/96
```

Listing 3.1: Example of Cisco NAT64 (NAT-PT) configuration

Address translation is allowed for traffic with a source address in `2001:DB8:111::/64` subnet and destination address with `64:ff9b::/96` prefix. The translation is done on the `GigabitEthernet0/0.112` interface, with its IPv4 address – without a pool. IPv6 CEF[7] must be turned off for proper NAT64 functionality. The `ipv6 enable` and `ipv6 nat` commands must be specified at the participating interfaces.

This router was also included in the performance testing.

Juniper

Juniper supports Stateful NAT64 on SRX [13] and MX [14] series. During my work, I had the opportunity to try an SRX 240 Services Gateway. An example of NAT64 configuration can be seen in 3.2.

[7] Cisco Express Forwarding – optimizes network performance and scalability for networks with large and dynamic traffic patterns [3].

```
nat {
    source {
        rule-set trust-to-untrust {
            from zone trust;
            to zone untrust;
            rule nat64src {
                match {
                    source-address 2001:db8:0:0::/64;
                    destination-address 0.0.0.0/0;
                }
                then {
                    source-nat {
                        interface;
                    }
                }
            }
        }
    }
    static {
        rule-set nat64 {
            from zone trust;
            rule ipv6-clients1 {
                match {
                    destination-address 64:ff9b::/96;
                }
                then {
                    static-nat {
                        inet;
                    }
                }
            }
        }
    }
}
```

Listing 3.2: Example of Juniper SRX NAT64 configuration [34]

The translation takes place in two steps. In the first step, the static destination NAT is executed - the destination IPv4 address is derived from the destination IPv6 address. In the second step, the dynamic source NAT is executed. An IPv6 transport address is replaced by the outgoing interface IPv4 address and dynamic port.

Hewlett Packard Enterprise (HPE)

Unfortunately, I haven't found a reliable way to verify NAT64 support with HPE. A tool like a feature navigator was not found. I haven't encountered the NAT64 keyword when I search the company website and command references. However, commands for NAT-PT configuration have been found in the command reference for MSR series routers [8]. In any case, insufficient documentation was found.

Huawei

Huawei supports NAT64 on NE40E series routers [9].

Arista

Arista supports NAT64 on 7170 Series routers [2].

Extreme networks, Mikrotik

No support found.

Chapter 4

Measurement methodology and results

The goal of the measurement is to examine the performance of individual NAT64 solutions and find a solution that will be suitable for integration into the NETX router. Appropriate solutions must reach, first and foremost, sufficient traffic throughput. The target throughput was set at 10 Gbps for both TCP and UDP protocol flows at the maximum packet size (MTU 1500). Apart from the throughput itself, the performance impact of individual solutions will be measured. The chosen solution must also be stable and sustainable in the future.

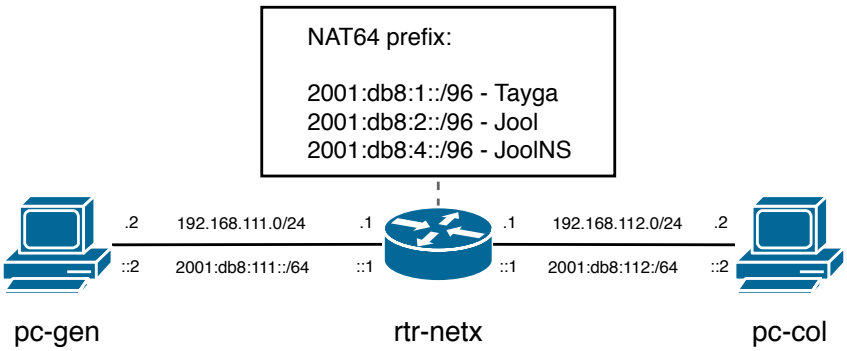
Measurements will be performed in a laboratory on the topology shown in Figure 4.1. We will use a NETX router as a NAT64 gateway. We will install and configure all chosen NAT64 implementations on the router. PC-GEN (generator) will generate packets to PC-COL (collector) via the NETX router. The hardware topology parameters are shown in Table 4.1.

Besides the throughput of each NAT64 solution, we will examine the impact on the network performance. Therefore, in addition to measuring the NAT64 traffic itself, we will measure even throughput of pure routing (without address translation) and routing with regular IPv4 NAT using iptables. At the same time, for all scenarios besides a throughput, we will measure the utilization of processor threads during the tests. From the chart of the utilization of individual threads and their average wages, we can compare the resource impact of individual traffic types.

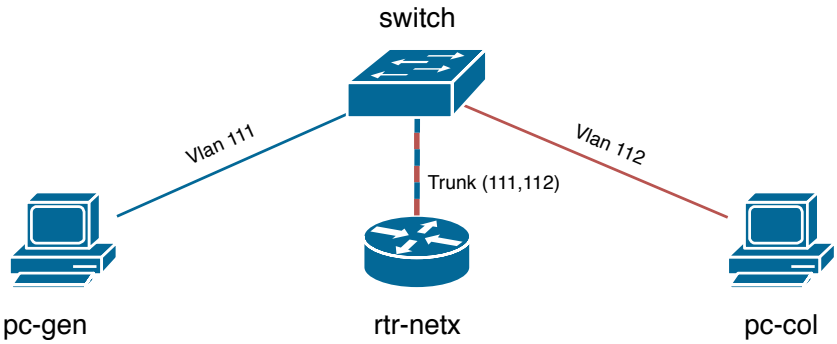
Originally the testing was planned using Iperf3. Since Iperf3 performance problems occurred at 10 Gbps link speed, Iperf was replaced by PF_RING and testing has focused mainly on UDP traffic.

PF_RING adds a new kind of network sockets to Linux kernel, making it able to work very efficiently and at much higher speeds [21]. PF_RING also offers applications (PF_COUNT and PF_SEND) that are built over the PF_RING API, and we will use them for our measurements. Similar to Iperf, PF_COUNT and PF_SEND is a client-server architecture. The PF_SEND utility can read a pcap file[1] and play it directly to the network card. Thanks to this straightforward approach, we can generate intrusive traffic with low CPU load. On the other side, PF_COUNT tool listens on a network card, calculates incoming packets, and generates a status report. For all tested scenarios, a

[1] Pcap is a file format that is capable of storing network traffic (packets). It is used, for example, in network communication capturing and its subsequent analysis.



(a) L3 topology



(b) L2 topology

Figure 4.1: L2 and L3 testbed topology.

rtr-netx	Intel(R) Xeon(R) CPU D-1537 @ 1.70GHz 4x 4GB DDR4 2133MHz Ethernet Connection X552 10 GbE SFP+ Linux 4.4.178-1.el7.netx.x86_64
PC-GEN	Intel(R) Xeon(R) CPU D-1587 @ 1.70GHz 4x 4GB DDR4 2133MHz Ethernet Controller XL710 for 40GbE QSFP Linux 3.10.0-693.17.1.el7.netx.x86_64 iperf 3.1.7
PC-COL	Intel(R) Core(TM) i3-4160 CPU @ 3.60GHz 2x 4GB DDR3 1600MHz RAM 2x Ethernet Controller X710 for 10GbE SFP+ Linux 3.10.0-327.36.2.el7.x86_64 iperf 3.1.31
SWITCH **PC-COL <=>rtr-netx** **PC-GEN <=>rtr-netx**	HPE 5900AF-48XG-4QSFP+ 10-Gig ethernet 40-Gig ethernet

Table 4.1: Testbed topology HW and SW parameters

long pcap file containing multiple UDP traffic flows was generated, and then played to the network card at the rate of 10 Gbps. On the other side, the number of successfully reached packets was count.

4.1 The measurement process and its challenges

The whole measuring process can be divided into three phases. In the first stage, a connection to the elements in the topology is made from a control station (personal laptop), and the necessary settings are made - running the NAT64 variant, executing the Iperf or PF_COUNT. In the second phase, traffic generation starts and the CPU threads load is being captured. In the final step, the results are collected, the configuration is cleared, and charts generating is done.

During the measurements, several problems had to be solved. At the start of the tests, it was found that the measured throughput did not reach the order of expected results. The reason was packet processing at stations. The packets that came from the network card queue were processed only in a single thread. It was because the scheduler hash function, which determined the distribution of packets from the network card queue, took into account only the source and destination addresses of the packet. However, in the chosen topology, all packets have the same source and destination addresses and therefore fall into the same thread. The solution was to switch the scheduler into a mode where it calculates the hash from the source and destination ports. At the same time, it was necessary to generate packets from/to multiple source ports. Changing the scheduler hashing function had to be done on all participating ethernet interfaces. The change had to be made for both IPv6 and IPv4 TCP and UDP, see listing 4.1.

```
[root@rt-netx-d ~]# ethtool -u tge3 rx-flow-hash udp6
UDP over IPV6 flows use these fields for computing Hash flow key:
IP SA
IP DA

[root@rt-netx-d ~]# ethtool -U tge3 rx-flow-hash udp6 sdfn
[root@rt-netx-d ~]# ethtool -u tge3 rx-flow-hash udp6
UDP over IPV6 flows use these fields for computing Hash flow key:
IP SA
IP DA
L4 bytes 0 & 1 [TCP/UDP src port]
L4 bytes 2 & 3 [TCP/UDP dst port]
```

Listing 4.1: Sample before and after application change of hash function for UDP IPv6 traffic.

Testing with Iperf was problematic. Using the Iperf tool, there was a problem with the performance of the measuring stations. Single Iperf3 instance is unable to use more than one CPU thread during traffic generation. Therefore, when examining 10 Gbps, the measuring stations were at the edge of their capabilities and the results are distorted. Iperf results were unstable with a large deviation. Therefore TCP results are tentative and for a better overview results from UDP are used.

Testing with PF_RING required small topological change. When tested using Iperf traffic flowed from VLAN 111 to VLAN 112 see L2 topology Figure 4.1b. However, when

working with PF_RING, we work on a lower layer and tagging ethernet frames no longer happen automatically. The way to guarantee frame tagging could not be found. Therefore it was necessary to make changes so that traffic could flow through native VLAN - untagged. Furthermore, it was necessary to use a zero-copy mode and set up a static mapping between IP and MAC addresses on the NAT64 router. While running PF_COUNT in zero-copy mode, the NIC[2] is fully occupied by PF_RING and other communication like ARP or ICMPv6 is stopped, and records from the ip neighbor table expired over time, which led to problems.

4.2 Evaluation of measured results

The evaluation is divided into two parts – TCP and UDP.

TCP

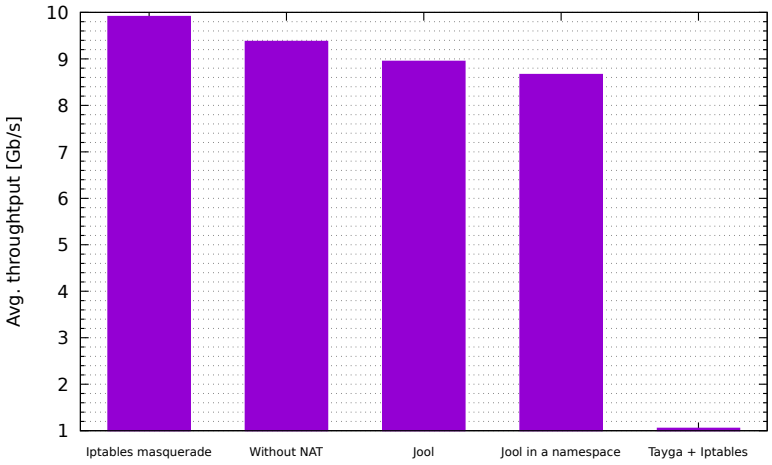


Figure 4.2: TCP throughput comparison

Figure 4.2 compares the average throughput of all measured variants and we can see that the testbed setup is capable of 10 Gbps throughput. It is also possible to observe that regular IPv4 NAT has a minimal performance impact and is paradoxical (due to measuring error) higher than pure routing. Tayga barely reaches 1 Gbps throughput. Jool attacks the 10 Gbps level and adding virtual network namespace has some performance impact.

However, as already mentioned, these results need to be taken as a proposal. More accurate data is provided by UDP measurements.

UDP

Entire range of packet sizes was tested in UDP tests. The packet size range is (64 B, 128 B, 512 B, 1024 B, 1024 B, 1280 B, 1500 B) and the size does not include *inter-frame gap* 12 B and *mac preambule* 8 B. Beside throughput in bps the number of packets (pps) that the router can forward were captured.

[2]NIC – Network interface controller

	Theoretical maximum		Pure Ipv4		Pure IPv6		NAT44		Tayga		Jool		Jool namespace		
64	14.88	10	2.6	1.4	2.3	1.2	2.4	1.3	0.000235	0.000122	0.7	0.3	0.35	0.2	
128	8.45	10	2.6	2.7	2.3	2.4	2.4	2.6	0.000235	0.000224	0.7	0.6	0.35	0.4	B
256	4.53	10	2.6	5.3	2.3	4.7	2.4	5.1	0.000240	0.000457	0.7	1.2	0.35	0.7	y
512	2.35	10	2.3	9.6	2.2	9.0	2.3	9.5	0.000250	0.000973	0.7	2.9	0.36	1.5	t
1024	1.20	10	1.2	9.8	1.1	9.8	1.2	9.8	0.051	0.41	0.7	5.9	0.36	2.9	e
1280	0.96	10	0.9	9.8	0.9	9.8	0.9	9.8	0.068	0.68	0.7	7.4	0.36	3.7	s
1500	0.82	10	0.8	9.8	0.8	9.8	0.8	9.8	0.076	0.91	0.7	8.6	0.41	4.9	
	Mpp/s	Gb/s	Mpp/s	Gb/s	Mpp/s	Gb/s	Mpp/s	Gb/s	Mpp/s	Gb/s	Mpp/s	Gb/s	Mpp/s	Gb/s	

Table 4.2: UDP traffic – measured by PF_RING. In the throughput in Gbps, the used tool does not include *inter-frame gap* and *mac preambule* into the calculation. Therefore, the maximum throughput is 9.8 Gbps instead of 10 Gbps.

$$bps = (packet\_size + 20) * pps * 8$$

The throughput results are in charts 4.3a, 4.3b and table 4.2. Chart 4.3a shows average throughput in gigabits per second depending on the packet size, and chart 4.3b shows average throughput in packets per second.

In addition to these values, processor load during the 1500 B packet size test was measured. With 1500 B packet size, the 10 Gbps line should already be saturated, and the CPU load should no longer be 100 %. Based on these values, it is possible to compare the effectiveness of individual solutions. Results are available in chart 4.4.

The results confirm the testbed ability to operate at 10-gigabit throughput and also confirm the minimal impact of using common IPv4 NAT. For pure routing and common IPv4 NAT, the theoretical maximum is reached with a packet length around 512 bytes.

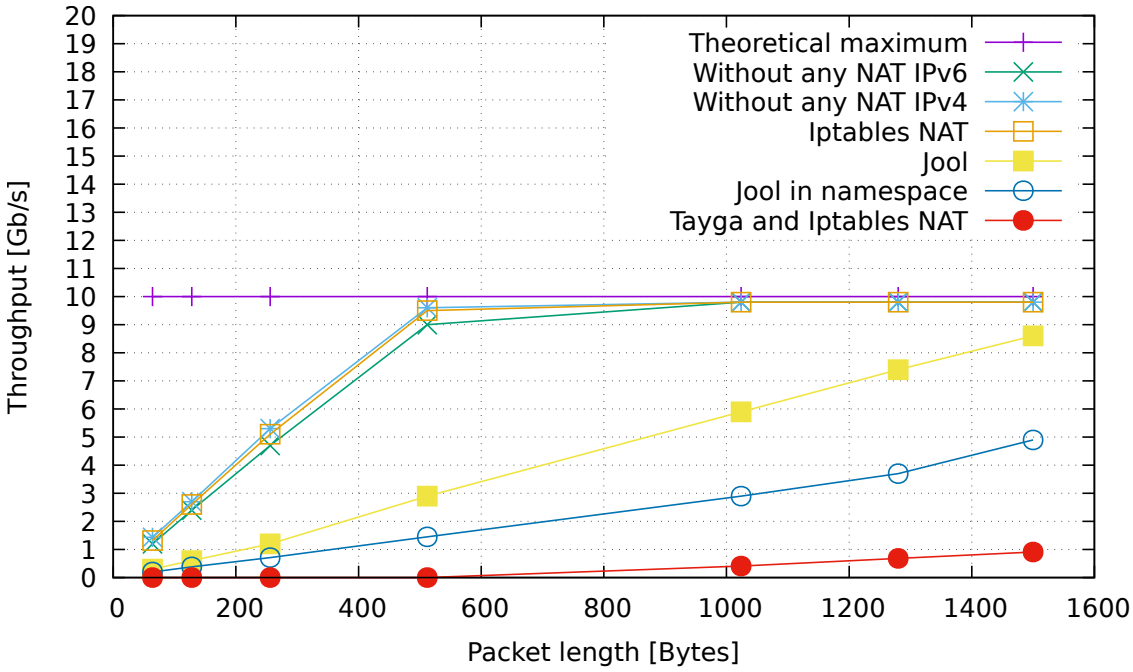
Tayga does not reach the desired throughput. In Tayga test, we may notice a dramatic drop in performance for packets smaller than 1024 B. Tayga is sensitive to congestion. Since the test generates traffic flow at the rate of 10 Gbps Tayga is overwhelmed, and its performance drops dramatically. The best result at 10 Gbps input rate was 70 kpps. 148 kpps was reached if the sending rate was reduced to Tayga's capability. However, even 148 kpps is a low result. The reason is apparent from the CPU load chart see 4.4f. Tayga works only in a single thread. There is no parallelism, and Tayga performance is limited by the performance of one core.

Jool is in the 700 kpps area, which means that 10-gigabit is not reached. Jool compared to common IPv4 NAT has a third of the performance. In a 1500 B packet test, the CPU load is still overloaded. The reasons and solutions for the high CPU usage are described in chapter 6.

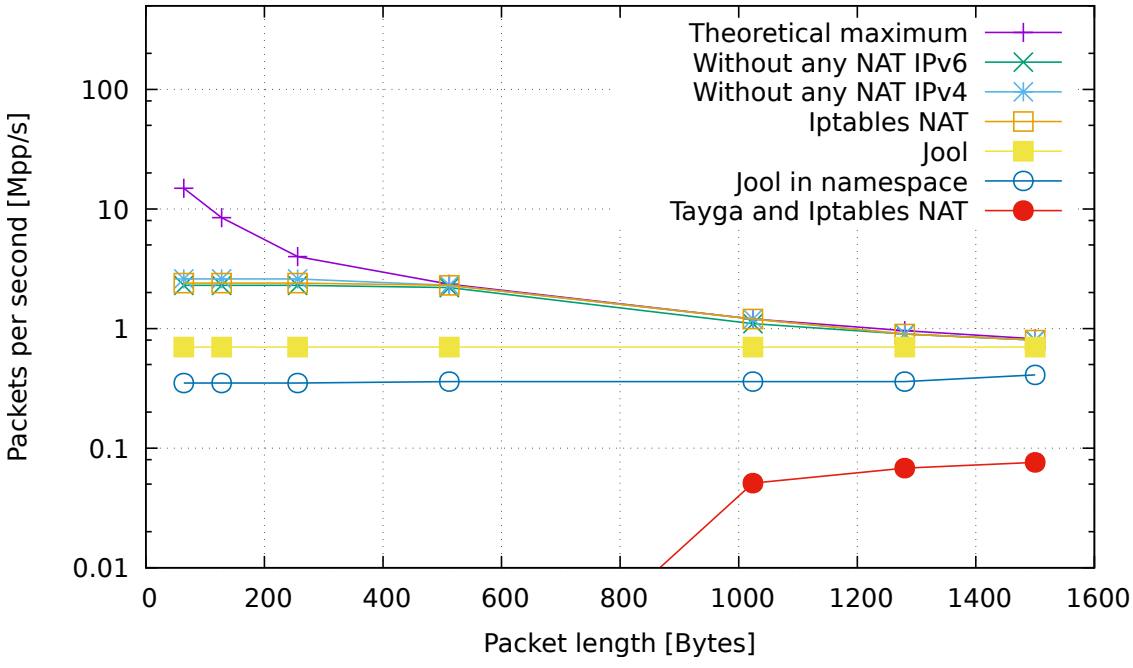
Using virtual network namespace reduce Jool performance by an additional 350 kpps to 350 kpps.

In the Appendix A, there is also a UDP test with Cisco router 2911. Cisco 2911 is only a gigabit router that is not suitable for comparing with the NETX router, so the results are listed here only as interest. In the results, you may notice that the Cisco router in NAT64 mode does not support CEF[3] and its NAT64 performance is a small fraction of pure routing.

[3]Cisco Express Forwarding – optimizes network performance and scalability for networks with large and dynamic traffic patterns [3].

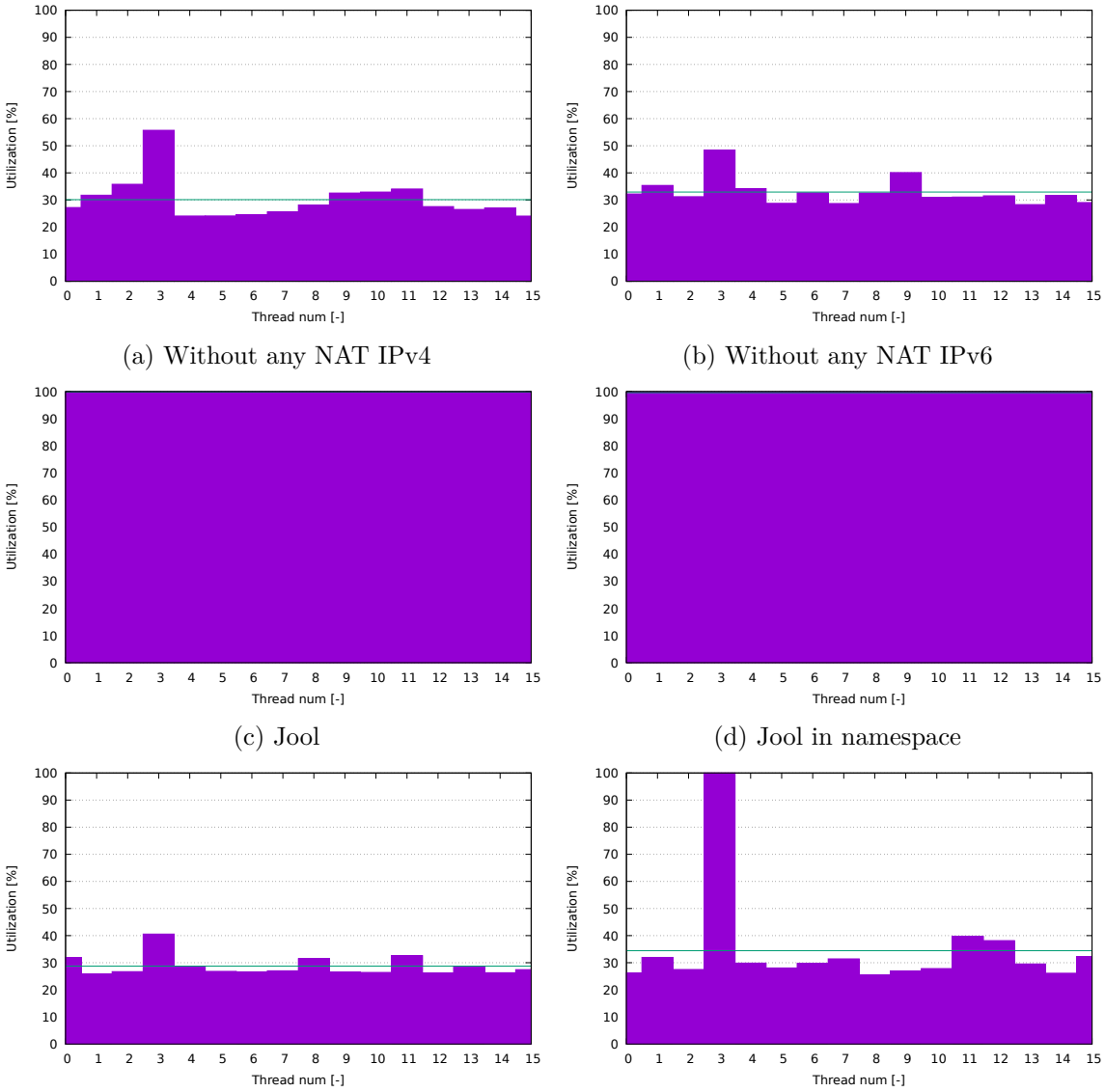


(a) Average throughput in Gbps



(b) Average throughput in Mpps

Figure 4.3: UDP traffic – measured by PF_RING



(a) Without any NAT IPv4

(b) Without any NAT IPv6

(c) Jool

(d) Jool in namespace

(e) Iptables NAT

(f) Tayga and Iptables NAT

Figure 4.4: CPU threads utilization during UDP test with packet size 1500 B

Chapter 5

Jool NAT64 integration into NETX router

NETX is a distribution based on enterprise Linux, that in combination with the appropriate hardware (both physical or virtual), forms a network router. NETX is a BUT university project. One of the goals of this work was to find and integrate a stateful NAT64 solution into the project.

Jool NAT64 in a virtual network namespace was selected for integration. Jool results were far from ideal. The required 10 Gbps throughput was not met. Still, Jool is the best possible solution, and with the optimization described in the following chapter 6 the performance is significantly better, and the 10 Gbps throughput is achieved.

Using the virtual network namespace option has several advantages. Especially:

- Allows NAT64 traffic filtering.
- Simplifies integration/administration through logical separation.
- Allows to generate/test NAT64 traffic directly from the router.
- Offers statistics from the interface between virtual and regular namespaces.
- Creates routes for redistribution.

Because of these advantages, it was a good idea to use virtual network namespace at the cost of overhead and lower pps. If maximum performance is needed and overhead of virtual network namespace would be too high, the classic setup method is still available through the shell.

Two of the key parts of the NETX distribution are a package repository and netc - NETX command line. The selected NAT64 solution had to be integrated into these two parts.

5.1 Netc – NETX command line integration

One of the certain benefits of NETX is its command line - netc. NETX is a collection of Linux programs and tools and thanks to netc it is possible to control them all in a uniform and standardized way. Netc can be called Industry Standard CLI or Cisco-like command line. It follows that even though Linux is running in the background, the configuration

process works like in any conventional network device. A network administrator works with a familiar configuration syntax, but if he encounters an advanced task, he can switch to Linux shell. In the shell environment, he can take full advantage of the Linux system. A simple example of netc syntax and the transition to the bash can be seen in listing 5.1.

```
rt-netx-d# show interface
INTERFACE       STATE          RX                    TX
                             b/s      p/s          b/s      p/s
ge1          A 100M-FD      3.6k     5.9         13.0k     4.5
ge2               down      0.0      0.0          0.0      0.0

rt-netx-d# interface ge1 ipv6 address 2001:db8:42:42::1/126
rt-netx-d# shell
[root@rt-netx-d ~]# uname -a
Linux rt-netx-d.net.vutbr.cz 3.10.0-862.el7.x86_64 #1 ...
```

Listing 5.1: Example of netc syntax and transition to Linux shell

The netc command line language has a tree structure. Each word in the command line moves through a syntactic tree and specifies the command to be nailed. Commands are of three types: *set*, *unset*, *show*. The semantic meaning of the *set* command is to perform an action/configuration and is not introduced by any prefix. By contrast, the *unset* command is used to negate the *set* command and is introduced by the word ***no***. The *show* is used to view the status and is introduced by the word ***show***. For example:

1. ipv6 route ::/0 2001:db8:0:1::1
2. ***show*** ipv6 route
3. ***no*** ipv6 route ::/0 2001:db8:0:1::1

The above example demonstrates work with IPv6 routes. All three commands are based in the same *ipv6 route* subtree. Command 1) performs a configuration, and command 3) then undo it with the keyword ***no***. Command 2) uses the keyword **show** and shows the current status.

5.1.1 Netc command line design for NAT64 section

Working with NAT64 will be placed in *ipv6 nat64* subtree. There will be four more main sections: instance, pool4, bib, and options.

Instance

1. ipv6 nat64 instance `<nat64-prefix>` `<p2p-ipv6[1]>` `<p2p-ipv4>`
2. ***no*** ipv6 nat64 instance
3. ***show*** ipv6 nat64 instance

The instance subtree is the cornerstone of the NAT64 configuration. It creates/destroys a Jool instance and the entire virtual network namespace.

[1] P2P – Point to point

Command 1) is the only command that is required to put the entire NAT64 into operation. `<nat64-prefix>` specifies the NAT64 prefix of the translated traffic – typically 64: ff9b::/ 96. `<p2p-ipv6>` and `<p2p-ipv4>` are the address ranges from which to address the point-to-point link between the regular and virtual NAT64 network namespaces. After performing this command, the following actions are performed:

1. Mode probing the Jool kernel module.
2. Creating a virtual network namespace.
3. Creating a virtual P2P interfaces.
4. Assigning one of the interface to the namespace.
5. Setting up the interfaces and setting up the addresses.
6. Creating a Jool instance with the NAT64 prefix in the namespace.
7. Setting IPv4 and IPv6 default routes from the namespace.
8. Setting route with the NAT64 prefix to the namespace.

Command 2) removes the Jool instance and the virtual network namespace. The virtual P2P interfaces and routes are removed automatically with the namespace.

Command 3) shows NAT64 status summary information and serves primarily to verify the configuration and its deployment.

```
rt-netx-d# ipv6 nat64 instance 64:ff9b::/96 2001:db8:0:64::/64 192.0.64.0/30
rt-netx-d# show ipv6 nat64 instance
Stateful NAT64 instance is runnnig
Is enabled: Yes
NAT64 prefix: 64:ff9b::/96
Pool4: 192.0.64.4/30
Pool4 route inserted: Yes
Pool4 Jools IPv4 counts: TCP 4, UDP 4, ICMP 4
BIB Jools entry counts: TCP 1, UDP 1, ICMP 1
```

Listing 5.2: Example of working with netc in instance section

Pool4

1. ipv6 nat64 pool4 `<pool-v4>`
2. ***no*** ipv6 nat64 pool4 `<pool-v4>`
3. ***show*** ipv6 nat64 pool4

Pool4 defines an IPv4 subnet that is used for translation.

Pool4 configuration is not required. If pool4 is not specified, the IPv4 address that is configured on the interface in the virtual namespace is used. However, a configuration without pool4 is not recommended, as the number of transport addresses is then limited. For address translation without pool4, port numbers 61001 to 65535 are then allocated.

Jool divides Pool4 into three categories, depending on which higher layer protocol is used - TCP, UDP, ICMP. And for each address individually, it requires specifying the port[2] numbers that are intended for translations. This detailed approach is too complicated for our purposes. Therefore, when configuring pool4 with netc, each IPv4 address from the subnet is used for all higher layer protocols with all available ports. It is the same approach we encounter with conventional routers.

Command 1) assign every IPv4 address from the subnet to the pool. IPv4 addresses are assigned to all three higher layer protocols with port numbers 1-65535. The assigned subnet needs to be routed to the virtual namespace. Therefore, after adding pool4, a static route is generated.

Command 2) flushes Jool's pool4 and removes the static route.

Command 3) shows the pool4 from Jool's point of view. Sequentially displays addresses and ports for TCP, UDP, and ICMP.

```
rt-netx-d# ipv6 nat64 pool4 192.0.64.4/30
rt-netx-d# show ipv6 nat64 pool4
+------------+-------+--------------------+-----------------+-------------+
|       Mark | Proto |     Max iterations |         Address |       Ports |
+------------+-------+--------------------+-----------------+-------------+
|          0 |   TCP |       2047 ( auto) |      192.0.64.4 |     1-65535 |
|            |       |                    |      192.0.64.5 |     1-65535 |
|            |       |                    |      192.0.64.6 |     1-65535 |
|            |       |                    |      192.0.64.7 |     1-65535 |
+------------+-------+--------------------+-----------------+-------------+
+------------+-------+--------------------+-----------------+-------------+
|       Mark | Proto |     Max iterations |         Address |       Ports |
+------------+-------+--------------------+-----------------+-------------+
|          0 |   UDP |       2047 ( auto) |      192.0.64.4 |     1-65535 |
|            |       |                    |      192.0.64.5 |     1-65535 |
|            |       |                    |      192.0.64.6 |     1-65535 |
|            |       |                    |      192.0.64.7 |     1-65535 |
+------------+-------+--------------------+-----------------+-------------+
+------------+-------+--------------------+-----------------+-------------+
|       Mark | Proto |     Max iterations |         Address |       Ports |
+------------+-------+--------------------+-----------------+-------------+
|          0 |  ICMP |       2047 ( auto) |      192.0.64.4 |     1-65535 |
|            |       |                    |      192.0.64.5 |     1-65535 |
|            |       |                    |      192.0.64.6 |     1-65535 |
|            |       |                    |      192.0.64.7 |     1-65535 |
+------------+-------+--------------------+-----------------+-------------+
```

Listing 5.3: Example of working with netc in pool4 section.

[2]Identifier in case of ICMP

BIB

1. ipv6 nat64 bib `<ip-port-v4> <ip-port-v6>`
2. ***no*** ipv6 nat64 bib `<ip-port-v4> <ip-port-v6>`
3. ***show*** ipv6 nat64 bib

Binding Information Base (BIB) is a table (collection of tables for TCP, UDP, ICMP) that keeps a binding between a source IPv6 transport address and a source IPv4 transport address. Entries to the table are dynamically inserted based on active connections. It is also possible to add a static record to the table and create a permanent binding. A static record is used if a connection from the IPv4 network needs to be established without prior communication — an analogy of IPv4 port forwarding.

Command 1) adds a static record to BIB tables. Jool, like in pool4, distinguishes between BIB for TCP, UDP, and ICMP. To facilitate configuration and more significant analogy to conventional routers, the command adds a record to all three tables.

Command 2) removes all three entries from the BIB tables.

Command 3) sequentially displays BIB records for TCP, UDP, ICMP.

```
rt-netx-d# ipv6 nat64 bib 192.0.64.5:80 2001:db8:0:42::5.80
rt-netx-d# ping 64:ff9b::8.8.8.8
rt-netx-d# show ipv6 nat64 bib
[Static TCP] 192.0.64.5#80 - 2001:db8:0:42::5#80
[Static UDP] 192.0.64.5#80 - 2001:db8:0:42::5#80
[Static ICMP] 192.0.64.5#80 - 2001:db8:0:42::5#80
[Dynamic ICMP] 192.0.64.5#606 - 2001:db8:0:64::1#16363
```

Listing 5.4: Example of working with netc in bib section

Options

1. ipv6 nat64 options `<option-key> <option-value>`
2. ***no*** ipv6 nat64 options `<option-key> option-value>`
3. ***show*** ipv6 nat64 options

The options subtree serves to configure Jool instance properties. The properties are defined as key/value pairs. Through the netc *ipv6 nat64 options* command it is possible to configure all available Jool properties except NAT64 prefix and properties related to state sharing across multiple Jool instances. Configuring NAT64 prefix takes place in the instance subtree, see above, and the state sharing between multiple Jool instances is not supported by netc. The complete list of all available options and its meaning can be found in Jool's official documentation[3].

Command 1) sets the value for the selected key.

Command 2) sets a default value for the selected key.

Command 3) shows all instance properties.

[3] https://www.jool.mx/en/usr-flags-global.html

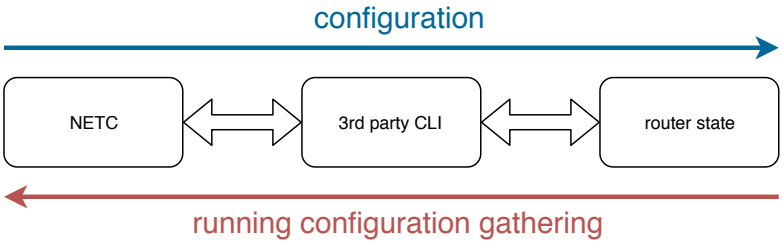


Figure 5.1: Netc operation flow diagram

```
rt-netx-d# ipv6 nat64 options address-dependent-filtering TRUE
rt-netx-d# show ipv6 nat64 options
  manually-enabled: true
  zeroize-traffic-class: false
  override-tos: false
  tos: 0
  mtu-plateaus: 65535,32000,17914,8166,4352,2002,1492,1006,508,296,68
  address-dependent-filtering: true
  drop-icmpv6-info: false
  ---
```

Listing 5.5: Example of working with netc in bib section

5.1.2 Details of NAT64 section implementation

Since version 1.14.0, netc has focused more on modularity and has started to use YAML[4] files for module definitions. The YAML files are used to define a command line structure and to specify the actions to be performed. Each logical unit should be defined in its YAML file. In our case, the whole ipv6 nat64 subtree is defined in netc.d/jool-nat64.yml.

```
NODE:       'ipv6/nat64/bib/'
DESCR:      'Binding Information Base'
SHOW:       'xcfg-netc-jool bib show'
---
NODE:       'ipv6/nat64/bib/%IP4PORT/'
GET:        'xcfg-netc-jool bib get ip4port'
---
NODE:       'ipv6/nat64/bib/%IP4PORT/%IP6PORT/'
GET:        'xcfg-netc-jool bib get ip6port'
SET:        'xcfg-netc-jool bib set %1 %2'
UNSET:      'xcfg-netc-jool bib unset %1 %2'
```

Listing 5.6: Sample from the YAML definition file for BIB subtree.

Sample 5.6 shows the definition for the ipv6 nat64 bib subtree. In addition to the *set* *unset* and *show* items, the significance of which has already been highlighted above, there is another necessary entry – *get*.

The configuration process through netc is shown in Figure 5.1.

[4]YAML – human-readable data serialization language.

When a netc command is called, netc translates this command to a third party utility call that handles the called part. For example, when *ipv6 route ::/0 2001:db8::cafe* is entered, the netc command is translated to the ip utility call, specifically to *ip -6 route add ::/0 2001:db8::cafe*. The result of the operation is a new state of the router, namely the default route is inserted into the routing table. The process is shown in Figure 5.1 with a blue arrow and corresponds to the *set unset* and *show* command types.

At the same time, there must also be a backward mapping. Netc must be able to determine from the state of the router which commands have it entered into this state. This backward mapping is required to determine the running-config. Running-config is a sequence of commands that lead to the current configuration state. Saved running-config becomes a startup-config, which is a sequence of commands to be executed to put router it into the configured state after a boot. Netc by default does not hold any configuration state. Configuration state is derived by reverse transformations. For example, if the default route as mentioned above is configured directly to by the ip route utility without using netc, netc would include the command to the running-config because it would be reversed mapped from the routers state. The process is shown in Figure 5.1 with an orange arrow and corresponds to the *get* command type.

The action *get* is called when gathering the running-config. The result of the call must be all configured entries for the called non-terminal. In the above example 5.6, the get action in the first case returns all configured IPv4 transport addresses and in the second case corresponding IPv6 transport addresses.

```
rt-netx-d# show running-config
!
ipv6 nat64 instance 64:ff9b::/96 2001:db8:0:64::/64 192.0.64.0/30
ipv6 nat64 options address-dependent-filtering TRUE
ipv6 nat64 pool4 192.0.64.4/30
ipv6 nat64 bib 192.0.64.5:80 2001:db8:0:42::5.80
!
```

Listing 5.7: Sample from running-config section NAT64.

Translation of commands between netc and third-party utility can be simple, and in this case, it is possible to perform mapping directly in a YAML file. However, in most cases, it is not so straightforward, and an auxiliary script is used. The script can be programmed in any language, typically Perl or Bash, to maintain a unified platform. In the case of Jool integration, the ./bin/xcfg-netc-jool Perl script is used to cover all netc calls.

```
sub actionBibSet {
        if (system("ip netns exec nat64NS jool -i netc instance display &> /dev/null") != 0){
                syslog(LOG_INFO, "Jool instance is not running - bib set canceled");
                print("NAT64 instance has to be running!");
                return;
        }

        my $trans4Arg = shift @ARGV;
        my $trans6Arg = shift @ARGV;

        my $trans4 = $trans4Arg =~ s/\:/\#/r;
        my $trans6 = $trans6Arg =~ s/\./\#/r;

        my $commandBase = "ip netns exec nat64NS jool -i netc bib add";

        system(sprintf("$commandBase %s %s --tcp", $trans6, $trans4));
        system(sprintf("$commandBase %s %s --udp 2>&1 | logger", $trans6, $trans4));
        system(sprintf("$commandBase %s %s --icmp 2>&1 | logger", $trans6, $trans4));
}
```

Listing 5.8: Code sample from ./bin/xcfg-netc-jool – procedure for show ipv6 nat64 bib ntec call

In the code example 5.8, a procedure that serves a set action in the BIB section is shown. Generally, these operating procedures consist of three parts: checking input conditions, transforming input, calling third-party utility. In example 5.8, a check is made to see if the Jool instance is active. If the instance is not active, it is not possible to continue. Then, a transformation of the input transport addresses is performed. Netc uses a notation other than Jool CLI by default. In our case, the transformation is performed by a simple regular expression. Finally, a Jool CLI call is made. It can also be seen in the example that, as already mentioned, Jool makes the BIB for TCP, UDP and ICMP different, so the call is made three times. One of the advantages of the configuration script is that it logs to the syslog in detail. In case of a problem, it is possible to check the syslog where every configuration step is verbosely recorded.

5.2 Package building and distribution via RPM

Software distribution and updates to NETX installations are done through the RPM package system. NETX manages its own RPM repositories. Since NETX is based on the Linux distribution CentOS, the core repository content is the same as the CentOS. In addition to the usual packages, NETX builds its own packages whether its a specially adjusted kernel, the netc command line, or modified standard package.

Package build in RPM is controlled by a SPEC file. SPEC file is a prescription that defines how to create an installation package from the source code. In most cases, it is as follows. The source application is translated, and the resulting binary files are included in the RPM package. Along with binary files, information, where the binary files should be copied on the destination station, is also stored. The RPM installer during the installation copies the files as prescribed in the package. Of course, translation and binary files are not required, and any file can be subject to the RPM package. RPM can also run scripts

during installation, in addition to copying files. More information about RPM packaging can be found in [1] [30].

Jool does not provide pre-built software packages and is only available as source code. For these reasons, it was necessary to include the assembly of the Jool packages into the NETX packages assembly process. It means devising the installation process and describing it with the SPEC file and harmonize it with the automatic system in NETX.

5.2.1 Building Jool RPM package

To build a Jool package, two components need to be resolved, the userspace CLI and the kernel module. The userspace CLI can be packed in the usual way. On the other hand, the kernel module has several issues to be solved. Since it is a kernel module, it cannot be built in a kernel environment other than the one that the module is targeted to. The solution is to build the module separately for each version of the kernel or build the module on the target station. Jool offers the possibility to build the module using DKMS. DKMS was used throughout the testing. Thus, this method has proven itself and has made the build of the module considerably easier. For this reason, DKMS is the preferred method when assembling the package.

Dynamic Kernel Module Support (DKMS) is a Linux framework/subsystem that manages kernel modules that are not a direct part of the kernel. The source code of the module is registered in the DKMS, and the system then takes care of its compilation and installation. When the kernel is updated, the system re-compiles and installs the module itself.

```
[root@rt-netx-e ~] cp -r $module-$version /usr/src/
[root@rt-netx-e ~] dkms add  $module/$version
[root@rt-netx-e ~] dkms build  $module/$version
[root@rt-netx-e ~] dkms install  $module/$version
```

Listing 5.9: Demonstration of work with DKMS

Example 5.9 shows steps to install a module through DKMS. The source code for modules is located in the `/usr/src` directory by default. Keeping the correct naming convention - *name-version* is important. If there is a configuration file for DKMS in the module source code, the module can be installed.

The creation of a package relying on DKMS looks like this. A package is created that does not contain any binary files, only includes the complete source code. When the package is installing, it copies the source codes to `/usr/src` directory and runs the DKMS commands in post-install hook, see example 5.9.
DKMS offers command `dkms $module/$version mkrpm --source-only` to create such an RPM package automatically. However, this package would not contain the necessary Jool CLI application, and in the end, this method would not fit into the NETX automatic package assembly flow, see Chapter 5.2.2.

Finally, the Jool package build and installation process is as follows. The Jool CLI source code is compiled, and the result binary files are included in the package. Besides, the complete source is added to the package. The final package is placed in the repository where the NETX router will download it. When installing the package, the Jool CLI binaries are copied to the required locations and the source code is copied to `/usr/src`. After that, the post-installation hook runs and adds the module to DKMS and starts the DKMS installation. In case of uninstalling, the package manager will delete the copied

files (CLI binaries and the source code), but before that, a pre-uninstall hook triggers the module remove from the DKMS.

5.2.2 NETX automatic package build flow

NETX has an automated package assembly system. The main idea of the system is to build packages in containers, specifically in Docker. There are many advantages to packaging in containers. One of the great benefits is the independence of the operating system. The NETX (CentOS) package can thus be built on any operating system running Docker. A further advantage is that a clean environment is prepared for each assembly and disappear upon completion of the work. However, we also encounter certain limits. Because the container shares a kernel with a host PC, we are limited in kernel module translations. In case of a kernel module, it would be possible to translate and build the package only for the same kernel as the host PC kernel.

In the case of Jool, we decided to translate the kernel module during the installation on a target station, and therefore this problem is not critical. The principle works as follows. An essential Docker image is created, based on the target platform. In our case, the image is based on CentOS 7.5. The essential image includes only the basic utilities that are needed in most package build cases. When assembling the package, a container is created from the image. Source code folder and a folder for the resulting package are mounted. The system detects from the SPEC file which tools and libraries will be needed to build, and those that are missing from the base image are downloaded. Then it builds the packages and returns the result to the prepared folder. Once the process is complete, the container is removed.

To automate the entire process, NETX uses the docker-rpm-builder [6] tool to take care of the entire process. Package source codes are managed in a GIT repository. Including a new package in the system means creating a new folder in the repository. If the correct directory structure is followed, the system will take care of everything else.

```
.
|-- jool
|   |-- SOURCES
|   |   `-- jool-4.0.0.tar.gz
|   `-- SPECS
|       `-- jool.spec
|-- Makefile
| ...
```

Listing 5.10: Jool directory in NETX package build repository.

In example 5.10, a Jool section of the package build repository structure can be seen. The package is built with the `make jool` command.

Chapter 6

Jool's performance bottlenecks analysis

The Jool solution has been selected for NETX platform integration. However, from UDP results, Jool did not achieve the expected packet throughput. The same pps result as regular IPv4 NAT was expected. For this reason, bottleneck inspection was performed.

6.1 Performance analysis using perf tool

Figure 6.1a shows perf results during test execution - under the full load of the NAT64 traffic. From the results it is evident, that processor spends a huge portion of time waiting for access to a critical section. We are working with a stateful algorithm where each packet that flows through NAT64 creates a state so that all packets of the same flow are mapped to the same transport addresses and packets that flow back into the network are passed to the host who initiated the connection. Since packets are processed in parallel, the need for critical sections and mutual exclusion is expected.

In a further analysis, Figure 6.1b indicates that the CPU threads are waiting at the lock in the *rfc6056_f* function. When analyzing this function, it was found that the function is used to count an MD5 checksum based on packet header data. As the name of the function suggests, it is an implementation of RFC 6056, which describes how to randomly assign port numbers, so that they cannot be predicted and its knowledge cannot be used in network attacks [19]. Here, the MD5 checksum is calculated using the *crypto_shash* family functions in a critical section. An explanation of why calling these functions inside the critical section is missing. A critical section would be meaningful if calling these functions is not thread-safe. This would mean that a parallel call to the same function could lead to data inconsistency. However, no indications have been found to inoculate this problem during analysis. Also, there is a commentary where the author himself suggests that it would be appropriate to examine this part of the code to improve performance. For the experiment, the need for a critical section was removed.

The result was almost no performance increase in output. Figure 6.1c shows the perf output after removing the critical section. We may notice that now the threads are waiting in the *get_random_bytes* function. The *get_random_bytes* function is part of the C standard library and is used to generate random values. It is a quality random number generator. Quality random number generators can generate reliably random values but at a low pace. *get_random_bytes* is used to generate a random identifier in the IPv4 packet

headers. Since every translated packet needs its IPv4 identification, it is likely that the used random number generator will not be able to generate values fast enough. Challenges with generating IPv4 identification when translating packet will be discussed in the following section. For the test, a static value was used instead of random values for the experiment. Pps performance growth was about 50%.

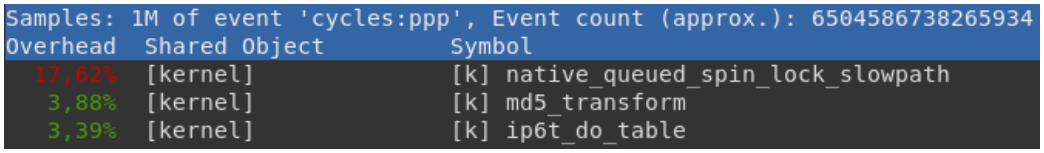
In the next analysis by the perf tool, Figure 6.1d, it was found that the threads are now waiting on the lock, which controls access to a table that manages connection records. This situation is already in line with expectations, and the next step would be to optimize the work within the critical section.

6.2 IPv4 identification field and fragmentation issues

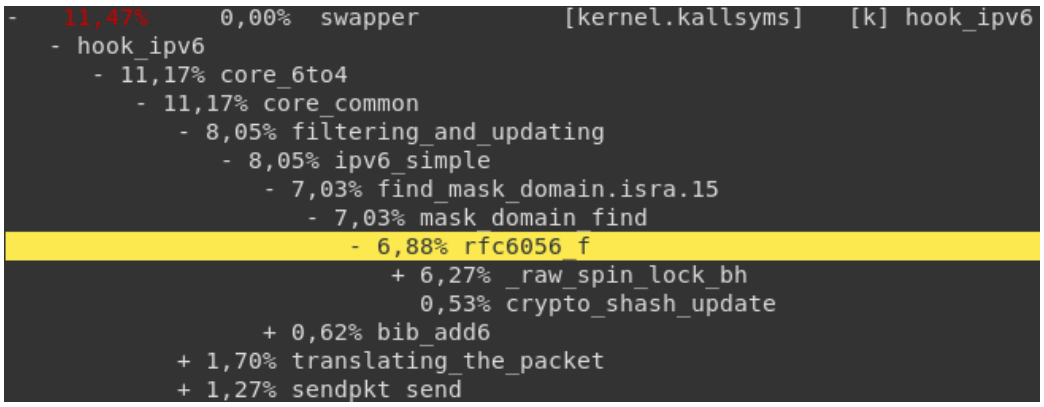
IPv4 identification (IPv4-ID) is a 16-bit value in the IPv4 packet header. The purpose of this value is to identify packets in a flow. The flow is defined as a source/destination address and an upper layer protocol. Every packet in the flow needs a unique identifier for the time it can occur in the network. There was a big discussion about the usage of these unique numbers. Finally, RFC 6864 clearly laid down rules for using the field. RFC 6864 says IPv4-ID can only be used for packet fragmentation purposes [28]. In IPv4, fragmentation can occur in two modes. If a flag *do not fragment* (IPv4-DF) is set to 0 in an IPv4 packet, routers along the path will do fragmentation if required. If the IPv4-DF is set to 1, the fragmentation is done by the transmitting side and if the router is unable to forward the packet for reasons of packet size it informs the transmitting side by ICMPv4 message. If the IPv4-DF is set to 1 and the transmitting side is responsible for the packet fragmentation, RFC 6864 says the IPv4-ID is optional and can be arbitrary. Otherwise, IPv4-ID is mandatory, and its uniqueness requirement remains. However, RFC 6864 admits that the need for the uniqueness of this value is difficult to accomplish and is often overlooked. For example, if we consider a 1500 byte packet flow at a 1 Gbps rate, the 16-bits space is exhausted in less than seconds. RFC 4963 suggests that the most appropriate approach to the IPv4-ID is to generate a random value for the first packet in the flow and then increment it with each packet [18]. This ensures that the 16-bit space is used as effectively as possible.

IPv6 approach to fragmentation is different. In IPv6, the transmitting side is always responsible for the fragmentation - the equivalent to IPv4-DF set to 1. The minimum MTU in IPv6 is 1280 bytes - 68 bytes in IPv4. IPv4-ID in IPv6 header (without extension) has no equivalent. These differences lead to many fragmentation problems. The RFC that defines packet translation has been released in three versions: RFC 2765 [20], RFC 6145 [32], RFC 7915 [5], and approach to fragmentation and IPv4-ID generation in packet translations has been modified in each version. In the original RFC 2765, it was determined that the translated IPv4 packet would always have the IPv4-DF set to 1 and the IPv4-ID set to 0. This eased approach turned out to be a problem mainly due to different minimal MTUs between protocol versions. Therefore, in the following RFC 6145, it was determined that this approach is only suitable for packets with a size $>$ 1280 bytes or $<=$ 88 bytes. In current RFC 7915, this option remains only to IPv4-DF flag and IPv4-ID must always be generated.

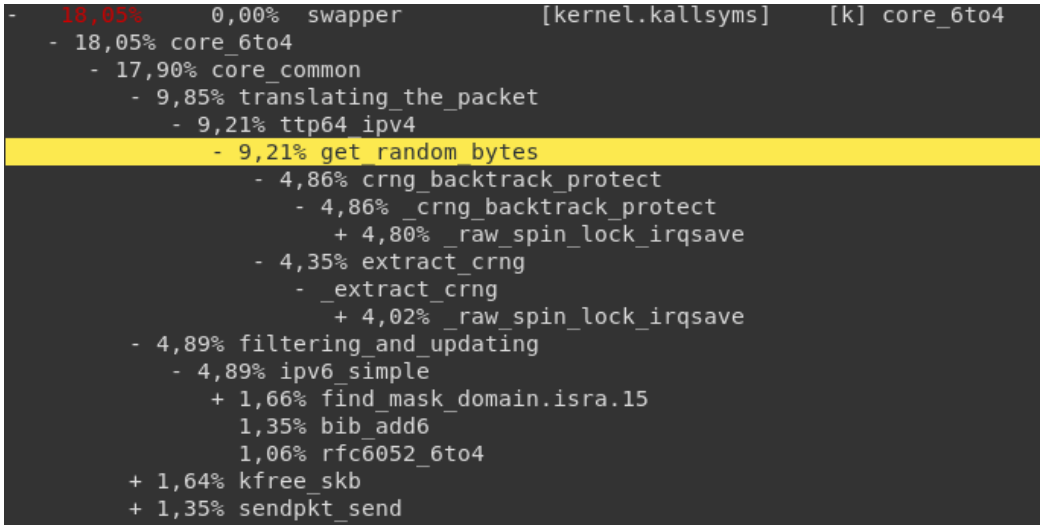
Examining Tayga's implementation and Cisco NAT64 solution, it was found that they always set IPv4-DF flag to 1 and sets IPv4-ID to 0. Jool adheres to the latest RFC but generates an IPv4-ID for each packet randomly. As a result, the IPv4-ID is repeated before it is necessary and at the same time, it reduces performance.



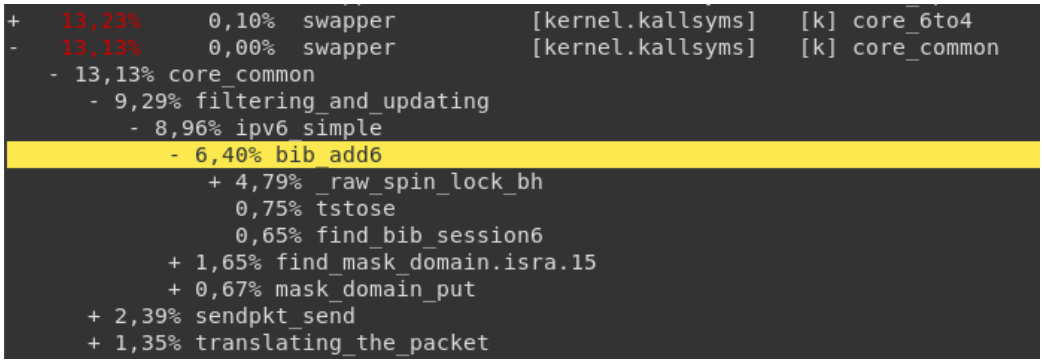
(a) Quick result before experimentation. The *native_queued_spin_lock_slowpath* indicates a large wait for access to the critical section.



(b) More detailed result before experimenting. The result indicates waiting in the function *rfc6056_f*.

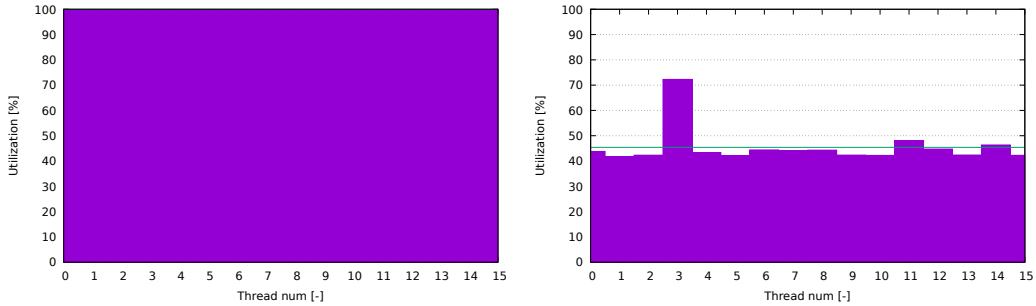


(c) Result after removing critical section while working with an MD5 checksum. The result indicates waiting in the function *get_random_bytes*



(d) Result after excluding random IPv4 packet identifier generation. The threads are now waiting to work with state table of the algorithm.

Figure 6.1: Perf outputs during tests execution - under the full load of the NAT64 traffic.



(a) Standard Jool version 4.0.0 (b) Jool with proposed optimization

Figure 6.2: Comparison between CPU threads utilization during UDP test with packet size 1500 B before and after optimization

6.3 Cooperation with developers and the resulting performance gain

The above findings have been consulted with Jool developer Ing. Alberto Leiva Popper [10]. It was confirmed that closing the *crypto_shash* functions to the critical section was unnecessary. The use of a critical section was preventive, and there is no reason for it. An IPv4-ID generation has been discussed, and improvement has been proposed. The IPv4-ID generating is now done as the kernel does it when sending IPv4 packets. The *get_random_bytes* was replaced by *___ip_select_ident*. Function *___ip_select_ident* generates only one random number for each flow and then increments it with each subsequent packet. This method is in line with RFC 4963 and RFC 7915. Along with those changes, another call to *get_random_bytes* has been removed, which has proved to be unnecessary.

Using the *___ip_select_ident* function demanded a higher version of the kernel >= 4.2. The used NETX router was therefore upgraded to 4.4.178-1.el7.elrepo.x86_64 kernel. Tests showed a 58.3 % increase in pps performance compared to standard 4.0.0 version. Pps throughput rose from 0.72 Mpps to 1.14 Mpps. This noticeable increase in performance can also be seen in the process load decrease, see Figure 6.2.

The changes were declared successful and are scheduled for the next Jool 4.0.1 release.

Chapter 7

Conclusion and results summary

At the beginning of my work, I introduced readers to the problems of transition algorithms between IP protocols. I introduced the conceptual problems associated with transition algorithms and what are their general solutions. Based on these concepts, I described the Stateful NAT64. I tried to capture all the fundamental properties, and the residues refer to the appropriate RFCs. In the NAT64 excerpt, I have included all essentials such as address mapping, DNS64, 464XLAT, so that the reader has a complete understanding of how NAT64 works.

In my research on existing NAT64 implementations, I came across four open-source candidates – Tayga, Jool, Ecdysis, WrapSix. Three solutions were carefully examined and tested on the lab router. The two solutions that have proved to be acceptable (Tayga and Jool) have been described in the document. The Ecdysis solution was eliminated at the beginning because it has not been developed for some time, so it does not support new kernels. The WrapSix solution even after several attempts still had errors in the form of duplicate packets and was therefore also eliminated. With the remaining solutions, besides the descriptions, necessary procedures for putting the solutions into operation were introduced. I have also been slightly devoted to Linux network namespaces, which are an interesting option for running NAT64 solutions.

In addition to open-source solutions, I also explored the support for NAT64 with well-known network router vendors. The research has shown that there is already a ready-made solution on the market. For example, Juniper supports NAT64 on two device series (SRX and MX) and provides excellent documentation.

I have passed two chosen solutions to performance tests. I have used a methodology that, in addition to examining a traffic throughput, has also revealed the network performance impact and resource impact on the router's processor.

I encountered several challenges during testing because of the relatively high 10 Gbps throughput requirements. The use of the Iperf tool proved to be almost unusable at such speeds. The examined router was not fully utilized, but the test stations could not generate more intense traffic. Overall, Iperf's results were unstable with a large deviation. For UDP tests, I was recommended to use the PF_RING tool, which did not suffer from these problems, and the UDP results are far more accurate.

From the measured results, I concluded that a solution that can reach the specified 10-gigabit throughput must be able to utilize multiple CPU threads. Unfortunately, it turned out that Tayga does not meet this feature. Tayga's unmistakable strength is its simplicity, but until Tayga is not able to harness multiple processor threads, it can not be recommended for NETX router. At the same time, Tayga is very sensitive to congestion.

When congested, its performance drops dramatically. Jool with optimization meets the required throughput, but its packet per second performance is still not ideal. Jool is a comprehensive solution with many advanced features, and its development is still active. For these reasons, Jool was elected for integration.

Integration into NETX was successful. A command line scheme for Jool control has been designed. The scheme was programmed and integrated into the netc command line. Netc now supports working with Jool instance, pool and BIB configurations, translation parameters tuning, and several verifications commands. Jool has been included in the NETX package repositories. Both the Jool CLI application and the Jool kernel module were included in one RPM package. The CLI application is distributed in binary form; the kernel module is compiled during installation using DKMS.

Jool performance bottlenecks analysis has been performed. The findings have been consulted with Jool developers. In collaboration with the developers, a significant performance increase of 58.3 % was achieved.

In the follow-up work, it would be possible to extend the netc command line with additional features — for example, multiple IPv4 pools, state synchronization support or multiple instances. In addition, it would be possible to try to find more optimization to achieve even better performance results. Jool also offers a stateless NAT64 (SIIT) variant that could also be integrated into NETX.

Bibliography

[1] Adam Miller, M. D., Maxim Svistunov: RPM Packaging Guide. 2019.
Retrieved from: https://rpm-packaging-guide.github.io/

[2] Arista: Arista 7170 Series.
Retrieved from: https://www.arista.com/en/products/7170-series

[3] Cisco: Cisco Express Forwarding Overview.
Retrieved from: https://www.cisco.com/c/en/us/td/docs/ios/12_2/switch/configuration/guide/fswtch_c/xcfcef.html

[4] Cisco: Cisco Feature Navigator.
Retrieved from:
https://cfn.cloudapps.cisco.com/ITDIT/CFN/jsp/by-feature-technology.jsp

[5] Congxiao Bao, F. B. T. A. F. G., Xing Li: IP/ICMP Translation Algorithm. RFC 7915. June 2016. doi:10.17487/RFC7915.

[6] Franzoni, A.: docker-rpm-builder.
Retrieved from: https://github.com/alanfranz/docker-rpm-builder

[7] Google: Google Public DNS64. 2018.
Retrieved from: https://developers.google.com/speed/public-dns/docs/dns64

[8] Hewlett-Packard: HP A-MSR Router Series Layer 3 - IP Services Command Reference.
Retrieved from:
https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c02659297

[9] Huawei: NE40E V800R010C00 Configuration Guide - NAT and IPv6 Transition 01.
Retrieved from:
https://support.huawei.com/enterprise/en/doc/EDOC1100028531?section=j04v&topicName=nat64-overview

[10] Jan Pokorný, A. L. P.: ICMx/Jool - Issue 282: NAT64 performance evaluation. 2019.
Retrieved from: https://github.com/NICMx/Jool/issues/282

[11] Jari Arkko, F. B.: Guidelines for Using IPv6 Transition Mechanisms during IPv6 Deployment. RFC 6180. May 2011. doi:10.17487/RFC6180.

[12] Jool: Introduction to Jool. 2018.
Retrieved from: https://www.jool.mx/en/intro-jool.html

[13] Juniper: IPv6 NAT64.
Retrieved from:
https://apps.juniper.net/feature-explorer/feature-info.html?fKey=3635&fn=IPv6%20NAT6

[14] Juniper: NAT64 support for MS-MIC and MS-MPC interface cards.
Retrieved from: https://apps.juniper.net/feature-explorer/feature-info.html?fKey=6055&fn=NAT64%20support%20for%20MS-MIC%20and%20MS-MPC%20interface%20cards

[15] Lutchansky, N.: TAYGA - Simple, no-fuss NAT64 for Linux. 2011.
Retrieved from: http://www.litech.org/tayga/

[16] Martinez, J. P.: NAT64/464XLAT Deployment Guidelines in Operator and Enterprise Networks. draft. April 2018.
Retrieved from:
https://tools.ietf.org/html/draft-ietf-v6ops-nat64-deployment-03

[17] Masataka Mawatari, C. B., Masanobu Kawashima: 464XLAT: Combination of Stateful and Stateless Translation. RFC 6877. April 2013. doi:10.17487/RFC6877.

[18] Matt Mathis, J. H., Ben Chandler: IPv4 Reassembly Errors at High Data Rates. RFC 4963. July 2007. doi:10.17487/RFC4963.

[19] Michael Larsen, F. G.: Recommendations for Transport-Protocol Port Randomization. RFC 6056. January 2011. doi:10.17487/RFC6056.

[20] Nordmark, E.: Stateless IP/ICMP Translation Algorithm (SIIT). RFC 2765. February 2000. doi:10.17487/RFC2765.

[21] ntop: PF_RING - High-speed packet capture, filtering and analysis.
Retrieved from: https://www.ntop.org/products/packet-capture/pf_ring/

[22] Philip Matthews, I. v. B. M. B., Andrew Sullivan: DNS64: DNS Extensions for Network Address Translation from IPv6 Clients to IPv4 Servers. RFC 6147. April 2011. doi:10.17487/RFC6147.

[23] Philip Matthews, M. B., Iljitsch van Beijnum: Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers. RFC 6146. April 2011. doi:10.17487/RFC6146.

[24] Robert E. Gilligan, E. N.: Basic Transition Mechanisms for IPv6 Hosts and Routers. RFC 4213. October 2005. doi:10.17487/RFC4213.

[25] Scott Rose, D. M. R. A. R. A., Matt Larson: DNS Security Introduction and Requirements. RFC 4033. March 2005. doi:10.17487/RFC4033.

[26] S.J.M. Steffann, R. v. R., Iljitsch van Beijnum: A Comparison of IPv6-over-IPv4 Tunnel Mechanisms. RFC 7059. November 2013. doi:10.17487/RFC7059.

[27] Srisuresh, P.; Tsirtsis, G.: Network Address Translation - Protocol Translation (NAT-PT). RFC 2766. February 2000. doi:10.17487/RFC2766.

[28] Touch, D. J. D.: Updated Specification of the IPv4 ID Field. RFC 6864. February 2013. doi:10.17487/RFC6864.

[29] Viagenie: Ecdysis: open-source implementation of a NAT64 gateway. Retrieved from: https://ecdysis.viagenie.ca/

[30] Vrbanec, M.: Packaging kernel modules/drivers using DKMS. 2015. Retrieved from: https://schneide.blog/2015/08/10/packaging-kernel-modulesdrivers-using-dkms/

[31] xHire: WrapSix. Retrieved from: https://www.wrapsix.org/

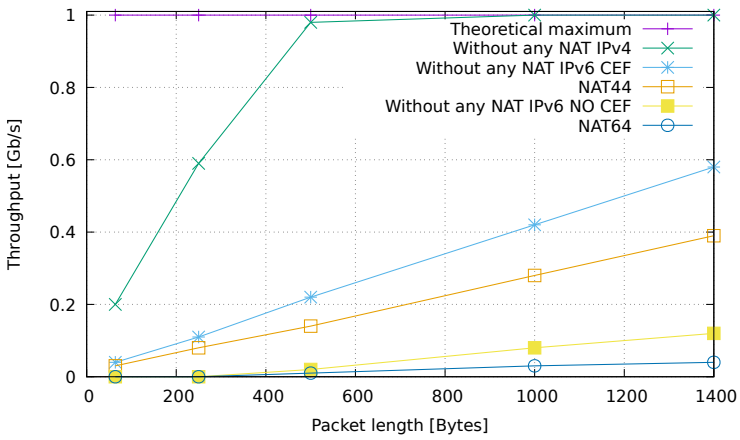
[32] Xing Li, C. B., Fred Baker: IP/ICMP Translation Algorithm. RFC 6145. April 2011. doi:10.17487/RFC6145.

[33] Xing Li, C. H. M. B. C. B., Mohamed Boucadair: IPv6 Addressing of IPv4/IPv6 Translators. RFC 6052. October 2010. doi:10.17487/RFC6052.

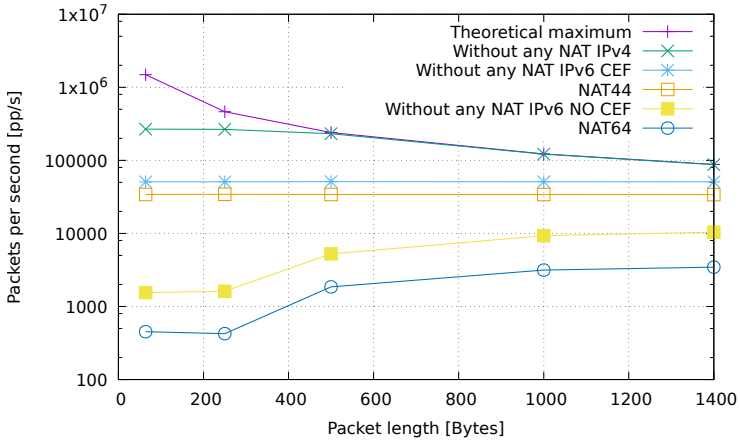
[34] Yilmaz, G.: How do NAT64 and DNS64 work? Retrieved from: http://rtodto.net/how-do-nat64-and-dns64-work-in-ipv6-world-and-srx-config/

Appendix A

UDP traffic measured by PF_RING on Cisco 2911 IOS 15.3(3)M6



(a) Average throughput



(b) Average packet per second

Figure A.1: UDP traffic measured by PF_RING on Cisco 2911 IOS 15.3(3)M6

Appendix B

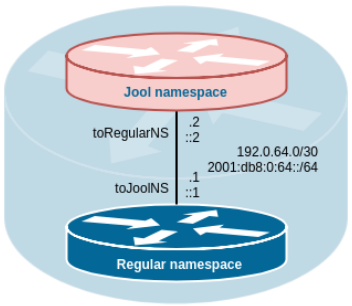
NETX NAT64 documentation

Copy of the created NAT64 documentation for router NETX available at https://docs.netx.as/.

Command reference

For Stateful NAT64 (RFC 6146) the NETX platform uses Jool developed by NIC MÉXICO and Tecnológico de Monterrey

Jool is integrated into the NETX platform using virtual network namespace. After you create a new NAT64 instance, the following virtual topology is created.



NetX router

The Jool system is integrated into the *netc* interface. All NAT64 related commands are available in *ipv6 nat64* context.

Instance

Instance command creates a new virtual network namespace with a Jool instance. For communication between regular network namespace and Jool's network namespace, virtual interfaces are used, which need to be addressed.

```
netx# ipv6 nat64 instance <NAT64 prefix> <IPv6 P2P subnet> <IPv4 P2P subnet>
netx# no ipv6 nat64
netx# show ipv6 nat64
netx# show ipv6 nat64 session
netx# show ipv6 nat64 stats
```

For example:

```
netx# ipv6 nat64 instance 64:ff9b::/96 2001:db8:0:64::/64 192.0.64.0/30
netx# show ipv6 nat64
Stateful NAT64 instance is runnig
Is enabled: Yes
NAT64 prefix: 64:ff9b::/96
Pool4: not set. P2P address is used!
BIB Jools entry counts: TCP 0, UDP 0, ICMP 0
```

The first command argument is a NAT64 prefix. It is possible to use Well-Known prefix 64:ff9b::/96 or any other prefix with length /96. A static route to the Jool's virtual network namespace is automatically created for the selected NAT64 prefix.

The second argument is an IPv6 subnet, that is used for addressing the P2P network between namespaces.

The third argument is an IPv4 subnet, that is used for addressing the P2P network between namespaces.

In this step, NAT64 is already running, and translations are taking place. For translation, the IPv4 address used on the P2P link between the namespaces is used. However, the number of usable ports is limited (61001 to 65535) therefore it is advisable to add an IPv4 pool.

show ipv6 nat64 displays information about nat64. Displays whether the instance and the virtual network namespace are running and if it is enabled. Instance and the virtual network namespace can be running but can be manually disabled via *ipv6 nat64 options manually-enabled false*. *no ipv6 nat64* destroys the instance a deletes NAT64 configuration. Manually disabling is good for temporary disabling the NAT64 without losing current configuration.

Furthermore, you can see if the IPv4 pool is used and how many addresses are available in the pool. If an IPv4 pool is configured the commands check if a proper route is installed.

```
Pool4: not set. P2P address is used!
```

vs

```
Pool4: 80.254.236.128/25
Pool4 route inserted: Yes
```

The last line of the show command displays the current number of entries in the BIB table. The number of records = number of static records + number of currently opened connections.

show ipv6 nat64 session shows currently active sessions/connections.

show ipv6 nat64 stats show some NAT64 stats.

IPv4 pool

Adds a pool of IPv4 address for translating IPv6 sources.

```
netx# ipv6 nat64 pool4 <ipv4 pool>
netx# no ipv6 nat64 pool4 <ipv4 pool>
netx# show ipv6 nat64 pool4
```

For example:

```
netx# ipv6 nat64 pool4 192.0.64.128/25
```

A static route to the Jool's virtual network namespace is automatically created for the selected IPv4 pool.

show ipv6 nat64 pool4 shows IPv4 addresses in the pool in detail as Jool internally manages them.

BIB

Binding Information Base (BIB) is a table that keeps a binding between a source IPv6 transport address and a source IPv4 transport address. Entries to the table are dynamically inserted based on active connections. It is also possible to add a static record to the table and create a permanent binding. A static record is used if a connection from the IPv4 network needs to be established without prior communication — an analogy of IPv4 port forwarding.

```
netx# ipv6 nat64 bib <ipv4 transport address> <ipv6 transport address>
netx# no ipv6 nat64 bib <ipv4 transport address> <ipv6 transport address>
netx# show ipv6 nat64 bib
```

For example:

```
netx# ipv6 nat64 bib 192.0.64.129:80 2001:db8:0:42::5.80
netx# ping 64:ff9b::8.8.8.8
netx# show ipv6 nat64 bib
[Static TCP] 192.0.64.129#80 - 2001:db8:0:42::5#80
[Static UDP] 192.0.64.129#80 - 2001:db8:0:42::5#80
[Static ICMP] 192.0.64.129#80 - 2001:db8:0:42::5#80
[Dynamic ICMP] 192.0.64.5#606 - 2001:db8:0:64::1#16363
```

B.0.1 Jool's instance options

Mantains subset of supported Jool's instance options.

Supported options are:

- address-dependent-filtering
- drop-externally-initiated-tcp
- drop-icmpv6-info
- f-args
- handle-rst-during-fin-rcv
- icmp-timeout
- logging-bib
- logging-session
- manually-enabled
- maximum-simultaneous-opens

- mtu-plateaus
- override-tos
- source-icmpv6-errors-better
- tcp-est-timeout
- tcp-trans-timeout
- tos
- udp-timeout
- zeroize-traffic-class

The option meanings are listed in the official Jool documentation https://www.jool.mx/en/usr-flags-global.html

B.1 NETX NAT64 basic tutorial

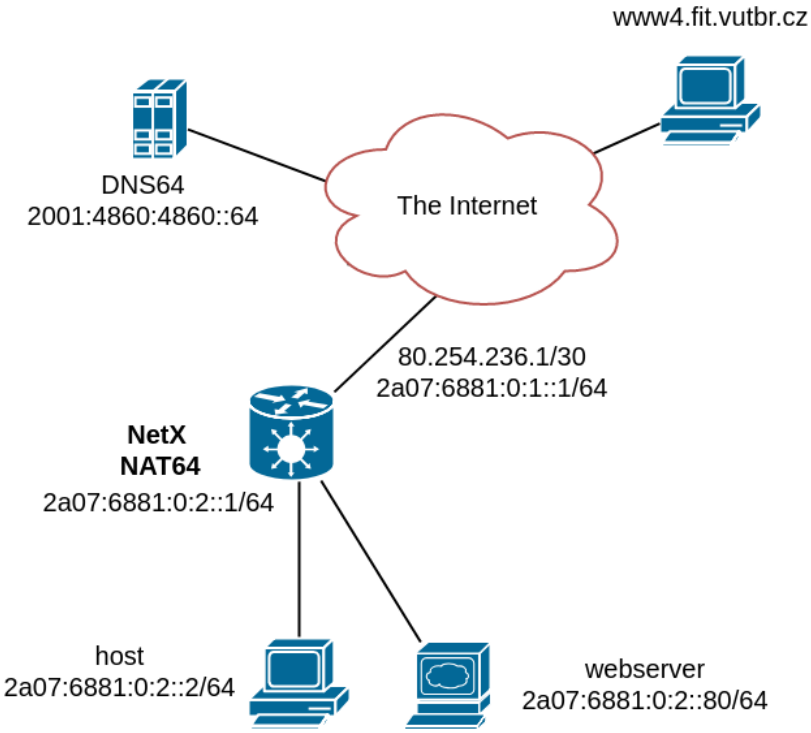
This example shows a simple configuration that connects IPv6 only network with the rest of the Internet. IPv4 only hosts are still reachable through NAT64 AFTR running on NETX router.

Whenever an IPv6 host wants to contact a host on the Internet, it asks DNS64 for address translation. If the DNS64 server detects that the requested service does not have a matching AAAA record, the response is created by concatenation of NAT64 prefix and queried host IPv4 address. All hosts now seem to have IPv6 connectivity. The used DNS64 server in this example is public Google DNS64 server 2001:4860:4860::64.

DNS resolution must take place before work with IP literals. Some applications with hard-coded IPv4 literal could have problems.

There is also a static record in the BIB table so that a web server inside the internal IPv6 only network is also available for external IPv4 only clients.

Topology



Address plan

NETX router uses the following interfaces and addresses:

- Internet connectivity: interface *ge1*, 80.254.236.1/30 2a07:6881:0:1::1/64
- Local IPv6 only network: interface *ge3*, 2a07:6881:0:2::1/64
- NAT64 prefix: 64:ff9b::/96
- IPv4 pool: 80.254.236.128/25
- P2P subnets for NAT64 virtual namespace: 80.254.236.4/30 2a07:6881:0:64::/64

The following commands can be used to set up IP addresses:

```
! set up ge1 addresses
netx# interface ge1 ipv4 address 80.254.236.1/30
netx# interface ge1 ipv6 address 2a07:6881:0:1::1/64
netx# ipv4 route 0.0.0.0/0 80.254.236.2
netx# ipv6 route ::/0 2a07:6881:0:1::2

! set up tge3 addresses
netx# interface tge3 ipv6 address 2a07:6881:0:2::1/64
```

Host config (Linux)

```
host# ip addr add 2a07:6881:0:2::2/64 dev eno1
host# ip route add ::/0 2a07:6881:0:2::1
host# cat "nameserver 2001:4860:4860::64" > /etc/resolv.conf
```

Basic address, default route and DNS server assignment.

Webserver config (Linux)

```
host# ip addr add 2a07:6881:0:2::80/64 dev eno1
host# ip route add ::/0 2a07:6881:0:2::1
host# cat "nameserver 2001:4860:4860::64" > /etc/resolv.conf
```

Basic address, default route and DNS server assignment.

NETX NAT64 config

NAT64 instance creation

```
netx# ipv6 nat64 instance 64:ff9b::/96 2a07:6881:0:64::0/64 80.254.236.4/30
```

IPv4 pool adding

```
netx# ipv6 nat64 pool4 80.254.236.128/25
```

The pool4 subnet has to be properly routed towards the NETX router.

Static BIB record for webserver adding

```
netx# ipv6 nat64 bib 80.254.236.129:80 2a07:6881:0:2::80.80
```

Webserver is now reachable via 80.254.236.129.

Verification

Instance verification

```
netx# show ipv6 nat64
Stateful NAT64 instance is runnig
Is enabled: Yes
NAT64 prefix: 64:ff9b::/96
Pool4: 80.254.236.128/25
Pool4 route inserted: Yes
Pool4 Jools IPv4 counts: TCP 128, UDP 128, ICMP 128
BIB Jools entry counts: TCP 1, UDP 1, ICMP 1
```

DNS64 verification

```
host# dig AAAA +noall +answer www4.fit.vutbr.cz
www4.fit.vutbr.cz.  14399 IN  CNAME tereza.fit.vutbr.cz.
tereza.fit.vutbr.cz.  14399 IN  AAAA  64:ff9b::93e5:916
```

traceroute6 verification

```
host# traceroute6 www4.fit.vutbr.cz
traceroute to www4.fit.vutbr.cz (64:ff9b::93e5:916), 30 hops max, 80 byte
 1  2a07:6881:0:1::1 (2a07:6881:0:1::1)  0.175 ms  0.163 ms  0.152 ms
 2  2a07:6881:0:64::2 (2a07:6881:0:64::2)  0.336 ms  0.339 ms  0.280 ms
 3  64:ff9b::50fe:ec05 (64:ff9b::50fe:ec05)  0.265 ms  0.269 ms  0.252 ms
 4  64:ff9b::50fe:ec01 (64:ff9b::50fe:ec01)  0.431 ms  0.423 ms  0.415 ms
 5  64:ff9b::b901:1904 (64:ff9b::b901:1904)  0.380 ms  0.351 ms  0.339 ms
 6  64:ff9b::93e5:fc71 (64:ff9b::93e5:fc71)  1.993 ms  2.583 ms  1.342 ms
 7  64:ff9b::93e5:fd35 (64:ff9b::93e5:fd35)  0.777 ms  1.141 ms  1.327 ms
 8  64:ff9b::93e5:fd38 (64:ff9b::93e5:fd38)  0.799 ms  0.812 ms  0.691 ms
 9  64:ff9b::93e5:fd3b (64:ff9b::93e5:fd3b)  2.993 ms
 10  64:ff9b::93e5:feda (64:ff9b::93e5:feda)  9.291 ms  9.316 ms  9.445 ms
 11  64:ff9b::93e5:916 (64:ff9b::93e5:916)  0.546 ms  0.510 ms  0.513 ms
```

1 Local network gataway
2 Jool network namespace interface
3 Regular network namespace interface (64:ff9b::80.254.236.5)
4 Internet network gateway (64:ff9b::80.254.236.1)
...

BIB and session verification

```
host# wget www4.fit.vutbr.cz

netx# show ipv6 nat64 bib
[Static TCP] 80.254.236.129#80 - 2a07:6881:0:2::80#80
[Dynamic TCP] 80.254.236.129#62916 - 2a07:6881:0:2::2#54228
[Static UDP] 80.254.236.129#80 - 2a07:6881:0:2::80#80
[Static ICMP] 80.254.236.129#80 - 2a07:6881:0:2::80#80
[Dynamic ICMP] 80.254.236.191#60559 - 2001:db8:111::2#14787

rt-netx-d# show ipv6 nat64 session
TCP
---------------------------------
UDP
---------------------------------
ICMP
---------------------------------
rt-netx-d# show ipv6 nat64 session
TCP
---------------------------------
(V4_FIN_V6_FIN_RCV) Expires in 00:01:40.949
Remote: tereza.fit.vutbr.cz#http        2a07:6881:0:2::2#54228
Local: 80.254.236.129#62916       64:ff9b::93e5:916#80
---------------------------------
UDP
---------------------------------
ICMP
---------------------------------
Expires in 00:00:59.494
Remote: 147.229.9.22#536702001:db8:111::2#15786
Local: 80.254.236.188#5367064:ff9b::93e5:916#15786
---------------------------------
```

Appendix C

Content of the attached data carrier

```
.
|-- thesis-latex
|-- measuring
|-- sources
    |-- netx-docs
    |-- netx-netc
    `-- netx-rpm
```

For more information, see the README files in each folder.